LAB WORK NO. 1
CONVERSION AND OPERATIONS IN DIFFERENT NUMERATION BASES

1. Objective of the lab work

The purpose of this lab is to understand the way an integer or decimal number is converted from one base to another. This paper will focus on converting integer and decimal numbers from base 10 to a random base, especially bases 16, 2 and 8, and also on the reverse conversion, from a random base to base 10, especially from bases 16, 2 and 8. Direct conversion from base 16 to bases 2 or 8, and vice versa, will also be studied. Simple operations (add, subtract) in different numeration bases will be presented later.

2. Theoretical considerations

A numeration system consists of all the number representation rules by means of certain symbols, called digits.

For any numeration system, the number of distinct symbols for the system's digits is equal with the base (b). So, if the base b=2 (the numbers are written in binary), the symbols will be the 0 and 1 digits. For base b=16 (hexadecimal) the symbols will be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. One can notice that for numbers written in a base higher than 10, other symbols (letters) are used besides the common digits for base 10. Therefore, in the case of numbers written in hexadecimal, the letters A, B, C, D, E, and F have the associated values of 10, 11, 12, 13, 14, and 15.

To make an easier distinction between numbers written in a certain base, at the end of the number another letter is written, which represents the base, for instance:

B for numbers written in binary (base 2)
Q for numbers written in octal (base 8)
D for numbers written in decimal (base 10)
H for numbers written in hexadecimal (base 16)

Usually, numbers written in decimal don't necessarily have to be followed by the „D" symbol, because this base is considered to be implicit.

There are others ways to write a number, like writing the base at the end of the number in brackets, for instance: 100101001 (2) or 17A6B (16).

If a number in a certain base „b" is given, as an integer and a decimal part:

$Nr\ (b) = C_n\, C_{n-1}\, C_{n-2} \ldots C_2 C_1\, C_0\, ,\, D_1 D_2 D_2 D_3 \ldots$,

than its value in base 10 will be:

$$Nr\ (10) = C_n * b^n + C_{n-1} * b^{n-1} + \ldots + C_2 * b^2 + C_1 * b^1 + C_0 * b^0 + \\ + D_1 * b^{-1} + D_2 * b^{-2} + D_3 * b^{-3} + \ldots$$

2.1. Number conversion from base 10 to a random base

First of all, it must be pointed out that for converting a number that consists of both integer and decimal part, both parts have to be converted separately.

2.1.1. Converting the integer part

The simplest algorithm consists of successively dividing the number written in decimal to the base that the conversion will be made to (the number is divided by the base, then the quotient is divided by the base and so on, until the quotient becomes 0), after which the rests that were obtained are taken, reversing the order they appeared in, which represents the value of the number in that base.

Examples:

- Converting the integer numbers 347 and 438 from base 10 to bases 16, 2 and 8.

First, the numbers will be converted to base 16 because this operation is done with fewer divisions than the conversions to bases 2 and 8.

347	16		
32	21	16	
27	16	1	16
16	**5**	0	0
11	←	**1**	

(which is „**B**"?)

Therefore, reversing the order of the rests we have 15B (H).

Further on, one can convert to bases 2 and 8 in the same manner, but there is also a quicker way to convert numbers between bases 2, 8 and 16, given the fact that for each hexadecimal digit there is a corresponding 4 binary digit number, and that for each digit in octal there are 3 corresponding binary digits, as seen in the table below:

Value in decimal	Value in hexadecimal	Binary number corresponding to hexadecimal	Binary number corresponding to octal
0	0	0000	000
1	1	0001	001
2	2	0010	010
3	3	0011	011
4	4	0100	100
5	5	0101	101
6	6	0110	110
7	7	0111	111
8	8	1000	
9	9	1001	
10	A	1010	
11	B	1011	
12	C	1100	
13	D	1101	
14	E	1110	
15	F	1111	

There also has to be considered that when a number is passed between bases 2, 8, 16, grouping the digits is done „from the comma towards the extremities", meaning that for integer numbers it is done from the right to the left (by completing the number with zeros on the left side in case this is required, that is on the side that doesn't affect its value), and for decimal numbers the grouping is done after the comma, from the left side to the right, by filling in with zeros at the right side of the number.

Example:

- 347 (D) = 15B (H) = 1 0101 1011 (B) = 533 (O)
- 438 (D) = 1B6 (H) = 1 1011 0110 (B) = 666 (O)

2.1.2. Converting the decimal part

To convert a sub-unitary number (that is the fractional part of a number) from base 10 to a random base, a series of successive multiplications of the fractional parts has to be done, until the fractional part becomes null, a period is found or the representation capacity is surpassed (a sufficient number of digits has been obtained, although the algorithm hasn't finished). The digit that surpasses the decimal part at each multiplication is a digit of that number, in the base that the conversion is made to.

As an example, it is easy to use the following scheme, in which the digits of the representation are more clearly represented with the help of the two lines. Also, the position of the comma is more obvious (the numbers below the first multiplication, that is the ones under the line, are positioned after the comma). One must notice that only what is on the right side of the comma is multiplied.

Example:

- Conversion of the number 0, 47 (D) to binary, octal and hexadecimal:

0,	47*2
0	94
1	88
1	76
1	52
1	04
0	08
0	16
0	32
0	64
1	28
0	56
1	12
0	24
0	48
0	96
1	. . .

So: 0, 47 (D) = 0, 0111 1000 0101 0001 (B) = 0, 7851 (H) = 0, 3605 (O)

Converting a number that has both an integer and a decimal part is done by successively converting the integer part and the decimal one.

Example:

- Representation in bases 2 and 16 of the real number 14, 75

We obtain: 14 (D) = 1110 (B) = E (H)
and 0, 75 (D) = 0, 11 (B) = C (H).
Therefore 14, 75 (D) = 1110, 11 (B) = E, C (H)

2.2. Converting a number from a random base to base 10

For converting a number from a random base to base 10 one can use the formula that was defined in the first part of this lab, meaning that if a number is written in a random base „b" as an integer and decimal part:

$Nr(b) = C_n C_{n-1} C_{n-2} \ldots C_2 C_1 C_0 , D_1 D_2 D_2 D_3 \ldots$

than its value in base 10 will be:

$$Nr(10) = C_n * b^n + C_{n-1} * b^{n-1} + \ldots + C_2 * b^2 + C_1 * b^1 + C_0 * b^0 + D_1 * b^{-1} + D_2 * b^{-2} + D_3 * b^{-3} + \ldots$$

Example:

- The number 3A8 (H) is given in hexadecimal and its value in decimal is required:
  $N = 3*16^2 + 10*16^1 + 8 = 3*256 + 160 + 8 = 936$ (D)

- The fractional number 0, 341 (Q) written in base 8 is given and its value in decimal is required:
  $N = 3*8^{-1} + 4*8^{-2} + 1*8^{-3} = 3/8 + 4/64 + 1/512 = 0.4394$ (D)

- The binary number 110, 11 (B) is given and its values in hexadecimal and decimal are required:
  N = 110, 11 (B) = 6, C (H) = 6, 75 (D)

2.3. Simple operations with numbers written in different bases

Further, the adding and subtracting of integer unsigned numbers written in binary, octal and hexadecimal is presented.

