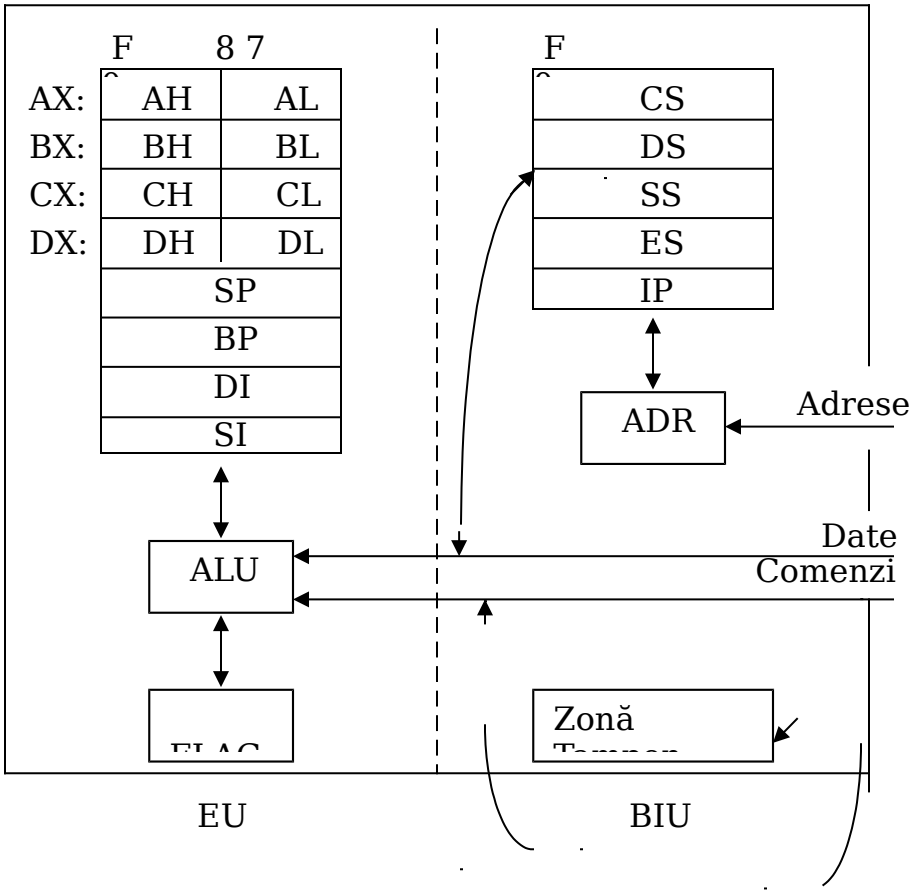
2.6. THE 8086 MICROPROCESSOR ARCHITECTURE

2.6.1. 8086 Microprocessor's structure

The 8086 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 6 bytes buffer, where from 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

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

BX – *base register*. Mostly used in address computation.

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

DX – *data register* - frequently used with AX for doubleword results (DX:AX).

AX, BX, CX, DX are word registers (16 bits). Every one of them may also be seen as the concatenation of 2 byte subregisters: the high part (the most significant 8 bits of the word – and we have though registers **AH, BH, CH, DH)** and the low part (the least significant 8 bits of the word – and we have though registers **AL, BL, CL, DL).**

SP and **BP** are *stack* registers. The stack is a LIFO memory area.

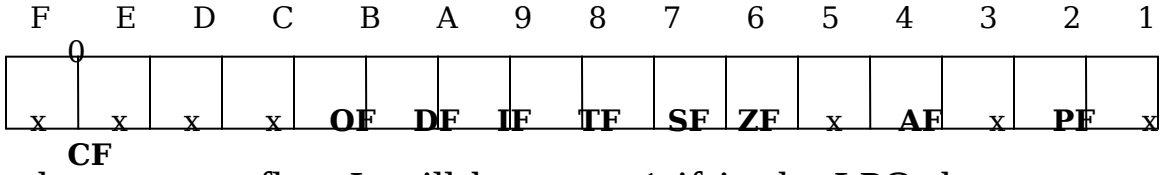
Register **SP** (*Stack Pointer*) points to the last element put on the stack (the element from the top of the stack).

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

DI and **SI** 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.

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 last performed operation (LPO). The FLAGS register has 16 bits but only 9 are actually used.



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 +
0111 0011 there is transport and CF is set therefore to 1
1 0000 0110

PF (*Parity Flag*) – Its value is set so that together with the bits 1 from 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 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 (signed interpretation) didn't fit the reserved space, then OF will be set to 1 and will be set to 0 otherwise.

2.6.4. Address registers and address computation

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

Design decision - 8086 - RAM size - 1 MB = 2^{20} bytes ; so, we need 20 bits for address representation. Registers size is 16 bits. So the problem is how can we obtain a 20 bits address using 16 bits registers?

The solution was introducing the concept of memory segment. *A memory segment* represents a continuous sequence of bytes having the following three properties:

(i) it starts at an address multiple of 16 (bytes) ; it follows that the 4 least significant bits of such an address are 0 ! So 16 bits are enough for pointing to the beginning of a memory segment

(ii) its length is a multiple of 16

(iii) its size is limited at 64 KB = $2^6 * 2^{10}$ bytes = 2^{16} bytes, so **16 bits are enough for representing any address inside a memory segment**.

