

VANCEA

## DEFINITIONS!!

**IA-32 / 32 bit computing architecture** is an abstract model of a microprocessor specifying the microprocessor's elements, structure and instruction set, based on the previous **Intel 8086 computing architecture**

**CPU = computer's central processing unit**

## X86 MICROPROCESSOR'S STRUCTURE:

1. **ALU** = Arithmetic and logic unit
  - Is a component of the CPU
  - It can apply the instruction code and activate a circuit and utilizes an already fixed program
  - Works only with addition, subtraction, multiplication, and division
  - Works **only** with **addresses!!** (no values or commands)
  - Native architecture **doesn't work** with **real** numbers, **ONLY INTEGERS!** **Mathematical co-processor** take care of the real numbers.
  - Any command goes on a command bus (for not mixing everything) → we need **BIU** (bus interface unit) to **access the operators and operands**
  - Will set accordingly the values from the EFLAGS register

### Operations performed by ALU:

- A) arithmetic operators - bit subtraction and addition (addition used in place of multiplication and subtraction in place of division)
- B) bit shifting operators – shifting the location of a bit to the right or left by a particular number of places, responsible for a multiplication operation
- C) logical operations – AND, OR, NOT, XOR

2. **EU** = Executive unit
  - Runs the machine instructions by means of ALU
3. **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 (*region of memory used to temporarily hold data while it is being moved from one place to another*), 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.

**Word size** = the number of bits processed by a computer's CPU in one go. (data bus size, instruction size, address size are usually multiples of the word size)

## FLAGS:

Flag – indicator represented on a bit

EFLAGS register – shows a synthetic overview of the execution of each instruction  
It has 32 bits, but **only 9 are 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

1. **CF** = transport flag

If in the LPO (last performed operation) there was a transport digit outside the representation domain of the obtained result => CF = 1

It represents the **overflow in the UNSIGNED interpretation!!!**

2. **PF** = parity flag

It is set such that together with the bits 1 from the least significant byte of the representation of the LPO's result, it is **obtained an odd number of 1's**.

PF = 1 => it says that there is an **even number of bits with value 1**

PF = 0 => it says that there is an **odd number of bits with value 1**

3. **AF** = auxiliary flag

Shows the transport value from bit 3 to bit 4 of the LPO's result.

4. **ZF** = zero flag

ZF = 1 => the result of LPO = 0

ZF = 0 => the result of LPO != 0

5. **SF** = sign flag

Is the **first bit** of the LPO

6. **TF** = trap flag

Is a **debugging flag** and it is **NOT SET** by the result of the LPO

TF = 1 => machine stops after every instruction

7. **IF** = interrupt flag

IF = 1 => interrupts are **allowed**

IF = 0 => interrupts will not be handled

8. **DF** = direction flag

Used for operating string instructions.

DF = 0 => string parsing performed from left to right (from beginning to end)

DF = 1 => string parsing performed from right to left (from end to beginning)

9. **OF** = overflow flag

It represents the **overflow in the SIGN interpretation!!!**

If the result of LPO doesn't fit in the reserved space => OF = 1, else OF = 0

Ex: **AF=0**

$$\begin{array}{r}
 100110011 + \\
 011110011 \\
 \hline
 100000110
 \end{array}
 \qquad
 \begin{array}{r}
 93(16) + \\
 73(16) \\
 \hline
 106(16)
 \end{array}$$

CF = 1 PF = 1 SF = 0 ZF = 0

FLAGS CATEGORIES	
Set as <b>direct effect</b> of the execution of LPO	Set by the <b>programmer</b> to influence the next instructions
CF PF AF SF OF ZF	CF TF DF IF

### INSTRUCTIONS TO SET THE FLAGS VALUES

- There are 7 such instructions

CLC => CF = 0	STC => CF = 1	CMC => complements the value of CF
CLD => DF = 0	DTS => DF = 1	
CLI => IF = 0	STI => IF = 1	*can be used by the programmer <b>only on 16 bits!!!!</b> OS restricts the access on 32 bits!!!

### INSTRUCTIONS WHICH USE THE VALUES OF FLAGS

1. **ADC** – add the CF
2. **SBB** – subtract the CF
3. **Conditional Jumps**
  - a. JZ – if ZF = 1
  - b. JNZ – if ZF = 0
  - c. JC – if CF = 1
  - d. JNC – if CF = 0
  - e. JPE – if PF = 1 (parity even)
  - f. JPO – if PF = 0 (parity odd)
  - g. JO – if OF = 1
  - h. JNO – if OF = 0
  - i. JS – if SF = 1
  - j. JNS – if SF = 0

### INSTRUCTIONS TO SAVE THE VALUES OF FLAGS

1. PUSHF – save the values on the stack
2. POPF – pop the values from the stack

## 2'S COMPLEMENT:

At the level of x86 microprocessor, the negative numbers are represented just like the positive numbers, but the most significant bit (sign bit) is 1. In order to represent a negative number, we use 2's complement.

The 2's complement representation of a **negative** number is  $2^n - nr$ , where  $nr$  is the absolute value of the number.

Ex:  $93h = 147 = 1001\_0011 \rightarrow$  **unsigned interpretation**  $1001\_0011 = 147$   
 But in the **signed interpretation**  $1001\_0011 = -109$

### How to compute the 2's complement?

- 1) Subtract the binary contents of the location from  $2^n$ ,  $n$  – representing the number of bits of the location to be complemented

Ex:

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

- 2) Reverse the values of all bits of the initial binary number and add 1 to the obtained value  
 $100\ 0011 \rightarrow 0110\ 1100 + 1 = 0110\ 1101 = 109$
- 3) Starting from the right, we left **unchanged** all the bits until the **first bit 1** (this one inclusive) and **reverse** the values of all the other bits to the left

$\begin{array}{c} 1001\ 0011 \\ \uparrow \dots \uparrow \uparrow \uparrow \\ 0110\ 1101 \end{array}$ 
→ first bit 1

- 4) **Practical ONLY for base 10** – The sum of the absolute values of the 2 complementary values is the cardinal of the set of values representable on that size  
 on 8 bits (1 byte)  $\rightarrow 2^8 = 256$  values ( $[0\dots255]$  or  $[-128\dots+127]$ )  
 on 16 bits (2 bytes)  $\rightarrow 2^{16} = 65536$  values ( $[0\dots65535]$  or  $[-32768\dots+32767]$ )

$$256 - 147 = 109$$

We can talk about **interpretations ONLY when we have numbers written IN OTHER BASES!!!!** **BASE 10 IS ALREADY AN INTERPRETATION!!**

We discuss the **interpretation of numbers that in base 2 START WITH 1!!!** For a binary number starting with 0, its **interpretation will BE THE SAME in both SIGNED AND UNSIGNED interpretations.**

If we start from a representation of the form **1xxx...** of value **+abc**, we **CANNOT OBTAIN** the value **-abc ON THE SAME REPRESENTATION SIZE!!** Its negative variant will also have

Representation in base 2

to begin with 1 as its associated binary representation, but complementing a binary value of the form 1xxx... will provide a binary value **starting with 0 on a representation size identical to the initial one!**

Number X in binary representation begins with	-X begins with	-X is represented on	Examples:
0	1	Same sizeof as X	109 = 01101101 ; -109 = 10010011
1	1	2 * sizeof(X)	147 = 10010011; -147 = 11111111 01101101

## POINTER ARITHMETIC

The arithmetic operations allowed with pointers are the ones that **can result in a correct location in memory**. This meaning, using arithmetic expressions which have addresses as operands.

- i) **Addition of a constant to a pointer** (ex:  $a[7]$ ,  $q + 9$ ) => **a POINTER!**
- ii) **Subtraction of a constant to a pointer** (ex:  $a[-4]$ ,  $p - 7$ ) => **a POINTER!**
- iii) **Subtraction of 2 POINTERS!!** → it results in the **number of bytes** between 2 addresses => **a SCALAR VALUE!!**

**OBS!!** You can't add 2 pointers!! It does not make sense!!

$a[7] = *(a+7) = *(7+a) = 7[a]$  - works both in C and assembly!

