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*

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
```

In the perspective of evaluating the effect of instructions such as PUSH ESP or POP dword [ESP], the order in which the component (sub)operations of the PUSH and POP instructions are performed should be specified even more clearly:

a). The source operand of the instruction is evaluated (ESP for PUSH and dword [ESP] respectively for POP)
b). ESP is updated accordingly (ESP := ESP-4 for PUSH and ESP := ESP+4 for POP respectively)
c). The assignment involved in the effect of the instruction is performed on the destination operand (the new top of the stack for PUSH and dword [ESP] respectively (this being now after the subtraction of ESP from b) the element immediately below the initial top of the stack - for POP)

Assuming that the initial situation is ESP = 0019FF74, after PUSH ESP we will have ESP = 0019FF70 and the contents of the top of the stack will now be 0019FF74.

Assuming that the initial situation is ESP = 0019FF74 and that in this location is the value 7741FA29 (the top of the stack), after POP dword [ESP] we will have ESP = 0019FF78 and the content of this location (the content of the location at the top of the stack) will be 7741FA29 (so we could say that "the top of the stack moves one position lower"!!).

PUSH and POP only allow you to deposit and extract values represented by word and doubleword. 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). This restriction comes from the fact that both operands must be L-values !! 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'
numar resb 1
        .       .       .
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.