2.3.1. Addition

Adding is done by the same rules that exist in decimal, with the remark that the highest digit in a base „b" will be b-1 (9 in decimal, 7 in octal, 1 in binary and F in hexadecimal). Therefore, when a result higher

than b-1 is obtained by adding two digits of rank „i", a carry will appear to the digit having the „i"+1 rank. On the position ranked „i" will be the rest of the division of the result obtained by adding the rank „i" digits, by the base. The carry to the „i"+1 rank digit will become a new unit to the adding of the rank „i"+1 digit, meaning that a 1 will be added (the carry).

Examples:

1111 1	11	11
01010110 (B)+	1364 (Q)+	6D8A32 (H)+
10110101 (B)	3721 (Q)	33E4C8 (H)
100001011 (B)	5305 (Q)	A16EFA (H)

The 1 unit carry to the higher ranked digit was marked by writing a 1 above the superior ranked digit to which the carry was made. The adding operation in binary is useful for representing a number in complement by 2 , when one chooses to add a 1 to the number's representation in complement by 1 (see lab about data representation).

Examples:

- The students have to add the 2 integer numbers 347 (D) and 438 (D) that were converted earlier in the lab to bases 16 and 8, and to check the result by converting it to base 10.

347 (D)+ 438 (D) = 785 (D)
15B (H) + 1B6 (H) = 311 (H). Checking: 311 (H) = 3*256+1*16+1 = 785
533 (Q) + 666 (Q) = 1421 (Q). Checking:1421 (Q) = 1*512+4*64+2*8+1 = 785

2.3.2. Subtraction

The subtraction rules in decimal are also used for other bases: if one can't subtract two digits of rank „i" (if the minuend's digit is lower than the subtrahends), then a unit is „borrowed" from the next ranked digit („i"+1). If the digit from which borrowing is desired is 0, than one borrows further from the next ranked digit.

Examples:

. .	. .
01011010 (B) –	A3D4 (H) –
01001100 (B)	751B (H)
00001110 (B)	2EB9 (H)

- The students have to subtract the two integer numbers 347 (D) and 438 (D) that were converted earlier in the lab to the numeration bases 16 and 8, and to check the result by converting it to decimal.

438 – 347 = 91 (D)
1B6 (H) – 15B (H) = 5B (H). Checking: 5B (H) = 5*16+11 = 91
666 (Q) – 533 (Q) = 133 (Q). Checking: 133 (Q) = 1*64+3*8+3 = 91

Subtraction is useful when it is desired to represent numbers in complement by 2 and a subtraction from $2^{no\_of\_reprez\_digits + 1}$ of the absolute value of the number is made.

3. Lab tasks

1. Convert integer numbers from base 10 to bases 2, 8 and 16
2. Convert integer numbers between bases 2, 8 and 16
3. Convert integer numbers from bases 2, 8 and 16 to base 10
4. Convert real numbers to bases 2 and 16
5. Add and subtract numbers in bases 2 and 16

Notes

LAB WORK NO. 2
THE INTERNAL DATA REPRESENTATION

1. Object of lab work

The purpose of this work is to understand the internal representation of different types of data in the computer. We will study and illustrate different ways of representing integer numbers (in magnitude and sign (MS), in complement to 1 (C1), complement to 2 (C2), in binary, decimal, packed and unpacked (BCD)), and real numbers (the IEEE short, long and temporary format)

2. Theoretical considerations

2.1 The representation of the integers in magnitude and sign (MS) in Complement to 1 and 2 (C1, C2)

Integer numbers can be represented on byte, on word (2 bytes) double words (4 bytes) or quadwords (8 bytes). For all representations, the most significant bit represents the sign bit, and the rest of the representation (the other bits) are used for representing in binary the number (the negative numbers have a different representation in the 3 representation forms)

There are two parts in representing the whole numbers: the sign bit and the absolute value. In all the tree forms of representation if the sign bit is 0 it represents a positive numbers and 1 in the sign bit represents negative numbers.

The field for the absolute value is represented like this:

- In the magnitude and sign (MS) representation, the module of the number is represented, so a number is represented putting 0 or 1 on the sign byte, according to the positive or negative value of the number and in the rest of the representation it is going to be used the value of the module in binary.
- In the complement to 1 (C1) representation, if the number is positive, the representation is identical as in magnitude and sign, the module of the number is represented and the sign byte is implicit 0. If the number

is negative then all the representation bits of the number in absolute value are completed, the 1→0 and 0→1. The sign bit will be 1.

- In C2 representation, if the number is positive the representation is identical as in magnitude and sign and in complement on 1. If the number is negative, then the representation of the number in absolute value is complemented to 2, namely the representation of the module is subtracted from the value 2^{n+1} (n represents the number of bits to be represented, the sign bit will become 1). Another way of obtaining the representation in C2 of a negative number is by adding 1 to the representation in C1.

From the modes of representing the numbers in the three forms, results that the positive numbers have the same representation in magnitude and sign as in complement on 1 and complement on 2.

A greater attention must be accorded to the minimum space (the minimum number of bytes) on witch a number can be represented in the three modes of representation. For example when we want to find the minimum number of bytes on which the number 155 can be represented we must defer to the fact that for representing the module there is one byte less (the sign byte) from the representation space. In this case, even if the value of its module fits on an byte (155 = 9Bh), the number can not be represented on an byte in either of the three representation modes, because the sign byte must be represented separately, so at the interpolation of the 9Bh representation, the first byte being 1, the representation will be of a negative number in stud of the desired number. Conclusively for representing the number 155 there is going to be needed minimum 2 bytes (the representation is done on multiple of bytes), and the number will be represented like this: 009Bh, being positive in all the 3 representation modes.

Examples:

Represent on 4 bytes the following numbers: 157, 169, -157 and -169

157(D) = 1001 1101(B) = 9D(H)

So the representation in MS, C1 and C2 is 00 00 00 9D (H)

169(D) = 1010 1001(B) = A9(H)

So the representation in MS, C1 and C2 is: 00 00 00 A9 (H)

For –157, the module is represented firs (it has been computed above) and the result is:

MS: 1000 0000 0000 0000 0000 0000 1001 1101 (B) = 80 00 00 9D (H)

C1: 1111 1111 1111 1111 1111 1111 0110 0010 (B) = FF FF FF 62 (H)
C2: 1111 1111 1111 1111 1111 1111 0110 0011 (B) = FF FF FF 63 (H)
For –163 analogous:
MS: 80 00 00 A9 (H)
C1: FF FF FF 56 (H)
C2: FF FF FF 57 (H)

2.2. Representing the real numbers in the IEEE format

The IEEE standard of representing the real numbers proposes 3 modes of representing for real numbers:

- The short format on 4 bytes
- The long format on 8 bytes
- The temporary format on 10 bytes

The real numbers are represented in the short an long formats in the computer's memory, and the temporary format are found in loading the real numbers in the mathematic coprocessor.

