OVERFLOW CONCEPT ANALYSIS

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 !

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

Definition (general, compressed and incomplete). An *overflow* is a mathematical situation/condition which expresses the fact that the result of an operation didn't fit the reserved space for it. (neither -147 "fits" in the range [-128..+127] nor on a single byte, but it is more difficult to understand that the definition includes and refers to this case as well..)

Definition (more exact and complete). At the level of the assembly language an *overflow* is a 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 for that size 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 +	**(these are the correct**
1011 0011	179	B3h	- 77	**mathematical results !!)**
1 0100 0110	**326**	1 46h	**- 186** !!!!	
(binary representation)	**(unsigned interpr.)**	**(hexa repr.)**	**(signed interpretation)**	

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

From the discussion on the admissible representation intervals and respectively from the analysis of the type of problems "Which is the minimum number of bits on which can be represented... 326 and -186 respectively" it will result that

326 € [0..511] and -186 € [-256..+255] and thus the MINIMUM number of bits on which 326 and -186 can be represented is 9, and the representation of -186 is: 512-186 = 326 = 1 46h = 1 0100 0110

As a result, ALL the above operations are MATHEMATICALLY CORRECT on 9 bits and the final operands and results CAN fit in the reserved space, IF the operations are on 9 bits !!

But unfortunately, the above addition is performed at the 8-bit processor level (since in assembly language we have ADD b+b → b) and as a result, MATHEMATICALLY, it will NOT perform correctly on 8 bits, nor 326 nor -186 not fitting 1 byte!! (WHICH will NOT run correctly on 8 bits exactly? – base 2, base 10, base 16 ?... NONE will run correctly on 8 bits !!! and that's why CF and OF = 1 both…)

This is signaled SIMULTANEOUSLY by the flags CF (for unsigned interpretation) and OF (for signed interpretation), both flags being set to the value 1.

As a result, what we will get as results and especially as EFFECTS on 1 byte in the program is the following:

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	1 46h	+70 !!!!
	(unsigned)		**(hexa)**	**(signed)**
	CF= 1			**OF=1**

By setting both CF and OF to 1, the « message » from the processor is that both interpretations in base 10 of the above base 2 addition are incorrect mathematical operations on 8 bits !

0101 0011 +	83 +	53h +	83 +	**(these are the correct**
0111 0011	115	73h	115	**mathematical results !!)**
1100 0110	198	C6h	198 !!!!	
(unsigned OPERANDS interpretation)		**(hexa)**	**(signed OPERANDS interpretation)**	

- 198 is the correct result in base 10 for both the corresponding interpretations of the binary OPERANDS from the above addition, BUT now we have to see if the result fits on 8 bits (YES – it fits, therefore we will have CF=0) and respectively if the result of the binary operation in the **signed** interpretation, it is consistent with the correctness of the mathematical operation performed (it is NOT, because 11000110 is NOT a positive number in the signed interpretation!), so what we will get as results on 1 byte will be:

  (ADD b+b → b, so what we obtain in the computer 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 above base 2 addition is a correct one and the operation functions properly both at the level of the interpretation of the OPERANDS and the RESULT. OF will however be set by the processor to the value 1, this meaning that the signed interpretation of the base 2 addition above reflects an incorrect operation (more precisely the RESULT obtained is an incorrect one in the signed interpretation!)

OF will be set to 1 (*signed overflow*) if for the addition operation we are in one of the following two 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 -	**(-102 and 154 are the correct**
1100 1000	200	C8h	-56	**mathematical results in base 10!!)**
1001 1010	-102	9Ah	154 !!!!	
(unsigned OPERANDS)		**(hexa)**	**(signed OPERANDS)**	

However, if we also take into account the **interpretations of the RESULT** obtained in the program, we have:

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

Both interpretations of the result obtained in base 2 ARE MATHEMATICALLY INCORRECT, so CF and OF will both be set to the value 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 final 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…

w/b → b 1002/3 = 334 = division overflow – fatal – Run time error ('Divide overflow', 'Division by zero' sau 'Zero divide') – technically an **INT 0** will be issued ! – Why such a message is issued even if we divided at 3 ?

Because from the mp point of view regardless of whether we divide by ZERO or something different from zero, the result DOESN'T FIT in the reserved space, the situation we end up with is the SAME: run-time error because it DOESN'T FIT!!

334 does not fit in a byte, and INFINITY DOESN'T FIT IN ANYTHING!!!!!!!!!

number/0 – Zero divide = IT IS OVERFLOW because INFINITE does NOT fit anything !!!!!!!!!

Why do we need CF and OF in EFLAGS SIMULTANEOUSLY?? Isn't a single flag enough to show us IN TURN if we have an overflow or not, either in the signed interpretation or in the unsigned one ? NO, because when performing an addition or subtraction operation in base 2, in fact 2 operations are actually performed SIMULTANEOUSLY in base 10: one in the signed interpretation and the other in the unsigned interpretation.

As a result, two different flags are needed **SIMULTANEOUSLY** to deal each of them **SEPARATELY** with **one** of the 2 possible interpretations in base 10: CF – for the unsigned interpretation ; OF – for the signed one**.**

This happens because the operation of addition or subtraction expressed IN BASE 2 is performed IDENTICALLY, therefore, REGARDLESS OF THE signed or unsigned INTERPRETATION of the operands and the result !!!! This is also the reason why there is **NO IADD or ISUB** in assembly language! Because even if they existed, they would NOT work differently from ADD and SUB respectively!

ADD = IADD, SUB = ISUB – because in base 2 the addition and subtraction are performed in the same way INDEPENDENTLY OF THE INTERPRETATION !!!!

- **So, why do we need then IMUL and IDIV ??**

Because unlike addition and subtraction, which work the same in base2, regardless of the interpretation (signed or unsigned), SIGNED and UNSIGNED multiplication and division work DIFFERENTLY in the signed case compared to the unsigned case !!

As a result, in case of addition and subtraction there is no need to specify BEFORE the computation how we want the operands and the result to be interpreted, because the 2 operations work anyway the same BINARY regardless of how we want to interpret them. It is enough to decide AFTER performing the operation how we want the operands and the result to be interpreted.

In contrast, multiplication and division do NOT work BINARY the same in the 2 interpretations, here there is the need to specify BEFORE a multiplication or division how we want the operands to be interpreted and this is done precisely by specifying MUL and DIV (if we want unsigned operands) or respectively IMUL and IDIV (if we want signed operands).