**4.3.2.3.** **Commented examples**

The action mode of conditional jump instructions and the **cmp** instruction

mov al,80h ;al := 128 = 10000000b = -128 ! (Interesting! – we remark that, because of the complementary code representation rules 128 and -128 have the same binary representation, which is 10000000b!)

(\*) cmp al,0 ;the cmp instruction does not interpret in any way the value in AL(as signed or unsigned), it only computes the fictional subtraction **al-0** and affects the flags accordingly: SF=1 , CF=ZF=OF=PF=AF=0.

jl et ;The use of the JL instruction (Jump if Less than) leads to the signed interpretation of the al<0 comparison (see table 4.2), namely cf. of the table 4.1 it tests if SF≠OF and since SF=1 and OF=0, the fulfillment of the condition is decided and the jump at the et label. We deduce that the interpretation of the compared values was up to the programmer who, by the use of the JL instruction, decided that he wants to compare -128 with 0 and since -128 is less than 0, the condition was fulfilled (echiv. with jnge et). In contrast, jnl et or jge et , (which will test if SF=OF) will NOT be fulfilled and will NOT cause the jump at the specified label.

jb et ;The use of the JB instruction (Jump if Below), leads to the signed interpretation of the al<0 comparison (see table 4.2), which means cf. of the table 4.1, it is tested if CF=1 and since CF=0, the non-fulfillment of the condition is decided so the jump to the et label will not be performed. We deduce that if the interpretation of the compared values was up to the programmer, which by using the JB instruction, he decided that he wants to compare 128 with 0 and since 128 is NOT “below” 0, the condition is NOT fulfilled (equivalent with jnae et with jc et).

jae et1 ;unsigned testing if al ≥ 0 (128 ≥ 0?) - CF=0, so the condition is fulfilled (equivalent with jnc et1 or jnb et1) – the jump to the et1 label is performed

jbe et2 ;signed testing if al ≤ 0 (128 ≤ 0?) – CF = ZF = 0, so the condition (CF=1 sau ZF=1) is NOT fulfilled and as a consequence the jump to the et2 will not be performed – consistent result with jb et, because jbe implies jb ;(equivalent with jna 1et2)

ja et3 ;unsigned testing if al > 0 (128 > 0?) – CF = ZF = 0, so the condition (CF=0 şi ZF=0) is fulfilled and as a consequence, the jump to et3 will be performed (equivalent with jnbe et3) and a consistent result with jbe et2, because if jbe is not fulfilled, so ja has to be.

je et4 ;It is tested if al = 0 (128 = 0 ?) – the sign does not matter if what is tested is the equality! – since ZF=0, the ZF=1 condition is not fulfilled so the jump to the label et4 will not be performed (equivalent with jz et4). In contrast, jne ;et4 or jnz et4 (it will be tested if ZF=1) will be fulfilled and will lead to the jump to the specified label.

jle et5 ;Signed testing if al ≤ 0 (-128 ≤ 0?) – OF = ZF = 0 and SF=1 , so the condition (ZF=1 or SF≠OF) is fulfilled and as a consequence the jump to the et5 is performed (equivalent with jng et5) and as a consistent result with jl et, because ;jle implies jl.

jg et6 ;Signed testing if al > 0 (-128 > 0?) – OF = ZF = 0 and SF=1, so the condition (ZF=0 şi SF=OF) is NOT fulfilled and as a consequence the jump to the et6 label will NOT be performed (equivalent with jnle et6) and a consistent result with jle ;et5, because if jg is not fulfilled, then jle has to be.

jp et7 ;It is tested if PF=1 - PF=0, so the condition is fulfilled – the jump will not be performed (equivalent with jpe et7 – *Jump if Parity Even*). In contrast, jnp et7 ;(which tests if PF=0 – equivalent with jpo et7 – *Jump if Parity Odd*) will be fulfilled and the jump will be performed

jo et8 ;It is tested if OF=1 – OF=0, so the condition is not fulfilled - the jump

is not performed (there is no overflow). In contrast, jno et8 (which

tests if OF=0) will be fulfilled and the jump will be performed

js et9 ;It is tested if in the signed interpretation the result of the comparison is negative (because as we specify when presenting the CMP instruction, it is not about the signed or unsigned interpretation of the ***operands*** of the fictional subtraction d-s, but the ***final result*** of it!), which means testing if SF=1 - the condition is fulfilled in our case and as a consequence, the jump will be performed! In contrast, jns et9 (which tests if SF=0) will NOT be fulfilled and the jump will NOT be performed.

cmp 0,al ;syntax error: *“Illegal immediate*”, because the syntax of the cmp instruction prohibits the specification as a first operand of an immediate value (constant); however, if we want to force a comparison of this kind (0-al) we can use as a first operand a register initialized with the value 0.

mov bl,0

cmp bl, al ;performs the fictional subtraction bl-al (0-al = 0-80h = 0-10000000b = ;10000000b) and affects the flags accordingly: CF=SF=OF=1, ;ZF=PF=AF=0.