All the three formats contain 3 parts:

Sign	Characteristic	**Mantissa**

- The sign bit S
- Characteristic C (on 8, 11, or 15 bits, for short, long and temporary format)
- Mantissa M (on 23, 52, or 64 bytes)

For each representation:
S is 0 if the number is positive and 1 if the number is negative.
Characteristic = Exponent + 7Fh (or 3FFh for long IEEE and 3FFFh for temporary format)
In order to compute mantissa, first the number is represented in binary. This representation is normalized and written in the following format:
NO = 1.<binary digits> * $2^{exponent}$.

For the IEEE short and long format, the mantissa is formed by the digits after the decimal point, so the first 1 before the decimal point is not represented in the mantissa. For the temporary format all the digits of the mantissa are represented (including the leading 1).

Examples:

Represent in the IEEE short and long format the number 17,6(D).

The integer and the fractional part are converted separately, the results are:

The integer part: 17(D) = 11(H) = 1 0001(B)

The fractional part: 0,6(D) = 0,(1001)(B) (to be observed that the number is periodic), has infinite number of bits.

So 17,6(D) = 10001,(1001)(B)

The number is normalized: 17,6(D) = 10001,(1001)(B) = 1,0001(1001) * 2^4 (instead of 2^4 it would be more correct to have 10^{100}(B) because the notation was in binary, the fact that the characteristic is easer to calculate in hex then in binary can be a motivated excuse.

From this representation we can deduce the mantissa (the part after the decimal point, without that 1 before the point which is not represented by convention):

M = 0001(1001)(B).

After that the characteristic is calculated: C = exponent + 7F(H) = 4+7F(H) = 83(H) = 1000 0011(B)

The bit for the sign will be written and we can write the representation:

0	1000 0011	00011001100110011001100
sign	charact.	mantissa

In order to write the representation in hex we will group 4 binary digits. Attention to the fact that grouping 4 digits will not correspond to the characteristic because of the sign bit that shifts a position. So the hex digits of the characteristic will not be found in the representation written in hex.

The result of the representation is: 41 8C CC CC(H).

In practice, a rounding will appear at the last bit, and the representation is:

41 8C CC CD(H).

Similarly we will represent –23,5(D):

23(D) = 17(H) = 1 0111(B)

0,5(D) = 0,1(B)

So 23,5(D) = 10111,1 = 1,01111 * 2^4 it results that M = 0111100000000000… (23 bytes)

Characteristic = 7F(H) + 4(H) = 83(H)
The sign bit becomes 1.
The representation directly in hex is C1 BC 00 00(H).

In the following t you have the representation of a number in IEEE short format and you have to find the real number that is represented.

Example:

Given the representation 43 04 33 33 (H), you have to calculate the decimal value of the number represented

The representation in binary:

0100 0011 0000 0100 0011 0011 0011 0011

From here we deduce that:
The sign is 0
The characteristic is C = 1000 0110 (B) = 86(H)
The exponent is 86(H) – 7F(H) = 7 (H)
Mantissa M = 0000 1000 0110 0110…
The number is No=1,mantissa * $2^{exponent}$ = 1,0000 1000 0110…*2^{7}=
= 1000 0100,00110011…= 128 + 4 + 0.125 + 0.0625 + … ≈
≈ 132,1875 which approximates 132,2.

2.3. The representation of the numbers in packed BCD and unpacked BCD (Binary Coded Decimal)

Beside the modes of representation of the integers in MS, C1 and C2, there is the representation in Packed BCD and unpacked BCD.

In the Packed BCD representation one decimal digit is represented on 4 bits, so there are 2 decimal digits in a byte.

In the Unpacked BCD representation unpacked one decimal digit is represented on one byte (so we put 0 on the first 4 bits).

This representation modes are used for a better readability of the numbers from the programmers point of view, even if this is done by losing part of the available memory space (for packed BCD, only values 0-9 are used on 4 bits, and for unpacked BCD 4 more bits are left unused).

In order to compute operations with numbers represented in BCD, there are additional instructions for correcting the result after addition, multiplication, that will be studied in labs regarding instructions for arithmetic operations.

Example:

The number 3912(D) is going to be represented in BCD

- packed: 39 12(H) on 2 bytes;
- unpacked: 03 09 01 02(H) on 4 bytes.

3. Lab tasks

- Represent in MS, C1 and C2 the +35, -127 and 0.
- Represent in IEEE short format two real numbers.
- Given a representation in IEEE short format, find the represented number.

LAB WORK NO. 3
TURBO DEBUGGER ENVIRONMENT

1. Objective of the lab work

The purpose of this lab is to be able to debug programs written in assembly language and general executables, using a debugging tool.

2. Theoretical considerations

2.1 Turbo Debugger Environment

Turbo Debugger environment allows the testing and tracing of any executable program (.exe or .com) and allows:

- displaying memory and register content;
- their modification;
- step by step program execution;
- program execution till encountering a breakpoint;
- instruction insert in assembly language;
- memory area dump and disassembly.

To launch the Debugger type:

td [options] [program_name [arguments]]

Parameters between square brackets are optional. “Program_name” parameter represents the program to debug. If there is no extension we suppose it is .exe. “Arguments” parameter represents the arguments (input parameters) of the program to debug. Turbo Debugger options must be placed in front of the name of the program to debug.

If no option is given, program name or argument, Turbo Debugger will load without any program and with default options.

Examples:

td –r prog1 a

will run the Debugger with –r option (remote debugging), will load “prog1” program with parameter “a”.

td prog2 –x

will start Turbo Debugger and will open “prog2” program with parameter “-x”.

Some of the most important options are:

- the possibility of launching the environment with a configuration file;
- different ways of display refresh;
- the possibility of process switching depending on “id”;
- recording the keys pressed;
- remote debugging;
- using the mouse;
- program debugging in Windows.

To find more detailed information about the option for launching the Turbo Debugger, use the “td /?” command, or consult the Help page from Turbo Debugger which refers to the command line options.

2.2. Turbo Debugger windows

2.2.1 Code window

In the code window you can see the disassembled instructions of the program. The title shows the processor on which the program runs. Line source numbers and the labels appear in front of the lines on which they will be used.

On the left side we can see the memory address of the instruction (ex. CS:0100, meaning at the address resulted from the Code Segment (CS) and the offset 0100h). It follows a hexadecimal code of variable length (representing the machine code in hexadecimal) of the instruction of which the mnemonic is on the next column.

