**1. THE x86 MICROPROCESSOR ARCHITECTURE (IA-32)**

**1.1. 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.

**1.2. The EU general registers**

* EAX - accumulator register.
* EBX - base register.
* ECX - counter register.
* EDX - data register.
* ESP (Stack Pointer) points to the last element put on the stack.
* EBP (Base pointer) points to the first element put on the stack.
* EDI and ESI - destination Index and source Index registers - usually used for accessing elements from bytes and words strings.

**1.3. Flags**

A flag is an indicator represented on 1 bit. A configuration of the FLAGS register shows an overview of the execution of each instruction.

EFLAGS - the status register - has 32 bits but only 9 are actually used.

*Flags categories:*

1. reporting the status of the LPO (having a previous effect) – CF, PF, AF, ZF, SF, OF

* ADC; Conditional JUMPS (23 instructions – JA = JNBE; JG = JNLE; JZ; …)

1. 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

*Flags:*

1. **CF** (*Carry Flag,* transport flag) - 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 flags the UNSIGNED overflow!**
2. **PF** (*Parity Flag*) - 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.
3. **AF** (*Auxiliary Flag*) - shows the transport value from bit 3 to bit 4 of the LPO’s result.
4. **ZF** (*Zero Flag*) - set to 1 if the result of the LPO was zero and set to 0 otherwise.
5. **SF** (*Sign Flag*) - set to 1 if the result of the LPO is a strictly negative number and is set to 0 otherwise.
6. **TF** (*Trap Flag,* debugging flag) - if set to 1, then the machine stops after every instruction.
7. **IF** (*Interrupt Flag*) - if set to 1, interrupts are allowed, if set to 0 interrupts will not be handled.
8. **DF** (*Direction Flag*) - if set to 0, then string parsing will be performed in from the beginning to its end and from end to the beginning if set to 1.
9. **OF** (*Overflow Flag*) - 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. **OF flags the signed overflow!**

*Specific instructions to set the flags values:*

Assembly language provides specific instructions to set the values of the flags that will have a future effect. Seven instructions:

1. **CLC** – CF=0 |
2. **STC** – CF=1 |
3. **CMC** – complements the value of the CF |3 instructions for CF
4. **CLD** – DF=0 |
5. **STD** – DF=1 |2 instructions for DF
6. **CLI** – IF=0 |
7. **STI** – IF=1 |2 instructions for IF

IF instr. can be used only on 16 bits programming. On 32 bits, the OS restricts the access to these instr.

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

**2. Two’s complement**

**2.2. Variants of obtaining**

*Variant 1*

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).

*Variant 2*

Reversing the values of all bits of the initial binary number, after which we add 1 to the obtained value.

*Variant 3*

We left unchanged the bits starting from the right until the first bit 1 inclusive and we reverse the values of all the other bits.

*Variant 4*

The sum of the absolute values of the two complementary values is the cardinal of the set of values representable on that size.

**2.2. Admissible representation interval**

**[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**

*Example*

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

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

**2.3 Interpretation vs. Representation**

Base 2 à Base 10 ß Base 16

(representation) INTERPRETATION (representation)

of base 2 representation

↙ ↘

Unsigned Signed

(absolute value) (positive and/or negative)

**3. Overflow concept**

**3.1. 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 (Carry Flag) – unsigned overflow**

**OF (Overflow Flag) – signed overflow**

**3.2. CF vs. OF**

*Addition (b + b = b & w + w = w …)*

**CF = 0** – the **unsigned** **interpretation** in base 10 of the base 2 addition is mathematically **correct.**

**CF = 1** – the **unsigned** **interpretation** in base 10 of the base 2 addition is mathematically **incorrect.**

**OF = 0** – the **signed** **interpretation** in base 10 of the base 2 addition is mathematically **correct**.

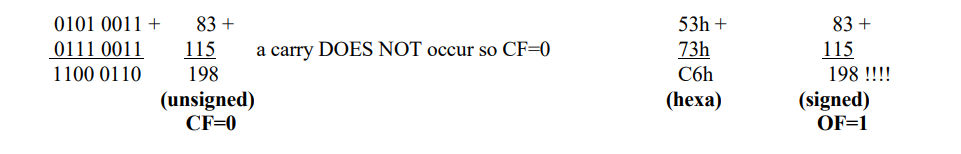
**OF = 1** – the **signed** **interpretation** in base 10 of the base 2 addition is mathematically **incorrect**.