**LHS (left hand side of an assignment = L-VALUE = address) = RHS (right h.s. //==// = R-VALUE = contents!)**

LHS – variables or something that can store a value, a content, this is why you see it mentioned as an address above

RHS – things that does not need to store a value, such as numerical values, but it has a content


Ex:

$i = i + 1$  -  $i$  is a L-value, but it can also be a R-value

$1 = i + 7$  – this is a syntax error!!! **1 can't be a L-value**

## **C++ reference variables (&)**

- i) Passing variables by reference at subprogram calls (ex:  $\text{float } f(\text{int } \&x, y)$ )
- ii) Defining ALIASES (ex:  $\text{int } \&j = i$ ,  $j$  becomes an ALIAS for  $i$ )
- iii) Returning L-values as a result of function calls

<pre>int f(x, i){     .....     return v[i]; }</pre> <p>rez = f(a, i)</p>  <p>this is a function which returns</p> <p>a R-VALUE</p>	<pre>int &amp;f(x, i) {     .....     return v[i]; }</pre> <p>f(a, 7) = 79</p> <p>v[7] = 79</p> <p>this is a function which returns</p> <p>a L-VALUE</p>
--	--

## ASSEMBLY LANGUAGE BASICS:

**Machine language** of a computing system = 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



**Symbolic language!!**    Symbols – mnemonics + labels

**An assembler works with:**

- **Labels** (etichete) – user-defined names for pointing to data or memory areas

Ex:

```
repeat1:      ;label of a loop instruction
a db 10       ;label of a variable declaration
```

**OBS:** a valid variable name starts with a **letter**, **\_** or **?**.

- 1) **Code labels** – present at the level of instructions, define the destinations of the control transfer during a program execution (can also appear in data segment)  
Ex: jumps, function calls, subroutines
- 2) **Data labels** – provide symbolic identification for some memory locations (can also appear in code segment)  
Ex: define data directives

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 **offsets of data labels and code labels** are values computable at **ASSEMBLY TIME** and they remain constant during the whole program's run time.

Ex:

Mov eax, 8

Mov eax, [var]

→ A variable once allocated in the memory segment **will never change its location** → information determinable at assembly time based upon the order in which variables are declared and due to the dimension of representation inferred from the associated type

Accessing a variable:

- [a] → access the value of the variable
- a → represents the address of the variable

ex: mov eax, v ;eax ← offset of v

mov eax, [v] ;eax ← the content from address v – 4 bytes

lea eax, [v] ;eax ← offset of v – 4 bytes

[ ] → accessing an operand memory (the content)

- mnemonics* {
- **Instructions** – mnemonics which suggest the underlying action  
→ Assembler **generates the bytes** that codify the corresponding instruction  
Ex: jmp, add, pop

### GUIDE THE PROCESSOR!!!

- **Directives** – indications given to the assembler for correctly generating the corresponding bytes

**GUIDE THE ASSEMBLER!!!** Specify the particular way in which the assembler will generate the object code

Ex: db, dw, end (see a few pages below)

1. **SEGMENT directive** – allows targeting the bytes of code or data emitted by an assembler to a given segment

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

ex: segment data use32 class=data

*optional*

The optional arguments give to the link-editor and the assembler the necessary information regarding the way in which segments must be loaded and combined in memory

### Type:

- a) **code** (or text) – contain code → content cannot be written but can be executed
- b) **data** (or bss) – data → allowing reading and writing, but not execution
- c) **rdata** – segment that can be only be read, containing definitions of constant data

**Alignment** – multiple of the bytes number from which that segment may start (only powers of 2, between 1 and 4096)

- when it is missing, it is **implicitly** considered **align = 1**

**Combination** – how similar named segments from other modules will be combined with the current segment

- a) **public** – it is concatenated with other segments with the same name  
len = sum(all segments)
- b) **common** – the beginning of this segment must overlap with the beginning of all segments with the same name  
len = max(all segments)
- c) **private** – the segment cannot be combined with others
- d) **stack** – segments with the same name will be concatenated, during run time the resulting segment will be the stack segment
- when it is missing, it is **implicitly** considered **combination = public**

**Usage** – allows choosing another word size than the default 16 bits one

**Class** – allows choosing the order in which the link editor puts segments in memory

### 2. Data definition directives

**UNIQUE**

**not UNIQUE**

**UNIQUE**

Data definition = declaration (attributes specification) + allocation (reserving required memory space)

Data definition source line:

*[name] data\_type expression\_list*

**Data\_type:** db, dw, dd, dq, dt

*[name] allocation\_type factor*

**Allocation\_type:** resb, resw, resd, resq, rest – uninitialized data reservation directive

*[name] times factor data\_type expression\_list*

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 are visible in the entire source code, but .a and .b are local labels, that can be accessed with the local name .a or .b until another common label is defined or they can be accessed from anywhere by their complete name var1.a, var2.a, var2.b.

**OBS!!! TIMES** directive can also be applied to instructions

ex: TIMES 32 add eax, edx ;eax = eax + 32 \* edx

3. **EQU directive** – allows assigning a value to a label without allocating any memory space or bytes generation

**OPERANDS** = parameters which define the values to be processed by the instructions or directives (ex: registers, constants, labels, keywords)

- **Location counter** – an integer number managed by the assembler for every separate memory segment
  - ➔ value of location counter = number of the generated bytes correspondingly with the instructions and the directives already met in that segment (**the current offset inside that segment**)
  - ➔ every segment has its **OWN** location counter

**\$** - address of “here”

**\$\$** - address of the start of the current section

**\$ - \$\$ = DISTANCE FROM THE BEGINNING OF THE SEGMENT => a scalar**

#### SOURCE LINE FORMAT:

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

Examples:

here: jmp here ;label + mnemonic + operand + comment

repz cmpsd ;prefix + mnemonic + comment

a dw 23424, 234 ;label + mnemonic + 2 operands + comment

**PREFIXES** - are assembly language constructs that appear optionally in the composition of a source line (explicit prefixes) or in the internal format of an instruction (prefixes generated implicitly by the assembler in two cases) and that modify the standard behavior of those instructions (in the case of explicit prefixes) or which signals the processor to change the default representation size of operands and/or addresses, sizes established by assembly directives (BITS 16 or BITS 32)

**EXPRESSIONS** – operands + operators ➔ **evaluated at ASSEMBLY TIME**

Operators – indicate how to combine the operands for building an expression

**Operands specification modes:**

<b>Immediate operands</b> Direct addressed operands → <b>the offset part only!</b>	Computed at <b>ASSEMBLY TIME</b>
<b>Register operands</b> Indirectly accessed memory operands	Computed at <b>RUN TIME</b>
<b>Memory operands</b> in direct addressing mode (involves address relocation process)	Computed at <b>LOADING TIME</b>

Assembly time: ??: offset

Loading time: 0708:offset

**1. Immediate operands** – constant numerical data

- are specified through different bases, which can be identified using the specific base letter at the end of the numbers (in this case the numbers must start with digits because 0ABCH is different than ABCH – which is interpreted as a symbol) or by adding 0x, 0b, 0d in front of the number

ex: express the hexa number B2A → 0xb2a, 0xb2A, 0hB2A, 0b2Ah, 0B2aH

**2. Register operands**

- Direct using: mov eax, ebx
- Indirect using: mov eax, [ebx] – used for pointing to memory locations

**3. Memory addressing operands**

- a. Direct addressing operands – a symbol representing the address of an instruction or data

The **offset** is computed at **assembly time**, the address of every operand relative to the executable program's structure is computed at **linking time**, the actual physical address is computed at **loading time**

Ex: labels (jmp *et*), procedures names (call *proc1*), the value of the location counter (b db \$-a)

**The effective address** – refers to a segment register (that can be **explicitly** specified by the programmer or a register which is **implicitly** associated by the assemble)

- CS – code labels target of the control transfer instructions (jmp, call, ret, jz)

- SS – in 2am formula when using EBP or ESP as **base**
- DS – for the rest of data accesses
- ES – used only in explicit specifications (ES:[var], ES:[ebx+eax\*2-a]) or certain string instructions like movsb

b. Indirect addressing operands – use registers for pointing to memory addresses → **FORMULA DE LA 2 NOAPTEA**

**OPERATORS** – used for combining, comparing, modifying and analyzing the operands

They perform computations only with constant **SCALAR** values computable at **assembly time** (without adding or subtracting a constant from a pointer and without 2am formula)

**OBS:** there is a difference between operators and instructions because instructions may perform computations with values unknown until run time.