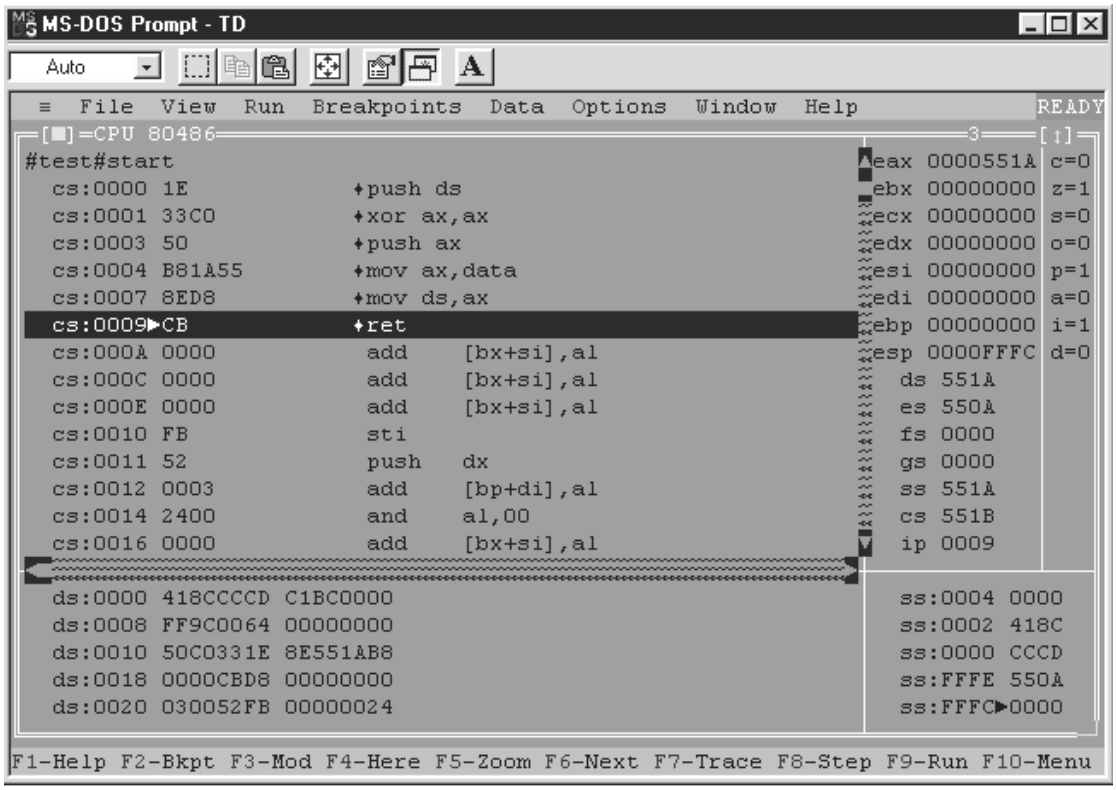
A distinguished sigh (an arrow), placed between the instruction address and its code (in case of an active CPU window, that line is coloured), and symbolizes the current instruction, the next instruction to be executed.

2.2.2. Register window

This window shows the processor registers. Their content is displayed in hexadecimal, in word size (2 bytes). The window local menu contains the following commands:

- Increment – allows adding the value 1 to the marked register content.

- Decrement – allows subtracting the value 1 from the marked register content.
- Zero – to set the register value on 0.
- Change – modifying the value of selected register.
- Registers 32-bit – allows changing the display mode of registers in 32 bits format (extended registers EAX, BAX, including segment registers FS and GS, etc.).



2.2.3. Flag window

This window shows the flag status (0 or 1). Every flag is indicated by a significant letter (C – Carry, Z – Zero, S – Sign, O – Overflow, P-Parity, A – BCD Carry (auxiliary carry), I – Interrupt, D - Direction).

The local menu of this window has only one command: Toggle – it switches the flags values 0 or 1 (to activate the command press “space” or “enter”).

2.2.4. Data window

This window shows, in hexadecimal, a part of the memory area from a segment. Window content displays the address (as "segment:offset") and the effective memory content.

Local menu of this window contains the following commands:

- GoTo – it sets the address on which the data display will begin in the data window. This address may be given in different ways:
  - offset (ex. 0100h), in this case current segment is taken by default;
  - segment:offset (ex. DS:0000h, or DS:0100h), and the positioning is effected to the absolute address given by segment and offset;
  - segment:offset with explicit values (ex. 54F7:0008h), the positioning will be effected to 54F7 segment and 8h offset;
  - variable_name (ex. No1), in case the assembly was done with debug information included.

- Search – searches a byte sequence in the memory.
- Next – searches the next appearance of the byte sequence given in Search command.
- Change – allows memory content modification from the current location by introducing a sequence of bytes.
- Follow – allows the positioning in the data window to a new address based on the number of bytes from the address specified by the current position (ex. code Near, Far, data offset, segment:offset).
- Previous – positions the window to the initial address, before modifying with GoTo or Follow.
- Display As – permits the choosing of the displaying mode of data in data window and has the options: Byte (1 byte), Word (2 bytes), Long (4 bytes), Comp (8 bytes), Float (real number, 4 bytes), Real (real number, 6 bytes), Double (real number, 8 bytes), Extended (10 bytes).
- Block – allows operating with memory blocks.

2.2.5. Stack window

Stack window shows the current content of the top of stack (last few elements) indicated by registers SS:SP.

Local menu of this window has the following commands:

- GoTo – allows the modification of the address from which the display is made in the window.
- Origin – it effects the return to the base address (SS:SP).
- Follow – allows setting the address from which the display is done as the value of the word selected from the stack. The command is useful when it is needed the following of the stack pointer content.
- Previous – positions the stack window to the initial address, before the modification with GoTo or Follow.
- Change – allows the modification of the stack content from the current location by inserting a new value (word).

2.3 Turbo Debugger Menus

In this part we will describe the most common part of Turbo Debugger menus.

- "File" menu has the following submenus:
  - "Open" – opens a dialog box for loading an executable file for debugging.
  - "Change dir" – allows the changing of current working directory (from where the file loading is made).
  - "Get info" – offers information about the loaded program and conventional memory status and EMS.
  - "DOS shell" – restores the control to the operating system, without closing Turbo Debugger (can go back using the command "exit").
  - "Resident" – allows quitting Turbo Debugger, with the possibility to have it resident in the memory (to activate it, use the command Ctrl-Break).
  - "Symbol load" – loads a symbol table specified by the user.
  - "Table relocate"- allows the relocation of the symbol table.
  - "Quit" – exits Turbo Debugger environment.

