- ~~negative numbers 23 Jan 2019~~ + 29 Jan 2024
- ~~call code, entry code etc 24 Jan 2023 + 8 Feb 2018~~ + 30 Jan 2024
- addresses 31 Jan 2020 + 22 Jan 2024 (& some machine instructions)
- ~~overflow 9 Feb 2018 + Model 1~~
- ~~datatypes 14 Feb 2022~~
- ~~flags 18 Feb 2019~~
- basic elements Model 2

Negative numbers – 23 January 2019

- Explain and exemplify how negative numbers are represented and operated at the level of the 80x86 architecture. Detail the representation mechanism and give 2 adequate examples. How can we obtain the corresponding signed value in base 10 (interpretation) for a given representation in base 2? Which are the possible methods to be applied?

Mathematically, the value $2^n - V$, where V is the absolute value of the represented number, is the representation of a negative number (2's complement).

Let us take $-\overline{abc}$ as a number. $\overline{abc}$ is simply the binary representation of the value, but for $-\overline{abc}$ we must use 2's complement.

There are a few methods used for getting the 2's complement of a number:

1) $2^n - V$: for example $136 = 10001000b = 88h$

the fact that the sign bit is 1 means that we have 2 interpretations for this representation: signed and unsigned

what is the signed interpretation? $-$(2's complement of 1000 1000)

n is the number of bits +1 e.g. here is 9

$$\begin{array}{r} 100000000 - \\ 10001000 \\ \hline 01111000 \end{array}$$

7 8 $\longrightarrow$ $01111000b = 78h = 120$

$\Rightarrow$ the sign interpretation of 10001000 is -120.

$-120 \in [-128, 127]$ signed interp. on 8 bits

but $-376 \notin [-128, 127)$, therefore we must represent it on 16 bits ($\in [-2^{15}, 2^{15}-1]$)

$2^{16} - 376 = 65160 = 1111111010001000b = FE88h$

$$\begin{array}{r} 10000\ 0000\ 0000\ 0000 - \\ 1\ 0111\ 1000 \\ \hline 1111\ 1110\ 1000\ 1000 \end{array}$$

2) invert all bits and add 1

$10001000 \longrightarrow 0111\ 0111 + 1 = 0111\ 1000$

3) keep the bits from righ to left, including the first 1 and invert all the other ones: $1000\ 1000 \longrightarrow 0111\ 1000$

(stays the same)

* to verify that our interpretation is correct, the sum of the absolute values of the 2 complementary numbers should always be the cardinal of the set of values representable on that size: $256 - 136 = 120 \Rightarrow -\overline{abc} = -120$

↳ this is also the quickest way to find the interpretation of a number, if that what interests us

- Present how the programmer operates with negative numbers at the level of asm. in the case of each of the 4 basic arithmetic operations (1 source code each, explained and justified)

1) Addition – ADD and Subtraction – SUB

↳ regardless of the interpretation, the addition at the level of asm is done the same way for both signed and unsigned numbers (this is the reason why we don't have iADD or iSUB)

```
mov al, 3      ; AL = 0000 0011
mov bl, -2     ; BL = 1111 1110
add al, bl     ; AL = 0000 0001   also CF=1
                  = 1 => correct result
```

```
mov al, -4     ; AL = 1111 1001
mov bl, 3      ; BL = 0000 0011
add al, bl     ; AL = 1111 1100 => < 252 unsigned
                                    -4 signed => correct result
```

```
mov al, -7     ; AL = 1111 1001
mov bl, -8       BL = 1111 1000
sub al, bl       AL = 0000 0001 = 1 correct
```

```
mov al, -7     ; AL = 1111 1001
mov bl, 3      ; BL = 0000 0011
sub al, bl       AL = 1111 0110 => < 246
                                    -10 correct
```

