LABORATORY NO. 4
THE ELEMENTS OF THE ASSEMBLY LANGUAGE AND THE FORMAT OF THE EXECUTABLE PROGRAMS

1. Objective of laboratory

The purpose of this lab is the presentation of the instruction format in assembly language, of the most important pseudo-instructions and the structure of the executable programs: .COM and .EXE.

2. Theoretical considerations

2.1. The elements of the assembly language TASM

2.1.1. The format of the instructions

An instruction may be represented on a line of maximum 128 characters, the general form being:

[<label>:] [<opcod>[<operatives>][;<comments>]]

where:

<label> is a name, maximum 31 characters (letters, numbers or special characters _,?,@,..), the first character being a letter or one of the special characters. Each label has a value attached and also a relative address in the segment where it belongs to.

<opcod> the mnemonic of the instruction.

<operatives> the operative (or operatives) associated with the instruction concordant to the syntax required for the instruction. It may be a constant, a symbol or expressions containing these.

<comments> a certain text forego of the character “;” .

The insertion of blank lines and of certain number of spaces is allowed. These facilities are used for assuring the legibility of the program.

2.1.2 The specification of constants

Numerical constants – are presented through a row of numbers, the first being between 0 and 9 (if for example the number is in hexadecimal and starts with a character, a 0 will be put in front of its). The basis of the number is specified through a letter at the end of the number (B for binary, Q for octal, D for decimal, H for hexadecimal; without an explicit specification, the number is considered decimal).

Examples: 010010100B, 26157Q (octal), 7362D (or 7362), 0AB3H.

Character constants or rows of characters are specified between quotation (“ “) or apostrophes (‘ ‘).

Examples: “row of characters”, ‘row of characters’

2.1.3. Symbols

The symbols represent memory locations. These can be: labels or variables. Any symbol has the following attributes:

- the segment where it is defined
- the offset (the relative address in the segment)
- the type of the symbol (belongs to definition)

2.1.4. Labels

The labels may be defined only in the code part of the program and then can be used as arguments of CALL or JMP instructions.

The attributes of labels are:

- the segment (generally stored in CS) is the start address the segment. When a reference is made to the label, the value is found in CS (the effective value is known only during runtime)
- the offset is the distance in btes of the label beside the start of the segment where it has been defined
- the type determines the reference manner of the label; there are two types: NEAR and FAR. The NEAR type reference is offset ONLY, the FAR type reference specifies also the segment and offset (segment: offset).

  The labels are defined at the beginning of the source line. If a label is followed by “:” character then the label is of NEAR type.

2.1.5. Variables

The definition of variables (data labels) may be made with space booking instructions.

The attributes of variables are:

- segment and offset – similarly to labels with the distinction that there may be other ledger segments
- the type – is a constant, which shows the length (in octets) of the booked zone:

BYTE (1), WORD (2), DWORD (4), QWORD (8), TWORD (10), STRUC (defined by the user), RECORD (2).

Examples:

```
DAT         DB    0FH, 07H     ; occupies one octet each, totally 2
DATW        LABEL WORD         ; label for type conversion

MOV         AL,DAT             ; AL<-0FH
MOV         AX,DATW            ; AL<-0FH, AH<-07H
MOV         AX,DAT             ; type error
```

2.1.6. Expressions

The expressions are defined through constants, symbols, pseudo-operatives and operatives (for variables are considered only the address and not the content, because when compiling, only the address is known).

