C H A P T E R 4
ASSEMBLY LANGUAGE INSTRUCTIONS

General form of an ASM program in NASM + short example:

```
global start                    ; we ask the assembler to give global visibility to the symbol called start
                                ;(the start label will be the entry point in the program)

extern ExitProcess, printf      ; we inform the assembler that the ExitProcess and printf symbols are foreign
                                ;(they exist even if we won't be defining them),
                                ; there will be no errors reported due to the lack of their definition

import ExitProcess kernel32.dll  ;we specify the external libraries that define the two symbols:
                    ; ExitProcess is part of kernel32.dll library (standard operating system library)
import printf msvcrt.dll            ;printf is a standard C function from msvcrt.dll library (OS)

bits 32                     ; assembling for the 32 bits architecture

segment code use32 class=CODE          ; the program code will be part of a segment called code
        start:
                ; call printf(”Hello from ASM”)
                push dword string   ; we send the printf parameter (string address) to the stack (as printf requires)
                call   [printf]          ; printf is a function (label = address, so it must be indirected [])

                ; call ExitProcess(0), 0 represents status code: SUCCESS
                push dword  0
                call   [ExitProcess]

segment data use32 class=DATA   ; our cariables are declared here (the segment is called data)
        string:    db "Hello from ASM!", 0
```

4.1. DATA MANAGEMENT / 4.1.1. Data transfer instructions

4.1.1.1. General use transfer instructions

MOV *d,s*	<d> <-- <s> (b-b, w-w, d-d)	-
PUSH *s*	ESP = ESP - 4 and transfers („pushes") <s> in the stack (s – doubleword)	-
POP *d*	Eliminates („pops") the current element from the top of the stack and transfers it to d (d – doubleword) ; ESP = ESP + 4	-
XCHG *d,s*	<d> ↔ <s> ; s,d – have to be L-values !!!	-
[reg_segment] XLAT	AL ← < DS:[EBX+AL] > or AL ← < segment:[EBX+AL] >	-
CMOVcc d, s	<d> ← <s> if cc (conditional move) is true	-
PUSHA / PUSHAD	Pushes in the stack EAX, ECX, EDX, EBX, ESP, EBP, ESI and EDI	-
POPA / POPAD	Pops EDI, ESI, EBP, ESP, EBX, EDX, ECX and EAX from stack	-
PUSHF	Pushes EFlags in the stack	-
POPF	Pops the top of the stack and transfers it to Eflags	-
SETcc d	<d> ← 1 if cc is true, otherwise <d> ← 0 (byte set on condition code)	-

If the destination operand of the MOV instruction is one of the 6 segment registers, then the source must be one of the eight 16 bits EU general registers or a memory variable. The loader of the operating system initializes automatically all segment registers and changing their values, although possible from the processor point of view, does not bring any utility (a program is limited to load only selector values which indicates to OS preconfigured segments without being able to define additional segments).

PUSH and **POP** instructions have the syntax **PUSH** *s* and **POP** *d*

Operands d and s MUST be doublewords, because the stack is organized on doublewords. The stack grows from big addresses to small addresses, 4 bytes at a time, ESP pointing always to the doubleword from the top of the stack .

We can illustrate the way in ehich these instructions works, by using an equivalent sequence of MOV and ADD or SUB instructions:

```
push eax  ⇔     sub esp, 4          ; prepare (allocate) space in order to store the value
                mov [esp], eax      ; store the value in the allocated space

pop eax   ⇔     mov eax, [esp]      ; load in eax the value from the top of the stack
                add esp, 4          ; clear the location
```

These instructions only allow you to deposit and extract values represented by word and double. Thus, PUSH AL is not a valid instruction (syntax error), because the operand is not allowed to be a byte value. On the other hand, the sequence of instructions

```
PUSH  ax     ; push ax in the stack
PUSH  ebx    ; push ebx in the stack
POP   ecx    ; ecx <- the doubleword from the top of the stack (the value of ebx)
POP   dx     ; dx <- the word from the stack (the value of ax)
```

Is a valid sequence of instructions and is equivalent as an effect with:

```
MOV   ecx,   ebx
MOV   dx,    ax
```

In addition to this constraint (which is inherent in all x86 processors), the operating system requires that stack operations be made only through doublewords or multiple of doublewords accesses, for reasons of compatibility between user programs and the kernel and system libraries. The implication of this constraint is that the PUSH operand16 or POP operand16 instructions (for example, PUSH word 10), although supported by the processor and assembled successfully by the assembler, is not allowed by the operating system, might causing what is named the incorrectly aligned stack error: the stack is correctly aligned if and only if the value in the ESP register is permanently divisible by 4!