2) IMUL and IDIV

↳ unlike for addition and subtraction, multiplication and division work differently for signed and unsigned numbers, this is why we need to specify BEFORE the operation takes place the interpretation for the result

```
mov ax, 2      ; AX = 0002h
mov bx, -3     ; BX = FFFDh
imul bx        ; DX:AX = FFFF FFFAh
```

notice how DX (or the high part of the result, giving the sign) was NOT computed the same way, leaving the latter representation with only a positive interpretation (SF=0), which is false

whereas:

```
mov ax, 2      ; AX = 0002h
mov bx, -3     ; BX = FFFDh
mul bx         ; DX:AX = 0001 FFFAh
```

* same goes for IDIV

```
mov ax, -8     ; AX = FFF8h
mov bl, 2      ; BL = 02h
idiv bl        ; AX / BL → AL = FCh = -4 signed
```

- Present and exemplify every asm. instruction (or category of instructions if they are similar) capable to operate with such numbers.

  The operations presented before, (i.e. ADD, iMUL, SUB, iDIV) are obviously capable of operating negative numbers. Another category is the conversion instructions. such as CBW, CWD, CWDE since they work with extending the sign byte.

```
mov al, -3   ; AL = FD
cbw          ; AX = FFFD
```

? every instruction, pretty much, works with negative numbers, since we talk about interpretation

- How is the value 0 considered and why?

  0 is considered a positive value, since the sign bit is 0 ⇒ it is NOT a negative number, therefore it is positive :)

Call code, entry code and exit code - 27 ian 2023

- Define and explain the concept of call code, entry code and exit code in the general context of programming languages, after which analize how they are reflected at the level of linking a C module with a x86 asm. module. Explain the ? task of each stage, giving source code adequate schematic examples to clarify the involved responsabilities and needs. Who is responsible for generating these codes and when exactly? Why is needed the asm. language involvement in working with these concepts? Describe the STDCALL and CDECL call conversions, explain where are they used. Present and explain the communication mechanism between the modules involved in a multimodule ASM - ASM programming and ASM - C resp.

* definition → what's given → implication → conclusion

with the call code you "open" some parantheses that need to be closed with the exit code
↳ every action must be unrolled somewhere below

- In the general sense, call code, entry code and exit code are what part of what we call subroutine call implementation. Any call implementation, independently of the OS has something similar, that differ by the call conventions, but the steps remain the same.

Call code: ⟶ the caller's responsability

1) Saving volatile resources (EAX, ECX, EDX, eFlags)
   ↳ the volatile resources are those registers that the calling convension defines as belonging to the call subroutine.
   ↳ this is why the caller, as part of the call code, is responsible to save their values and, after the subroutine is called, to restore the initial vals.

*) Extra step: Assure the compliance
   ↳ this means checking that the stack is alligned, DF=0 etc.

2) Passing parameters
   ↳ pushing them onto the stack, since it is a one of the shared fiscal resources, for them to be accesed later from another file.

3) Saving the return address and performing the actual call

ex:

```
extern exit, printf
import exit msvcrt.dll
import printf msvcrt.dll
global start
segment data use32 class="data"
    format db "%d", 10, 13, 0
```

```
segment code use 32 class = "code"
start:
    mov ecx, 10     ; ecx = 00 00 00 0Ah
    xor eax, eax    ; eax = 0

urmator:
    push eax        } saves the volatile resources
    push ecx
                    (*) stack is alligned
    push eax
    push dword format } pushing the params for a CDECL call
    call [printf]
    add esp, 4*2
    pop ecx         } restore the volatile resources
    pop eax

    inc eax
loop urmator
push dword 0
call [exit]
```

* the volatile resources need to be saved because printf uses them to return an error code

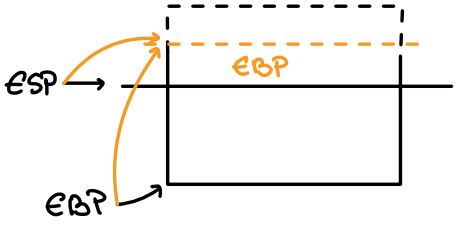
Entry code → caller's responsability

1) Configuring the stack frame

↳ stack frame = data structure stored in the stack, of fixed dimention, containing:
- params prepared by the caller
- return address
- copies of the non-volatile resources
- local variables