Priority	Operator	Type	Result
7	-	unary, prefix	Two's complement (negation): $-X = 0 - X$
7	+	unary, prefix	No effect (provides simetry to „-“): $+X = X$
7	~	unary, prefix	One's complement: <code>mov al, ~0 =&gt; 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: $-24 // 4 // 2 = -3$ ( <b>-24 / 4 / 2 ≠ -3!</b> )
6	%	Binary, infix	Remainder of unsigned division: $123 \% 100 \% 5 = 3$
6	%%	Binary, infix	Remainder of signed division: $-123 \% 100 \% 5 = -3$
5	+	Binary, infix	Sum: $1 + 2 = 3$
5	-	Binary, infix	Subtraction: $1 - 2 = -1$
4	<<	Binary, infix	Bitwise left shift: $1 << 4 = 16$
4	>>	Binary, infix	Bitwise right shift: $0xFE >> 4 = 0x0F$
3	&	Binary, infix	AND: $0xF00F \& 0xFF6 = 0x0006$
2	^	Binary, infix	Exclusive OR: $0xFF0F \wedge 0xFF = 0xFF0$
1		Binary, infix	OR: $1   2 = 3$

The segment specification operator (:) – performs the FAR address computation of a variable or label relative to a certain segment

Syntax: **segment: expression**

Ex:

[ss: ebx + 4] ;offset relative to SS

[es: 082h] ;offset relative to ES

10h: var ;segment address specified by 10h selector, the offset is the value of the var label

Type operators – specify the types of some expressions or operands stored in memory

These are **non-destructive temporary conversion operators**

For **memory-stored operators**, type can be: BYTE, WORD, DWORD, QWORD, TWORD (10)

For **code labels**, type can be: NEAR (4 bytes address), FAR (6 bytes address)

Where we need a data size specifier:

- mov [mem], 12
- (i)div [mem] ; (i)mul [mem]
- push [mem] ; pop [me

**OBS!!!!** Push 15 it works because a constant will always be pun on 32 bits!!!

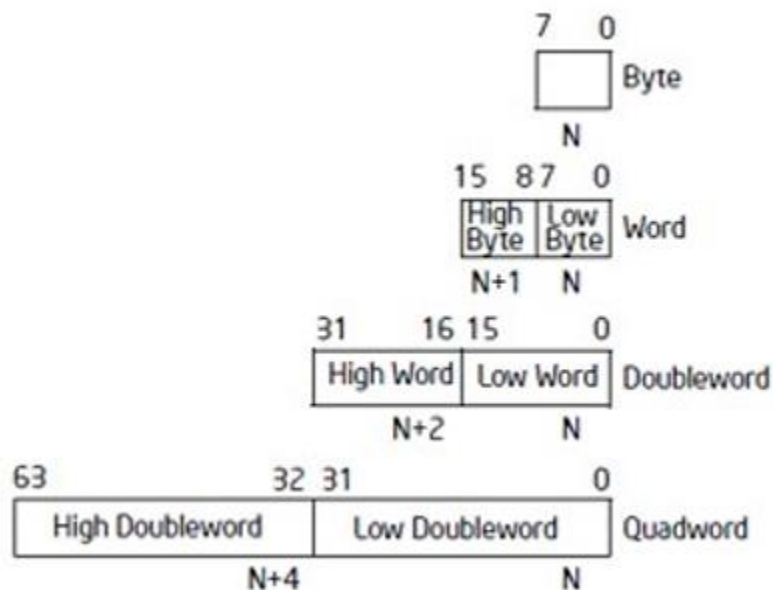
**BIT** = the **smallest unit of REPRESENTING the information** – represented by a binary digit (0/1)

**BYTE** = the **smallest ACCESSIBLE unit** – data represented on 8 bits

**WORD** = contains **2 bytes** (16 bits)

**DOUBLEWORD** = contains **4 bytes** (32 bits)

**QUADWORD** = contains **8 bytes** (64 bits)



#### CONSTANTS:

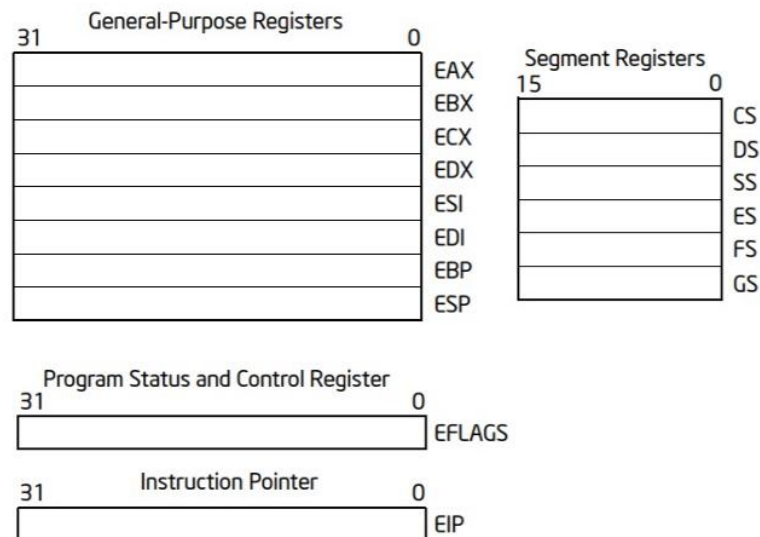
- Numbers
- Characters
- String

#### VARIABLES:

- Pre-defined variables

- o **CPU REGISTERS**

- **General registers**



- **EAX** – accumulator, used as an operand for the most of the instructions
- **EBX** – base register
- **ECX** – counter register
- **EDX** – data register
- **ESP** – stack register → points to the **last element** put on the stack
- **EBP** – stack register → points to the **first element** put on the stack
- **EDI** – index register (**destination**), used for accessing elements from bytes and words strings
- **ESI** – index register (**source**), used for accessing elements from bytes and words strings

- **Segment registers**

During run time there is only at most active segment of any type

- CS, DS, SS, ES – contain the values of the selectors of the active segments. Determine the **starting addresses** and the **dimensions of the 4 active segments**
- FS, GS – store selectors pointing to other auxiliary segments without having predetermined meaning

- **Other registers**

- **EIP** – contains the **offset of the current instruction inside the current code segment** (managed exclusively by BIU)
- **EFLAGS**

- User-defined variable  
→ has a name, a data type, a value, and a memory location

## INSTRUCTIONS:

### MACHINE INSTRUCTIONS REPRESENTATION:

An instruction = a sequence of 1 to 15 bytes

### INTERNAL FORMAT OF AN INSTRUCTION:

*[prefixes] + code + [ModeR/M] + [SIB] + [displacement] + [immediiate]*

0-4 bytes    1-2 bytes    0-1 byte    0-1 byte    0-4 bytes    0-4 bytes

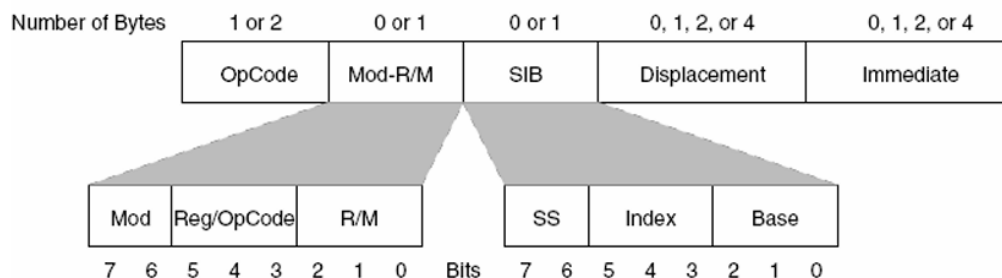
Prefixes → how an instruction is executed (may request repetitive execution or may block the address bus)