The **XCHG** instruction allows interchanging the contents of two operands having the same size (byte, word or doubleword), at least one of them having to be a register (the other one being either a register or a memory address). Its syntax is

XCHG *operand1, operand2*

XLAT "translates" the byte from AL to another byte, using for that purpose a user-defined correspondence table called *translation table.* The syntax of the XLAT instruction is

[reg_segment] XLAT

translation_table is the direct address of a string of bytes. The instruction requires at entry the far address of the translation table provided in one of the following two ways:

- DS:EBX (implicit, if the segment register is missing)
- segment_register:EBX, if the segment register is explicitly specified.

The effect of **XLAT** is the replacement of the byte from AL with the byte from the translation table having the index the initial value from AL (the first byte from the table has index 0). EXAMPLE: pag.111-112 (course book).

For example, the sequence

```
mov  ebx, Table
mov  al,6
ES  xlat                    AL ← < ES:[EBX+6] >
```

transfers the content of the 7th memory location (having the index 6) from *Table* into AL.

The following example translates a decimal value "number" between 0 and 15 into the ASCII code of the corresponding hexadecimal digit :

```
segment data use32
        .       .       .
TabHexa     db      '0123456789ABCDEF'
        .       .       .
segment code use32
mov   ebx, TabHexa
        .       .       .
mov   al, numar
xlat                        ; AL ← < DS:[EBX+AL] >
 ES xlat                    ; AL ← < ES:[EBX+AL] >
```

This strategy is commonly used and proves useful in preparing an integer numerical value for printing (it represents a conversion *register numerical value – string to print*).

How many ACTIVE code segments can we have ?... 1 – CS
How many ACTIVE stack segments can we have ?... 1 - SS
How many ACTIVE data segments can we have ?... 2 – DS and ES BOTH are reffering to DATA segments

4.1.1.3. Address transfer instruction - LEA

LEA *general_reg*, *contents of a memory_operand*	general_reg ← offset(mem_operand)	-

LEA (***L**oad **E**ffective **A**ddress*) transfers the offset of the *mem* operand into the destination register. For example

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

loads into EAX the offset of the variable v, the instruction equivalent to `mov  eax, v`

But **LEA** has the advantage that the source operand may be an addressing expression (unlike the mov instruction which allows as a source operand only a variable with direct addressing in such a case). For example, the instruction:

```
lea   eax,[ebx+v-6]
```

is not equivalent to a single MOV instruction. The instruction `mov eax, ebx+v-6`

is syntactically incorrect, because the expression ebx+v-6 cannot be determined at assembly time.

By using the values of offsets that result from address computations directly (in contrast to using the memory pointed by them), LEA provides more versatility and increased efficiency: versatility by combining a multiplication with additions of registers and/or constant values and increased efficiency because the whole computation is performed in a single instruction, without occupying the ALU circuits, which remain available for other operations (while the address computation is performed by specialized circuits in BIU)

Example: multiplying a number with 10

```
mov eax, [number]        ; eax <- the value of the variable number
lea eax,   [eax * 2]     ; eax <- number * 2
lea eax,   [eax * 4 + eax]   ; eax <- (eax * 4) + eax  = eax * 5 = (number * 2) * 5
```

4.1.1.4. Flag instructions

The following four instructions are *flags transfer instructions*:

LAHF (***Load register AH from Flags***) copies SF, ZF, AF, PF and CF from FLAGS register in the bits 7, 6, 4, 2 and 0, respectively, of register AH. The contents of bits 5, 3 and 1 are undefined. Other flags are not affected (meaning that LAHF does not generate itself other effects on some other flags – it just transfers the flags values and that's all).