- "View" menu has the following submenu:
  - "Breakpoints" – allows the viewing of the breakpoints and their characteristics from the active program being debugged.
  - "Stack" – allows the viewing of stack window.
  - "Log" – opens log window (which has also a submenu).
  - "Watches" – opens a window for variable watch – the value of the variables you want to follow. With the local submenu, variables can be added, edited, deleted, as well as their content.
  - "Variables" – opens the visualization window of the variables defined in the program. The window contains a submenu to offer more operations on the variables.
  - "Module" – allows the selection for visualization of one of the program modules loaded from a list.
  - "File" – loads a file for visualization (the window can be used for viewing the source file, being the initial file of the executable file to debug).
  - "CPU" – opens the CPU window, referred in "Turbo Debugger Windows".
  - "Dump" – opens data window for displaying memory content (see "Turbo Debugger Windows" chapter). The window contains many commands in the local submenu.
  - "Registers" – opens the window for displaying and modifying register content (see "Turbo Debugger Windows" chapter). The window contains many commands in the local submenu.
  - "Numeric processor" – opens the math co-processor window, displaying the internal stack, as well as the flags. This window is used for debugging the programs that are using co-processor dedicated instructions.
  - "Execution history window" – opens the window that shows the last instructions executed by the central unit. Previous executed instructions are stored only when running with Trace command (from Run menu). If the tracing is made in a module visualization window, option "Full history" must be set on "yes". Local submenu has the following commands:
    - "Inspect" – opens a module visualization window in which it can be seen the source code of the instruction selected from those being executed.
    - "Reverse executed" – executes backwards the instructions from the current one to the selected one. Excepting the I/O

instructions, the status is the same as the one before the execution of the instruction up to which the return is made.
    - "Full history" – it is a switch that allows a slower and more complete way, or faster and less complete way of reverse execution.
  - "Hierarchy" – opens a window useful for debugging C++ or Pascal program that contain objects.
  - "Windows messages" – opens a message window when debugging programs under Windows.
  - "Another" – allows opening of another module, memory or file window (as described above).

- "Run" menu has the following submenu:
  - "Run" (F9) – runs the program till meeting a breakpoint, till breaking the program by the user with break keys, or till the end of the program. If the program is stopped with break keys (usually Ctrl-Break) there can be examined registers and program status.
  - "Goto cursor" (F4) – runs the program till reaching the selected line from the source code (CPU window or module visualization window).
  - "Trace into" (F7) – executes one instruction or code line (it is used the most in program debugging).
  - "Step over" (F8) – executes one instruction or code line as "Trace into" command, with the specification that procedure calls are being executed in one step, so it is not entering in procedures with the debugging.
  - "Execute to" (Alt-F9) – runs the program and is stopping to a specified location within the program. The user will be asked to insert an address to which the program to stop.
  - "Until Return" (Alt-F8) – runs the program being debugged till the current procedure or function is finished (to the first "return"). The command is used when accidentally "Trace into" is used instead of "Step over" and it is entered by mistake into a procedure, or in case of a procedure debugging and it is wanted the execution of the rest of it without stopping.
  - "Animate" – similar to "Trace into", being repeated. Instructions are executed continuous till key pressing. The debugger changes his status to notice the execution changes. The user is being asked for the instruction execution rate.

- "Back trace" – remakes the status by backwards execution of the last executed instruction (undo).
- "Instruction trace" – executes one machine instruction. The command is used to trace a break call in CPU window, for tracing a function into a module, which does not contain debugging information.
- "Run Arguments" – allows arguments changing from the command line of the program being debugged. The command is used when a program is debugged and it needs one or more input parameters, not given (or erroneously given).
- "Program reset" (Ctrl-F2) – reloads the current program. The command is used when re-running of a program is wanted.

- "Breakpoints" menu has the following submenu:
  - "Toggle" (F2) – marks (on/off) a breakpoint on current instruction; in this point the program will stop at every run.
  - "At…" (Alt-F2) – marks a breakpoint to a specified address.
  - "Change Memory Global" – sets a breakpoint, which will change the value of a memory area.
  - "Expression True Global" – sets a breakpoint that will take action when an inserted expression becomes true.
  - "Hardware breakpoint" – sets a hardware breakpoint by his detailed specification in the afferent dialog box.
  - "Delete all" – deletes all declared breakpoints.

- "Data" menu has the following submenu:
  - "Inspect" – allows the inspection of some variable or references inserted in memory at request in the dialog box.
  - "Evaluate/Modify" – evaluates an arbitrary expression, allows variable names as well as formulas, and displays the result in decimal and hexadecimal.
  - "Add Watch" – adds an expression or a variable in the variable watch window.
  - "Function return" – allows the inspection of the value that will be returned by the current function.

- "Options" menu has the following submenu:
  - "Language" – allows the specification of the way Turbo Debugger interprets the expressions user inserted.
  - "Macros" – creates, modifies and deletes macros assigned to certain keys (ex. command sequences mostly used).
  - "Display Options" – opens a dialog box for setting the display mode on the display, how to display numbers, as well as the way in which the display refresh is made.
  - "Path for Source" – allows setting of the way where Turbo Debugger searches the source files that compose the program.
  - "Save Options" – opens a dialog box for selecting the configuration part to be saved, as well as the configuration file.
  - "Restore Options" – allows configuration loading from a configuration file previously saved with "Save Options" command.

- "Window" and "Help" menus are similar to any other application, so they won't be described in this laboratory.

3. Lab tasks

A short program will be edited, assembled, link-edited and debugged, using the most used commands and functions from Turbo Debugger menu.

For the beginning, the following program will be edited, using an editor like Notepad. The program only declares some variables in the data segment. The code segment contains only the instructions needed to activate the data segment. The program will be used especially for exemplifying the visualization mode of data in memory. The name of the program will be "test.asm".

```
;------------------------------------------------------------ COPY FROM HERE
DATA SEGMENT PARA PUBLIC 'DATA'
NO1  DD 17.6
NO2  DD -23.5
NO3  DW 100
NO4  DW -100
DATA ENDS

CODE SEGMENT PARA PUBLIC 'CODE'
ASSUME CS:CODE, DS:DATA
START     PROC FAR
          PUSH DS
          XOR  AX,AX
          PUSH AX
          MOV  AX,DATA
          MOV  DS,AX
;OTHER PROGRAM INSTRUCTIONS
          RET
START     ENDP
CODE ENDS
END  START
;------------------------------------------------------------COPY UNTILL HERE
```

The program will be assembled with the command:

Tasm /zi test.asm

This command will generate the object module test.obj (in case of successful assembling). If /la option (expanded listing) is used, an .lst file will be obtained. This text file will contain information about line numbers, relative address on which the instructions are assembled, machine code

resulted after the assembly, as well as the initial form of the instructions. At the end the symbol table will be listed in detail.

For additional details referring Turbo Assembler parameters run “tasm /?” command, or simpler, without parameters, only “tasm”.

Next is step is to link-edit the .obj module, only one for this simple example, but there could be several modules. To obtain the executable file, write the command:

Tlink /v test.obj

As a result of this command, if no errors occur, the file test.exe will be generated. Option /v is used for easier debugging of the program and signifies the inclusion of the information from the symbol table so that Turbo Debugger will list and use variables and labels instead of memory addresses. This option cannot be used in a .com program. To obtain a .com program, the option /t must be used (and the code must have been written to meet .com program conditions, e.g. to have only one segment, that starts at address 100h, etc.).

For additional details referring Turbo Link parameters, use “tlink /?” or “tlink” command.

For testing and debugging the program the command line is:

Td test.exe

This will launch the Turbo Debugger and the program test.exe will be automatically loaded. Without a parameter you could load your program from the file menu.

