ASC Lecture 2 → the general registers of the EU + flags

Notes | lecture notes 2, 3

Why are they called **general**? Because they have a general purpose.

In contrast, the BIU is designed to work only with a specific purpose: CS - code, DS - data, etc

(E comes from extended → when they were moved from 16-bits (AX) to 32-bits(EAX))

EAX - Accumulator register

→ used by most instructions as **one of the *implicit* operands**

EBX - Base register

ECX - Counter register

→ used for loops (it will look into ECX and will perform as many iterations as its value)

EDX - Data register

→ used usually together with EAX in computations in which the result exceeds a double word

! EAX → EDX:EAX; *mul* will always go into AL → byte, AX → word, DX:AX → dword, EDX:EAX →qword

ESP - Stack pointer

→ contains the offset of (a pointer to) the element at the **top** of the stack

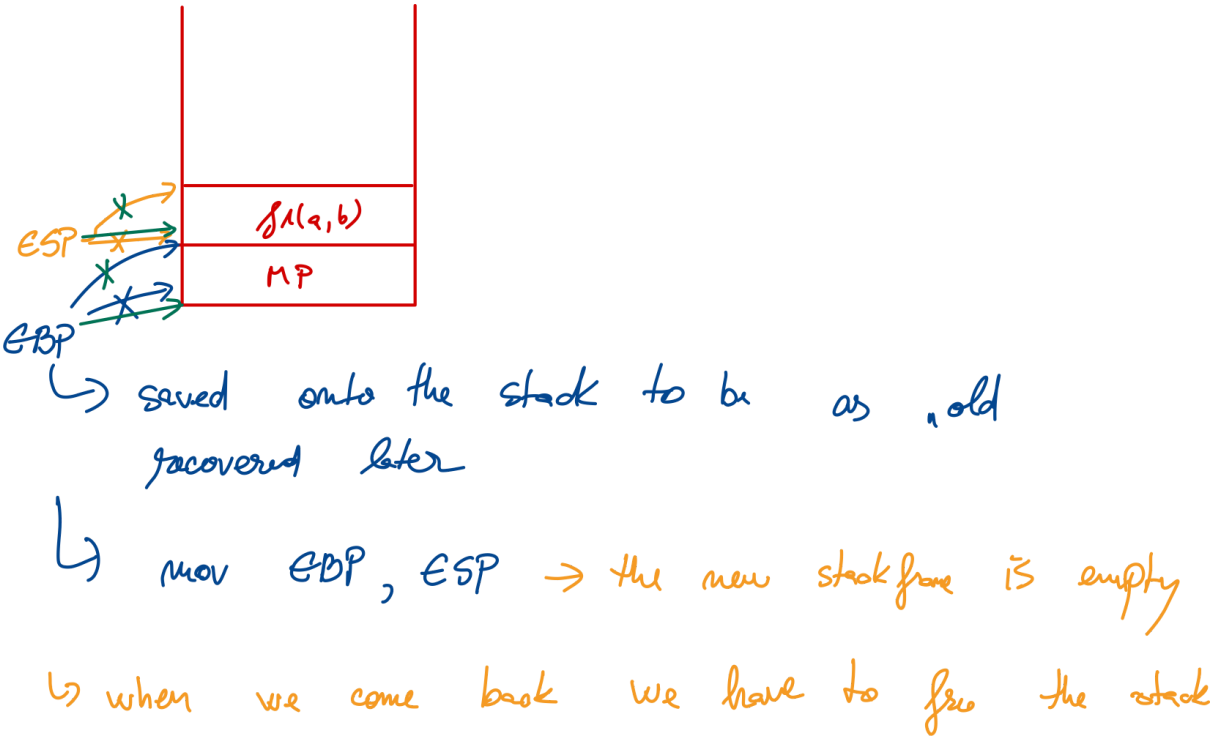
EBP - Base pointer

→ contains the offset of (a pointer to) the **base (first)** element of the stack

stack = a data structure that respects LIFO

queue = a data structure that respects FIFO

→ **SS gets executed first**

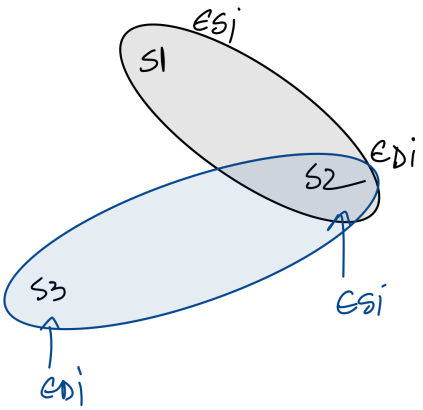


EDI - Destination Index (pointer, offset)

ESI - Source Index

→ strings are not defined data structures in .asm, that's why we need a "source pointer" and a "destination pointer"

→ why are there just 2? because you usually work with 2 strings. but what if you want to work with 3 strings?