***Overflow for addition:***

positive + positive = negative

negative + negative = positive

*Example*

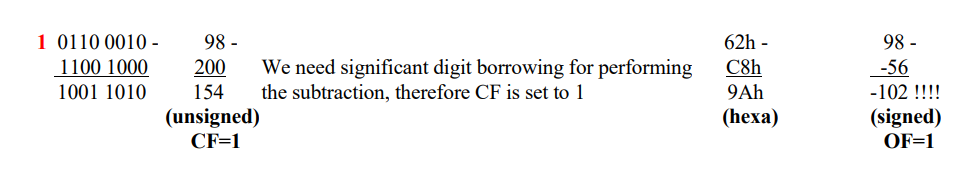
**

***Overflow for subtraction:***

positive – negative = negative

negative – positive = positive

*Example*

**

***Overflow for multiplication:***

The multiplication operation does not produce overflow, the reserved space for the result being enough for both interpretations. Anyway, in the case of multiplication:

CF = OF = 0 if b\*b = b or w\*w = w or d\*d = d

CF = OF = 1 if b\*b = w or w\*w = d or d\*d = q

***Overflow for division:***

The worst effect in case of overflow is in the case for the division operation: if the quotient does not fit in the reserved space (the space reserved by the assembler being byte for word/byte, word for doubleword/word and doubleword for quadword/doubleword) then the division overflow will signal a ‘Run-time error’ and the operating system will stop the program issuing one of the 3 semantic equivalent messages:

‘Divide overflow’ or ‘Division by zero’ or ‘Zero divide’

In the case of a correct division CF and OF are undefined.

**4. Address registers and address computation**

**4.1 Definitions**

**Address of a memory location** – nr. of consecutive bytes from the beginning of the RAM memory and the beginning of that memory location.

**Segment** – an uninterrupted sequence of memory locations, used for similar purposes during a program execution. 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**.

**Offset** – the address of a location relative to the beginning of a segment. An offset is valid only if his numerical value, on **32 bits**, doesn’t exceed the segment’s limit which he refers to.

**The offsets of the variables defined in a program are always constant values, determinable at assembly/compiling time.**

**Address specification** **= segment selector : offset**

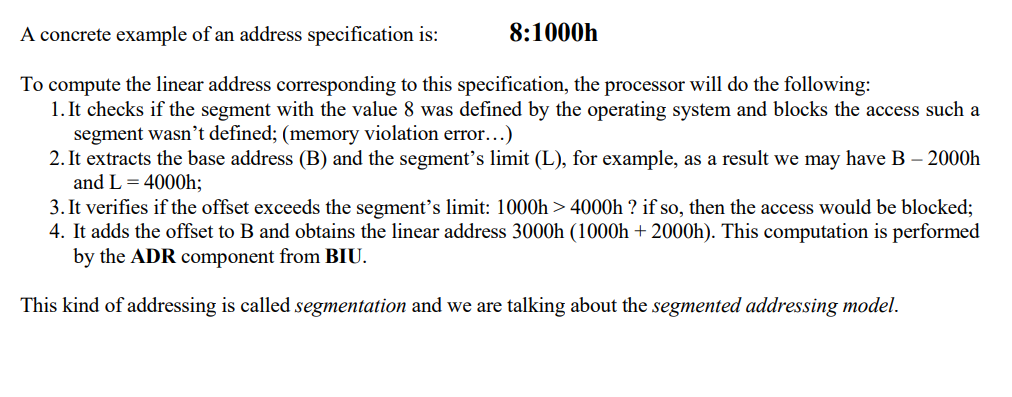
Address specification – **FAR address**

Offset – **NEAR address**

**Segment selector** – 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!

**Linear address** – segment beginning + offset, represents the result of the segmentation computing.

***Example of linear address computing:***

**

When the segments start from address 0 any offset is automatically valid and segmentation isn’t practically involved in addresses computing. This particular mode of using the segmentation, used by most of the modern operating systems is called the ***flat memory model***.

**4.2. Segments**

The x86 architecture allows 4 types of segments:

* **code segment** – contains instructions.
* **data segment** – contains 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.