**Proposed exercise**: Resume the discussion about the effect of all the conditional jump instructions from above (analysed for the comparison case (\*) cmp al,0) in the conditions in which this comparison is replaced by the last two presented instructions, hence in the case in which we perform cmp bl,al cu bl=0.

**Which would be the justification that in the case cmp bl,al we have CF = OF = SF = 1 and in the case *cmp al,0* only SF=1 and CF = OF = 0 ?**

For justifying the different ways of setting the flags we have to take into account the practical rules of setting these flags. These general rules are:

* SF takes the value of the sign bit of the obtained result
* CF takes the value of the transport digit: if we are talking about an addition, we analyze if the obtained result result caused (CF=1) or not (CF=0) a transport outside the representation space; if we are talking about a subtraction d-s we have: if |d| ≥ |s| then CF=0 (there is not need for a borrow digit in order to perform the subtraction) and if |d| < |s| then CF=1 (we need a borrow for performing the subtraction and this fact is reflected in CF)
* OF is set to value 1if there exists an overflow in the signed interpretation of the result (“*OF is set if there exists a signed overflow*”), which means that if the obtained result does not fit in the accepted interpretation interval (this being [-128..+127] if we are talking about byes [-32768..+32767] and about signed interpreted words).

The last two rules derive from the implementation manner of the overflow concept in the 80x86 processor.

**In the case of the unsigned operation/operands the overflow will be reported by setting the CF (*carry flag*). In the case of signed operations/operands the overflow will be reported by setting the OF (*overflow flag*).**

**How do we detect the overflowing situations in the case of additions and subtractions?** Which are the practical rules that should be applied for understanding and justifying correctly the setting of the flags which we will remark in the run programs? In the following discussion, we will mainly focus on justifying the ways of setting the OF (*overflow flag*), because due to its name, it is the main responsible factor for the characterization of a situation by the programmers as being an overflow or not.

Pay attention to what is always ignored in this context, which is that a CF=1 situation (with OF=0) signals an overflow, but in the case of the unsigned numbers.

**For ADDITION: if we add two numbers of the same sign and the fresult is of a differnt sign, that signals an overflow** (OF=1), **otherwise** (OF=0). This is then what we could name *the rule of addition overflow* (RAO) in the case of signed interpretation.

For example, in the case of a byte, if we consider the addition 100 + 50 = 150, we will get a signed overflow (!) (seems surprising, doesn’t it ?). Justification: 100 (= 64h = 01100100b) + 50 (= 32h = 00110010b) = 150 (= 96h = 10010110b). The operands have the same sign, but the result is of a different sign, so according to RAO we will get OF=1. Intuitively, the overflow can be justified by 150 ∉ [-128..127], so we will obtain an error of the*“out of range”* sort. Even though we could remark that 150 = 10010110b = -106 (in signed interpretation), and -106 ∈ [-128..127], this last interpretation can not be accepted because the operands (100 and 50) have positive values in both interpretations (the sign bit being 0). As a result, the sum of two positive numbers can’t result in a negative number and so, the only accepted interpretation in this context for 10010110b is 150 ∉ [-128..127], so the OF is set to 1.

On the other hand, CF = 0 (there is no transport digit outside of the representation space), so there is no overflow in the unsigned interpretation: 100 + 50 = 150 ∈ [0, 255] (the accepted interval for the interpretation of unsigned numbers)

Similarly, in the signed interpretation the sum of two negative numbers can’t provide a positive number. We take the example:

10010110 +

10000010

**1** 00011000

We can see from the binary representation that there exists a transport digit outside of the accepted representation space of those 8 bits, so intuitively it is enough in the signed interpretation that the sum of two negative numbers (they are negative because the sign bite is 1 for both numbers) should provide a positive number: 00011000b = 18h = 24. This value is in fact a truncation of the correct binary value (on 9 bits) which should be obtained (100011000b = 118h), and the truncation takes place because of the overflow. As a consequence, we can’t obtain a positive number by adding two negative numbers (only through a truncation and the necessity of a truncation means overflow!). We can see that such a truncation always means the appearance of a transport digit 1 outside of the representation dimension of the result, so we will automatically have CF=1.