💡 the size of a **word (always a 16 bit value)** defines the 32 bits type architecture

Q: Why are there 3 registers that work with the stack?

→ SS, ESP, EBP

! runtime stack is the main part of the program

→ the role of ESP and EBP is to **always define the currently executed stack-frame**

→ unlike for ADD and SUB, where the interpretation can be decided afterwards, for **MUL and DIV you have to decide BEFORE the implementation regarding the future interpretation (mul / imul or div / idiv)**

💡 ASM is a **value *oriented*** language ⇒ everything that flows in this language is **a numerical value**

Flags

→ indicator represented on 1 bit

A *flag* is an indicator represented on 1 bit. A configuration of the FLAGS register shows a synthetic overview of the execution of the each instruction. For x86 the EFLAGS register (the status register) has 32 bits but only 9 are actually used.

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

CF → carry flag

→ set to 1 if in the **last performed operation** there was a digit outside the representation domain

$$\begin{array}{r} 1001\ 0011 + \\ \underline{0111\ 0011} \\ 1\ 0000\ 0110 \end{array} \qquad \begin{array}{r} 147 + \\ \underline{115} \\ 262 \end{array} \qquad \begin{array}{r} 93h + \\ \underline{73h} \\ 106h \end{array} \qquad \begin{array}{r} -109 + \\ \underline{115} \\ 06 \end{array}$$

```
mov ah, 93h
mov al 73h
add ah, al ; AH = 06h, CF = 1
```

```
; how to fix it?
mov al, cf ; syntax error
```

```
; solution for adding the carryflag
mov al, ah
mov ah, 0
adc ah, 0 ; brings into AH the value of CF
```

PF → parity flag

→ together with the bits 1 from the least significant byte representation of the last performed operation results an odd number of 1 digits obtained

→ the number of bits 1 from is **odd ⇒ PF = 1**

- used in data transmition, to see if it was done in a correct way

AF → auxiliary flag

→ shows the transport value from bit **3** to bit **4**

→ every 4 bits represent 1 hexa digit **⇒ nibble (half of a byte)**

→ shows if we have a transport digit from the lower to the higher part, in the example bellow **AF = 1**

$$\begin{array}{r} 01101100 + \\ 11000100 \\ \hline 100110000 \end{array}$$

ZF → zero flag (a truth value flag)

→ if LPO was a 0, ZF = 1

SF → sign flag (also a truth value flag)

→ takes the value of the left-most bit

→ SF = 0 if the result is positive and SF = 1 otherwise

→ answers the question: Was the value just represented a *strictly negative number* or not

TF → trap flag

→ debugging flag, every time you press F7 the *trap flag* is set to 1

→ when you break at an instruction, the TF = 1 before that line

IF → interrupt flag

→ don't have access to it

→ treats exceptions

→ *critical sections* are sections that cannot interrupted

→ when such a section starts, IF = 0 and the code cannot be interrupted until it is again set to 1

DF → direction flag

→ the processor is not "aware" of strings, they are just continuous chunks of data, that can be "read" ascending or descending

→ destined for string instructions

→ if DF = 0 ⇒ ascending, DF = 1 ⇒ descending

→ works with **EDI** and **ESI** (indexes in strings)

OF → overflow flag

→ is set to 1 if the operation exceeded the reserved memory, 0 otherwise

→ the result of the LPO didn't fit the operands reserved space (admissible representation interval)

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

They will, in most situation, not have the **same value**

💡 **CF signals overflow for UNSIGNED interpretation, while OF signals the SIGNED overflow**

1001 0011 +	147 +
0111 0011	115
1 0000 0110	262

unsigned

byte + byte = byte

⇒ 147 + 115 = 6 FALSE

CF = 1

CF tells you that the operation went wrong

signed

the only correct one

OF = 0

93h +	-109 +
73h	115
106h	06

It might seem like flags are meant to show us what just happened, but that is not the case for all of them:

💡 Flags that show us what **just** happened, reporting the status of the LPO:

CF, OF, PF, ZF, AF, SF

Flags that have to be set by the programmer for future effect on instructions that follow:

TF, IF, DF, CF

> If you have special instructions that take in consideration the value of one flag it would mean that it might let you set it

Instructions that use the values from flags

Conditional JUMPS

ja = jbne ; jg = jnge ; jz ...