2.1.7. Operators (in the order of priorities)

1. Brackets () []

. (dot) - structure_name.variable – serves for binding the name of a structure with its elements

LENGTH – number of elements in memory

SIZE – the memory length in bytes

WIDTH – a field’s width from a RECORD

Example:

```
EXP   DW    100 DUP (1)
```

Then:

LENGTH EXP has the value 100

TYPE EXP has the value 2

SIZE EXP has the value 200

2. segment name: - explicit segment reference

Example:

```
MOV AX, ES:[BX]
```

3. PTR – redefinition of variable type

Example:

```
DAT   DB    03
MOV   AX, WORD PTR DAT
```

OFFSET – furnishes the offset of a symbol

SEG – furnishes the segment of a symbol

TYPE – a variable type

THIS – creation of an attributed operative (segment, offset, type) date

Example:

```
SIRC  DW    100   DUP(?)
SIRO  EQU   THIS BYTE
```

SIRC is a defined of 100 WORDS (200 byte in length); the variable SIRO has the same segment and offset as SIRC but it is of BYTE type.

4. HIGH – addresses the high part of a word

LOW – addresses the low part of a word

Example:

```
DAT   DW    2345H
MOV AH, HIGH DAT        ; AH<-23
```

5. * / MOD

Example:

```
MOV CX, (TYPE EXP)*(LENGTH EXP)
```

6. + -
7. EQ, NE, LE, LT, GE, GT
8. NOT –logic operative
9. AND
10. or, xor
11. SHORT – forces the short appeal

Example:

```
JMP   label          ; direct jump
JMP   SHORT label  ; IP is relative
```

2.1.8. Pseudo instructions

Pseudo-instructions are commands (orders, instructions) for assembler, necessary for the proper translations of the program and for the facility of the computer programmer's activity.

Only the pseudo-instructions indispensable in writing the first programs are shown.

2.1.9. Pseudo-instructions work with segments

Any segment is identified with a name and class, both specified by the user. When defined, the segments receive a series of attributes, which specifies for the assembler and for the link-editor the relations between segments.

The segments definition are made through:

segment_name SEGMENT [align_type] [combine type] ['class']
... ...
segment_name ENDS

where:

segment_name – is the segment's name chosen by the user (the name is associated with a value, corresponding to the segment's position in the memory).

align_type – is the segment's alignment type (in memory). The values, which it may take, are:

PARA (paragraph alignment, 16 octets multiple)
BYTE (octet alignment)
WORD (word alignment)
PAGE (page alignment – 256 octets multiple)

combine_type – is actually the segment's type and represents an information for the link-editor specifying the connection of segments with the same type. It may be:

PUBLIC – specifies the concatenation

COMMON – specifies the overlap

AT expression – specifies the segment's load having the address expression *16

STACK – shows that the current segment makes part of pile segment

MEMORY – specifies the segment's location as the last segment from the program

'class' – is the segment's class; the link-editor continually arranges the segments having the same class in order of its appearance. It is recommended to use the ‘code’, ‘data’, ‘constant’, ‘memory’, ‘stack’ classes.

2.1.10. The designation of the active segment

In a program may be defined more segments (code and date). The assembler verifies whether the dates or the instructions addressed may be reached with the segment register having a certain content. For a realization in proper conditions, the assembler of the active segment must be communicated, meaning that the segment register must contain the address of the loaded segment.

ASSUME <reg-seg>:<name-seg>, <reg-seg>:<name-seg> ...

reg-seg – the register segment

name-seg – the segment which will be active with the proper register segment

Example:

ASSUME CS:prg, DS:date1, ES:date2

Observations:

- the pseudo-instruction does not prepare the register segment but communicates to the assembler where the symbols must be looked for
- DS is recommended to be shown at the beginning of the assembler with a typical sequence:

```
ASSUME DS:name_seg_date
MOV   AX, name_seg_date
MOV DS, AX
```

- CS must not be initialized but must be activated with ASSUME before the first label
- instead of name-seg from ASSUME the NOTHING identifier may be used if we don’t want to associate a segment to the register.

