I. Flags ✓
II. Negative numbers ✓
III. Overflow ✓
IV. Addresses ✓
V. Basic elements of ASM + instructions
VI. Data types / Data def. directives ✓
VII. Machine instructions
VIII. Multimodule

I. Explain what are and which is the role of the flags:

A flag is an indicator represented on 1 bit. A configuration of the flags register shows a synthetic overview of the execution of each instruction. For X86 the EFLAGS register has 32 bits but only 9 are actually used: OF, DF, IF, TF, SF, ZF, AF, PF, CF

· Present the flags, the clasification and the influence:

→ CF 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. CF flags the unsigned overflow. CF signals if the LPO was not performed well.

ex:
```
mov al, 10001110b        1000 1110 +
mov bl, 01111000b        0111 1000
add al, bl              1| 0000 0110
```
the CF is set to 1

142+ / 120 / 262 > 256

-114+ / 120 / 6

→ ZF is the zero flag. Its value is set to 1 if the result of the LPO is equal to 0, 0 otherwise

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

→ SF is the sign flag, that is set to 1 of the result of the LPO is strictly negative, 0 otherwise

ex:
```
mov al, -2
mov bl, 5
imul bl         ⇒ AX = -10
```

```
mov al, -100 (9C)      100 | 64 | 36 | 32 | 4      0110 0100
mov bl, 10 (0A)                                    1001 1100
sub al, bl                                          9C -
    al = 92 ⇒ SF=1                                  0A
                                                    92
                                                    SF=1
```

→ OF is the overflow flag that flags the signed overflow. If the result of the LPO (in the signed interpr.) didn't fit the reserved space, or the result belongs to another interval of repr. or the result is, mathematically, incorrect

ex:
```
mov al, 0110 0000b   ; 96+
mov bl, 0100 0000b   ; 64
                     160 > 128
add al, bl
```
⇒ al = 1010 0000b
(1 is neg but we added 2 positive numbers)
⇒ OF=1

Also, the only operations that set the OF are addition and subtraction because at multiplication CF=OF and their value represents if the result can be represented on a smaller nr. Division does not set the flags bcs. if the result does not fit ⇒ FATAL ERROR

→ DF direction flag is used for operating string instructions. If set to 0, then the string parsing will be performed in an ascending order, otherwise in a descending order

→ IF interrupt flag can not be set directly on 32 bits programming. If set to 1, interrupts are allowed, if set to 0 interrupts will not be handled. Used in critical sections for stopping every other process except for the current one

→ TF trap flag is a debugging flag. If set to 1, the machine stops after every instruction

→ AF auxiliary flag shows the transport from bit 3 to bit 4 of the LPO's result

→ PF parity flag. Its value is set so that together with the nr of bits with value 1 from the least sign. byte of the result of the LPO is an odd number. It is used in data translation to check if the transmision was made in a correct way

Flags can be divided into 2 categ: flags that are set based on the result of the LPO (CF, OF, PF, AF, ZF, SF) and flags set by the programmer having a future effect on instructions that follow (CF, TF, IF, DF)

Instructions for setting the flags: for CF – cmc, stc, clc ; for DF – std, cld ; for IF – sti, cli (used only on 16 bits)

PUSHF / POPF  LAHF / SAHF  bits: 0,2,4,6,7

· Which are the instructions having a strong connection with the flags values?

ADC, SBB, conditional jumps (23 instructions ja, je, jz, jc), rcl, rcr, sal, sar

ja: CF=ZF=0 ; jz: ZF=1 ; jc: CF=1

· Why 2 flags for overflow?

Because OF → signed overflow  CF – unsigned overflow

FLAGS

Both signed and unsigned numbers in base 10 are represented at the level of the 80x86 architecture by their binary or hexa configuration. Base 2 and 16 are representation bases and base 10 is an interpretation base in which values can be interpreted as signed or unsigned

Starting from a negative decimal value such as -20, there are 3 methods that provide its binary configuration involving the 2's complement. Mathematically, the 2's complement representation of a negative number is $2^u - V$, where V is the absolute value of the number. First method represents the direct application of the definition:

-20 = 1110 1100b

20 = 0001 0100b ⇒ 1 0000 0000b − 0001 0100b = 1110 1100b

The second method represents switching all the bits of the repr. of the abs. value and add 1.