10010110b = 96h = -106 (in signed interpretation) = +150 (in unsigned interpretation)

10000010b = 82h = -126 (in signed interpretation) = +130 (in unsigned interpretation)

In unsigned interpretation we have 150 + 130 = 280 ∉ [0..255] (the intuitive justification of overflow). Technically, we saw that CF = 1 and it clearly follows that we have an overflow in the unsigned interpretation.

Hence, we can’t have -106 + (-126) = 24! (because 00011000b = 18h = 24 in both interpretations). This is in the case of applying RAO. Another intuitive justification of overflow would be this sort of situation:

In signed interpretation we have -106 + (-126) = -232 ∉ [-128..127], so OF=1.

This last motivation is more intuitive for justifying the overflow, but this kind of justifications are harder to express regarding an algorithm. Technically speaking, RAO remains ”the fastest applicable practical rule algorithmically speaking” if we can say so... (and look that we could!)

It results that in the case in which we add two numbers of a different sign there will never be an overflow. Also, if we add two numbers of the same sign, but the result has the same sign as the operands, there will not be an overflow either (it means that no truncation was needed for representing the result on the same dimension as the operands). It can be easily verified from a mathematical point of view that in neither of the cases do we get out of the accepted interpretation interval.

For SUBTRACTION: the operands are respectively interpreted as signed, the solicited subtraction is computed among the corresponding configuration of bits and if the obtained signed interpreted result fits in the accepted interpretation interval (the [-128...127] interval] for the signed bytes and respectively [-32768..32767] for signed words) then an overflow is reported, and so OF=1. We can name this formulation as the *rule of subtraction overflowing* (RSO) in the case of signed interpretation.

In the case of an unsigned subtraction overflow: the need to perform a subtraction with a borrow digit reported by the processor by setting CF=1, which we can semantically interpret as an ”unsigned subtraction overflow”.

Let’s analyze more examples with the purpose of clarifying the rules from above, like their impact over the way of setting the flags.

Examples:

**i).** mov ah,82h ;82h = 130 (unsigned interpretation) = -126 (signed interpretation)

; = 10000010b (the sign bit being 1; the two interpretations differ)

mov bh,2ah ;2ah = 42 (both in signed interpretation and unsigned interpretation)

; = 00101010b (the sign bit being 0, the two interpretations coincide)

cmp ah,bh ;the fictional subtraction is performed: ah-bh=10000010b - 00101010b

= 01011000b

; = 58h = 88 (both in signed and unsigned interpretation, because the sign bit is 0)

This subtraction sets the flags as follows:

SF = 0 (because the sign bit for the result of 58h = 01011000b is 0)

CF = 0 (because |82h| > |2ah| the problem of a subtraction with a borrow is not in question; so we won’t have an overflow in the unsigned interpretation which is performed:

130 – 42 = 88)

OF = 1 (the signed interpretation is performed, meaning ah-bh = -126 – 42 = -168

and since -168 ∉ [-128..127] a *signed overflow* is reported and as if follows, OF=1)

cmp bh,ah ;the fictional subtraction is performed: bh–ah = 00101010b-10000010b

= 10101000b

; = A8h = 168 (in unsigned interpretation) = -88 (in signed

interpretation)

This subtraction sets the flags as it follows:

SF = 1 (because the sign bit for the result A8h = 10101000b is 1)

CF = 1 (because |2ah| < |82h| the subtraction with a borrow digit is in question; in unsigned interpretation, the subtraction becomes 42 – 130 = 168 (!) which actually comes from the need of a subtraction of the type (256 + 42) – 130 = 168 and as a result of the need of a borrow, an overflow will be reported regarding the unsigned overflow, which is interpreted here as “this subtraction can’t be performed correctly without using a borrow digit”)

OF = 1 (the subtraction is performed in signed interpretation, so bh-ah = 42-(-126) = +168 and since +168 ∉ [-128..127] a *signed overflow* is reported and as a result OF=1)

**ii).** mov ah,126 ; equivalent with mov ah,7eh because 126 = 7Eh = 01111110b (the sign bit being 0 , the two interpretations coincide and as a result, the content of AH is 126 both in signed and unsigned interpretation)

mov bh,2ah ;2ah = 42 (both in signed and unsigned interpretation)

