C H A P T E R 4
ASSEMBLY LANGUAGE INSTRUCTIONS

General forma of an ASM program :

```
assume  cs:code, ds:data        ;ASSUME is detailed in  3.3.1.2.

data segment                    ;data is here the name of the data segment – this name may ne changed
          .....                 ;data definitions - DB, DW, DD.
data ends

code segment               ;code is here the name of the code segment – this name may ne changed
start:    mov ax,data
          mov ds,ax        ;loading the DS segment register with the data segment starting address
          . . .
          mov ah,4ch
          int 21h          ;calling the 4ch function of interrupt 21h for ending
                           ;the run of the current program
code ends
end start                       ;indicates the end of the source file (the END directive) together with defining
                    the program's entry point – code label "start"
```

We may develop programs containing only one data segment and one code segment, having predefined mandatory names (*data* and *code*). For this purpose the <u>simplified directives</u> may be used (TASM only). Such a program has the following structure:

```
     .model    small
     .data
          .    .    .               ;contents of the data segment
```

```
.code
Start:
        mov ax,@Data
        mov ds,ax
        .    .    .          ; contents of the code segment
        mov ah,4ch
        int 21h              ;call of  4ch function of  21h interrupt – program termination
end  Start
```

A source program is edited in a file with the .asm extension: *name*.ASM

where *name* is a user-defined identifier. In this case the following sequence of MS-DOS commands :

```
...>TASM  name
...>TLINK name
...>TD  name
```

accomplishes the assembling, link-editing and the assisted run of the respective program.

4.1. DATA MANAGEMENT/ 4.1.1. <u>Data transfer instructions</u>

4.1.1.1. <u>General use transfer instructions</u>

MOV *d,s*	<d> <-- <s>	-
PUSH *s*	transfers („pushes”) <s> into the stack	-
POP *d*	eliminates („pops”) the current element from the top of the stack and transfers it to d	-
XCHG *d,s*	<d> <--> <s>	-

XLAT [*translation_table*]	AL <-- < seg:[BX+<AL>] >	-

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

Operands d and s MUST be words, because the stack is organized on words. The stack grows from big addresses to small addresses, 2 bytes at a time, SP pointing always to the word from the top of the stack.

The **XCHG** instruction allows interchanging the contents of two operands having the same size (byte or word), at least one of them having to be a register. 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

XLAT [*translation_table*]

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:BX (implicit, if the operand is missing)
- segment_register:BX, if the XLAT operand is present, the segment register being determined based on the operand’s corresponding ASSUME directive.

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 (coursebook).

4.1.1.3. <u>Address transfer instructions</u>

These instructions are very useful when operating with strings, for passing parameters etc. In the below table *reg* is a general register.

LEA *reg,mem*	reg <-- offset(mem)	-
LDS *reg,mem*	reg <-- <mem>, DS <-- <mem+2>	_
LES *reg,mem*	reg <-- <mem>, ES <-- <mem+2>	_

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

```
lea   ax,v
```

loads in AX the offset of v, instruction equivalent to `mov ax, offset v`

But **LEA** has the advantage that the source operand may be indexed and also registers operands may be used according to the offset specification formula. For example

```
lea   ax,[bx+v]        is NOT equiv. to mov ax, offset [bx+v]    (syntax error !)
```

because OFFSET is an <u>operator and operators impose the evaluations to be made at assembly time</u>.

LDS (***L****oad pointer using* ***DS***) and **LES** (***L****oad pointer using* ***ES***) transfer the far address stored in *mem* (so the <u>**contents** of the doubleword variable *mem*</u>!) in the pair of registers DS:reg and ES:reg respectively. LDS and LES are an efficient way of preparing in registers the value of some pointers (far addresses of some specific memory locations).

For example, supposing that *varp* contains the value 1234h:5678h (possibly further interpreted as the far address), we may obtain the first byte of this memory area in AL by applying the following instruction sequence

```
        .   .   .
    varp dd    12345678h
    .   .   .
    lds   bx,varp     ;transfers the contents of  varp  in DS:BX - in DS:=1234h and in  BX:=5678h
    mov  al,[bx]      ;transfers in AL the byte from DS:[BX], namely the byte from 1234h:5678h
```

