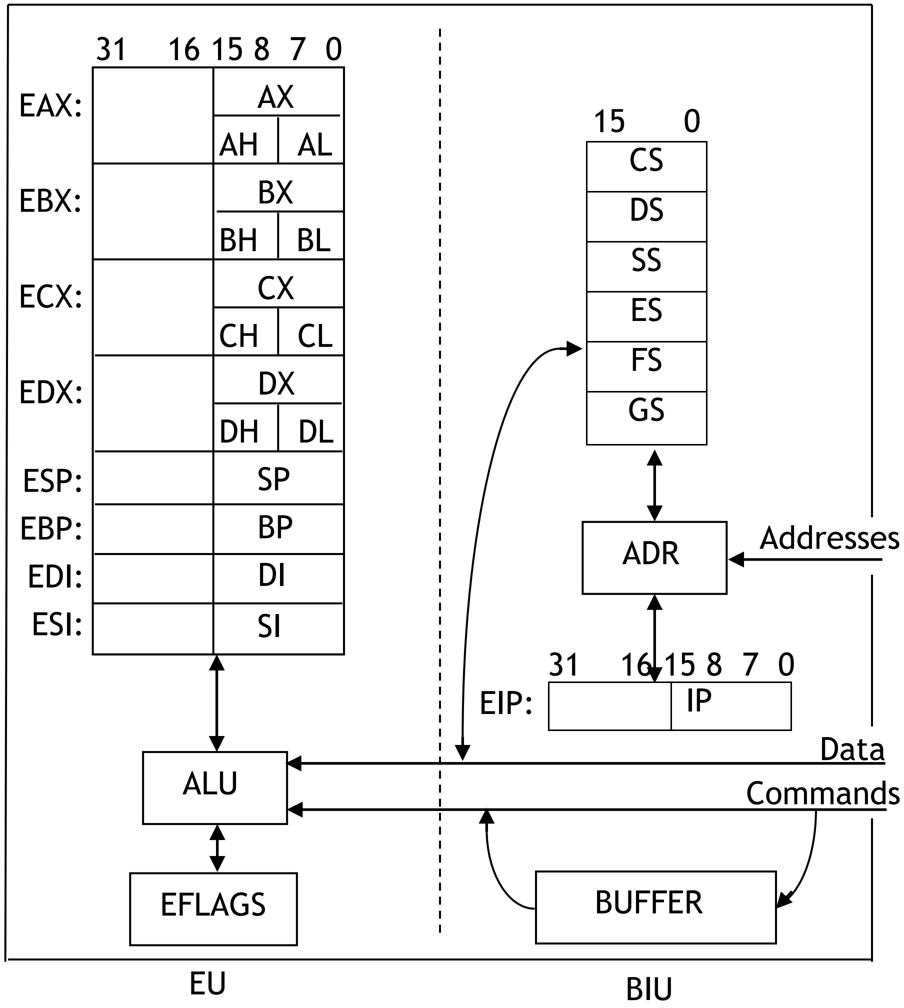
2.6. THE x86 MICROPROCESSOR ARCHITECTURE (IA-32)

2.6.1. x86 Microprocessor's structure

The x86 microprocessor has two main components:

- **EU** (*Executive Unit*) – run the machine instr. by means of **ALU** (*Arithmetic and Logic Unit*) component.
- **BIU** (*Bus Interface Unit*) - prepares the execution of every machine instruction. Reads an instruction from memory, decodes it and computes the memory address of an operand, if any. The output configuration is stored in a 15 bytes buffer, from where EU will take it.



EU and **BIU** work in parallel – while **EU** runs the current instruction, **BIU** prepares the next one. These two actions are synchronized – the one that ends first waits after the other.

2.6.2. The EU general registers

EAX - *accumulator*. Used by the most of instructions as one of their operands.

EBX – *base register*.

ECX - *counter register* – mostly used as numerical upper limit for instructions that need repetitive runs.

EDX – *data register* **-** frequently used with EAX when the result exceed a doubleword (32 bits).

"Word size" refers to the number of bits processed by a computer's CPU in one go (these days, typically 32 bits or 64 bits). Data bus size, instruction size, address size are usually multiples of the word size. So, for a CPU the "word size" is a basic attribute/feature influencing the above mentioned elements.

Just to confuse matters, for backwards compatibility, Microsoft Windows API defines a WORD as being 16 bits, a DWORD as 32 bits and a QWORD as 64 bits, regardless of the processor. So, WORD and DWORD DATA TYPES are ALWAYS on 16 and 32 bits respectively FOR THE ASSEMBLY LANGUAGE , regardless of the CPU's "word size" (16, 32 or 64 bits CPU).

ESP and **EBP** are *stack* registers. The stack is a LIFO memory area.

Register **ESP** (*Stack Pointer*) points to the last element put on the stack (the element from the top of the stack).

Register **EBP** (*Base pointer*) points to the first element put on the stack (points to the stack's basis).

EDI and **ESI** are *index registers* usually used for accessing elements from bytes and words strings. Their functioning in this context (*Destination Index* and *Source Index*) will be clarified in chapter 4.

EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI are doubleword registers (32 bits). Every one of them may also be seen as the concatenation of two 16 bits subregisters. The upper register, which contains the most significant 16 bits of the 32 bits register, doesn't have a name and it isn't available separately. But the lower register could be used as single so we have the 16 bits registers **AX, BX, CX, DX, SP, BP, DI, SI.** Among these registers, AX, BX, CX and DX are also a concatenation of two 8 bits subregisters. So we have **AH, BH, CH, DH** registers which contain the most significant 8 bits of the word (the upper part of AX, BX, CX and DX registers) and **AL, BL, CL, DL** registers which contain the least significant 8 bits of the word (the lower part).

2.6.3. Flags

A *flag* is an indicator represented on 1 bit. A configuration of the FLAGS register shows a synthetic overview of the execution of 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

We have 2 flags categories:
a). reporting the status of the LPO (having a so called previous effect) – CF, PF, AF, ZF, SF, OF
- ADC ; Conditional JUMPS (23 instructions – ja = jnbe; jg = jnle ; jz; …)

b). flags to be set by the programmer having a future effect on instructions that follows – CF, TF, IF, DF
- HOW ?... by using SPECIAL intructions – 7 instructions

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

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 !

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

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

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

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

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

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

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

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

Flags categories

The flags can be split into 2 categories:

a). with a previous effect generated by the Last Performed Operation (LPO): CF, PF, AF, ZF, SF and OF
b). having a future effect after their setting by the programer, to influence the way the next instructions are run: CF, TF, DF and IF.

Specific instructions to set the flags values

Considering the b) category, it is normal that the assembly language to provide specific instructions to set the values of the flags that will have a future effect. So, we have 7 such instructions:

CLC – the effect is CF=0
STC – sets CF=1
CMC – complements the value of the CF ; 3 instructions for CF

CLD – sets DF=0
STD – sets DF=1 ; 2 instructions for DF

CLI – sets IF=0
STI – sets IF=1 ; 2 instructions for IF – they can be used by the programmer only on 16 bits programming ; on 32 bits, the OS restricts the access to these instructions !

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