The general registers of the EU (executive unit)

Why general? Because they have a general purpose (the BIU is designed to work only with a specific purpose: CS, DS, ...)

EAX - Accumulator register

↳ most used processor

↳ it is used by most instructions as one of the operands

↳ E comes from "extended"

EBX - The Base register

* the processor understands datatypes as sizes, nothing else

db - byte
dw - word
dd - double-word
dq - quad-word

* arrays can be only

↳ continuous sequence of memory units (smallest accessible being the byte)

a[7] = *(a+7)

↳ pointer to the 7th memory unit starting from the address of "a"

a - base / starting address of unit

ECX - Counter register

↳ a loop will look into ECX and will perform as many iterations as it's 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, AX, EDX:EAX

* div you only specify the divider, the dividend is implicit: EAX, AX, EAX:EDX

↳ and the result is always EAX, AX, AL

H - highest part

L - lowest part

Two pairs :

ESP - Stack pointer

↳ contains a pointer to the element from the top of the stack

EBP - Base pointer

↳ cont. pointer to the element from the base (first) element of the stack

stack → data structure that respects LIFO - last in first out

queue → ——— // ——— FIFO - first in first out

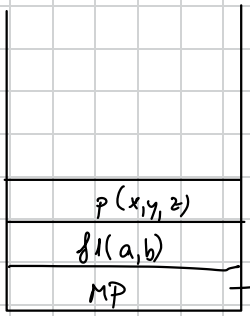
SS - stack segment → ESP - for start
EBP - for base

there's no queue segment. why?

exam Why 8 registers work with the stack?

Runtime stack is the main part the

the execution of an .exe file :



→ main program → a stack frame

run-time stack

implementation of running program → stack frame

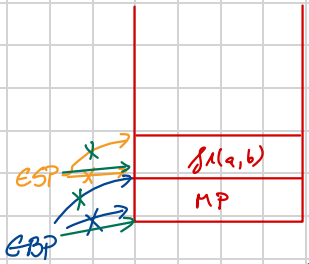
↳ information needed for the program to go on

every time the active execution frame is on TOP of the stack frame

MP always starts and stops the execution

Why do we need 3 registers for stack?

↳ the role of ESP and EBP is to always define the currently executed stack frame



↳ saved onto the stack to be as „old" recovered later

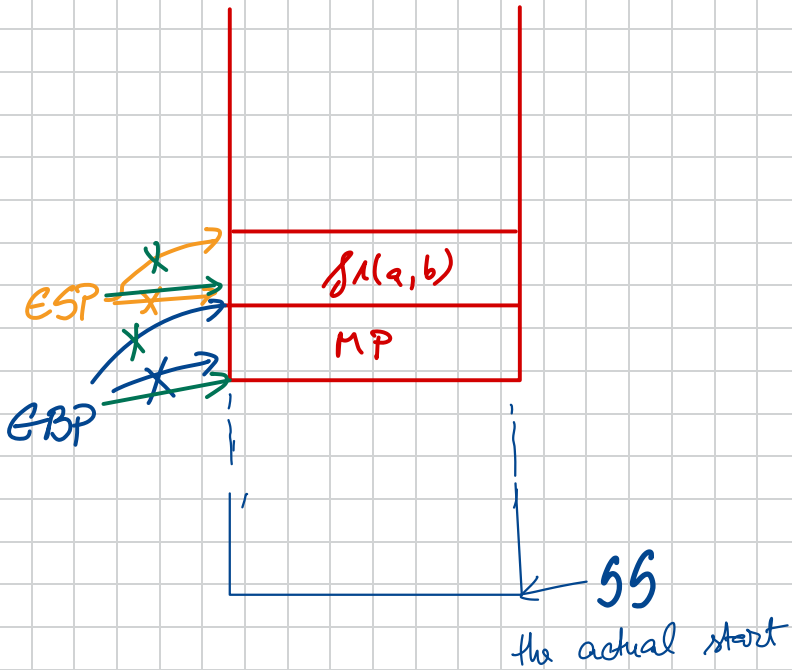
↳ mov EBP, ESP → the new stack frame is empty

↳ when we come back we have to free the stack

mov ESP, EBP

* MP is not the first

SS is the first



↳ logical segments:

CS – code segments

DS – data segments

SS – stack segments

ES – extra segments

* there's no extra logical segment in 32 bits programming

the 3 stack segments:

you don't need to start the execution of a program for SS to have value

you cannot access the high part of EAX, EBX... alone

→ the lower part of EAX ... is AX

EDi – Destination Index (pointers) !

esi – Source Index (pointers) !