Code → operation to be run

ModeR/M → specifies the nature and the exact storage of operands (register or memory)

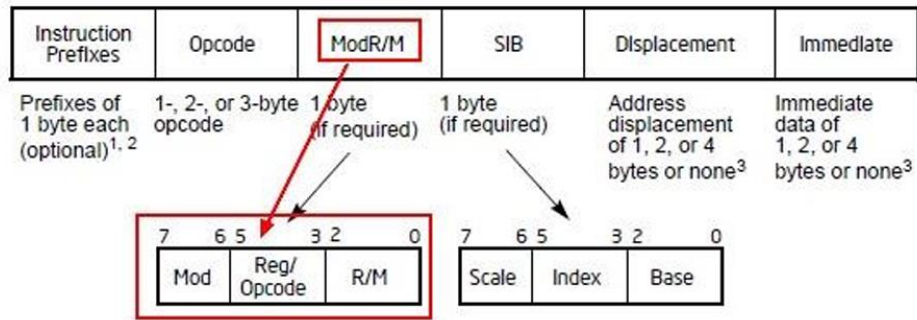
Number of Bytes	0 or 1	0 or 1	0 or 1	0 or 1
	Instruction prefix	Address-size prefix	Operand-size prefix	Segment override

(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.



### Type of prefixes:

- Instruction prefix (0 or 1)

**F3h** = REP, REPE/REPZ

**F2h** = REPNE/REPNZ

**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.

- Segment override prefix → explicit specification of a segment register

2Eh = CS

36h = SS

3Eh = DS

26h = ES

64h = FS

65h = GS

- **Operand size** prefix **66h** - Changes size of **data** expected by default mode of the—

ex: push word a → it will generate 66h in front

**bits 32**

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 bit – rez in EAX

EXPLICITLY provided  
by the programmer

Not explicitly  
provided by the

**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 reg on 16 bits (DX:AX)

cwde ; 66:98 - because here the 16 bits default mode is not followed – result in EAX

- **Address size prefix 67h** - **Changes size of address expected by the default mode** of the instruction

ex:

**bits 32**

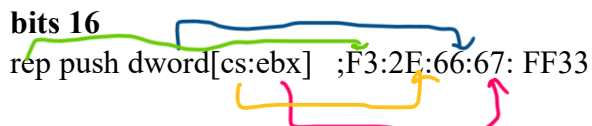
mov eax, [bx] ;67:8B07

**Bits 16**

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

**bits 16**

rep push dword[cs:ebx] ;F3:2E:66:67: FF33



**FORMULA DE LA 3 NOAPTEA – OFFSET PE 16 BITI**

$$\text{Offset} = [\text{BX/BP}] + [\text{SI/DI}] + [\text{const}]$$

- an instruction has at most two operands

- o **MOV** dest, source

- Dest, source – registers/variables/constants – they have the **SAME SIZE**

**CAN'T BE A CONSTANT**

If the dest is a segment register => source **must be** one of the 16 bits general registers or a memory variable!

- o **ADD** dest, source

- at most one operand can be a **memory location** (ex: add [a], 100)

- o **SUB** dest, source

Interpret operands as <b>unsigned</b> numbers	Interpret operands as <b>signed</b> numbers
DIV	IDIV



MUL	IMUL CBW CWD CWDE
-----	----------------------------

Regarding the arithmetic instructions with 2 operands:

- All operands must have **the same size/type**
- At least one operand must be a general register or a constant (but not to appear as a destination operand)

mov EAX, [a] – moves in EAX a doubleword **starting** at the address of the variable ‘a’

mov EAX, a – moves in EAX the **offset** of the variable ‘a’

- o **MUL** source  
Source = register or variable

*compatibility reasons with '86 C A*

Source is a <b>byte</b>	AX ← AL * source
Source is a <b>word</b>	DX:AX = AX * source
Source is a <b>dword</b>	EDX:EAX = EAX * source

*most significant* → *least significant part*

- o **DIV** source

Source is a <b>byte</b>	AL = AX / source AL – quotient AX – remainder
Source is a <b>word</b>	AX = DX:AX / source AX – quotient DX – remainder
Source is a <b>dword</b>	EAX = EDX:EAX / source EAX – quotient EDX – remainder

- o **ADC** dest, source  
Dest = dest + source + CF
- o **SBB** dest, source  
Dest = dest – source – CF

- o **CMP** a, b – performs a non-destructive *sub a, b* and **sets the flags** accordingly
- o **TEST** a, b – performs **fictive** a AND b and **sets the flags**
- o **XCHG** dest, source – dest  $\leftrightarrow$  source (interchanges the operands)  
dest, source – can be 2 general purpose registers or a register and a memory location (**have to be L-VALUES**)
- o [reg\_seg] **XLAT** – replace the byte from AL with the byte from the translation table whose offset is in EBX, having the index the initial value from AL  
**AL  $\leftarrow$  <DS: [EBX+AL] or AL  $\leftarrow$  <segment: [EBX+ AL]>**

Ex:

Segment data use32

....

tabhexa db '0123456789ABCDEF' (always the index of the table starts from 0)  
numar resb 1

....

Segment code use32

mov EBX, tabhexa

....

mov AL, numar

xlat ;AL  $\leftarrow$  <DS: [EBX+AL]>

ES xlat ;AL  $\leftarrow$  <ES: [EBX+AL]>

Explanation: if numar = 11, then AL = B

- o **PUSH s** – pushes <s> in the stack and ESP = ESP – 4  
Similar instruction:

```
sub ESP, 4 ;prepare the space in order to store the value
mov [ESP], EAX ;store the value
```

- o **POP d** – pops the current element from the top of the stack and puts it in d  
ESP = ESP + 4  
Similar instruction:

```
mov EAX, [ESP] ;load in EAX the value from the top of the stack
add ESP, 4 ;clear the location
```

- o **PUSHA/PUSHAD** – pushed the value of the registers in the stack
- o **POPA/POPAD** – pops the value of the registers from the stack
- o **PUSHF** – pushes EFlags in the stack

- o **POPF** – pops the top of the stack and transfers it to EFlags
- o **LEA** `general_reg, memory_operand` – `general_reg ← offset(mem_operand)`  
`lea EAX, [v] ⇔ mov EAX, v` ;in EAX is loaded the offset of v

**OBS!!** The source operand can be an addressing expression → `lea eax, [ebx+v-6]`

- o **CBW/D/DE** – it is the sign extension, converting AL/AX/AX to → AX, DX:AX, EAX
- o **MOVZX** `d, s` – d must be a **register** with a **size larger than s** (reg/mem), **extending s** with **zero**

Ex:

`mov AL, 10101100b`  
`movzx AX, AL` ; AX = 00000000\_10101100b

- o **MOVSX** `d, s` – **extends s** with the **sign bit**
- Ex:
- `mov AL, 10001110b`  
`movsx AX, AL` ; AX = 11111111\_10001110b

**OBS!!!**

`movsx AX, [v]` ;it works, it will consider [v] of dimension byte

`movsx EAX, [v]` ;it is **syntax error** because the operand size was not specified

`movsx ax, v` ; **syntax error** because v = offset(v) is a constant

- o **CMP** `d, s` – fiction subtraction `d – s` → it changes the flags
- o **TEST** `d, s` – non-destructive d and s → it modifies SF, ZF, PF
- o **CALL operand** – before performing the jump, call saves to the stack the address of the instruction following call (the returning address)  
 Operand can be: a procedure name, a register containing an address, a memory address

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

- o **RET** `[n]` – it frees from the stack n bytes and pops the returning address stored there by call

		B resd 1
		.....
RET n	⇔	pop dword [B] add esp, n

jmp [B]