SAHF (***Store register AH into Flags***) transfers the bits 7, 6, 4, 2 and 0 of register AH in SF, ZF, AF, PF and CF respectively, replacing the previous values of these flags.

PUSHF transfers all the flags on top of the stack (the contents of the EFLAGS register is transferred onto the stack).The values of the flags are not affected by this instruction. The **POPF** instruction extracts the word from top of the stack and transfer its contents into the EFLAGS register.

The assembly language provides the programmer with some instructions to set the value of the flags (the condition indicators) so that the programmer can influence the operation mode of the instructions which exploits these flags as desired.

CLC	CF=0	CF
CMC	CF = ~CF	CF
STC	CF=1	CF
CLD	DF=0	DF
STD	DF=1	DF

CLI, STI – they are used on the Interrupt Flag. They have effect only on 16 bits programming, on 32 bits the OS blocking the programmer's access to this flag.

4.1.2. **Type conversion instructions** (destructive)

CBW	converts the byte from AL to the word in AX (sign extension)	-
CWD	converts the word from AX to the doubleword in DX:AX (sign extension)	-
CWDE	converts the word from AX to the doubleword in EAX (sign extension)	-
CDQ	converts the doubleword from EAX to the quadword in EDX:EAX (sign extension)	
MOVZX d, s	loads in d (REGISTER !), which must be of size larger than s (reg/mem), the UNSIGNED contents of s (zero extension)	-
MOVSX d, s	load in d (REGISTER !), which must be of size larger than s (reg/mem), the SIGNED contents of s (sign extension) http://www.c-jump.com/CIS77/ASM/DataTypes/T77_0270_sext_example_movsx.htm !!!!!!!!!!!	-

CBW converts the signed byte from AL into the signed word AX (extends the sign bit of the byte from AL into the whole AH, thus destroying the previous content of AH). For example,

```
mov al, -1      ; AL=0FFh
cbw             ;extends the byte value -1 from AL to the word value -1 in AX (0FFFFh).
```

Similarly, for the signed conversion word - doubleword, the **CWD** instruction extends the signed word from AX into the signed doubleword in DX:AX. Example:

```
mov ax,-10000       ; AX = 0D8F0h
cwd         ;obtains the value  -10000  in  DX:AX (DX = 0FFFFh ; AX = 0D8F0h)
cwde        ; obtains the value  -10000  in  EAX     (EAX = 0FFFFD8F0h)
```

The **unsigned** conversion is done by „zerorizing" the higher byte or word of the initial value (for example, by `mov ah,0` or `mov dx,0` – a similar effect like applying the `MOVZX` instruction)

Why CWD coexists with CWDE ? The CWD instruction must remain for backwards compatibility reasons, but also to assure the proper functioning of the (I)MUL and (I)DIV instructions.

4.1.3. The impact of the little-endian representation on accessing data (pag.119 – 122 – coursebook)

If the programmer uses data consistent with the size of representation established at definition time (for example accessing bytes as bytes and not as bytes sequences interpretted as words or doublewords, accesing words as words and not as bytes pairs, accessing doubewords as doublewords and not as sequences of bytes or words) then the assembly language instructions will automatically take into account the details of representation (they will manage automatically the little-endian memory layout). If so, the programmer must NOT provide himself any source code measures for assuring the correctness of data management. Example:

```
a   db   'd', -25, 120
b   dw     -15642, 2ba5h
c    dd   12345678h
…
mov  al, [a]      ;loads in  AL the  ASCII code of  'd'
mov  bx, [b]     ;loads in BX the value -15642; the order of bytes in BX will be reversed compared to the
                  memory representation of b, because only the memory representation uses little-endian! At
                  register level data is stored according to the usual structural representation (equiv.to a big
                  endian representation).

mov edx, [c]     ;loads in  EDX the value of the doubleword 12345678h
```