0001 0100b ⇝ 1110 1011b + 1 = 1110 1100b

The third method represents switching from right to left the bits not including the first bits of value 0 until the first bit of value 1

0001 0100b ⇝ 1110 1100b

On n bits we can represent $2^n$ values { $[0, 2^n - 1]$ unsigned values; $[-2^{n-1}, 2^{n-1} - 1]$ signed values

From a binary configuration we can get the signed value in base 10 by respecting the following steps: if the binary config has the structure $\overline{0xx\ldots}$, the signed decimal value is obtained by a classic conv.; if the binary config has the structure $\overline{1xx\ldots}$, then the signed value is −(2's complement of that binary config)

ex: 0010 0000b has the same value both in signed and unsigned interpretation

= 32

1001 0000b < 144 unsigned; −(0111 0000b) = −112 signed

For addition and subtraction it is not taken into account the value's interpretation, but its binary configuration, so these operations are performed the same as for unsigned nr.

```
ex: mov al, 90h          mov al, 144          mov al, -112
    mov bl, 10h    <=>   mov bl, 16    <=>    mov bl, 16
    add al, bl           add al, bl           add al, bl
```

---

```
mov al, 90h
mov bl, 10h
sub al, bl
```

For multiplication and division, there are separate instructions for signed an unsigned values.

```
for unsigned :  mov al, 10          mov ax, 10
                mov bl, 2           mov bl, 2
                mul bl ; AX=20      div bl ; AL=5  AH=0

for signed :    mov al, -10         mov ax, -10
                mov bl, 2           mov bl, 2
                imul bl             idiv bl
                    AX = -20            AL = -5  AH=0
```

NEG. NUMBERS

---

All arithmetic operations can be performed with negative numbers (add, sub, imul, idiv), conversions (cbw, cwd, cwde, cdq) and also conditional jumps (jl, jg, jge, jle)

```
ex: mov al, -2
    cbw ; AX = -2   FF FE

    mov bl, -2
    mov al, -5
    cmp bl, al
    jle skip ; the jump is taken
       add al, 3
    skip:
```

0 is considered a positive number (always starts with 0)

At the level of assembly language an overflow is a situation/condition which expresses the fact that the result of the ISO didn't fit the reserved space for it OR does not belong to the admissible representation interval for that size OR the operation is a mathematic nonsense in that particular interpretation.

At addition, the OF is set only in these 2 cases: we add 2 negative numbers and the result is a pos. one or we add 2 pos nr and the result is a neg. nr

ex: 1000 0000 + 1011 0000 = 1 0011 0000

-128 + -80 = 48 (signed) incorrect

0100 0111 + 0101 0000 = 1001 0111

75 + 80 = -105 (signed) incorrect

At subtraction, the OF is set only in these 2 cases: from a negative nr, we subtract a pos one and the result is pos or from a pos nr we subtract a neg one and the result is neg.

ex: 1000 1110 - 0100 1001 = 0100 0101

-114 - 73 = 69 (signed) incorrect

b: 0110 1100 - 1010 0000 = 1100 1100

108 - 96 = -52 (signed) incorrect

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 !

OVERFLOW

- Address of a memory location = nr of consecutive bytes from the beginning of the RAM memory and the beginning of that mem. location
- Memory segment = an uninterrupted sequence of memory locations, used for similar purposes during a program execution, a logical section of a program's memory, featured by it's basic addres, limit and type.
- Offset = the address of a location relative to the beg. of the segment / the nr of bytes between the beg. of the segment and that mem. location ex: a db 1 (offset 0 relative to SS) b db 2 (offset 1 relative to SS)
- FAR addres = the pair segm selector + offset, defines completely both the segment and the offset inside it COMPLETE ADDRESS SPEC.

ex: mov ax, [DS:X]

spec. methods: $s_3s_2s_1s_0$: offset_spec, where SSSS constant
segm_reg: offset_spec, where segm_reg can be DS, CS, SS, ES, FS, GS
FAR [var], where var is of type Qword and contains 6 bytes repr. the FAR address