<b>LOOP</b>	Until ECX != 0
<b>LOOPE</b> <b>LOOPZ</b>	Until ECX != 0 and ZF = 1 Until ECX = 0 or ZF = 0
<b>LOOPNE</b> <b>LOOPNZ</b>	Until ECX != 0 and ZF = 0 Until ECX = 0 or ZF = 1

## STRING INSTRUCTIONS:

In order to work with string instructions, we must:

- Set the offset of the source string in ESI
- Set the offset of the destination string in EDI
- Set the parsing direction
  - DF = 0 => from left to right
  - DF = 1 => from right to left

<b>LODSB/W/D</b>	AL/AX/EAX ← <DS: ESI> DF = 0 → ESI += 1/2/4 DF = 1 → ESI -= 1/2/4
<b>STOSB/W/D</b>	<ES: EDI> ← AL/AX/EAX DF = 0 → EDI += 1/2/4 DF = 1 → EDI -= 1/2/4
<b>MOVSB/W/D</b>	<ES: EDI> ← <DS: ESI> DF = 0 → ESI += 1/2/4, EDI += 1/2/4 DF = 1 → ESI -= 1/2/4, EDI -= 1/2/4
<b>SCASB/W/D</b> ~changes the flags~	CMP AL/AX/EAX, <ES: EDI> DF = 0 → EDI += 1/2/4 DF = 1 → EDI -= 1/2/4
<b>CMPSB/W/D</b> ~changes the flags~	CMP <DS: ESI>, <ES: EDI> DF = 0 → ESI += 1/2/4, EDI += 1/2/4 DF = 1 → ESI -= 1/2/4, EDI -= 1/2/4

## FUNCTIONS :

*Printf*("%d + %d = %d", a, b, c)

Rules of the calling convention:

- Parameters are passed on the stack **from right to left** (each element on the stack is a dword)
- The default result is returned by the function in EAX
- The registers can be modified by the called function → **YOU NEED TO SAVE THEIR VALUES BEFORE CALLING THE FUNCTION**

- You must **free the parameters** from the stack

## WORKING WITH FILES:

1. Open a file		
FILE * fopen(const char* filename, const char * access_mode)		
ARGUMENTS		
Mode	Meaning	Description
r	read	- Open file for reading. - The file must exist.
w	write	- If the file does not exist, it creates a new file and opens it for writing. - If a file with the given name exists, it opens it for writing. It overwrites the content of the file.
a	append	- If the file does not exist, it creates a new file and opens it for writing. - If a file with the given name exists, it opens it for writing. It does not overwrite the content, it continues writing at the end of the file.
r+	Read+write for existing file	- Open file for reading and writing. - The file must exist.
w+	Read+write	- If the file does not exist, it creates a new file and opens it for reading and writing. - If a file with the given name exists, it opens it for reading and writing. It overwrites the content of the file
a+	Read+append	- If the file does not exist, it creates a new file and opens it for reading and writing. - If a file with the given name exists, it opens it for reading and writing. It does not overwrite the content, it continues writing at the end of the file.
RESULTS		
If the file is successfully opened, <b>EAX will contain the file descriptor</b> (an identifier) which can be used for working with the file (reading and writing). <b>If an error occurs, fopen will set EAX to 0</b>		

Example:

push dword modread ; for strings, the offset is pushed on the stack

push dword inputfile ; for strings, the offset is pushed on the stack

call [fopen]

add esp, 4\*2

## 2. Write into a file

<b>int fprintf(FILE * stream, const char * format, &lt;variable_1&gt;, &lt;variable _2&gt;, &lt;...&gt;)</b>
<b>RESULT</b>
In case of an <b>error</b> , <b>EAX</b> contains a value < 0

<b>3. Read from a file</b> <b>int fread(void * str, int size, int count, FILE * stream)</b>
<ul style="list-style-type: none"><li>• First argument is the string where the bytes that are read from the file are stored</li><li>• Second argument represents the size of the elements that are read from the file</li><li>• Third argument represents the maximum number of elements to be read</li><li>• Last argument is the file descriptor</li></ul>
<b>RESULT</b>
EAX will contain the number of elements read. If this number is below <b>count</b> , it means wither that there was an error, or that the function got to the end of the file.

<b>4. Closing an open file</b> <b>int fclose(FILE * descriptor)</b>
--

## MULTI-MODULE PROGRAMMING

- global – export a variable or procedure defined in the current module
- extern – import an external variable or procedure defined in another module

Passing the parameters to a function/procedure defined in another module:

- Parameters can be passed using the **registers**
- Parameters can be passed to the function in the other module by **declaring them global**; the problem with this is that it breaks an old and important principle of programming: modularization (i.e. a program is better maintained if it is formed by independent modules linked together, e.g. functions, source files etc.) and everything becomes global (part to the same namespace which can cause name clashes – the same symbol is defined in different places)
- Parameters can be passed using the **stack** – this is the most powerful and flexible solution

### ■ CODE FILES – code reusability and dates from assembly

%include → **THIS IS NOT multi-module programming** because at the compilation it will be only one module obtained through the concatenation of the included files

### ■ BINARY FILES – they are kept as obj

code reuse and dates from assembly + codes and dates from high level languages + libraries

The existence of separate binary files implies **separate compilation!!** The link-editor

## STATIC LINKING AT LINK-TIME

- Preprocessor: text → text
  - o processes the source text, resulting in an intermediary text source
  - o can be imagined as a component of the compiler or assembler
  - o may be missing
- Assembler: instructions (text) → binary encoding (object file)
  - o encodes the instructions and data from the preprocessed text source and builds an object file that consists of machine code and variable values, along with information about the content
- Compiler: instructions (text) → binary encoding (object file)
  - o Identifies instructions through which the functionalities described in the text source can be obtained and then generates an object file containing the binary codification of those instructions and variables
- Linking: object file → library or program
  - o Constructs the final result, a program .exe or a library, linking together the code and binary data from the object file

It is allowed to join multiple binary modules (object files or static libraries) in a single file.

**GLOBAL** and **EXTERN** directives are used to export and import data from different modules and these are made for **multi-module programming**.

In C all the variables and methods defined in the most exterior area of the program can be accessed from other modules. In order to restrict access to some data, it can be used the keyword **static**.

<b>; FILE1.ASM</b>		<b>; FILE2.ASM</b>
<b>global</b> Var1, Subroutine2	→	<b>extern</b> Var1, Subroutine2
<b>extern</b> Var3, Subroutine3	←	<b>global</b> Subroutine3, Var3
Subroutine1:		Subroutine3:
....		....
call(Subroutine3)		call(Subroutine2)
....		....
operations(Var3)		operations(Var1)
....		....
<b>Subroutine2:</b>		<b>Subroutine1:</b>
....		....
<b>Var1</b> dd ...		Var2 db ...
Var2 db ...		<b>Var3</b> dd ...

## HOW TO PASS THE PARAMETERS?

- 1) By value

- 2) By reference
- 3) Call by value or reference decision

## CALLING CONVENTIONS

- **CDECL** convention – it is specific to the C programming language  
The parameters are passed to subroutines by **pushing them on the stack**. We can pass any type of parameters but extended at least to DWORD. These will be transmitted in **reverse order** of declaration, from right to left. There is **no limit** when it comes to the **number of parameters** that can be passed. In the end, the arguments must be freed up and the **CALLER** is responsible for **cleanup actions**.
- **STDCALL** convention – it is specific to Windows OS  
It is similar to the CDECL convention, the only differences being that there is a **fixed number of parameters** that can be passed and the **cleanup** is performed by **CALLEE**.

For both conventions, the volatile resources are: EAX, ECX, EDX, and Eflags, and the result is stored in EAX, EDX:EAX or ST0.

	Parameters			Volatile resources	Results	Cleanup
	Storage	Order	Number			
<b>CDECL</b>	Stack	Reverse	<b>Any</b>	EAX, ECX, EDX, EFLAGS	EAX/EDX:EAX/ST0	<b>CALLER</b>
<b>STDCALL</b>	Stack	Reverse	<b>Fixed</b>	EAX, ECX, EDX, Flags	EAX/EDX:EAX/ST0	<b>CALEE</b>

**Volatile resources** – are represented by the registers that belong to the called subroutine. The **caller** is responsible **as part of the call code** to save their values and to restore them at the end of the call.