If we need accessing or interpreting data in a different form than that of definition then we must use explicit type conversions. In such a case, the programmer must assume the whole responsability of correctly accessing and interpreting data. In such cases the programmer must be aware of the little-endian representation details (the particular memory layout corresponding to that variable/memory area) and use proper and consistent accesing mechanisms **Ex pag.120-122.**

```
segment data
    a dw 1234h          ;because of the little-endian representation, in memorie the bytes have the
                        following placement:
    b dd 11223344h      ;34h   12h   44h   33h   22h   11h
                ; address    a     a+1    b     b+1    b+2    b+3

    c db -1

segment code

    mov al, byte [a+1] ;accessing a as a byte, calculating the address a+1, selecting the byte from the address
                        a+1 (the byte with the value of 12h) and transfer it in the AL register

    mov dx, word [b+2]      ;dx:=1122h

    mov dx, word [a+4]      ;dx:=1122h  because b+2 = a+4, these pointer type expressions compute the same
                            address, specifically the address of the byte 22h.

    mov dx, [a+4]           ;this instruction is equivalent to the previous one, specifying the conversion operator
                            WORD not being required.

    mov bx,  [b]            ;bx:=3344h

    mov bx,  [a+2]          ;bx:=3344h, because the following addresses are equal: b = a+2.

    mov ecx, dword [a]      ;ecx:=33441234h, because the doubleword that starts at the address of a is
                            composed of the following bytes: 34h  12h  44h  33h, which (because of the little-
                            endian representation) form the following doubleword: 33441234h.

    mov ebx, [b]                ; ebx := 11223344h

    mov ax, word [a+1]          ; ax := 4412h
```

```
mov eax, dword [a+1]        ; eax := 22334412h

mov dx, [c-2]               ; DX := 1122h because c-2 = b+2 = a+4

mov bh, [b]                 ;bh := 44h

mov ch, [b-1]               ;ch := 12h

mov cx, [b+3]               ;CX := 0FF11h
```

4.2. OPERATIONS.

4.2.1. Arithmetic operations

Operands are represented in complementary code (see 1.5.2.). The microprocessor performs additions and substractions "seeing" only bits configurations, NOT signed or unsigned numbers. The rules of binary adding or substracting two numbers do not impose previously considering the operands as signed or unsigned, because independently of interpretation, additions and subtractions works the same way. So, at the level of these operations, the signed or unsigned interpretation depends on a further context and is left to the programmer.

The addition and the subtraction are evaluated in the same way (adding or subtracting the binary configurations) not taking into account the sign (interpretation) of these configurations! This does not apply to multiplication and division. When using these operations we need to know beforehand if the operands will be interpreted as signed or unsigned.

For example, if A and B are bytes:

A = 9Ch = 10011100b (= 156 in the unsigned interpretation and -100 in the signed interpretation)
B = 4Ah = 01001010b (= 74 , both in signed and unsigned interpretation)

The microprocessor performs the addition C = A + B obtaining

C = E6h = 11100110b (= 230 in the unsigned interpretation and -26 in the signed one)

We though notice that the simple addition of the bits configuration (without taking into account a certain interpretation at the moment of addition) assures the result correctness, both in signed and unsigned interpretation.

ARITHMETIC INSTRUCTIONS – page 123 (coursebook)
4.2.1.3. Examples – page 129-130 (coursebook)

4.2.2. Logical bitwise operations (AND, OR, XOR and NOT instructions).
AND is recommended for isolating a certain bit or for forcing the value of some bits to 0.
OR is suitable for forcing certain bits to 1.
XOR is suitable for complementing the value of some bits.

4.2.3. Shifts and rotates.
Bit shifting instructions can be classified in the following way:

- Logic shifting instructions
  - left - **SHL**
  - right - **SHR**
- Arithmetic shifting instructions
  - left - **SAL**
  - -right - **SAR**

Bit rotating instructions can be classifed in the following way:

- Rotating instructions without carry
  - left - **ROL**
  - right - **ROR**
- Rotating instructions with carry
  - left - **RCL**
  - right - **RCR**

For giving a suggestive definition for shifts and rotates let's consider as an initial configuration one byte having the value X = abcdefgh, where a-h are binary digits, h is the least significant bit, bit 0, a is the most significant one, bit 7, and k is the actual value from CF (CF=k).