- NEAR addres = only the offset is specified, the segment address being implicitly taken from a segment register; a near addres is always inside one of the 4 active segments
ex: mov eax, [v]
- Segmentation = memory management mechanism that divides the physical memory into variable-sized segments
ex: CS, DS, ES, SS
- Linear addres = (address computation) composed of base and offset ~32 bits
ex: $a_7a_6a_5a_4a_3a_2a_1a_0 = b_7b_6b_5b_4b_3b_2b_1b_0 + 0_70_60_50_40_30_20_10_0$

base = 1000h
offset = 2000h $\Rightarrow$ address = 1000h + 2000h = 3000h

- Flat memory model = linear addres with base 0
ex: $a_7a_6a_5a_4a_3a_2a_1a_0 = 00000000 + 0_70_60_50_40_30_20_10_0$
used by most of the modern OS. (Windows)

ADDRESS

- Physical effective addres = final result of segmentation plus paging eventually. The final addres obtained by the BIU points to physical memory (hardware) - at least 32 bits

offset_spec_formula = [base] + [index * scale] + [const]

- Direct addressing = only the constant is present ex: mov eax, [v]
- Based addressing = there is a base register ex: mov eax, [ebx]
- Scale-indexed addressing = one of the index reg is present ex: mov eax, [ecx]
- Indirect addressing = a non-direct addressing mode (based or/and scale-indexed) $\rightarrow$ at least one reg. between []
ex: mov eax, [eax + 2*ebx + v + 6]

Rules for performing the association with a segm. reg:
- CS for code labels target of the control transfer instructions (jmp, call, ret) ex: jmp there
- SS in SIB addressing when using ESP or EBP as a base ex: mov eax, [esp + 4]
- DS for the rest of the data accesses

At the level of 80x86 architecture, the memory can be accessed only by using the offset computation formula, without variables names associated to a data type.
The task of a data def. directive is not to specify an associated data type to a var., but to generate the corresponding list of bytes for a named memory area following the little endian representation

The general form of a data def. source line is:

I. [name] data_type expression_list
- name: label for a data referral
- data_type: type of representation

data_type can be: db, dw, dd, dq, dt

ex:
```
a db 1
  dw '12'
c dd 30h
d dq 'a','b','c','d'
e dt 1000h
```

II. [name] allocation_type factor
- allocation_type: uninitiali data rezervation directives
- factor: nr. that shows how many times the allocation type is repeated

ex: a resb 3 → reserves 3 byte starting from the address of a

allocation_type: resb, resw, resd, resq, rest

III. [name] TIMES factor data_type expression_list

TIMES directive allows repeated assembly of an instruction or data def.

ex: sir TIMES 10 db 2 → creates an array of 10 bytes starting from the address of sir, every elem is equal to 2

The EQU directive allows assigning an immediate value during assembly time to a label without allocating any memory bytes or byte generation

ex: two equ 2

Conversions classification:

1. distructive <-> non-destructive

a) distructive: cbw, cwd, cwde, cdq, movsx, movzx; mov ah,0 ; mov dx,0 ; mov edx,0

ex:
```
mov al, -2
cbw   ; => AX = FFFE
```
```
mov al, 6
mov ah, 0
```

b) non-distructive: byte, word, dword, qword ex: mov al, byte[a] ; Only AL is modified

2. by enlargement <-> by narrowing

a) by enlargement: distructive ones + word, dword, qword
```
a db 10
.....
mov ax, word[a]
```

b) by narrowing: byte, word, dword
```
a dd 100h
.....
mov al, byte[a]
```

3. a) signed cbw, cwd, cwde, cdq, movsx
```
mov al, -1
cbw  ; AX = FFFF
```

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

```
mov al,5
movzx ax,al
```

⋮ AX = 5

4. implicit vs explicit

float → integer

int → float (implicit)

There are cases when the data type does not need to be specified

ex: mov al, 5 because the size of al is taken into account

but at push [var] it is syntax error

the size must be specified : push dword[var]

DATA TYPE