If the program has been assembled and link-edited with the debug options, the initial source code will show up. This can be debugged and ran by Turbo Debugger commands. For a better understanding of what is happening in the computers memory it is recommended to open the CPU window (View - CPU); it contains CPU windows, the registers, the flags, the stack and the data segment. You may modify ore insert new instructions, but it is not recommended. You can not save your work in an .asm file.

Now, tracing the program may commence. The most used commands will be: step by step run: Run – Trace (F7) which executes one instruction at every step, or establishing breakpoints: Breakpoints – Toggle and then running with Run – Run, or positioning the cursor on a specified

instruction and using Run – Go to cursor. To return to the beginning of the program, use Run – Program reset.

While tracing the program, at every step the registers, flags, stack, and data area values can be inspected or even modified as needed.

For data visualization directly in data segment, switch to the data window and position to the beginning address of the data segment – Go to (DS:0000h) command, the address can be also written directly using DS value (ex. 551A:0000h). Another way to position is writing variable name (only in case of debugging information is included at assembly). The variable values can be also seen in Data menu: Data – Inspect or Data – Evaluate/Modify (function that can be used later as a simple decimal – hexadecimal conversion tool).

Another possibility is the window Data – Add watch, which monitors the specified variable values.

Usually, data window is positioned on Byte status, meaning data is displayed byte by byte, and on the right their ASCII representation is shown. For our program, the data segment starts with no1 = 17.6 value (in short IEEE format, so on 4 bytes), no2=-23.5 (same format as no1) and it is continued with no3=100 (on 2 bytes in C2) and no4=-100 (2 bytes in C2).

To view no1 and no2 values, you will set the Display as – Float, to view their representation you will use Display as – Byte, the memory content (17.6 is represented as 418CCCCD, and –23.5 is represented as C1BC0000).

To view no3 and no4 you should display memory content from address ds:0008h (so, offset 2*4 bytes as much as those 2 values in short IEEE format representation); the visualization is done with Display as Word. You may notice that 100 is represented in memory as 0064h, and –100 as FF9Ch.

The program can be traced till end. Take care, some programs may block or end unexpected, not as the programmer wanted. You may see strange instructions executed, this is the result of the earlier mentioned error.

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

LABORATORY NO. 5
ARITHMETICAL, LOGICAL, ROTATE AND SHIFT INSTRUCTIONS FOR 80X86 MICROPROCESSOR

1. Object of laboratory

The purpose of this lab is to present the arithmetical, logical, rotate and shift instructions for x86 microprocessors.

2. Theoretical considerations

The 16 bits microprocessors of the x86 family have computing instructions to allow operations on 8 or 16 bits and to implement routines for multiple bytes or multiword operations. By computing we mean:

- arithmetical operations: add, subtract, multiply, divide, increment, decrement, complement of 2 and compare;
- logical operations: and, or, xor, complement of 1 and test;
- rotate and shift operations.

2.1. Arithmetical instructions

Arithmetical operations are using numbers in byte or word size, in unsigned or C2 representation. Add and subtract operations can also use operands of type BCD unpacked (one decimal digit per byte) or packed BCD (two decimal digits per byte). Multiply and divide operations can be used also for unpacked BCD. In the following table <s> and <d> represent the „source" operand and „destination" operand. Arithmetical instructions generally affect the following flags: AF, CF, OF, DF, PF, ZF. These flags are generally set according to the result of the instruction.

General form	The effect	Affected flags
ADD <d>, <s>	<d> ← {<d>}+{<s>}	AF,CF,OF,PF,SF,ZF
ADC <d>, <s>	<d> ← {<d>}+{<s>}+{CF}	AF,CF,OF,PF,SF,ZF
INC <d>	<d> ← {<d>}+1	AF,OF,PF,SF,ZF

AAA	Decimal correction after addition in unpacked BCD (implicit AL)	AF,CF unmodified OF,PF,SF,ZF undefined
DAA	Decimal correction after addition in packed BCD (implicit AL)	AF,CF,PF,SF,ZF modified OF undefined
SUB <d>, <s>	<d> ← {<d>}-{<s>}	AF,CF,OF,PF,SF,ZF
SBB <d>, <s>	<d> ← {<d>}-{<s>}-{CF}	AF,CF,OF,PF,SF,ZF
CMP <d>, <s>	Only flags are set according to d-s, result is not stored	AF,CF,OF,PF,SF,ZF
DEC <d>	<d> ← {<d>}–1	AF,OF,PF,SF,ZF
NEG <d>	<d> ← [0-{<d>}]	AF,CF,OF,PF,SF,ZF
AAS	Decimal correction after subtraction in unpacked BCD (implicit AL)	AF,CF modified OF,PF,SF,ZF undefined
DAS	Decimal correction after subtraction in packed BCD (implicit AL)	AF,CF,PF,ZF,SF modified OF undefined
CBW	Conversion from byte stored in AL to word stored in AX (sign extension)	---
CWD	Conversion from byte stored in AX to double word stored in DX, AX (sign extension)	---

MUL <s>	if <s> is a byte:
	AX ← (AL)*{<s>}
	if <s> is a word:
	DX, AX ← (AX)*{<s>}
	Operands are handled as unsigned integer
		CF,OF modified
		AF,PF,SF,ZF undefined
		If CF and OF are 1 then AH (resp. DX) store values different from 0

IMUL <s>	if <s> is a byte:
	AX ← (AL)*{<s>}
	if <s> is a word:
	DX:AX ← (AX)*{<s>}
	Operands are handled as signed integer.
		CF,OF modified
		AF,PF,SF,ZF undefined
		If CF and OF are 1 then AH (resp. DX) store values different from 0

AAM	Decimal correction after	SF,PF,ZF modified

	multiplication in BCD unpacked. MUL is used to multiply and then correction is set. AX stores the result.	OF,AF,CF undefined

DIV <s>	if <s> is a byte:
	AL ← [(AX)/{<s>}]
	AH ← (AX) mod {<s>}
	if <s> is a word:
	AX ← [(DX,AX)/{<s>}]
	DX ← (DX,AX) mod {<s>}
	Operands are handled as unsigned integer.
	If the quotient exceeds destination's capacity a level 0 interrupt will be generated.
		AF,CF,OF,PF,SF,ZF undefined

IDIV <s>	if <s> is a byte:
	AL ← [(AX)/{<s>}]
	AH ← (AX) mod {<s>}
	if <s> is a word:
	AX ← [(DX,AX)/{<s>}]
	DX ← (DX,AX) mod {<s>}
	Operands are handled as signed integer.
	If the quotient exceeds destination's capacity a level 0 interrupt will be generated.
		AF,CF,OF,PF,SF,ZF undefined

AAD	Decimal correction before a division in BCD unpacked. The correction is made and then DIV is used for division.	PF,SF,ZF modified AF,CF,OF undefined

The operands involved in addition or subtraction are unsigned integers or signed integers represented in C2. The developer of the program must choose how to represent the operands, how to evaluate the result properly and take efficient actions in case of overflow.