**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.

**CS, DS, SS and ES** determine the starting addresses and the dimensions of the 4 active segments: code, data, stack and extra segments.

**FS and GS** can store selectors pointing to other auxiliary segments without having predetermined meaning.

**CS, DS, SS ES, FS and GS** are called **segment registers** (or **selector registers**).

Register **EIP** (you can access the less significant word **IP**) contains the offset of the current instruction inside the current code segment, this register being managed exclusively by **BIU**.

**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! (stack segment is automatically generated).

**4.3. Machine instructions representation**

**Instruction** – a sequence of 1 to 15 bytes, these values specifying:

* an operation to be run
* the operands – maximum 2 operands – **source and destination:**
  + only one may be stored in the RAM memory
  + the other one must be either one EU register, either an integer constant
* possible supplementary modifiers

*Instruction general form:*

**instruction\_name destination, source**

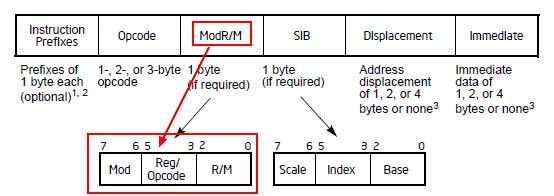
*The* ***internal format of an instruction*** *varies between 1 and 15 bytes, and has the following* ***general form****:*

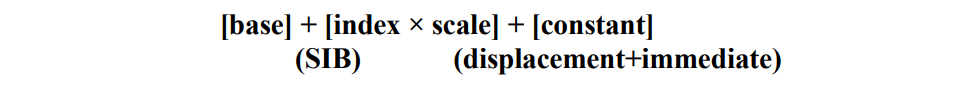


**Prefixes** – control how an instruction is executed. These are optional (0 to maxim 4) and occupy one byte each. (For example, they may request repetitive execution of the current instruction or may block the address bus during execution to not allow concurrent access to operands and results.)

**Code (opcode)** – the operation to be run, identified by 1 to 2 bytes, being the only mandatory bytes, no matter of the instruction.

**ModeR/M (register/memory mode)** – specifies for some instructions the nature and the exact storage of operands (register or memory), identified by one byte. This allows the specification of a register or of a memory location described by an offset.





**SIB:**

* **Base:** EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP
* **Index:** EAX, EBX, ECX, EDX, EBP, ESI, EDI
* **Scale:** 1, 2, 4, 8

**The ESP register is available as base but cannot be used as index!**

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

**Immediate value** – a numeric constant on 1, 2 or 4 bytes. When present, appears always at the end of instruction.

**4.4. FAR addresses and NEAR addresses**

To address a RAM memory location two values are needed: the segment and the offset.

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**. A FAR address is a COMPLETE ADDRESS SPECIFICATION** and it may be specified in one of the following 3 ways:

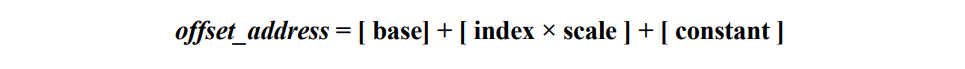
* s3s2s1s0 : offset\_specification – s3s2s1s0 is a constant
* segment register : offset\_specification – segment registers are CS, DS, SS, ES, FS or GS
* FAR [variable] – variable is of type QWORD and contains the 6 bytes representing the FAR address.

**4.5. Computing the offset of an operand**

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

* **register mode** – operand is a register (ex. mov **EAX**, 17)
* **immediate mode** – operand is a value (not its address or a register holding it) (ex. mov EAX, **17**)
* **memory addressing mode** – operand is located somewhere in memory

In **memory addressing mode**, the offset is computed using the formula:



**4.6. Addressing modes**

Modes of addressing 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 modes 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).

**An indirect addressing is when we have at least one register specified between squared brackets.**

In the case of the jump instructions, we have **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** – represented on a byte with values between [-128, 127]
* **relative NEAR addresses** – represented on a doubleword with values between [-2147483648, 2147483647]

**5. Basic elements of Assembly language**

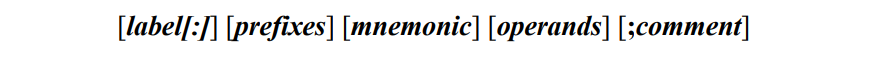
**5.1. Definitions**

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. The programmer can use:
  + **$** - evaluates to the assembly position at the beginning of the line containing the expression (**means address of “here”**)
  + **$$** - evaluates to the start of the current section (**means address “of address of start of current section”**)

**5.2. Source line format**

In the x86 assembly language the source line format is:



*Labels:*

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. **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.