; = 00101010b (the sign bit being 0, the two interpretations coincide)

cmp ah,bh ;the fictional subtraction is performed:

ah-bh= 01111110b- 00101010b = 01010100b

; = 54h = 84 = 126 - 42 (both in signed and unsigned interpretation, because the sign bit of the result is 0)

This subtraction sets the flags as it follows:

SF = 0 (because the sign bit of the result 54h = 01010100b is 0)

CF = 0 (because |126| > |42| subtraction with a borrow digit is not in question, so there won’t be any overflow in the unsigned interpretation)

OF = 0 (the subtraction is performed in signed interpretation, which means ah-bh = 126 – 42 = 84 and since

84 ∈ [-128..127] does NOT report a *signed overflow* as a result OF=0)

cmp bh,ah ;the fictional subtraction is performed: bh–ah = 00101010b-01111110b = 10101100b

; = 42-126 = ACh = 172 (in unsigned interpretation) = -84 (in signed interpretation)

This subtraction sets the flags as it follows:

SF = 1 (because the sign bit for the result ACh = 10101100b is1)

CF = 1 (because |42| < |126| subtraction with a borrow digit is in question; in unsigned interpretation, the subtraction becomes 42 – 126 = 172 (!) coming from the necessity of a subtraction of the type (256 + 42) – 126 = 172 and as a result of the necessity of the borrow, an overflow will be reported in unsigned interpretation by setting the carry flag)

OF = 0 (the subtraction is performed in signed interpretation, meaning bh-ah = 42-126 = -84 and

since -84 ∈ [-128..127], there is NO *signed overflow* and as a result OF=0)

As a general rule, from the point of view of the binary representation, if the result of the subtraction a-b ∈ [-127..127] then b-a ∈ [-127..127] (the particular situation in which a-b = -128 will be discussed beneath). Similarly, for word representations in the interval

[-32767..32767] with regard to the particular case -32768. As a result, we can conclude that the instructions **cmp a,b** and **cmp b,a** will always provide the same values for OF.

**iii). - Discussion regarding the cases cmp 80h,0 and cmp 0,80h**

mov ah,80h ;80h = 128 (unsigned interpretation) = -128 (signed interpretation)

; = 10000000b (the sign bit being 1, the two interpretations differ)

mov bh,0 ;bh:=0

cmp ah,bh ;the fictional subtraction is performed: ah-bh= 10000000b-00000000b = 10000000b

; = 80h = 128 (unsigned interpretation) = -128 (signed interpretation)

This subtraction sets the flags as it follows:

SF = 1 (because the sign bit of the result 80h = 10000000b is 1)

CF = 0 (since |80h| > |0|, subtraction with a borrow digit is not in question, so we can’t talk about an overflow in unsigned interpretation)

OF = 0 (the subtraction in signed interpretation is performed, meaning ah-bh = -128 – 0 = -128 and

since -128 ∈ [-128..127] does NOT report a *signed overflow*, as a result OF=0)

cmp bh,ah ; the fictional subtraction is performed: bh-ah= 00000000b-10000000b = 10000000b

; = 80h = 128 (unsigned interpretation) = -128 (signed interpretation)

This subtraction sets the flags as it follows:

SF = 1 (because the sign bit for the result 80h = 10000000b is1)

CF = 1 (because |0h| < |80h|, subtraction with a borrow digit is not in question, in unsigned interpretation the subtraction becomes 0 – 128 = 128 (!), which comes from the necessity of subtraction like (256 + 0) – 128 = 128 and as a result of the necessity of a borrow, an overflow in unsigned interpretation will be reported by setting the carry flag)

OF = 1 (the subtraction is signed interpretation is performed, meaning bh-ah = 0 – (-128) = +128 and

since+128 ∉ [-128..127] , a *signed overflow* is reported and as a result,

OF=1)

CF = 1 in the case cmp 0,80h because a subtraction with a borrow digit is performed:

0 - 10000000b = **1** 00000000 –

10000000

**0**10000000

and the borrow digit is transferred in CF.

Let’s analyze in this context what it means and how we got to the domain of the ”signed numbers that can be represented on 1 byte” , respectively the domain of the ”signed numbers that can be represented on one word”.