An incorrect result, for unsigned operands, can be checked by testing the value of the CF set by the operation. For signed operands the error can be checked by examining the value in the OF.

Overflow can be tested through conditional jump instructions JC, JNC, JO, JNO for handling errors.

Example:

```
DATA        SEGMENT
MEM8        DB      39
DATA        ENDS


CODE SEGMENT
;... ...                                    unsigned    signed
        MOV AL, 26  ;load al                    26        26
        INC AL      ;increment al               1         1
        ADD AL, 76  ;add immediate date         76        76
                    ;                           ----      ----
                    ;                           103       103
        ADD AL, MEM8;add memory                  39        39
                    ;                           ----      ----
                    ;                           142       -114+OF
        MOV AH, AL ;copy to ah                  142
        ADD AL, AH ;add  register               142
                    ;                           ----
                    ;                           28+CF
;... ...
CODE ENDS
```

For this example the add operation was on 8 bits. When the sum is over 127 the OF is written, when over 255 the CF is written.

Example:

```
DATA        SEGMENT
MEM8        DB      122
DATA        ENDS

CODE SEGMENT
;                                              UNSIGNED      SIGNED
;… …
        MOV AL, 95  ;load  al                  95            95
        DEC AL      ;decrement                 -1            -1
        SUB AL, 23  ;subtract immediate value  -23           -23
                                               ----          ----
                                               71            71
        SUB AL, MEM8        ;subtract memory   -122          -122
                                               ----          ----
                                               205+CF        -51
        MOV AH, 119;load ah
        SUB AL,AH   ;subtract register                       -119
                                                             ----
                                                             86+OF
;… …
CODE ENDS
```

The instructions ADC and SBB allow implementation for multi-byte or multi-word operations. They perform the action of ADD and SUB and also add or subtract the value of CF indicator.

Example:

```
DATA        SEGMENT
MEM32       DD      316423
DATA        ENDS

CODE SEGMENT
… …
        MOV AX, 43981
        SUB DX, DX                          ;load dx, ax          43981
        ADD  AX, WORD PTR MEM32[0] ;add inf. word
        ADC  DX, WORD PTR MEM32[2] ;add sup. word          316423
                                            ;                 --------
                                            ;result in dx:ax  360404
… …
```

```
CODE ENDS
```
Example:

```
DATA        SEGMENT
MEM32A      DD      316423
MEM32B      DD      156739
DATA        ENDS

CODE SEGMENT
… …
            MOV AX, WORD PTR MEM32A[0]      ;load inf. word
            MOV DX, WORD PTR MEM32A[2]      ; load sup. word
            SUB AX, WORD PTR MEM32B[0]      ;subtract inf. word
            SBB DX, WORD PTR MEM32B[2]      ; subtract sup. word
… …
CODE ENDS
```

MUL is used for multiplying unsigned numbers. IMUL is used for multiplying signed numbers. The syntaxes are :

```
MUL   {register | memory}
IMUL  {register | memory}
```

For multiplication one of the operands must be loaded in the accumulator register (AL for 8 bits operands and AX for 16 bits operands). This is an implicit register, it is not specified in the instruction. The information stored in this register will be destroyed by the result. The second operand must be specified as an operand in register or memory. This operand will not be destroyed by the operation unless it is DX, AH or AL. Multiplying two 8 bits numbers leads to a 16 bits result stored in AX. Multiplying two 16 bits numbers leads to a 32 bits result stored in DX, AX. For both cases if the high part of the result is 0 (for unsigned MUL) or it coincides with the sign extension (for IMUL in sign representation), the indicators CF and OF are set on 0; otherwise are set on 1. The other indicators have undefined values.

Example:

```
DATA        SEGMENT
MEM16       DW      –30000
DATA        ENDS
```

```
CODE SEGMENT
                    ;unsigned multiply on 8 bits
        MOV AL, 23  ;load al                    23
        MOV BL, 24  ;load bl                    24
        MUL BL      ;multiply with bl
                    ;                           ----
                    ;result in ax               552
                    ; CF and OF are set
                    ;multiply with sign on 16 bits
        MOV AX, 50  ;load ax                    50
        IMUL MEM16  ;multiply with mem.         -30000
                    ;                           --------
                    ;result in dx,ax            -1500000
                    ; CF and OF are set
… …
CODE ENDS
```

DIV instruction is used for dividing unsigned numbers; IDIV is used for signed values. The syntaxes are:

```
DIV   {register | memory}
IDIV  {register | memory}
```

In order to divide a 16 bits number by an 8 bits number the first operand is loaded in AX. The result overwrites the content of AX. If the divider is on 8 bits, register or memory location, after the division AL holds the quotient and AH the rest.

In order to divide a 32 bit number by a 16 bit number the first operand is loaded in the pair DX: AX. The information stored in DX and AX will be lost after the operation. After the division AX stores the quotient and DX the rest.

For dividing 2 numbers of equal length (8 or 16 bits) the first action is to convert to a double length (16 or 32 bits) the first operand. For unsigned numbers the conversion consists in deleting the upper byte of the first operand, register AH, and respectively the most significant word, register DX. For sign numbers conversion consists in sign extension and is obtained through CWB and CWD instructions.

If the divider is 0 or the quotient exceeds it's assigned register (AL or AX) then the processor generates a level 0 interruption. If this interruption is not handled by the developer the operating system will abandon the program. There are two methods for dealing with the situation: testing the

divider before the operation takes place and calling, when needed, a routine for handling errors; writing your own routine for handling errors to replace the routine for level 0 interruption.

Example:

```
DATA        SEGMENT
MEM16       DW    –2000
MEM32       DD    500000
DATA        ENDS

CODE SEGMENT
                        ; unsigned  division of a 16 bits operand
                        ;by an 8 bits operand
        MOV AX, 700     ;load  operand                  700
        MOV BL, 36      ;load divider                   36
        DIV BL          ;unsigned division
                        ;quotient is in al              19
                        ;rest is in ah                  16
                        ;
                        ;signed division of  a 32 bits operand
                        ;by a 16 bits operand
        MOV AX, WORD PTR MEM32[0]  ;load ax
        MOV DX, WORD PTR MEM32[2]  ; load dx            500000
        IDIV MEM16      ; signed division
                        ; quotient is in ax             -250
                        ; rest is in dx                 0
                        ; signed division of  a 16 bit operand
                        ; by a 16 bit operand

        MOV AX, WORD PTR MEM16    ; load operand        -2000
        CWD                       ;convert to double word
        MOV BX, -421              ; load divider        -421
        IDIV BX                   ; signed division
                                  ; quotient is in ax   4
                                  ; rest is in dx       -316
CODE ENDS
```

2.2. Operations in unpacked BCD:

The instruction set has 4 instructions for unpacked BCD or ASCII correction: AAA (ASCII Adjust after Addition), AAS (ASCII Adjust after Subtraction), AAM (Ascii Adjust after Multiplication) and AAD (ASCII Adjust **before** Division), these instruction will correct the result to unpacked BCD format.. Arithmetical operations are computed on byte size operands only. The result must be in AL register implicitly used by the adjust instructions. If an operation implies 2 one digit operands with a result of two digits, the adjust instruction for correction will place the least significant digit in AL, and the most significant in AH. If the result stored in AL generates carry to AH or needs to borrow from AH the flags CF and AF are set.