**Non-volatile resources** – are any memory addresses or registers which do not belong explicitly to the called subroutine, but if they need to be modified, it is necessary to be saved at the entry as part of the entry code and restored back at exit, as part of the exit code. The **callee** is saving and restoring the values of non-volatile resources.

10

## SUBROUTINE CALL

1. **CALL CODE** – represents the call preparation and execution

It must:

- Save the volatile resources in use: push register
- Assure the compliance with constraints (to align esp, df = 0)
- Prepare the arguments (stack, by convention): push



- Call execution:
  - ➔ call *subroutine* – if subroutine is *statically linked*
  - ➔ call *[subroutine]* – if subroutine is *dynamically linked* (at link-time)
  - ➔ call *register* or call *[variable]* – for run-time dynamic linking

ex:

push ecx     ;save the non-volatile resources in use

push eax     ;pass the parameters

call proc     ;execute the call

- Example: call printf from asm to display digits from 0 to 9

```
import exit msvcrt.dll
import printf msvcrt.dll
extern exit, printf
global start
```

If the call was made from C, the compiler would have generated by itself the call code!  
But the call is made from assembly, so the call code must be written by us!

```
segment code use32
```

```
start:
```

```
mov ecx, 10
```

```
xor eax, eax
```

```
.next:
```

```
push eax
```

```
push ecx
```

```
push eax
```

```
push dword format_string
```

```
call [printf]
```

```
add esp, 2*4
```

```
pop ecx
```

```
pop eax
```

```
inc eax
```

```
loop .next
```

```
push dword 0
```

```
call [exit]
```

```
segment data use32
```

```
format_string db "%d", 10, 13, 0
```

1. Save the volatile resources

2. DF=0, stack is aligned (only DWORDs were pushed)

3. Prepare the arguments for CDECL call

4. Call execution

Recover the volatile resources (the caller)

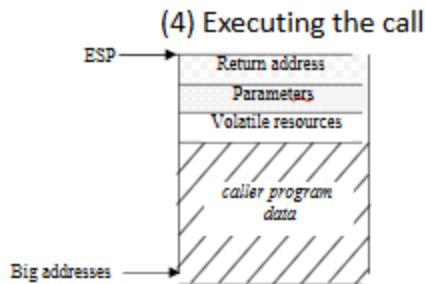
- Effect of the call code on the stack

```

push eax
push ecx
push eax
push dword format_string
call [printf]
add esp, 2*4

```

1. Saving volatile resources
2. DF=0, the stack has only been used on DWORD
3. Preparing arguments for the call CDECL
4. Executing the call



## 2. ENTRY CODE – represents procedure entry and preparation of execution

It must:

- Configure a **stack frame**: ebp or esp needed to be handled
- Prepare local variables of the function: `sub esp, nr_bytes`
- Save a copy of the non-volatile resources that are modified: `push register`

**Stack frame** – is a data structure stored in the stack, of fixed dimension, containing:

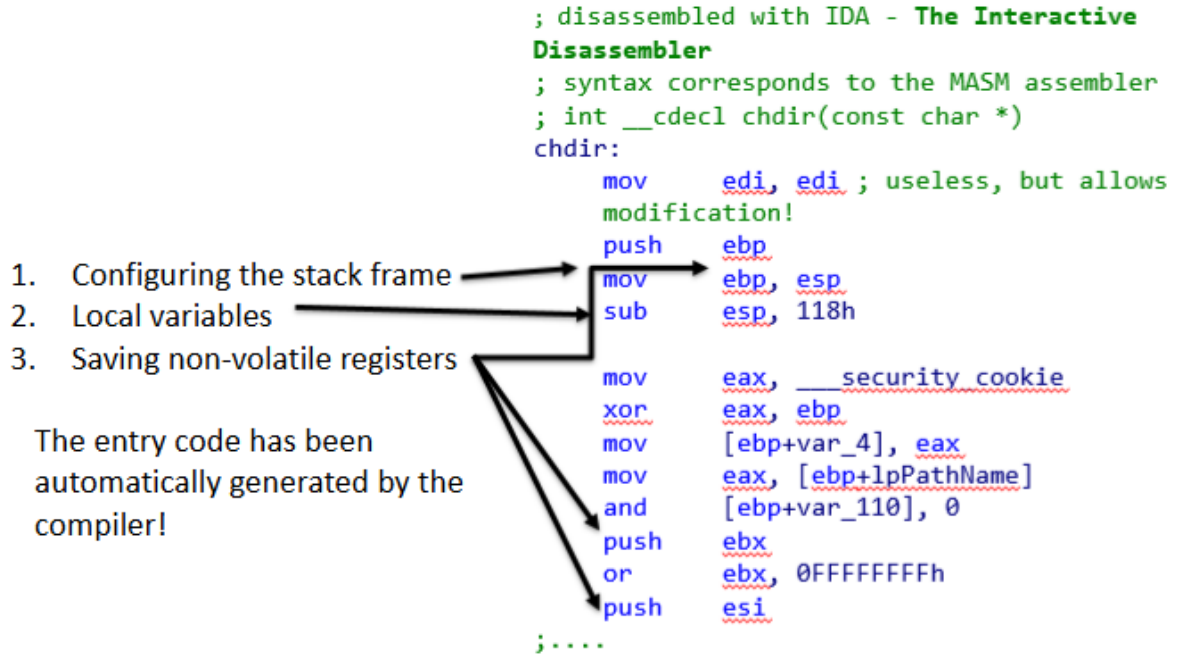
- the parameters prepared by the caller
- return address (to the instruction that follows the call instruction)
- copies of the non-volatile resources used by the subroutine
- local variables

`push EBP` ;for restoring the base of the current stack frame when returning

`mov EBP, ESP` ;the birth of the new stack frame

`sub esp, 8` ;reverse a necessary space to allocate local variables

- Example: entry code to the function CDECL `chdir`, generated by the C compiler



### 3. EXIT CODE – returns and frees up out-of-date resources

It must:

- Restore altered non-volatile resource: `pop register`
- Release the local variables of the function: `add esp, nr_bytes_locals`
- Deallocating the stack frame: `pop ebp`
- Return from the function and release arguments

➔ CDECL:

Called subroutine: `ret`

Calling procedure: `add esp, size_of_arguments`

➔ STDCALL:

`Ret size_of_arguments`

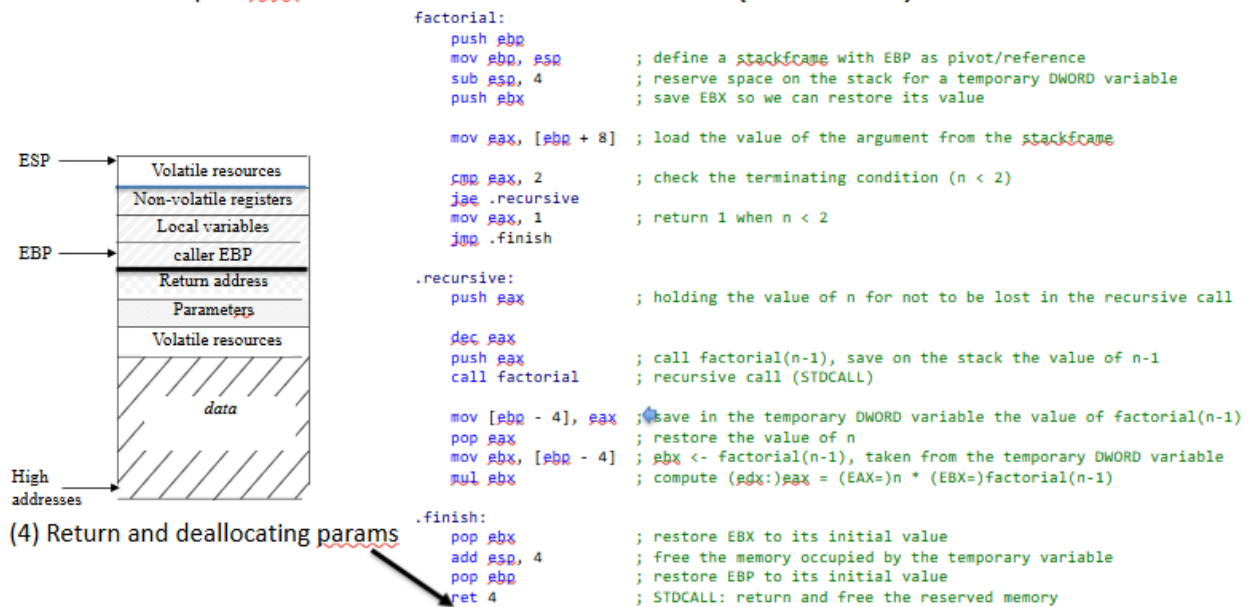
Ex:

`pop ebp` ;restore the non-volatile resources

`add esp, 8` ;free the spaced reserved on the stack for the local variables

`mov esp, ebp` ;restore the initial stack frame

- Example: asm exit code from a STDCALL function (recursive call) - stackframe



The steps are implemented automatically in the code generated by compilers of high-level languages, but in assembly, these are the responsibility of the programmers!

### Responsibilities 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)

### IMPORTANT RULES!!!

In memory → it is applied little endian (NOT ON THE CPU REGISTERS!)

That means:

a dw 1234h ; | 34 | 12 |

BUT

When we work with registers:

```
mov ax, 1234h ;will remain the same!!! ax = 1234
```

Or:

```
b dw 1234h ; | 34 | 12 |
```

```
mov al, [b] ; al = 34
```

```
mov ax, [b] ; ax = 12 34
```

Complex example:

```
a db 1122h ; => 22
```

```
b dw 1234h ; => 34 12
```

```
c db 0Ah ; => 0A
```

```
d dd 1122h ; => 22_11_00_00
```

Data seg

22	34	12	0A	22	11	00	00
----	----	----	----	----	----	----	----

*a* *b* *c* *d*  
*le*  
This is how the variables looks like in the memory

```
mov al, [a] ; al = 22
```

```
mov al, [b] ; al = 34
```

```
mov ax, [b] ; ax = 12_34
```

STIE SA TINA CONT DE LITTLE ENDIAN

```
mov ax, [a] ; ax = 34_22  
tine cont de little endian
```

ia wordul care se regaseste de la offsetul lui a si

```
mov eax, [b] ; eax = 22_0A_12_34
```

Concluzia: registrele iti vor lua valoarea din memorie incepand cu offsetul pe care il dai si tinand cont de little endian o sa o memoreze!!

Cand dam numere in hexa → de exemplu 12d = Ch, TREBUIE PUS UN 0 IN FATA CA SA IL CONSIDERE NUMAR SI NU STRING => 12d = 0Ch

**A REPRESENTATION IN BASE 2 – HAS ALWAYS 2 INTERPRETATIONS IN BASE 10**  
(signed interpretation and unsigned interpretation)

**You can't have an interpretation of a number which is in base 10!!!!!!! BASE 10 IS ALREADY AN INTERPRETATION!!**

If we have  $N$  bits  $\Rightarrow$  the values that have the same representation in signed and unsigned interpretation are :  $[-2^{N-1}, 2^{N-1}-1]$

**WE HAVE 6 SEGMENT REGISTERS:**

- CS - code segment
- DS - data segment
- SS - stack segment
- ES - extra segment
- FS
- GS

The segment registers contain the **SEGMENT SELECTOR** corresponding to the currently active \_\_\_ segment **NOT THE STARTING ADDRESSES**

**WE HAVE 7 ADDRESS REGISTERS:** the one mentioned above + **EIP** register – which is an extension of the instruction pointer

A segment is defined by its:

- Basic address
- Sizeof
- Type

**THE MEMORY SEGMENT = is a logical section of a program's memory, featured by its BASIC ADDRESS, SIZE AND TYPE**

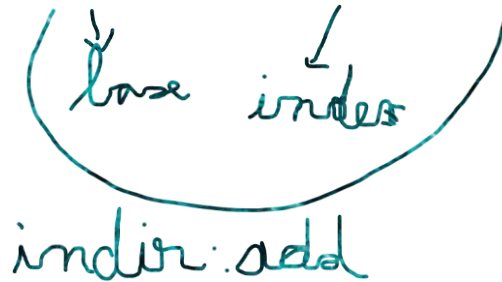
**Address specification** = a pair of **segment selector** (numeric values on 16 bits which selects uniquely the accessed segment) and an **offset**

s3s2s1s0: o7o6o5o4o3o2o1o0

**ADDRESSING MODES**

1 2 4 8  
↑

$Offset\_addressed = [base] + [index * scale] + [constant]$



direct add

[ ] – means the optionality of the element

**Base registers:** EAX, EBX, ECX, EDX, EBP, ESI, EDI, **ESP (can't be an INDEX register!!)**

**Index registers:** EAX, EBX, ECX, EDX, EBP, ESI, EDI

**!!!OFFSETUL SE CALCULEAZA LA MOMENTUL ASAMBLARII !!!**

**Valoarea unui registru se calculeaza la RUN TIME !! La fel si valoarea unei variabile !!**

**Address relocation → apare la LOADING TIME!!**

**DIRECT OFFSET ADDRESSING** = modifies an address using arithmetic operators

Ex:

```
byte_table db 14, 15, 22, 45
```

```
mov cl, byte_table[2]
```

```
mov cl, byte_table + 2
```

**INDIRECT MEMORY ADDRESSING** = leverages the computer's ability to address memory using data segment

Ex:

```
my_table times 10 dw 0 ; allocates 10 words initialized to 0
```

```
mov ebx, [my_table] ; effective address of my_table in ebx
```

```
mov [ebx], 110 ; my_table(0) = 110
```

## ADDRESS COMPUTATION

**a segment selector** – is defined and provided by THE OPERATING SYSTEM!

$$a7a6a5a4a3a2a1a0 := b7b6b5b4b3b2b1b0 + o7o6o5o4o3o2o1o0$$

$a7a6a5a4a3a2a1a0$  = the computed address in hexadecimal form

➔ Call back the **address specification**

**s3s2s1s0** → shows a segment access which has the **base address** **b7b6b5b4b3b2b1b0** and a limit **1716151413121110** (both obtained by THE PROCESSOR after performing a SEGMENTATION PROCESS)

$$o7o6o5o4o3o2o1o0 \leq 1716151413121110$$

How is made the **address computation**? => 8:1000h

- Checks if the **segment** with value **8** was defined by the **operating system**
- Extracts the **base address** (B) and the **segment limit** ( B – 2000h and L = 4000h )
- Verifies if the **offset exceeds the segment's limit**: 1000h > 4000h
- Add the offset to B and obtain the *linear address* ( 1000h + 2000h = 3000h ) –

**COMPUTATION PERFORMED BY THE ADR component from BIU**

Notion	Representation	Description
Address specification, logical address, FAR address	Selector <sub>16</sub> :offset <sub>32</sub>	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 <sub>32</sub>	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).



**Effective Address = BaseReg + IndexReg \* ScaleFactor + Disp**

The base register (BaseReg) can be any general-purpose register; the index register (IndexReg) can be any general-purpose register except **ESP**; Displacement values (Disp) are constant offsets that are encoded within the instruction; valid scale factors (ScaleFactor) include 1,2,4, and 8. The size of the final effective address (EffectiveAddress) is always 32 bits.