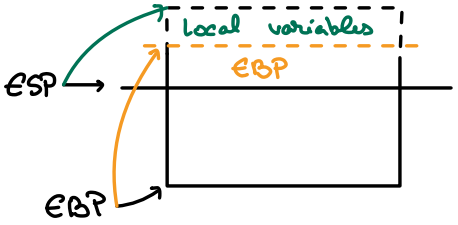
```
push EBP
mov EBP, ESP
```
} ! always at the beginning of the entry code

↳ here we construct a new empty stack frame



2) Allocating space for local variables

```
sub ESP, nr_bytes   ; the new formed stack gets bigger
```

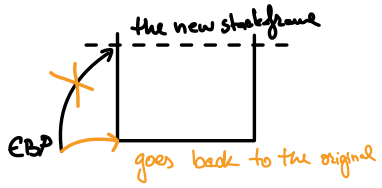


3) Saving the non-volatile resources (exposed to be modified)
↳ something that is modified, but is not volatile

Exit code - Callee

1) Restore altered non-volatile resources
2) Restore the local variables
3) Deallocate the stackframe
4) Return from the function and deallocate passed parameters

ex:
```
pop ebx      ; initial value
add esp, n   ; free the stack
pop ebp      ; restore the initial value of EBP
ret 4        ; returns to the initial stackframe
```



- When linking a C and .asm module we need ways and methods to communicate since they involve separate compilation we obtain different .obj files

Linkers check if everything was defined only once (declaration + allocation)

- The communication between modules happens through:
  - ↳ extern and global directives
  - ↳ registers and stack which are shared fixed resources } ASM + ASM
    - ↳ in C + asm

- CDECL
  - ↳ has a variable number of parameters
  - ↳ the caller is responsible for cleaning the stack
    (in asm everything is your responsability)
  - ↳ is specific to the C language
  - ↳ volatile resources EAX, ECX, EDX, EFlags
  - ↳ result stored: EAX, EDX:EAX, ST0

- STD CALL
  - ↳ used by windows
  - ↳ fixed numb. of params
  - ↳ callee is responsible for cleaning the stack

- Why use .asm for multimodule?
  Since .asm works with registers directly it is very fast for getting computations done, therefore it is usefull to have subroutines written in .asm

Overflow – 9 feb 2018 + Model 1

Elaborate an analysis regarding the "overflow" concept at the level of the arithmetic operations, focusing afterwards on how the x80 architecture reacts when overflow appears in the case of each of the 4 basic arithmetic operation (addition, subtraction, multiplication and division).

- Which are the cases considered by the .asm language "overflows" and how does the microprocessor react to them? Present for each of the 4 cases 2 adequate source code examples illustrating the appearance of such situations and the lack of them respectively
- Explain which is the exact reaction of the computing system for each presented case and what can the programmer do after such a situation arises (how can he take into account, use or interpret the issue of such situation by the processor) How can the programmer avoid overflow situations?

At the level of .asm language, an overflow is a situation or condition which expresses that the result of the last performed operation didn't fit the reserved space for it or it doesn't belong to that admissable representation (sign bit) or is a mathematical nonesense in that particular operation

1) Addition

↳ In the case of addition, since the two operands are of the same size, overflow occurs when result didn't fit the reserved space (add byte, byte = byte) or it is a mathematical nonesense

```
 1000 1001 +
 0111 1001
(1)0000 0010
```

```
 89h +
 79h
(1)02h
```

signed

```
-119 +
 121
-----
   2
```

unsigned

```
 137 +
 121
-----
 258
```

⇒ CF = 1 since a carry occured for the most significant digit

```
 1000 1001 +
 1010 1001
(1)0011 0010
```

```
 89h +
 A9h
(1)32h
```

```
-119 +
 -87
-----
-206
```

```
 137 +
 169
-----
 306
```

but 32h = 50 ≠ −206 + 306

OF = 1 since it is a mathematically incorrect conclusion: the sum of two negative numbers cannot be a positive one

