ASC Lecture 4 → 2′s complement and overflow

Notes	lecture notes 4, 8

2′s complement

Mathematically, the **2′s complement** *representation* of a negative number is $2^n - V$, where V is the absolute value of the represented number

How to compute it?

1. $2^n - v$
2. reverse all bits and add 1
3. leave unchanged the bits until the first bit 1, including it and *reverse* all the rest
4. if you want only the value in base 10, **the sum of a number and it′s complement in absolute value is the cardinal of the values representable on that size** (e.g. on 8 bits → 2^8)

2′s complement is a translation mechanism, helps us understand numbers in binary/hex

The discussion makes sense ONLY when we care about negative numbers (signed interpretation)

→ in base 10

On 8 bits we have the possibility to represent 2^8 valies:

unsigned byte → [0, 255]

signed byte → [-128, 127]

⇒ $[0, 255] \cap [-128, 127] = [0, 127]$ every number from here has the same signed / unsigned *interpretation* (they start with a 0)

$-147 \notin [0, 255] \cup [-128, 127]$ ⇒ **it doesn′t fit a byte**

Number X in binary representation begins with	-X begins with	-X is represented on	Examples:
0	1	Same sizeof as X	109 = 01101101 ; -109 = 10010011
1	1	2 * sizeof(X)	147 = 10010011; -147 = 11111111 01101101

(exceptions are the representations of the form $\overline{100\ldots}$ (-128, +128, −32768, +32768 *etc*)

Q: what is the minimum number of bits on which we can represent -147?

→ On n bits in the general case we can represent 2^n-1 values, either signed [-2^n-1, 2^{n-1} - 1] or unsigned [0, 2^{n} - 1]

⇒ on 9 bits we have [0, 2^9] or $-147 \in [-2^8, 2^8 - 1]$ ⇒ **the minimum number of bits needed for representing -147 is 9**

Q: not sure, vezi lecture 4.2 sfarsit, somewhere around 38′

Overflow

> At the level of .asm an overflow is a situation or condition which expresses the fact that **the result of the last performed operation didn't fit the reserved space for it** OR **it does not belong to that admissible representation (sign bit)** OR **is a mathematical nonsense in that particular operation**

1001 0011 +	147 +	93h +	-109 +
1011 0011	179	B3h	- 77
1 0100 0110	**326**	1 46h	**- 186** !!!!
(binary representation)	**(unsigned interpr.)**	**(hexa repr.)**	**(signed interpretation)**

!! 147 + 179 = 326 BUT **CF = 1** expressing that 326 cannot fit a byte (can be obtained on 9 bits, but NOT on 8)

💡 **byte + byte → BYTE**

These calculations are *mathematically* correct in base 10, but not in ASSEMBLY LANGUAGE, where add byte, byte IS ALWAYS a BYTE

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**

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**

> Obtains an impossible, mathematical incorrect conclusion → **TWO POSITIVE NUMBERS CANNOT RESULT A NEGATIVE ONE**

Where does overflow occur?

ADD

0..... + 0...... = **1.......**

1...... + 1...... = **0.......**

these two situations denote impossible mathematical results

SUB

1..... - 0...... = **0.......**

0...... - 1...... = **1.......**

e.g. 1 0110 0010 - 1100 1000 = **1001 1010** which would mean 98 - 200 = 154 in unsigned interpretation and 98 - (-56) = -102 in signed interpretation ← **none of the is correct, therefore CF = OF = 1**

"we need a borrow and we don't have it"

MUL

! **the only operation that will never trigger *only* OF**

if b*b:

1. byte → **CF = OF = 0** ⇒ **it shows that it can be restrained to a byte ("no overflow")**
2. word → **CF = OF = 1** ("**overflow**")

for multiplication, CF and OF are always the same

DIV

*the worst case of all

rule: w/b = byte

e.g. 1002 / 3 = 334 **BUT 344 doesn't fit a byte**

💡 The 2 flags are set exactly for letting you intervene with a correction **(ADC, SBB, CBW etc)**, but **FOR DIVISION IT IS FATAL ⇒ runtime error, the OS stops the running of the program (adică programul crapă)**

Possible errors:

→ "Zero divide"

→ "division overflow"

→ "Division by zero"

! they are equivalent

Q: Which of the following operations will set OF and CF to different values?

→ if there's any **mul** you can eliminate them from the start

Q: Do we **really** need both flags?

→ an addition or subtraction in base 2 in asm what happens is in fact that we have **2 simultaneously different operations in base 10**

⇒ **this is why CF and OF are necessary to be together present and associated with this operation**

Q: How do you tell the processor to perform an addition in the unsigned interpretation?

→ you **cannot**! it will always have 2 possible interpretations

Q: Why isn't there IADD and ISUB?

→ addition and subtraction works the same in base 2, **independently on the interpretations in base 19** ⇒ they function identically for both interpretations simultaneously

→ the analysis (interpretation) can be done afterwards

Q: What values will be associated with CF and OF in case of DIVISION?

→ YOU CANNOT CHECK

→ if you have **division overflow** and the program crashes, nobody cares about these values and they cannot be checked

→ if you don't have division overflow and everything goes well, it still means that **they have no use**

→ **you don't need these flags to tell you that the division went well, instead THE FACT THAT THE PROGRAM DIDN'T CRASH IS ENOUGH**

official answer: **in case of division CF and OF are UNDEFINED**

Q: Why do we need IMUL and IDIV?

→ in contrast from addition and subtraction which function the same independently of the interpretation, **multiplication and division function differently,** they have to be told **before the operation** what should be the interpretation of the operands

Q: Why do we have ADC but not “add with overflow”?

→ both CF and OF are set to 1 at the same time