On 1 byte, we can represent 256 values, whether we are talking about the signed or unsigned interpretation. In unsigned interpretation, these values are the ones in the interval [0..255]. Which are these 256 values representable in signed interpretation? Are we talking about the interval [-128..127] or about the interval [-127..128] ? Because it couldn’t possibly be about the interval [-128..128], because in this interval we have 257 values! In other words, somebody had to choose one of the two possibilities and at the same time, to clarify that the numbers -128 and +128 can’t coexist between the limits of the same interval of representation at the same time! (we remind you that in assembly language *data type* = *dimension of representation*)

This way we can also see the impact of this representation style on high level programming languages: for example, both **shortint** and **byte** in Turbo Pascal accept the value 80h (-128 as *shortint* and +128 as *byte*), but 80h **can’t have two distinct interpretations in the same data type!** We won’t ever see the values -128 and +128 being represented in the same data type when it comes to any high level programming language.

As a result, the decision that the accepted interval of signed values representable on 1 byte should be the interval [-128...+127] has been taken (which is exactly the domain of the values and of **shortint** data type from Turbo Pascal): so **+128 is not accepted as a signed value representable on 1 byte!**

However, as we can very easily identify, the instructions **mov ah, 128** and **mov ah,-128** are both accepted by the assembler, the effect in both cases being loading in *ah* the binary configuration 10000000b ! This is because in the first case, we are talking about unsigned interpretation for 80h and in the second case we are talking about interpretation. Loading a register with a certain binary configuration does not imply the necessity of interpreting the said configuration in a specific way. The target of that configuration, signed or unsigned, will be carried on by the following instructions which will use these values as operands. For instance, the usage of IMUL instead of MUL will result in the interpretation of the said binary configuration as a signed operand, instead of an unsigned one. Similarly, the usage of DIV instead of IDIV will result in the interpretation of the same operand as being unsigned, and so on.

In the case cmp 80h,0 what is performed is 80h–0 = 80h = 10000000b (128 - 0 = 128 in unsigned interpretation), without the need of a transport digit borrowed for performing the subtraction, so we don’t have an overflow in the unsigned interpretation, so CF = 0. In signed interpretation of the operands and of the final result, we get -128 - 0 = -128 ∈ [-128..127] so there is no overflow in the signed interpretation either, meaning OF = 0.

On the other side, we obviously have, in both cases, SF=1. *Intuitive* justification: in signed interpretation , the value 10000000b represents a strictly negative number, meaning -128. *Technical* justification: the sign bit of the binary representation 10000000b is 1, so SF=1.

**iv).** Let’s analyze now the ways in which we can compare the values 0 and 1 (and then 0 and -1) and what effects does the cmp instruction have among the flags in each situation.

The **cmp 1,0** situation (highlighted at a source text level, for instance cmp ah,0 with ah=1) will perform the fictional subtraction 1-0 = 1 = 00000001b. The effect over the flags will be CF = SF = OF = ZF = PF = AF = 0. The reasoning is obvious, based on the discussion from the previous examples.

The **cmp 0,1** situation (highlighted at a source text level, for instance cmp ah,1 with ah=0) will perform the fictional subtraction 0-1 = -1 = 11111111b:

0 - 00000001b = 1 00000000 –

00000001

0 11111111

The effect over the flags will be CF = SF = PF = AF = 1 şi ZF = OF = 0. The justification of the values of CF and SF is also highlighted here, based on the previously discussed examples and OF=0 because the result in signed interpretation is -1, and -1 ∈ [-128..127].

The **cmp -1,0** situation (highlighted at source text level, for instance cmp ah,0 with ah = -1) will perform the fictional subtraction -1- 0 = -1 = 11111111b. The effect over the flags will be SF = PF = 1 şi CF = OF = ZF = AF = 0. SF=1 because the sign bit is 1. OF=0 because the result in signed interpretation is -1, and -1 ∈ [-128..127]. CF=0 because performing a subtraction with a borrow in not imposed.

The **cmp 0,-1** situation (highlighted at source text level for instance cmp ah,-1 with ah = 0) will perform the fictional subtraction 0 – (-1) = +1 = 00000001b:

0 - 11111111b = 1 00000000 –

11111111

0 00000001

The effect over the flags will be: CF = AF = 1 şi OF = SF = ZF = PF = 0. SF = 0 because the sign bit is 0. OF=0 because 0 – (-1) = +1 ∈ [-128..127]. CF = 1 because performing a subtraction with a borrow is imposed. We can also justify like this: in unsigned interpretation, this subtraction actually means 0 - 255 = 1 (!), which should be justified by (256+0) – 255 = 1, so we need a borrow digit ad so, an overflow is reported in the case of the unsigned interpretation , so CF = 1.