Addressing Form	Example
Disp	<code>mov eax, [MyVal]</code>
BaseReg	<code>mov eax, [ebx]</code>
BaseReg + Disp	<code>mov eax, [ebx+12]</code>
Disp + IndexReg * ScaleFactor	<code>mov eax, [MyArray+esi*4]</code>
BaseReg + IndexReg	<code>mov eax, [ebx+esi]</code>
BaseReg + IndexReg + Disp	<code>mov eax, [ebx+esi+12]</code>
BaseReg + IndexReg * ScaleFactor	<code>mov eax, [ebx+esi*4]</code>
BaseReg + IndexReg * ScaleFactor + Disp	<code>mov eax, [ebx+esi*4+20]</code>

## POINTER ARITHMETIC

The ONLY operations that can be performed with pointers:

- **Addition of a constant to a pointer**  $\Rightarrow$  **A POINTER**  
Ex:  $p + 5 \rightarrow$  pointer
- **Subtraction of a constant to a pointer**  $\Rightarrow$  **A POINTER**  
Ex:  $9 - 5 \rightarrow$  pointer
- **Subtracting 2 pointers**  $\Rightarrow$  **SCALAR**  
Ex:  $q - p \rightarrow$  scalar



Used to find out the size of the memory between two pointers

**Asamblorul = TRANSLATOR**, ia ce scrii tu si il translateaza in cod obiect

**Sarcina PRINCIPALA a ansamblorului este de A GENERA BYTES (GENERATING BYTES)** corespunzatori instructiunilor si directivelor

**Assembler  $\rightarrow$  BUILDS THE DATA SEGMENT!!**

The elements an assembler works with:

- **Labels** ( start: ..... ; count\_digits: ..... dar si numele de variabile sunt labeluri)

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

- **Instructions** - mnemonics which suggest the underlying action

Ex: jmp, add, pop etc

- **Directives** - indicate the way the code and the data are generated at **assembly time**

Ex: db, dw, dd, dq, end, ends, endp, equ

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

a db ...

b dw...

c dd...

a **IS NOT A BYTE** – **is only an OFFSET**, symbol representing the start of a memory area **WITHOUT HAVING AN ASSOCIATED DATA TYPE**

- **Location counter \$** – every segment has its own location counter  
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

**\$** - address of “here”

**\$\$** - address of the start of the current section

**\$ - \$\$ = DISTANCE FROM THE BEGINNING OF THE SEGMENT => a scalar**

**Expressions** are evaluated at **assembly time** (their values are computable at **assembly time**, except for the operands representing **registers** contents, which can be evaluated only at **run time** – the offset specification formula)

Orice offset va fi **prefixat** cu unul dintre segmentele **CS, DS, SS !!**

**CS = is the destination of jmp, call, ret**

**SS = EBP or ESP was used as BASE REGISTER**

**DS = we have the rest of the register as bases**

mov eax, [ebx] ;DS

mov eax, [ebp] ;SS

mov eax, [ebp\*2] ;mov eax, dword ptr[SS: ebp + ebp]

mov eax, [ebp\*3] ;SS

mov eax, [ebp\*4] ;DS mov eax, dword ptr[DS: ebp \* 4] => index not a base

mov eax, [ebx + esp] mov eax, [esp + ebx] ;SS – ESP CAN'T BE AN INDEX, ESP IS ONLY A BASE REGISTER

mov eax, [ebx + ebp \* 2] ;DS

mov eax, [ebx\*1+ebp] ;SS

mov eax, [ebp\*1+ebx] ;DS

mov eax, [ebx\*1+ebp\*1] ;SS - the first found scaled element is taken as index !! EBP - base

mov eax, [ebp\*1+ebx\*1] ;DS - the first found scaled element is taken as index !! EBX - base

mov eax, [var] – in OllyDBG you will find mov eax, DWORD PTR [DS:004027AB]

**DIRECT ADDRESSING OPERANDS:** constants or symbols representing the address (segment and offset) of an instruction or some data

Ex: jmp et, call proc1, b db \$-a, mov eax, 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 is computed at LINKING TIME**

**The ACTUAL PHYSICAL ADDRESS is computed at LOADING TIME**

**INDIRECT ADDRESSING OPERANDS**

**[base\_register + index\_register \* scale + constant]**

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

## DIFFERENCE BETWEEN OPERATORS AND INSTRUCTIONS!!!!!!

**OPERATORS** perform computations only with CONSTANT SCALAR VALUES computable **AT ASSEMBLY TIME!!!!**

Scalar values = constant immediate values

**INSTRUCTIONS** perform computations with values that MAY remain UNKNOWN (mostly all of the time!!!) until **RUN TIME!!!!**

THAT'S WHY,

`1 << 12` IS DIFFERENT OF `shl eax, 12` (where `eax = 1`) !!!!



**IT DOESN'T SET CF**



**IT WILL SET THE CF!!!**

**WE ALWAYS NEED A DATA SIZE SPECIFIER for the given operands:**

- `mov <mem>, 12`
- `div <mem>`
- `mul <mem>`
- `push <mem>`
- `pop <mem>`

data size specifiers: BYTE / WORD / DWORD / QWORD

**Without using these data size specifiers, it will be AN AMBIGUITY!!! => SYNTAX ERROR**

`mov word[a], 12`    `mov [a], word 0`    `mov [a], 12`

`push eax`    `push dword[ecx]`    `push [ecx]`

`push 15`    => a constant will always be put on 32 bits (translated by the assembler)

`pop 15`    => **15 IS NOT A L-VALUE!!!**

`pop v`    => **YOU CAN'T MODIFY AN ADDRESS – v is a R-VALUE NOT L-VALUE**

`pop dword b`

pop dword[v]

push word[v]

push byte[v]

pop [v]

pop word[15]

=> iti pune in memorie la adresa 15 ce ai pe stiva

mov ah, b

=> syntax error b is an **OFFSET** which can be on 16/32 byts

mov ax, b

mov eax, b

mul v

mul word v

mul 15

=> **YOU CAN'T MULTIPLY BY A CONSTANT**

*correct syntax: mul <reg>/<mem>*

mul word[v]

mul [v]

mul ax

mul byte[ecx]

mul [ecx]

pop byte[v]

=> **STACK IS ORGANIZED ON 16/32 BITS!!!**

pop qword[v]

=> **INSTRUCTION NOT SUPPORTED IN 32 BITS!!!**

### ERROR TYPES!!!

- **SYNTAX ERROR** – diagnosed by ASSEMBLE/COMPILER ~ **assembly error**
- **RUN-TIME ERROR** – program crashes – it stops executing ~ **execution error**
  - o DIVISION BY 0
  - o MEMORY VIOLATION ERROR
- **LOGICAL ERROR** - program runs until its end or remains blocked in an infinite loop
- **LINKING ERROR** – when a variable is defined multiple times in a multimodule program

**INSTRUCTIONS WHERE BOTH THE OPERANDS MUST HAVE THE SAME SIZE, BUT NOT THE BOTH OF THEM ARE FROM MEMORY!!**

ADD	<REG>,<REG>
SUB	<REG>,<MEM>
ADC	<REG>,<MEM>
SBB	<REG>,<CON>
AND	<REG>,<CON>
OR	<MEM>,<CON>
XOR	<MEM>,<CON>
TEST	<MEM>,<REG>

**INSTRUCTIONS WHERE THE OPERAND MUST BE A REGISTER OR A VARIABLE**  
**(!!!NOT A CONSTANT!!!)**

MUL DIV IMUL IDIV INC DEC NEG NOT	<REG>
	<MEM>

**INSTRUCTIONS WHERE BOTH THE OPERANDS ARE FROM MEMORY, BUT ONLY ONE IS EXPLICIT and the other ONE IS IMPLICIT**

MUL DIV IDIV IMUL	<REG> <MEM>
PUSH	<REG> <MEM> <CON>
POP	<REG> <MEM>

**INSTRUCTIONS WHERE BOTH THE OPERANDS ARE IMPLICIT FROM MEMORY**

CBW	AL → AX
CWD	AX → DX:AX
CWDE	AX → EAX
MOVS <sub>B/W/D</sub>	[EDI] ← [ESI]
CMPS <sub>B/W/D</sub>	CMP [ESI], [EDI]
LODS <sub>B/W/D</sub>	AL/AX/EAX ← [ESI]
STOS <sub>B/W/D</sub>	[EDI] ← AL/AX/EAX
SCAS <sub>B/W/D</sub>	CMP [EDI], AL/AX/EAX

} ESI ± offset  
EDI = base  
base  
DF = 0/1