2.1.11. The Memory reservation

Usually the data is defined in a data segment. The instruction definition has the syntax:

<name> <type> [expression list] [<factor> DUP (<expression list>)]

where:

name – is the symbol’s name

type - is the symbol’s type:

DB – for byte reservation
DW – for word reservation (2 octets)
DD – for double word reservation (4 octets)
DQ – for quadruple word reservation (8 octets)
DT – for 10 byte reservation

expressions list – list of expressions, that can be evaluated, replaced by a constant at assembly time. Memory locations will be initialized with these constants. The “?” can be use as placeholder, no initial value
factor – a constant, which shows how many times the expression, is repeated after DUP:
Examples:

```
DAT     db      45
dat1    db      45h, ‘a’, ‘A”, 85h
dat2    db      ‘abcdefghi’     ; the text is generated
lg_dat2 db      $-dat2          ; the length of the given row dat2 ($ is the local current
                                  counter)
aa      db      100 dup(56h)    ; 100 octets having the value 56h
bb      db      20 dup (?)      ; 20 not initialized octets
ad      dw      dat1            ; contains the address (offset) of the given variable dat1
adr     dd      dat1            ; contains the address  (offset + segment) of given
                                  variable dat1
```

2.1.12. Other possibilities for defining symbols

- the definition of constants:

<name> EQU <expression>

The symbol “name” will be replaced with the value’s expression.

- labels declaration:

<name> LABEL <type>

<name> label will have the value of the segment where it is defined, the offset equal to the offset of the first instruction or memory location which follows and the type defined by the <type> which may be: BYTE, WORD, DWORD, QWORD, TBYTE, the name of a structure, NEAR or FAR.

Example: if we have the definitions

```
ENTRY         LABEL FAR
ENTRY1:
```

then:

```
JMP ENTRY   ; is FAR type jump
JMP ENTRY1; is NEAR type jump
```

2.1.13. Current Location Counter managment

ORG <expression> ; the CLC will be changed to the expression's value
Example:
ORG 100h ; counter at 100h
ORG $+2 ; skip 2 octets ($ is the current value of the CLC)

2.1.14. The definition of the procedure

A procedure may be defined as a sequence of instructions which ends with RET instructions and is reached with CALL. The definition is made with the sequence:

<procedure_name> PROC <[NEAR], FAR>
... the procedure's instructions
< procedure_name > ENDP

Example:
; DBADD procedure, which at (DX:AX) adds (CX:BX) with the result in (DX:AX)

```
DBADD       PROC  NEAR
        ADD AX,BX  ; add word LOW
        ADC DX,CX  ; add word HIGH with CARRY
            RET
DBADD       ENDP
```

The call is made with CALL DBADD from the same segment. From other segments the procedure is invisible.

Observations:

- no procedure may be called both with FAR and NEAR CALL. This function is established very carefully when projecting the programs (the solution for declaring all procedures as FAR is apparently simple but totally non-economic).
- It is possible to declare imbricated and overlapping procedure

2.2. The program's structure in assembly language

2.2.1. .COM programs

- The program contains only one segment, so the code and date may have, on the whole, maximum 64Ko; because of this the references are relatively made at the address from the beginning of the segment.
- The source program must begin with ORG 100H pseudo-instruction to keep space for PSP Program Segment Prfefix).
- Data may be put anywhere in the program, but it is recommended to be put at the beginning (great care must be paid not to execute by mistake the data,

- It is not necessary to initialize of segment registers, all are loaded with the common value from CS.
- Return to OS is done by calling system function INT 21H having the parameter in AX 4C00H.

2.2.2. Model for .COM programs

```
COMMENT *
        the presentation of the program
*

CODE SEGMENET PARA PUBLIC 'CODE'
        ASSUME CS:CODE, DS:CODE, ES:CODE
        ORG 100H
START:
        JMP ENTRY
;************** define your data here
ENTRY:
;************** program's instructions
        MOV AH,4CH
        INT 21H         ; exit to operating system
CODE ENDS
        END START
```

2.2.3. .EXE programs

- The programs may have several segments.
- For the correct execution, the user must explicitly initialize DS, ES and SS registers.
- It is recommended that the .EXE programs be conceived as a FAR type procedure (in order to be able to return to OS ore other application) Because of this, at the beginning of the program, through the sequence:

```
        PUSH DS
        XOR AX,AX
        PUSH AX
```