**v).** The previously studies cases (i-iv) were referring to subtraction operations, due to the analysis that we had regarding the effects of the cmp instruction. Let’s analyze the case of an overflow caused by addition, coming back to the discussion about applying the RAO rule:

mov ah,126 ;126 = 01111110b = 7eh (the same value 126 in both interpretations)

add ah, 2 ; 2 = 2h = 00000010b ; AH := 01111110b + 00000010b = 7eh + 02h =

; 10000000b = 80h (= 128 unsigned = -128 signed)

CF = 0 because: 01111110 +

00000010

10000000 - there is no transport outside the representation space of the result

SF = 1 because the sign bit of the result is 1 (in signed interpretation, the result of the operation is strictly negative = -128).

OF = 1 because:

- *technical* justification: according to ROA, we add the two numbers of the same sign (the sign bit is 0 for both of them) and the result is of a different sign (the sign bit is 1).

- *intuitive* justification: we add the two unsigned numbers, whose sum is 126 + 2 = 128. But the number +128 ∉ [-128..127], so there is a *signed overflow* and as a result OF=1.

**vi).** One of the surprising effects of both signed and unsigned interpretations refer to the situation in which the programmer initializes the operands with certain desired initial values (signed or unsigned, according to the necessities of the said problem) and expects the certain results or reactions, according to the given values. Attention! Usually, these values have two possible interpretations and will not be interpreted in any situation under the given form at initialization!

The subsequent use of instructions that force by their way of action, the complementary interpretation (signed/unsigned) of the initialized one can cause the appearance of certain situations in which a user, at first sight, either suspects errors from the assembly, or by representing the result in base 10, expects a hilarious interpretation. This happens if the double possible interpretation of binary manipulated configurations is not always taken into account. Let’s take an example:

mov al, 200 ; al = 11001000b = 0C8h = 200 (fără semn) = -56 (signed)

mov bl, -1 ; bl = 11111111b = 0FFh = 255 (fără semn) = -1 (signed)

cmp al, bl ; al-bl = 11001001b = C9h = -55 (signed) = 201 (unsigned)

(the flags are set accordingly OF=ZF=0 şi CF=SF=1)

What do we actually compare here? 200 with -1, as we specify the values from initialization?

Or maybe 200 with 255? Or -56 with -1 ? Or -56 with 255?

Answer: we compare 0C8h with 0FFh or in binary representation, 11001000 with 11111111. The effect can only be this: affecting the corresponding flags as a result of performing the fictional subtraction AL-BL. The way of expressing the comparison performed in base 10 correctly, is not deducted from the action of the CMP instruction (which does not distinguish at all the 4 possible variants from above), but based on some instruction that will follow, which will have the purpose of interpreting the performed comparison in one of the 4 ways from above. Let’s follow the variants of comparison identified below, by using the corresponding instructions of conditional jump:

jl et1 ; it is obvious that 200≮-1, so at first sight, the necessary condition for performing the jump is not met ... let’s not forget that JL (Jump If Less) interprets the comparison’s result as signed (so -55), this implying that the subtraction is also interpreted as

(-56 – (-1)) so both of the operands will be interpreted as signed... since -56 < -1, intuitively, the condition is verified (besides the technical justification of meeting the condition of the jump SF≠OF) so the jump will take place! So even though the programmer provided the values 200 and -1 at initailization, the use of the JL instruction lead to the interpretation of the comparison being between -55 and -1 and not between 200 and -1! (this explanation and the fact that the jump will be performed can help you ”prove” some colleagues how 200 can be smaller than -1!!!)

ja et2 ; because 200 > -1, in this case we would expect the jump to be performed.... but the use of the JA instruction (Jump If Above) implies unsigned interpretation, so the case of a correct comparison here is 200 with 255 and since 200 ≯ 255, the condition on not met and so the jump will not be performed (this is how we can prove that 200 is not superior to the value -1!!!). As confirmation, we can see that not even the condition imposed by JA is not met: we should have CF=ZF=0, but in our case CF=1, so the jump will not be performed.

jb et3 ; intuitively 200 < 255, and technically CF=1 so the jump is performed

jg et4 ; intuitively -56 ≯ -1, and technically even though ZF=0 is not met and the condition SF = OF so the jump will not be performed

As a result, in the 4 theoretically possible situations from above, we will actually find only 2:

* Unsigned comparison (200 with 255) - imposed by ”above” or ”below”
* Signed comparison (-56 with -1) - imposed by ”less than” or ”greater than”

Hence, we can’t compare 200 with -1 as the values were specified at initialization and neither -56 with 255, because **the interpretation is either signed, either unsigned for both operands.**

**vii).** We studied in the examples from above the reaction (the interpretation) of the 80x86 processor regarding the overflow notion in the case of addition and subtraction. When and how do the processors from the 80x86 family report the overflow in multiplication and division?