2) Subtraction

↳ the concept here is "we need a borrow but don't have it"

```
 0110 0010 -
 1100 1000
-----------
 1001 1010
```

```
 98 -
 200
-----
 154
```

```
 98 -
 -56
-----
-102
```

CF = OF = 1

3) Multiplication

↳ this is the only operation that will never trigger only OF

↳ since the result of a multiplication is always stored on a register double the size of the operand, we won't have overflow in the real sense

↳ CF and OF here signal if the result can be restrained to the size of the operand

if b * b :

1) CF = OF = 0 → the result fits on a byte ("no overflow")

2) CF = OF = 1 ⇒ the result only fits on a word ("overflow")

```
mov al, 250
mov bl, 2
mul bl    ; here CF=OF=1 because 250*2=500 ∉ [0,255] ⇒ doesn't fit a byte
```

4) Division

↳ the rule for division is that a register double the size of the explicit operand is divided and the result is stored on the register the same size of the operand (word / byte - byte)

↳ here the two flags don't tell you anything because the fact that the program didn't crash is enough to tell you that the division went well

* the possible errors that can occur are "zero devide", "division overflow" a "division by zero"

```
mov ax, 4096   ; AX = 0FFFh
mov bl, 2      ; BL = 02h
div bl  ——————→ program crashes because 4096 : 2 = 2048 ∉ [0,256] ⇒ doesn't fit
                                                  ↓
                                                  al
```

- The two flags OF and CF are set exactly for this reason, to allow you to correct the mistake by using instructions such as ADC or SBB, which are the equivalent of ADD and SUB, but take into account the value stored in the flags:

```
xor ax, ax    ; clear the register
mov al, 250   ; AL = FAh
mov bl, 7     ; BL = 07h
add al, bl    ; AL = 00h  CF=1
adc ah, 0     ; AH = 01h ⇒ AX = 0100h ⇒ true result
```

↳ therefore to obtain the right answer the programmer must take into account CF and OF

Datatypes — 14 feb 2022

- Define the concept of datatype at the level of x86 .asm language. What are the define mechanisms of a datatype in .asm. Give examples of variable declarations (with and without initialization) for every datatype. What are situations in which the type operands are (not) necessary? Justify and exemplify with adequate examples and sugestive source code for each of the presented cases. Make a classification of the type conversions, presenting and exemplifying the methods of type conversions at the level of x86 asm.

At the level of x86 architecture, the memory can be accessed only by using the offset specification formula [base + index * scale + constant], without variable naming being associated to it.
(base: all; index: all except ESP; scale: 1,2,4,8)

The task of data definition directives is not to specify an associated data type, as we understand it in high level programming languages, but rather it is to generate the corresponding number of bytes for a named memory area (following the little endian representation).

In the general form, a data definition line looks like this:

1) [name] data_type expression_list
- [name]: label for data referral
- data_type: size of repres. — db, dw, dd, dq

ex:
```
a db 'a'
b dw 100 (64h)
c dw 1000h
d dd 3000h
```
memory area:
'a' | 64 00 | 00 10 | 00 30 00 00
a | b | c | d

2) [name] allocation_type factor
- allocation_type: resb, resw, resd, resq
- factor: how many times the allocation is repeted

3) [name] TIMES factor data_type expression_list
- TIMES: allows repeted ASM instruction & data definition

4) EQU - allows signing a numeric value during assembly time to a label without allocating any memory area

label EQU value

The type operands are not necessary when the size of that

instruction is clear. For example:

mov al, 5 — doesn't need to be „mov al, byte 5" since you can only fit a byte on AL

push [label] — on the other hand would raise an "operation size not specified" error

push dword [label] - would be syntactically correct

- Type conversions classifications

1) a) destructive : cbw, cwd, cdq, cwde, movzx, movsx

```
mov al, -1      ; AL=FFh
cbw             ; AX=FFFFh
```
| the content in AH is destroyed while extending AL

```
mov bl, 100     ; BL=64h
mov bh, 0       ; BX=0064h
```
| BH was overwritten, destroying the previous contents

b) non-destructive : byte, word, dword, qword

```
a db 5
mov AL, byte [a]    ; only AL was modified
```