We then have:

SHL X,1 ;has the effect X = bcdefgh0 and CF = a
SHR X,1 ;has the effect X = 0abcdefg and CF = h
SAL X,1 ; identically to SHL
SAR X,1 ;has the effect X = aabcdefg and CF = h
ROL X,1 ;has the effect X = bcdefgha and CF = a
ROR X,1 ;has the effect X = habcdefg and CF = h
RCL X,1 ;has the effect X = bcdefghk and CF = a
RCR X,1 ;has the effect X = kabcdefg and CF = h

INSTRUCTIONS FOR BITS SHIFTING AND ROTATING – pag.134 (coursebook)

4.3. BRANCHING, JUMPS, LOOPS

4.3.1. Unconditional jump

Three instructions fall into this cathegory: JMP (equiv. to GOTO from other languages), CALL (a procedure call means a control transfer from the call's point to the first instruction from the called routine) and RET (control transfer back to the first executable instruction after the CALL).

JMP *operand*	Unconditional jump to the address specified by operand	-
CALL *operand*	Transfers control to the procedure identified by operand	-
RET [*n*]	Transfers control to the first instruction after CALL	-

4.3.1.1. JMP instruction

Syntax: **JMP** *operand*

where *operand* is a label, register or a memory address containing an address. Its effect is the unconditional control transfer to the instruction following the label, to the address contained in the register or to the address specified by the memory variable respectively. For example, after running the sequence

```
            mov  ax,1
            jmp  AdunaDoi
AdunaUnu:   inc  ax
            jmp  urmare
AdunaDoi:   add  ax,2
urmare:     .    .    .
```

AX will hold the value 3. **inc** şi **jmp** betwwen *AdunaUnu* and *AdunaDoi* will not be executed, unless a jump to *AdunaUnu* will be done from another step of the program.

As mentioned above, the jump may be made to an address stored in a register or in a memory variable. Examples:

(1)
```
mov  eax, etich

jmp  eax  ;register operand
          ; jmp [eax] ?
etich:    . . .
```

(2)
```
segment data
      Salt  DD  Dest ;Salt := offset Dest
      .     .   .
segment code
      .     .   .
      jmp  [Salt]      ;NEAR jump
.     ;memory variable operand
Dest :    . . .
```

If in case (1) we wish to replace the register destination operand with a memory variable destination operand, a possible solution is:

```
            b   resd  1
(1')            . . .
            mov  [b], DWORD etich ; b := offset etich
            jmp [b]          ; NEAR jump – memory variable operand
                             ;  JMP DWORD PTR DS:[offset_b]
```

Exemplul 4.3.1.2. – pag.142-143 (coursebook) – control transfer to a label. Analysis and comparison.

4.3.2. Conditional jump instructions

4.3.2.1. Comparisions between operands

CMP *d,s*	compares the operands values (does not modify them - fictious subtraction *d* - *s*)	OF,SF,ZF,AF,PF and CF
TEST *d,s*	non-destructive *d* **AND** *s*	OF = 0, CF = 0 SF,ZF,PF - modified, AF - undefined

Conditional jump instructions are usually used combined with comparision instructions. Thus, the semantics of jump instructions follows the semantics of a comparision instruction. Besides the equality test performed by a CMP instruction we need frequently to determine the exact order relationship between 2 values. For example we have to answer to: nr. 11111111b (= FFh = 255 = -1) is bigger than 00000000b(= 0h = 0)? The answer is .... IT DEPENDS !!!! This answer can be either YES or NO ! If we perform an unsigned comparision, then the first one is 255 and is obvious bigger than 0. If the 2 values are compared in the signed interpretation, then the first is -1 and is less than 0.

The CMP instruction does not make any difference between the two above cases, because as we mentioned in 4.2.1.1 addition and subtraction are performed always in the same way (adding or subtracting binary configurations) no matter their interpretations (signed or unsigned). So it's not the matter to interpret the operands of CMP as being signed or unsigned, but to further interpret the RESULT of the subtraction ! Conditional jump instructions are responsible to do that (Section 4.4.2.2).

4.3.2.2. Conditional jumps

Table 4.1. (pag.146 – coursebook) presents the conditional jump instructions together with their semantics and according to which flags values the jumps are made. For all the conditional jump instructions the general syntax is

<conditional_jump_instruction> label

The effect of the conditional jump instructions is expressed as ***"jump if operand1 <<relationship>> operand2"*** (where on the two operands a previously CMP or SUB instruction is supposed to have been applied) or relative to the actual value set for a certain flag. As easy can be noticed based on the conditions that must be verified, instructions on the same line in the table have similar effect.