Example illustrating how useful address transfer instructions could be:

```
data1      segment                                    data2      segment
    db 'first data segment'                           db  'the second data segment'
    aa    db  31                                          db  '01234567'
data1      ends                                           bb    dd  aa ;bb will contain the far address of  aa
                                                      data2      ends
            cod  segment
                assume  cs:cod, ds:data2, es:data1
            start:      .    .    .
                mov ax,data2
                mov ds,ax
                mov ax,data1
                mov es,ax
                lea   bx,aa              ;equiv. to   mov bx,offset aa
                lds   di,bb              ;ds:di := es:offset aa
    (practically  bb is used here as a pointer variable containing the far address of aa)
                    mov al, ds:[di]        ;al := 31 (the value from the address of the data label aa)
                    .    .    .
            cod  ends
                end  start
```

LDS and LES are the only assembly language instructions which allow both operands to be doublewords in an explicită manner (the contents of the <mem> doubleword is transfered in DS:reg or ES:reg).

We notice very much confusion regarding the mechanisms involved by LDS and LES relatively to that of LEA. Because all 3 of them are presented as "address transfer instructions" and because it is easy to understand the effect of LEA, there is a temptation to infere that LDS and LES are similar to LEA, in the sense that they will not load ONLY the offset of the source operand (as LEA does) but (and now there is a mistaken interpretation!!!) they will transfer ***the whole far address (segment:offset) of the source operand***! Here comes up the very frequent interpretation mistake done : ***LDS and LES do NOT transfer the far address of the source operand, but ITS CONTENTS !***

Re-analyzing the previous example in which we had

```
        lds  bx, varp            ; transfer of the contents of varp in DS:BX -  correct!
```

It is very important not to get into confusion and thinking in terms of LEA and to interpret that the effect of the above instruction to be "the transfer of the varp **address** in ds:bx" (INCORRECT conclusion !!!). If we really wanted something like that , the correct solution is :

```
            ; the following sequence loads in ds:bx the  far address of  the varp variable!

            mov ax, SEG varp     ;the SEG operatorul was introduced in 3.2.2.7.
            mov ds,ax            ;transfers the segment address of varp in DS
            lea bx, varp         ;transfer the offset of varp in BX
```

4.1.1.4. Flag instructions

The following 4 instructions are *flags transfer instructions* :

LAHF (***L****oad register* ***AH*** *from* ***F****lags*) copies SF, ZF, AF, PF and CF from FLAGS register in the bits 7, 6, 4, 2 and respectively 0 of register AH. The contents of bits 5,3 and 1 is 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 (***S****tore register* ***AH*** *into* ***F****lags*) 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 FLAGS register is transferred onto the stack)

POPF extracts the word from top of the stack and transfer its contents into the FLAGS register.

Also, the 8086 assembly language offers instructions for *setting the values* for 3 of the 9 existing flags.

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

STI	IF=1	IF

4.1.2. Type conversion instructions.

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)	-

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

```
mov al, -1
cbw             ;extends the byte value -1 from AL to the word value -1 in AX.
```

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

```
mov ax,-10000
cwd                 ;obtains the value  -10000  in  DX:AX.
```

The **unsigned** conversion is done by „zerorizing" the higher byte or word of the initial value:

```
mov ah,0 ; unsigned conversion of AL to AX
mov dx,0 ; unsigned conversion of AX to DX:AX
```

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 then 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 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*)*.

les dx, c ; loads in ES:DX the doubleword value 1234:5678h ; in ES - 1234h ; in DX - 5678h.
; the memory layout for c is 78h 56h 34h 12h ; after **les** the order of bytes in the registers
; will be the „normal" structural one , namely 12h 34h 56h 78h.

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.**

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. These means that NO special signed addition or signed subtraction instructions are needed (no need for ”IADD” or ”ISUB”).

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 moemnt of addition) assures the result correctness, both in signed and unsigned interpretation.

ARITHMETIC INSTRUCTIONS – pag.123 (coursebook)

4.2.1.3. Examples – pag.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.

Shift instructions :

- Logic shifts
  - left - **SHL**
  - right - **SHR**
- Arithmetic shifts
  - left - **SAL**
  - right - **SAR**

Instrucțiunile de *rotire* a biților în cadrul unui operand se clasifică în:

- Instrucțiuni de rotire fără carry
  - left - **ROL**
  - right - **ROR**
- Instrucțiuni de rotire cu 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, a is the most significant one and k is the actual value from CF (CF=k). We have then:

```
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 variable 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.

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