**”The overflow” in multiplication.** The MUL and IMUL instructions set CF=1 and OF=1 if the superior ”half” of the product (the superior byte if we are talking about a word-product of the superior word if we are talking about a doubleword-product) is a value different from 0. This is the definition of the ”overflow in multiplication” concept in the case of the 80x86 architecture. Let’s remark the fact that there is no distinction between MUL and IMUL and neither between CF and OF. Either both flags will be set to 1, signaling a ”multiplication overflow” as mentioned above, or both will be 0. The following example is on 8 bits:

mov al, 5  
mov bl,170  
mul bl     ;AX := AL \* BL = 5 \* 170 = 850 = 0352h and we will get CF=1

and OF=1 because the superior byte AH = 03 ≠ 0.

The IMUL variant will provide:

mov al, 5  
mov bl,170 ;170 = 0aah = -86 in signed interpretation   
imul bl     ;AX := AL \* BL = 5 \* (-86) = - 430 = 0fe52h and we’ll get CF=1

;OF=1 because the superior byte AH = 0feh ≠ 0.

In the case of some operations on 16 bits, we can have for example:

val1 DW 2000h  
val2 DW 0100h  
…  
mov ax, val1  
mul val2     ;DX:AX = 00200000h we’ll have CF=1 and OF=1 because the upper half of the product DX:AX, meaning the DX register contains 0020h ≠ 0.

These settings should be interpreted as errors. In no way is it about a potential loss of information as in the case of the other overflows (addition, subtraction or division). This is because even if we multiply maximal values that can be represented on the operand’s dimension (255 \* 255 for bytes and 65535 \* 65535 for words), we still don’t go over the double of the representation dimension of the operands, meaning the space that we have either way, by definition, because 255 \* 255 = 65025 < 65535 (the maximum unsigned number representable on a word) and 65535 \* 65535 = 4 294 836 225 < 4 294 967 295 (the maximum unsigned number representable on a doubleword).

In the case of signed multiplication (the IMUL instruction), the justification is similar: 127 \* 127 = 16129 < 32767 (the maximum signed number that can be represented on 1 word), and 32767 \* 32767 = 1 073 676 289 < 2 147 483 647 (the maximum signed number representable on a doubleword).

The overflow in the case of multiplication in the 80x86 assembly language is just a report of the fact that beginning with byte operands (or words) the product does not fit just in a byte (or a word), but a double dimension is rather needed for memorizing the result. In this way, see chapter 1, in which mathematically speaking, it was specified clearly that multiplication does not cause an overflow, precisely because of the allocation of a sufficient space for representing the product. In conclusion, we can say that mathematically speaking, the only operation that does not cause an overflow is the multiplication, but the 80x86 processors promote the notion of ”multiplication overflow” for differentiating the situations in which the products fits in a space of the dimension of the operands and in which they don’t.

The situations in which the product fits in the dimension of the operands will be characterized by the settings CF = OF = 0 (we don’t have a multiplication overflow). Here is an example:

mov al, 5  
mov bl, 51  
mul bl     ; AX := AL \* BL = 5 \* 51 = 255 = 00ffh and we’ll have CF=0 and

OF=0 because the superior byte AH = 0.

**Overflow in division:** In the of division, the specification of the operation in the form

(I)DIV *operand*

means that the specified operand is the divisor (it’s possible to be represented on either 8 or 16 bits) and the dividend is considered implicitly in AX (if the *operand* is a byte) or in DX:AX (if the *divisor* is a word). Performing this operation has the following effect:

AX / operand on 8 bits = quotient in AL şi remainder in AH;

DX:AX / operand on 16 bits = quotient in AX and remainder in DX;