*Mnemonics:*

There are 2 types of mnemonics:

* instructions names – actions that guide the processor
* directives names – guide the assembler

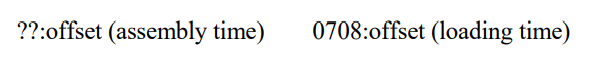
*Operands:*

The 3 operand types are:

* **immediate operands**
* **register operands**
* **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.

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**.



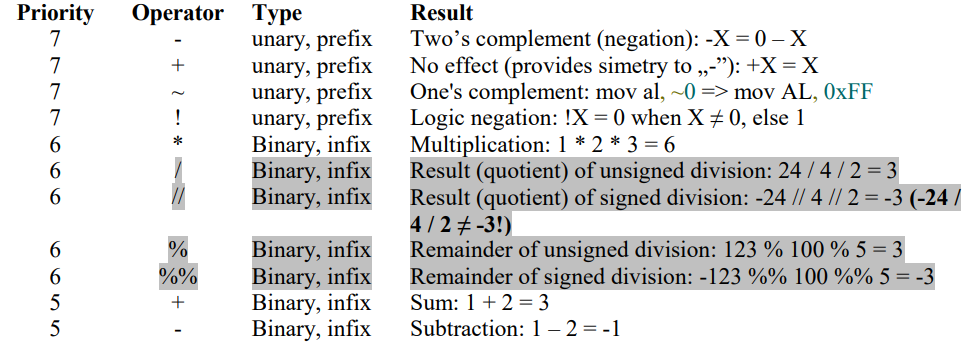
**5.2. Using operators**

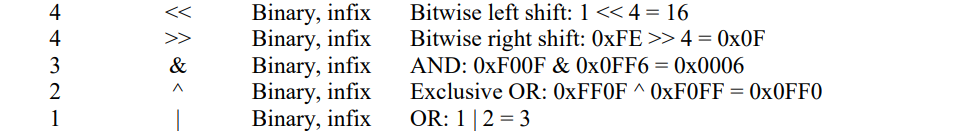
*Operators vs. Instructions:*

**Operators** perform computations only with constant **SCALAR values computable at assembly time** (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 until run time.

*Operators:*





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

**Type operators**: BYTE, WORD, DWORD, QWORD or TWORD

**5.3. Directives**

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

*The SEGMENT directive:*

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



1. *Name:*

The numeric value assigned to the segment **name** is the segment address (32 bits) corresponding to the memory segment’s position during run-time.

1. *Type:*

The optional argument **type** allows selecting the usage mode of the segment, having the 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

1. *Alignment:*

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.

1. *Combination:*

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. (implicit value)
* **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.

1. *Usage:*

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

1. *Class:*

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.

*Data definition directives:*

**DB, DW, DD, DQ, DT**

**RESB, RESW, RESD, RESQ, REST**

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

**EQU directive** allows assigning a numeric value or a string during assembly time to a label without allocating any memory space.

**6. Assembly language instructions**

**XCHG** instruction – allows interchanging the contents of two operands having the same size, at least one of them having to be a register. Its syntax is: **XCHG operand1, operand2**

**XLAT** instruction – "translates" the byte from AL to another byte, using for that purpose a user-defined correspondence table called translation table. The effect of XLAT is the replacement of the byte from AL with the byte from the translation table (whose offset is in EBX) having the index the initial value from AL. The syntax of the XLAT instruction is: **[reg\_segment] XLAT**

**LEA** (Load Effective Address) – transfers the offset of the memory operand into the destination register. LEA has the advantage that the source operand may be an addressing expression.

For example:

lea EAX, [v] loads into EAX the offset of the variable v, the instruction equivalent to mov EAX, v

lea EAX, [ebx+v-6] is not equivalent to a single MOV instruction

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

**PUSHF** instruction transfers all the flags on top of the stack (the contents of the EFLAGS register are transferred onto the stack).

