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	(=6?)		106h	06

CF flags the UNSIGNED overflow !

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.

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 vs. OF. The overflow concept.

1001 0011 +	147 +		93h +	-109 +
1011 0011	179	a carry of the most significant digit occurs	B3h	-77
1 0100 0110	326	so the value 1 is placed in CF	146h	-186 !!!!
	(unsigned)		**(hexa)**	**(signed)**
	CF= 1			**OF= 1**

- 326 and -186 are the correct results in base 10 for the corresponding interpretations

BUT, in ASSEMBLY language we have ADD b+b → b, so what we obtain as interpretations on 1 byte are :

1001 0011 +	147 +		93h +	-109 +
1011 0011	179	a carry of the most significant digit occurs	B3h	-77
1 0100 0110	70	so the value 1 is placed in CF	146h	+70 !!!!
	(unsigned)		**(hexa)**	**(signed)**
	CF= 1			**OF=1**

By setting both CF and OF to 1, the « message » from the assembly language is that both interpretations in base 10 of the base 2 addition are incorrect mathematical operations !

0101 0011 +	83 +		53h +	83 +
0111 0011	115	a carry DOES NOT occur so CF=0	73h	115
1100 0110	198		C6h	198 !!!!
	(unsigned)		**(hexa)**	**(signed)**
	CF=0			**OF=1**

- 198 is the correct result in base 10 for both the corresponding interpretations

BUT, in ASSEMBLY language we have   ADD   b+b → b, so what we obtain as interpretations on 1 byte are :

```
    0101 0011 +     83 +                                                    53h +        83 +
    0111 0011      115      a carry DOES NOT occur so CF=0                  73h         115
    1100 0110       198                                                     C6h        - 58   !!!!
                 (unsigned)                                                (hexa)     (signed)
                    CF=0                                                                OF=1
```

Setting CF to 0 expresses the fact that the unsigned interpretation in base 10 of the base 2 addition is a correct one and the operation functions properly. Setting OF to 1 means that the signed interpretation is NOT a correct mathematical operation !

OF will be set to 1 (*signed overflow*) if for the addition operation we are in one of the following situations (*overflow rules for addition in signed interpretation*). These are the only 2 situations that can issue overflow status for the addition operation:

```
0……+      or      1…..+     (Semantically, the two situations denote the impossibility of mathematical acceptance
0……               1…..        of the 2 operations: we cannot add two positive numbers and obtain a negative result
--------          -------      and we cannot add two negative numbers and obtain a positive result).
1……               0…..
```

In the case of subtraction, we also have two overflow rules in the signed interpretation, a consequence of the two overflow rules for addition:

```
1……-      sau     0…..-     (Semantically, the two situations denote the impossibility of mathematical acceptance
0……               1…..      of the two operations: we cannot subtract a positive number from a negative one and
------             -------    obtaining a positive one, neither can we subtract a negative number from a positive one
0……               1…..     and obtaining a negative one).
```

1	0110 0010 -	98 -		62h -	98 -
	<u>1100 1000</u>	<u>200</u>	We need significant digit borrowing for performing	<u>C8h</u>	<u>-56</u>
	1001 1010	154	the subtraction, therefore CF is set to 1	9Ah	-102 !!!!
		(unsigned)		**(hexa)**	**(signed)**
		CF=1			**OF=1**

None of the interpretations above is mathematically correct, because in base 10, the subtraction 98-200 should provide -102 as the correct result, but instead 154 is provided in the unsigned interpretation for the subtraction - which is an incorrect result! In the SIGNED interpretation we have: 98-(-56) = -102 (Again an incorrect value!!) instead of the correct one 98+56=154 (the value of 154 result interpretation is valid only in unsigned representation).

In conclusion, in order to be mathematically correct, the results of the two subtractions from above should be switched between the two interpretations; so the two interpretations associated to the binary subtraction are both mathematically incorrect. For this reason the 80x86 microprocessor will set CF=1 and OF =1.

Technically speaking, the microprocessor will set OF=1 only in one of the 4 situations presented above (2 for addition and 2 for subtraction).

The multiplication operation does not produce overflow at the level of 80x86 architecture, the reserved space for the result being enough for both interpretations. Anyway, even in the case of multiplication, the decision was taken to set both CF=OF=0, in the case that the size of the result is the same as the size of the operators (b*b = b, w*w = w or d*d = d) (« no multiplication overflow », CF = OF = 0). In the case that b*b = w, w*w = d, d*d = qword, then CF = OF = 1 (« multiplication overflow »).

The worst effect in case of overflow is in the case for the division operation: in this situation, if the quotient does not fit in the reserved space (the space reserved by the assembler being byte for the division word/byte, word for the division doubleword/word and respectively doubleword for division quadword/doubleword) then the « division overflow » will signal a ‘Run-time error’ and the operating system will stop the running of the program and will issue one of the 3 semantic equivalent messages: ‘Divide overflow’, ‘Division by zero’ or ‘Zero divide’.

In the case of a correct division CF and OF are undefined. If we have a division overflow, the program crashes, the execution stops and of course it doesn’t matter which are the values from CF and OF…