In the case of division, and overflow appears when the result of the division does not fit in the reserved space, according to the representation definition, which is when the quotient does not fit in AL or AX. In such a situation, the 80x86 processor issues a 0 interruption, the execution ending with a message issued by the routine of treating the 0 interruption, of the sort ”Divide by zero”, ”Zero divide” or ”Divide overflow” (depending on the processor type and/or the installed OS). It seems weird at first sight that a divisor by 0 (of type *div bh* with *bh=0*), which practically can’t be performed mathematically speaking, is treated similarly from the point of view of the assembly language as a division that mathematically can be performed. The sequence

mov ax,60000

mov bl,2

div bl

should issue, mathematically speaking, the quotient 30000. But according to the definition of division DIV, this quotient has to be memorized in the AL register, of byte size. Since the greatest representable value on 1 byte is 255, it is obvious that from the point of view of the assembly language it can’t be performed (similarly with the situation of type div 0) and as a result we understand now the decision of the architects of treating everything by issuing a situation similar to the one above. Let’s remark that in this way and the fact that the message ”*Divide overflow*” is accepted in this context as similar to ”*Divide by zero*”.

**viii)**. One of the frequent logical errors that inexperienced programmers make is mistaking the expressions ”*signed numbers*” and ”*unsigned numbers*” with the expressions ”*negative numbers*” and ”*positive numbers*”. The fact that a number is signed does not automatically mean that it is negative! Signed numbers can be both negative and positive. Unsigned numbers are always positive.

What conclusions can we draw, relatively to the way of interpretation (signed or unsigned) from the statement of a problem that asks that the execution of a certain action ”*if the number v (strictly) negative*” is asked? First of all, we will conclude that we are talking about signed interpretation. We can ask the question: how will we test if the signed number is negative or not? (let’s suppose that *v* is a byte). Because we are talking about signed interpretation, if the first bit of the configuration is 1, then the number is negative. So everything is reduced to a test over the first bit of the representation of the number. The following are 2 alternative for executing such a test:

a). We execute a movement of the first bit in CF and we test its value through an adequate instruction of conditional jump. The sequence

mov al,v ; for not affecting the content of variable v

shl al,1 ; shift left with 1 position => first bit goes in CF

jc este\_negativ ; if CF=1 then jump to the este\_negativ label

assures the testing that tells if the variable *v* is a negative number or not

b). We use the **cmp** instruction for a comparison related to 0:

cmp v,0 ; fictional subtraction v-0

jl este\_negativ ; if v<0 then jump to the este\_negativ label

or alternatively

cmp 0,v ; fictional subtraction 0-v

jg este\_negativ ; if 0>v then jump to the este\_negativ label

**ix).** We saw that at the level of performing additions and subtractions, the 80x86 processor does not make a difference between signed or unsigned additions/subtractions (technically speaking, they are both performed as binary operations with a result that can be interpreted after as signed or unsigned). However, in the moment when it comes to interpreting the result of an addition or subtraction in base 10, the following question comes: how do we represent, correctly from a semantical point of view, the operands of the said operations so that these expressions are consistent with the interpretation of the final result? Concretely:

00000101 + (= 5 in both interpretations)

11111110 (= 254 in unsigned interpretation and -2 in signed interpretation)

(1) 00000011 (= 3 in both interpretations of the 8 bits configuration)

represents 5 + 254 = 259 ( = 1 00000011 – configuration on 9 bits !) or represents 5 + (-2) = 3 ? As we will see, the answer here is that we can interpret it in both ways and justify in this way two separate ways of setting the CF and OF.

Because of the transport digit, we will have CF=1 (independently of the interpretation of the operands or of the final result as signed or unsigned, because we are talking about a technical consequence of the way of performing a binary operation of addition). As a result, in unsigned interpretation we have an overflow (evidently, because 259 > 255, so greater than the maximum number representable on 1 byte).

What happens with OF? Running the sequence

mov al, 5 ; = 5 in both interpretations

mov bl, 254 ; = -2 in signed interpretation

add al, bl ; AL := AL+BL = 5+(-2) = 3

does not set the OF flag to 1, so in the situation from above there is no ”overflow” in signed interpretation! From the point of view of justifying the way of setting the OF, the sequence from above would be more correct if written as:

mov al, 5

mov bl, -2

add al, bl ; so 5 + (-2) = 3

and it is obvious that in this interpretation we are not talking about an overflow (which is why OF=0).

Let’s remind ourselves in this context the examples given in the RAO and RSO: the addition 100 + 50 = 150 will report a signed overflow, according to RAO and the subtraction 130 - 42 (interpreted as -126 -42 = -168 ∉ [-128..127]) şi 42 -130 (interpreted as the subtraction 42-(-126) = +168 ∉ [-128..127]) cause *signed overflow* and as a result, OF=1.