**POPF** instruction extracts the word from top of the stack and transfer its contents into the EFLAGS register.

**MOVSX d, s** – loads in d (REGISTER), which must be of size larger than s (reg/mem), the UNSIGNED contents of s (**zero extension**).

**MOVZX d, s** – load in d (REGISTER), which must be of size larger than s (reg/mem), the SIGNED contents of s (**sign extension**).

**6.1 Conversions classification**

1. **Destructive** – cbw, cwd, cwde, cdq, movzx, movsx, mov ah,0; mov dx,0; mov edx, 0

**Non-destructive** – byte, word, dword, qword

1. **Signed** - cbw, cwd, cwde, cdq, movsx

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

1. **By enlargement** – cbw, cwd, cwde, cdq, movzx, movsx, mov ah,0; mov dx,0; mov edx, 0, word, dword, qword

**By narrowing** – byte, word, dword

**7. Multimodule programming**

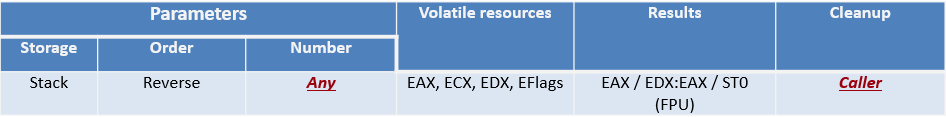
**7.1 Volatile vs. non-volatile resources**

- **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 and after that, at the end of the call to restore the initial values. **Part of the caller’s job to save volatile resources (as part of the call code)**. Also, the caller has to restore these values but not as part of a certain call/entry or exit code.

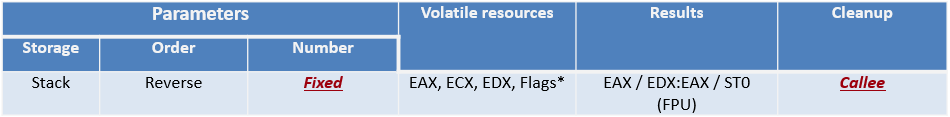
- **non-volatile** resources are any memory addresses or registers which do not belong explicitly to the called subroutine. **Part of the callee’s job to save and restore non-volatile resources as part of the entry code and exit code respectively**.

**7.2 Calling conventions**

**CDECL:**

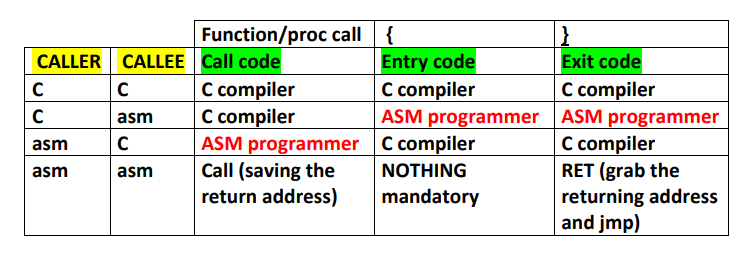


**STDCALL:**



*Steps:*

* **Call code**: call preparation and execution
* **Entry code**: procedure entry and preparation of execution
* **Exit code**: returning and freeing up resources

****

***Call code* (the caller):**

1. Saving the volatile resources (EAX, ECX, EDX, EFLAGS)
2. Passing parameters
3. Saving the returning address and performing the call

***Entry code* (the callee – called subroutine):**

1. Building the new stack-frame: PUSH EBP; MOV EBP, ESP
2. Allocating space for local variables: SUB ESP, nr\_bytes
3. Saving non-volatile resources exposed to be modified

***Exit code* (the callee):**

1. Restoring non-volatile resources
2. Freeing the space allocated for local variables (ADD ESP, nr\_bytes – mentioned here just as a reverse for the above b) from the entry code, but not really necessary because deallocating the stack-frame MOV ESP, EBP includes this action anyway from a practically point of view)
3. Deallocating the stack-frame MOV ESP, EBP and restoring the base of the caller stack-frame POP EBP (old EBP)
4. Returning from the subroutine (RET):
   1. CDECL – caller cleanup
   2. STDCALL – callee cleanup

**Stack frame** – data structure stored on the stack of fixed dimension containing: parameters, return address, copies of non-volatile resources and local variables.