We will call *offset* the address of a location relative to the beginning of a segment. An *address specification* is a pair of 16 bits numbers, one representing the segment's starting address and the second the offset of that memory location. In hexadecimal an address **specification** can be written as :

$$\mathbf{s_3s_2s_1s_0 : o_3o_2o_1o_0}$$

Based on such a specification the actual address computation will be performed as :

$$\mathbf{a_4a_3a_2a_1a_0 := s_3s_2s_1s_00 + o_3o_2o_1o_0}$$

This computation is done by the **ADR** component from **BIU**.

For example, the specification 7BC1 : 54A3 points actually to 810B3, as the result of the sum 7BC10 + 54A3.

This is not a bijective association. There is more than one possible specification for the same address. For example, 810B : 0003 points also to the location having the address 810B3. This addressing mechanism is called *Real Address Mode*. Starting with 80286, *Protected Virtual Address Mode* was introduced and starting with 80386 2 new modes were introduced : *paged mode* and *8086 virtual mode*, all these being designed to allow addressing more than 1 MB. Details may be found in chapter 10.

The 8086 architecture allows 4 types of segments:

- *code segment*, containing instructions ;
- *data segment*, containing data which instructions work on;
- *stack segment*;
- *extra 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 one active segment of any type. Registers **CS** (*Code Segment*), **DS** (*Data Segment*), **SS** (*Stack Segment*) and **ES** (*Extra Segment*) from **BIU** contain the starting addresses of the active segments, correspondingly to every type. Register **IP** contains the offset of the current instruction inside the current code segment, this register being managed exclusively by **BIU.**

2.6.5. Machine instructions representation

A 8086 machine instruction has maximum 2 operands. For most of instructions, they are called *source* and *destination* respectively. From these two operands, only one may be stored in the RAM memory. The other one must be either one **EU** register, either an integer constant. Therefore, an instruction has the general form:

instruction_name *destination**,** source*

The internal format of the 8086 instruction takes between 1 and 6 bytes. The first one, named *code* identifies the instruction to be run. The second, called *bytemode*, specifies for some of the instructions the nature and location of the operands (register, memory, integer constant etc.). Most instructions are represented on 1-2 bytes. The next maximum 4 bytes, if present, identify either a memory address, either a constant represented on more than 1 byte.

2.6.6. FAR addresses and NEAR addresses.

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 the segment's starting address is called a *FAR address.* A FAR address may ne specified in one of the three following ways:

- $s_3s_2s_1s_0$ *: offset_specification* where $s_3s_2s_1s_0$ is a constant;
- CS : offset_specification ; DS : offset_specification ;
  ES : offset_specification ; SS : offset_specification;
- VARDOUBLE , namely a doubleword variable containing a FAR address (pointer).

2.6.7. Computing the offset of an operand. Addressing modes.

For an instruction there are several ways of specifying a required operand:

- *register mode*, if the required operand is a register;
- *immediate mode*, when we use directly the operand's value (not its address and neither a register holding it);
- *memory addressing mode*, if the operand is located somewhere in memory. In this case, its offset is computed using the following formula:

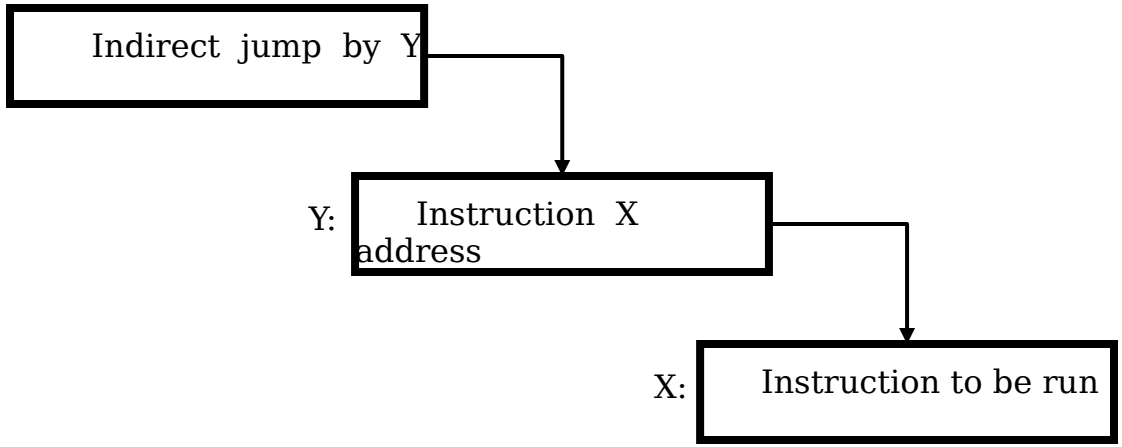
$$\textit{offset} = [\ BX \mid BP\ ] + [\ SI \mid DI\ ] + [\ \textit{constant}\ ]$$

based *indexed* *direct*

In the case of control transfer instructions (jumps) there are 2 additional addressing types :

Relative addressing – when the position of the next instruction to be run is expressed relative to the current position. This ”distance“ is expressed as the number of bytes to jump over, its range being between -128 and 127. Such an address is called a *short address*.

Indirect addressing – when the position of the next instruction to be run is expressed by an address, which at its turn is in a memory location, which address is given as the operand of the jump instruction:



Indirect addressing allows greater flexibility regarding instructions control. That is because depending on the context, the contents of Y may be different at two different moments of execution, allowing though running different instructions at different given moments.

Indirect addressing may be indirect NEAR addressing or indirect FAR addressing.