```
(1)  mov  ax, OFFSET etich          (2)  data segment
                                              Salt   DW  Dest ;Salt := offset Dest
     jmp  ax    ;register operand             .     .     .
                                         code segment
     etich:     . . .                         .     .     .
                                              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   dw   ?
        (1')         . . .
                 mov  b, offset etich
                 jmp b             ; NEAR jump – memory variable operand
```

JMP may also be used for transferring the control in another code segment (*far jump*). In such a case the target address must be identified as a FAR one (*segment:offset*).

A FAR jump may be accomplished in one of the following 4 ways:

a). declaring the destination label as FAR label using the LABEL directive (see 3.3.3). Example – pag.140.

b). directly specifying in the JMP instruction the FAR PTR type for the destination label as: JMP FAR PTR *label.* Such a specification has been used in 3.3.1.2 – the ASSUME directive. The PTR operator - 3.2.2.7.

c). explicitly prefixing with a segment register a NEAR address specified indirectly, namely *reg_segment: specificare_offset*. Example: jmp es:[bx+di].

d). specifying as an operand a doubleword variable containing the far address of destination – (pag.141).
Exemplul 4.3.1.2. – pag.142-143 (coursebook) – control transfer to a label. Analysis and comparison between direct transfer – indirect transfer.

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 follow 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 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 instruction are in charge to do that.

4.3.2.2. <u>Conditional jumps</u>

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

4.3.2.3. <u>Exemple comentate.............   pag.148-162 (coursebook).</u>

- Comparative analysis and discussion on : signed representations vs. unsigned, overflow, actual effects of conditional jump instructions.

4.3.3. <u>Repetitive instructions: loops (pag.162 – 164).</u>

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

<instruction> label

LOOP performs the repetitive run of the instructions block starting at *label*, as long as the contents of CX register is not 0. **It first performs decrementation of CX, then the test vs. 0 and then the jump if the result is not 0.** The jump is a NEAR one (max. 127 bytes – so pay attention to the "distance" between LOOP and the label!).

When the end of loop conditions are more complex **LOOPE** and **LOOPNE** may be used. **LOOPE** (***LOOP** while **E**qual*) ends the loop either if CX=0, either if ZF=1. **LOOPNE** (***LOOP** while **N**ot **E**qual*) ends the loop either if CX=0, either if ZF=0. Even if the loop exit is done based on ZF, CX decrementation is done anyway. **LOOPE** is the same as **LOOPZ** and **LOOPNE** the same as **LOOPNZ**. These are usually used preceeded by a CMP or SUB instruction.

JCXZ (***J**ump if **CX** is **Z**ero*) performs the jump to the specified label only if CX=0, being useful when we want to test the value from CX before entering the loop. The example below shows why JCXZ is useful – for avoiding enetring the loop with CX=0:

```
            .    .    .
        jcxz  MaiDeparte        ;if CX=0 a jump over the loop is made
    Bucla:
        Mov   BYTE PTR [si],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 CX=0, CX is first decremented, obtaining the value 0FFFFh (= -1, so diff. from 0, but also 65535 in the unsigned interpretation), the loop being resumed until 0 in CX will be reached, namely 65535 times !

<u>No loop instruction affects the flags</u>.

```
                          dec  cx
        loop Bucla   and  jnz  Bucla
```

also semantic equiv., 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. CALL and RET

A procedure call is done by using the **CALL** instruction (*direct* or *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 CS ] (only if the operand represents a FAR address)
A:  . . .        ⇔      push offset 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. If present, it frees also from the stack *n* bytes below the returning address.

RET may be far or near. As an effect we have:

```
                        B   dw   ?
    RET n               .    .    .
    (near return)  ⇔    pop  B
                        add  sp,n
                        jmp  B
                        B  dd  ?
RET n                       .    .    .
(far return)     ⇔      pop  word ptr B
                        pop  word ptr B+2
                        add  sp,n
                        jmp  B
```

Usually, CALL and RET are used in the following context

```
name  PROC
.    .    .
     ret  [n]
name  ENDP
.    .    .
CALL  name
```

PROC and ENDP were discussed in 3.3.4. The call and return are implicitly far or near, dependent on how procedure *name* is declared FAR or NEAR respectively.

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

```
call  bx    ;address taken from a register
call  vptr  ;address taken from a memory variable
```

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

- the name of a FAR or NEAR procedure
- the name of a register containing a NEAR address
- a NEAR or FAR memory address (as in the case of JMP).

4.4. STRING INSTRUCTIONS

* Data transfer instructions (**LODS, STOS** and **MOVS**)
* Instructions for data accessing and comparing (**SCAS** and **CMPS**)