2) a) signed cbw, cwd ...

```
mov AL, -1      ; AL=FFh
cbw             ; AX=FFFFh
```
the sign bit is extended

b) unsigned movzx ; mov ah, 0; mov dx, 0

```
mov AL, -1
movzx AX, AL
```
| AX = 00FF, extended with 0

3) a) by enlargement (1a) + 1b) - byte)

```
a db 10
mov AX, word [a]
```

b) by narrowing (byte, word, dword)

```
a dd 1000h
mov AL, byte [a] ; AL=00h
```

4) a) implicit

int → float since no information is lost

b) explicit

float → int

Flags - 18 Feb 2019

- Explain which is the role of the flags in the functioning of the x86 microprocessor. Present the flags, their classification and exemplify the influence of most important flags, giving .asm code examples (at least 4).
- Which are the instructions for directly accessing the flags? Which are the flags having the strongest connection with the flag values?
- How can we modify the configuration of EFLAGS if needed? Which are the flags reacting to an overflow, their role and rules (* done at overflow concept). Why is there more than one flag dealing with overflow?

A flag is an indicator on 1 bit. A configuration of EFLAGS shows an overview of the execution of each instruction. Some of them also dictate the way in which the execution of the next instruction is gonna go. EFLAGS register has 32 bits, but only 9 are used: OF, DF, CF, PF, ZF, IF, AF, TF, SF

CF - the "transport flag"
↳ set to 1 if the last performed operation had a digit outside of the representation domain, 0 otherwise
↳ signals unsigned overflow

```
mov al, 10001010b
mov bl, 01111001b
add al, bl
```

```
  1000 1010b +
  0111 1001b
-------------
(1)0000 0011
```

CF=1 ⇒ the operation was not performed well

PF - PF + # bits 1 from the lower byte of the LPO ⇒ odd numb.
↳ # bits 1 odd ⇒ PF=1
↳ used in data transmission to check if the information was correctly transmitted

AF - "carry" between nibbles
↳ shows that there's a transport digit between bits 3 and 4

```
  0110 1100 +
  0010 0110
-------------
  1001 0010
```

AF = 1

ZF → a truth value flag
↳ if LPO was 0 ⇒ ZF=0

```
mov al, 2
mov bl, 2
sub al, bl   ⇒ ZF = 1
```

SF → takes the value of the most significant bit
↳ SF=0 ⇒ positive numb, SF=1 ⇒ negative num.
↳ "Was the LPO a strictly negative numb?"

```
mov al, -2
mov bl, 10
imul bl      ; AX = AL * BL = 10·(-2) = -20  ⇒ SF = 1
```

TF → trap flag
↳ used in debugging, every time F7 is pressed, TF = 1

IF → interrupt flag
↳ when a critical section starts, IF = 0 ⇒ the code cannot be interrupted (rocket example)

DF → decides if the chuncks of data ("strings") are processed ascending or descending (DF = 0) (DF = 1)

OF → the equivalent of CF for signed interpretation

```
mov al, 10010011b
mov bl, 01110011b
add al, bl
```

```
10010011b +
01110011b
-----------
100000110
```

147 + 115 = 6 (false) CF = 1

-109 + 115 = 6 (true) OF = 0

Some flags show us what just happened (CF, OF, PF, ZF, AF, SF) and others dictate what'll happen next (TF, IF, DF, CF)

- Some of the instructions that directly access the flags are cond. jmp.
  ↳ at the level of the microprocessor, flags exist to offer the programmer the posibility to verify some conditions.
  - ja → CF = 0 ZF = 0
  - jz → ZF = 1
  - je → ZF = 1
  - jl → SF <> OF

  * also, instructions such as ADC or SBB, cwd, cbw, etc also work with the values of the flags.
  * instructions that set the flags: STC, STD, CLC, CLD, CMC
  * rcr, rcl, sal, sar
- The contents of EFLAGS can be changed using PUSHF / POPF