Moodle Enroll in course asc

! questions for exam

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, then OF will be set to 1 and will be set to 0 otherwise.

Which are the conditions in which overflow flag is set?

! Q: why do we have add with carry, but not add with overflow

Def! At the level of .asm an overflow is a situation / 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 admisable representation (sign bit) or math. nonesense in that particular operation

both CF and OF are set to 1 at the same time
the assembler gives us an incorrect result
on both interpretation

OF essentially tells us that the processor needs more space to perform that operation

$$\begin{array}{r} 93+ \\ 115 \\ \hline 198 \end{array}$$

→ signed interpretation gives us a negative number ⇒ OF is set

Addition (regula depăşirii la adunare) addition overflow rule

↳ if we add two negative numbers and get a positive number => mathematical nonesense

↳ same for two positive numbers and get a negative one

* only 2 situations for addition that set OF to 1

```
1 .... +        0 .... -
1 ....          0 ....
-------         -------
0 ....          1 ....
```

Subtraction

↳ subtracting from a positive numbe a negative number and get a negative one

↳ same for negative - positive = positive

```
0 .... -        1 .... -
1 ....          0 ....
-------         -------
1 ....          0 ....
```

Multiplication

m position number *
n position number
---
m+n positions ⟶ the only operation that will never trigger the OF

if b * b
- b (CF = OF = 0) ⇒ shows that it can be restrained to a byte ⇒ no "overflow"
- w (CF = OF = 1) "overflow"

* for multiplication CF and OF are <u>always</u> the same

! Trick Question
which of the 10 operations below will set CF and OF to dif. values?

Division

! which are the conditions at division for which OF and CF=1?

* must say before the operation the way you want it to be interpreted

↳ if the result doesn't fit the register the program STOPS.

fatal error: ex: 20000 (word) | 2 / 10000 (byte) → doesn't fit

↳ error message: { divide by 0 / 0 divide / division overflow

division by 0 ⟶ ∞ ⇒ overflow (μ / ε, $\varepsilon \to 0$)

* if there was a !ADD and !SUB they'd be the same but multiplication and division work diff.

CHAPTER 4 → ASSEMBLY LANGUAGE INSTRUCTIONS

General use transfer instructions

PUSH s has 2 operands, one implicit and one explicit

PUSH and POP transfer data to / from the stack AND update the value of ESP!

logical spaces { stack / heap / global ...

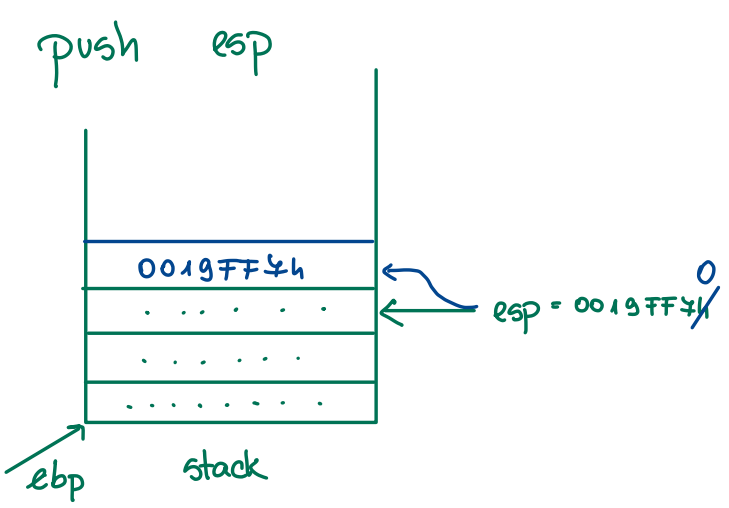
stack grows from big addresses to small addresses

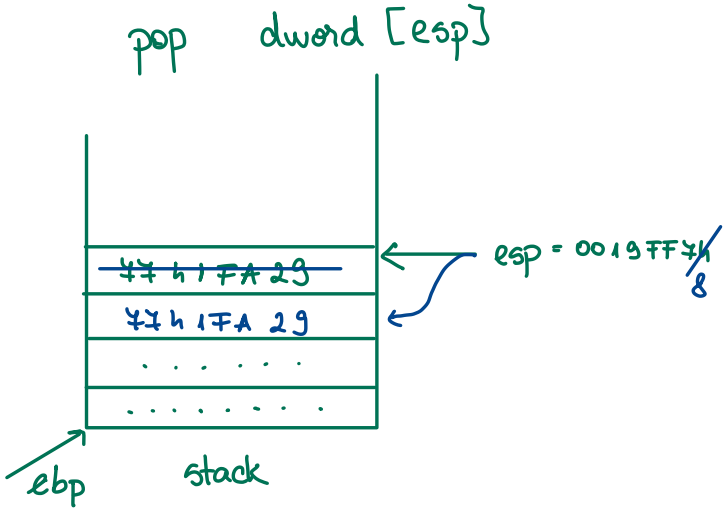
! pushing words to the stack raises a logical danger because ESP might not be a multiple of 4

↳ if you do, you must leave your place clean (a multiple of 4)

⇒ stack dealignment error

push eax ⟺ { sub esp, 4 ; allocate space to store the value / mov [esp], eax ; store the value





a) the operand is evaluated
b) ESP is updated accordingly
(-4 for PUSH and +4 for POP)
c) the assignment is performed

XCHG (exchange) → swaps the values from two entities (both can be registers, one register + one memory address)

[reg_segment] XLAT

↳ ex: you have 239 as int and want to get '239' as string
i + '0'

ex: 2AB37Fh → div 16
if remainder ∈ [0,9]
i + '0'
elif rem. ∈ [10,15]
i + 'A'

* you can define your own table like ascii (in ascii 0...9 and A..F are not cont.)

TabHexa db '0123456789ABCDEF'

AL ← DS:[EBX+AL]

mov ebx, TabHexa (must put your table into EBX)
mov al, number
Xlat → the byte AL will be overwritten by the value in your table (EBX+AL)

SS xlat ⟵ relative to SS
TabHexa from SS

at least one question will be from XLAT and LEA

LEA reg. dest, memory addressed op source (load effective address)

on 16 bits TASM

```
mov ax, v
mov ax, offset v
```

↕

```
LEA ax, v
```

equivalent of offset v

on 32 bits NASM

```
mov ax, [v]
mov eax, v      → transfer the offset of v
```

↕

```
lea eax, [v]
```

LEA takes the offset of .. ?

```
LEA eax, [ebx + esi*4 - 7]
```

↕

```
mov eax, ebx + esi*4-7
```

```
LEA eax, [ebx + esi*4 - 7]
```

↕

```
" mov eax, ebx + esi*4-7"
```

LEA gen_reg, contents of memory ⇒ gen_reg ⟵ offset (mem_operand)

Q: write one single expression that does .... some code
LEA solves that using offset specification formula

! How many ACTIVE data segments can we have at one?
2 ! because DS and ES are both considered data segments

```
PUSHF      (pushes EFlags to the stack)
POP eax
. . . . . .
PUSH eax
POPF
```

} put whatever to the EFlags