When two signed numbers are compared, **"less than"** and **"greater than"** terms are used and when two unsigned numbers are compared *"below"* and *"above"* terms are respectively used.

4.3.2.3. Examples along with comments..... pag.148-162 (coursebook).

- comparative analysis and discussion of the concepts of: signed vs. unsigned representations, overflow, actual effects of conditional jump instructions, specific flags (CF, OF, SF, ZF)

4.3.3. Repetitive instructions (coursebook pag.162 – 164)

These are: **LOOP, LOOPE, LOOPNE** and **JECXZ**. Their syntax is

<instruction> label

LOOP performs the repetitive run of the instructions block starting at *label*, as long as the value of CX register is different from 0. **It first performs decrementation of ECX, then the test and eventually the jump.** The jump is "short" (max. 127 bytes – so pay attention to the "distance" between LOOP and the label!). – PAY ATTENTION !! CHECK IT !!! (short jump is out of range!)

When the end of loop conditions are more complex **LOOPE** and **LOOPNE** may be used. **LOOPE** (***LOOP*** *while* ***E****qual*) differ from LOOP by ending condition, loop is ended either if ECX=0, either if ZF=0. In the case of **LOOPNE** (***LOOP*** *while* ***N****ot* ***E****qual*) the loop will end either if ECX=0, either if ZF=1. Even if the loop exit shall

be based on value of ZF, CX decrementation is done anyway. **LOOPE** is also known as **LOOPZ** and **LOOPNE** is also known as **LOOPNZ**. These are usually used preceeded by a CMP or SUB instruction.

JECXZ (***Jump if ECX is Zero***) performs the jump to the operand label only if ECX=0, being useful when we want to test the value in ECX before entering in a loop. In the following example, JECXZ instruction is used to avoid entering the loop if ECX=0:

```
            .       .       .
        jecxz MaiDeparte        ;if ECX=0 a jump over the loop is made
    Bucla:
        Mov   BYTE [esi],0      ;initializing the current byte
        inc   si                ;passing to the next byte
        loop  Bucla             ;resume the loop or ending it
    MaiDeparte:     .       .       .
```

If a loop is entered with ECX=0, ECX is first decremented, obtaining the value 0FFFF FFFFh (= -1, so a value different from 0), the loop being resumed until 0 in ECX will be reached, namely **2^32** = 4.294.967.296 more times !

It's important to say here that <u>none of the presented repetitive instructions affects the flags</u>.

```
                            dec  ecx
    loop Bucla    and       jnz  Bucla
```

although semantic equivalent, they do not have the same effect, because DEC modifies OF, ZF, SF and PF, while LOOP doesn't affect any flag.

4.3.4. <u>CALL and RET instructions</u>

A procedure call is done by using the **CALL** instruction, it can be a *direct* or an *indirect* call. The direct call has the syntax:

CALL *operand*

Similar to JMP, **CALL** transfers the control to the address specified by the operand. In addition to JMP, before performing the jump, CALL saves to the stack the address of the instruction following CALL (the returning address). In other words, we have the equivalence:

```
CALL operand              push dword A
A: . . .          ⇔       jmp  operand
```

The end of the called sequence is marked by a **RET** instruction. This pops from the stack the returning address stored there by CALL, transferring the control to the instruction from this address. The RET syntax is:

RET [*n*]

where *n* is an optional parameter. It indicates freeing from the stack *n* bytes below the returning address.

RET instruction can be illustrated by this equivalence:

```
                          B   resd 1
RET n                     .     .     .
(near return)     ⇔       pop dword [B]
                          add  esp,n
                          jmp  [B]
```

Usually, as it is natural, CALL and RET are used in the following context:

```
        procedure_label:
.       .       .
        ret  n
.       .       .
```

```
CALL  procedure_label
```

CALL may also take the transfer address from a register or from a memory variable. Such a call is identified as an *indirect call*. Example:

```
call  ebx       ;address taken from a register
call  [vptr]    ;address taken from a memory variable
```

Concluding, the destination operand of a CALL instruction may be:

- a procedure name
- the name of a register containing an address
- a memory address