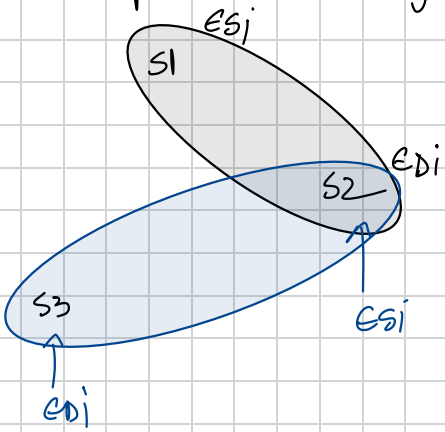
Strings are not defined structures for assembly

Why ? Because you usually work with 2 strings (input and output)

& base registers and

2 index registers

How to implement 3 strings? → pairs of 2, work on them and then add the third



push ESi
push EDi
↳ save the value onto the stack

→ again save the values ....

mov esi, 7 → index moreover ??

the size of "word" defines the 32 bits type architecture
↳ always a 16 bits value

Flags

↳ an indicator represented on 1 bit.

↳ the EFLAGS register has 32 bits but only 9 are actually used

CF - Carry flag → transport flag

↳ LPO (last performed operation)

↳ CF = 1 if I have a digit outside the domain, 0 otherwise

CF (*Carry Flag*) is the transport flag. It will be set to 1 if in the LPO there was a transport digit outside the representation domain of the obtained result and set to 0 otherwise. For example, in the addition

1001 0011 +	147 +		93h +	-109 +
0111 0011	115	there is transport and CF is set therefore to 1	73h	115
1 0000 0110	262		106h	06

CF flags the UNSIGNED overflow !

↑ without the carry flag this would be a 6, which is a mathematical impossibility

PF (*Parity Flag*) – Its value is set so that together with the bits 1 from the least significant byte of the representation of the LPO's result an odd number of 1 digits to be obtained.

AF (*Auxiliary Flag*) shows the transport value from bit 3 to bit 4 of the LPO's result. For the above example the transport is 0.

ZF (*Zero Flag*) is set to 1 if the result of the LPO was zero and set to 0 otherwise.

SF (*Sign Flag*) is set to 1 if the result of the LPO is a strictly negative number and is set to 0 otherwise.

TF (*Trap Flag*) is a debugging flag. If it is set to 1, then the machine stops after every instruction.

IF (*Interrupt Flag*) is an interrupt flag. If set to 1 interrupts are allowed, if set to 0 interrupts will not be handled. More details about IF can be found in chapter 5 (Interrupts).

DF (*Direction Flag*) – for operating string instructions. If set to 0, then string parsing will be performed in an ascending order (from the beginning to its end) and in a descending order if set to 1.

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

(1) → transport

99 +
99
1 98

byte + byte = byte → Always the same size

99 +	99 + 99 = 98 and I have a transport digit
99	
1 98	

- 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 that the result doesn't fit in the data size
- Why is ASM not allowing you instructions with >2 operands?
  - ↳ the pos. for CF are 0, 1, 2 BUT you cannot store a 2 on a bit
- each base 2 representation has 2 base 10 interpretations

  $1 \cdot 2^0 + 0 \cdot 2^1 \ldots$ → NOT efficient

  ↳ convert to base 16, then to base 10

CF flags the UNSIGNED overflow!

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

mov al, CF → ✗ syntax error

↳ make a conversion to word

```
mov al, ah
mov ah, 0
adc ah, 0
```

→ brings the carry flag into solution

adc → add with carry