Now it should be clear that at the level of the microprocessor, f**lags exist for offering the programmer the possibility of verifying some conditions**. In that aspect, conditional jump instructions were introduced in .asm for the programmers to be able to express the situations to be verified.

How do you set *flags*?

→ there are **7** instructions to change the value of 3 flags:

CF: clc, stc, cmc (complement carry)

DF: cld, std

IF: cli, sti ← can be used by the programmer only on 16-bits, but on 32-bits, the OS restricts access to this instruction

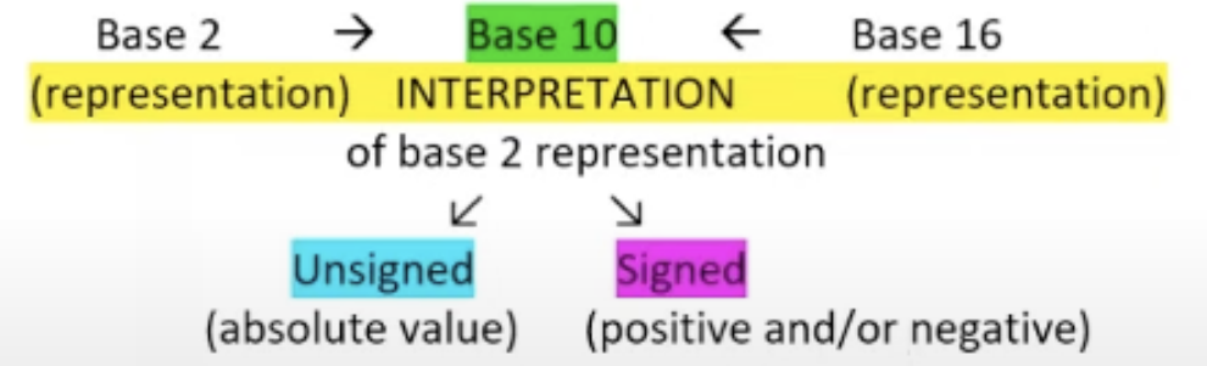
TF: should't go there, you are not allowed :)

The instructions that *are meant to set the flag values* **do not have explicit operands**, because each of them works on a *single* and *separate* flag.

Limits of intervals

⚠ on 8 bits / 1 byte: [0, **255**], [-128, **127**]

⚠ on 16 bits / 1 word: [0, **65535**], [-32768, **32767**]



you can have interpretation ONLY in base 10, base 2 and base 16 are representations!!

Q: Why the design of ADD and SUB is in such a way that the result must be the same size as the operands?

- CF is showing you if the result doesn't fit the data size. Moreover we don't need more than **one bit** for that since the "transport" can *at most be 1*

Q: Why doesn't ASM allow you instructions with more than 2 operands?

- the possibilities for CF in this case would be 0, 1 and 2 **but you cannot store 2 on a bit**

Q: Why did the designers provide 2 different flags for overflow? **16:40**

- the addition is only one **but** we have 2 different interpretations that give us **2 different values**

Q: How do you tell the processor to perform an addition in the unsigned interpretation?

- you **cannot** because you always have **2 possible interpretations**

Q: Why do we not have IADD and ISUB?

- because the addition goes the same and therefore, the implementation for these would be the exact same as ADD and SUB

Q: Why did the designers implement ADD and SUB with the result on the same size as the operands, not double like for MUL or DIV?

- because the only allowable **overflow is 1**, therefore it is useless, a waste of memory, to assign 1 whole byte only for 1 bit

Q: What are flags good for?

- they are used for checking conditions at the level of the programmer

Q: Why do we have 3 instructions for CF?

→ the 3rd one is **cmc** which complements the carry, used in 2's complement