The stack is prepared to return to OS through a far RET at the end of the program

2.2.4. Model for .EXE program

```
COMMENT * identification information for the program, author, data,
program's function, utilization *
;-------------------------------------------------------------------------------------
;  EXTERN  section
; the declaration of extern variables
;-------------------------------------------------------------------------------------

;-------------------------------------------------------------------------------------
; PUBLIC section
; the list of GLOBALE'S  variables defined in this file
-------------------------------------------------------------------------------------

;-------------------------------------------------------------------------------------
; CONSTANTE'S section
; The definitions of constants, including INCLUDE instructions, which read
; constant definitions
;-------------------------------------------------------------------------------------

;-------------------------------------------------------------------------------------
; MACRO section
; Macro definitions, structures, recordings and/or INCLUDE instructions
which
; read such definitions
;-------------------------------------------------------------------------------------

;-------------------------------------------------------------------------------------
; DATA section
; data definitions
;-------------------------------------------------------------------------------------

DATA SEGMENT PARA PUBLIC 'DATA'
;... ... define your data here
DATA ENDS

; more... ... other data segments if needed
```

```
;--------------------------------------------------------------------------------------
; STACK section
;--------------------------------------------------------------------------------------

STACK       SEGMENT PARA STACK 'STACK'
            DW STACK_SIZE    DUP (?) ; the pile will have 256 words
     STACK_START LABEL WORD     ; the top of the pile
STACK       ENDS

;--------------------------------------------------------------------------------------
; CODE section
;--------------------------------------------------------------------------------------
CODE        SEGMENT PARA PUBLIC 'CODE'
START       PROC FAR
            ASSUME CS:CODE, DS:DATA
            PUSH DS
            XOR AX,AX
            PUSH AX      ; the initialization for the returning
            MOV AX,DATA
            MOV DS, AX ; the initialization of DS date segment
;--------------------------------------------------------------------------------------
;... ... the main program's instructions your code
;--------------------------------------------------------------------------------------
            RET          ; return to OS
START       ENDP
;--------------------------------------------------------------------------------------
; PROCEDURES
; other procedures from the main program
;--------------------------------------------------------------------------------------

CODE        ENDS

;... ... other code segment if needed

;--------------------------------------------------------------------------------------
; the memory's segment section
;--------------------------------------------------------------------------------------

MEMORY      SEGMENT PARA MEMORY 'MEMORY'
;... ...  programs at high addresses
;... ... the definition of the memory's margins of the program
```

```
MEMORY      ENDS

        END START
```

2.3. Example of program in assembly language

The program calculates the sum of a row of numbers at SIR address and length specified in LGSIR variable; the result will be put in SUM location.

The first source program will be in the .COM type

```
CODE SEGMENT PARA PUBLIC 'CODE'
        ASSUME CS:CODE, DS:CODE
        ORG 100H

START: JMP ENTRY

SIR             DB 1,2,3,4
LGSIR           DB $-SIR
SUM             DB 0

ENTRY:
        MOV CH,0
        MOV CL,LGSIR        ; in CX is the length's row
        MOV AL,0    ; the initialization of the register where the sum is
                            ; calculated
        MOV SI,0            ; the index's initialization
NEXT:
        ADD AL,SIR[SI]      ; the add of the current element
        INC SI              ; passing at the next element in the row
        LOOP NEXT           ; CX decrementing and jump to next
                            ; element if CX differs from 0
        MOV SUM,AL
; end of program
        MOV AX,4C00h
        INT 21H
CODE ENDS
END     START
```

3. Lab tasks

- Study the presented example.
- Assemble, link and trace the given example
- Use Turbo Debugger to inspect content of registers and memory (SUM location).
- Rewrite the example in .EXE
- Make symbolic trace and debug
- Modify the code to add an array of words not bytes
- Modify the code to keep the sum in a double size location than the added values