Example:

```
        ; unpacked BCD addition
MOV AX, 9       ;load ax                    0009h
MOV BX,3        ; load bx                   0003h
ADD AL, BL      ;addition                   000ch
AAA             ;adjust after addition      ax=0102h
                ; AF and CF are set
        ;unpacked BCD subtraction
MOV AX, 0103H   ;load ax                    0103h
MOV BX, 4       ; load bx                   0004h
SUB AL, BL      ;subtract                   01feh
AAS             ; adjust after
                ;subtraction                ax=0009h
                ;AF and CF are positioned

        ;unpacked  BCD multiplication
MOV AX, 0903H           ; load ax           0903h
MUL AH          ;unsigned multiplication    001bh
AAM             ; adjust after MUL          ax=0207h

        ;unpacked BCD division
MOV AX, 0205H   ; load ax with dividend     25 unpBCD
MOV BL, 02      ; load bl with divisor       2 unpBCD
AAD             ; adjust before
                ;division                   AX=0019H
DIV BL          ;unsigned division          result is 010CH
```

```
                ;quotient in al                          0CH
                ;rest in ah                              01H
AAM             ;adjust after
                ;division the quotient  ax=0102H  12unpBCD
                ;the rest is lost
```

The rest will be lost. If needed, it must be saved in a different register before adjusting the quotient. The rest can also be corrected. For this it should be moved in AL.

2.3. Operations in packed BCD

The instruction set has two instructions for decimal correction DAA (decimal adjust after addition) and DAS (decimal adjust after subtraction) which allow adding and subtracting in packed BCD. ADD and SUB instructions are used to add and subtract followed by appropriate instructions to adjust the result.

Arithmetical operations must be on byte size in order to store the result in AL.

Instructions for decimal corrections in packed BCD never affect AH register. AF indicator is positioned in case of carry or borrow from the least significant digit to the most significant one. CF indicator is positioned in case of carry or borrow to exterior.

Example:

```
        ;Adding in packed BCD
MOV AX, 8833H       ;load ax                    8833H
ADD AL, AH          ;add to al                  al=0BBH
DAA                 ;decimal adjust
                    ;after adding               al=021H
                    ; CF is set
                    ;the result is  121H = 121 pBCD
        ;Subtracting in packed BCD
MOV AX, 3883H       ;load ax                    3883H
SUB AL, AH          ;subtract                   al=04BH
DAS                 ;decimal adjust
                    ;after subtraction          al=045H
                    ; CF is 0
```

2.4. Logical instructions

General form	Effect	Affected conditioning indicators
AND <d>, <s>	<d> ← {<d>} and {<s>}	CF,OF,PF,SF,ZF set AF undefined
TEST <d>, <s>	The indicators are set as for AND but {<d>} does not change	CF,OF,PF,SF,ZF set AF unmodified
OR <d>, <s>	<d> ← {<d>} or {<s>}	CF,OF,PF,SF,ZF set AF undefined
XOR <d>, <s>	<d> ← {<d>} xor {<s>}	CF,OF,PF,SF,ZF set AF undefined
NOT <s>	<s> ← not <s> (complemental to 1)	----

Logical instructions operate on bits, over bits of same rank of two operands. There are 5 logic instructions: AND, TEST, OR, XOR and NOT.

The syntax:

AND {register | memory}, { register | memory | immediate date}
TEST { register | memory }, { register | memory | immediate date }
OR { register | memory }, { register | memory | immediate date }
XOR { register | memory }, { register | memory | immediate date }
NOT { register | memory }

Example:

```
        ;example for  AND
MOV AL, 35H       ;load al                  00110101
AND AL, 0FBH      ;and with immediate value 11111011
                  ;                         ------------
                  ;                         00110001
AND AL, 0F8H      ;                         11111000
                  ;                         ------------
                  ;                         00110000
```

```
        ;example for OR
MOV AL, 35H        ;load al                      00110101
OR AL, 08H         ;or with immediate value      00001000
                   ;                             ----------
        OR AL, 07H ; or with immediate value     00000111
                   ;                             ----------
                   ;                             00111111
        ;example for XOR
MOV AL, 35H        ;load al                      00110101
XOR AL, 08H        ;xor with immediate value     00001000
                   ;                             ----------
                   ;                             00111101

XOR AL, 07H        ; xor with immediate value    00000111
                   ;                             ----------
                   ;                             00111010
```

Logical instructions can be used to compare an operand with 0 (OR BX, BX instead of CMP BX, 00) or to initialize with 0 (XOR CX, CX; SUB CX, CX instead of MOV CX, 00) having a more compact form.

2.5. Shift and rotation instructions:

General form	Effect	Affected conditioning indicators
SHL <s>, 1 SAL <s>, 1	Logic shift to left CF will store the most significant bit that was shifted. If <CF> <> the initial sign OF becomes 1.	CF,OF,SF,ZF,PF AF undefined
SHL <s>, CL SAL <s>, CL	Logic shift to left with a number of positions indicated by CL. CF will store the last shifted bit.	CF,OF,SF,ZF,PF AF undefined
SHR <s>, 1	Logic shift to right. Zeroes are inserted. CF will store the most significant bit. If the most significant bits of the result are different OF becomes 1.	CF,OF,SF,ZF,PF AF undefined
SHR <s>, CL	Logic shift to right with a number of positions indicated by CL. CF will store the last shifted bit.	CF,OF,SF,ZF,PF AF undefined

SAR <s>, 1	Arithmetic shift to right. Sign extension. The least significant bit will be stored by CF. If the most significant bits of the result are different OF becomes 1.	CF,OF,SF,ZF,PF AF undefined
SAR <s>, CL	Arithmetic shift to right with a number of positions indicated by CL. CF will store the last shifted bit.	CF,OF,SF,ZF,PF AF undefined
ROL <s>, 1	Rotate left by carry. If CF <> sign then OF becomes 1	CF, OF
ROL <s>, CL	Rotate left by carry with a number of positions indicated by CL.	CF, OF
ROR <s>,1	Rotate right by carry. If (CF) <> sign OF becomes 1.	CF, OF
ROR <s>, CL	Rotate right by carry with a number of positions indicated by CL.	CF, OF
RCL <s>, 1	Rotate left with carry. If (CF) <> sign OF becomes 1.	CF, OF
RCL <s>, CL	Rotate left with carry with a number of positions indicated by CL.	CF, OF
RCR <s>, 1	Rotate right with carry. If (CF) <> sign OF becomes 1.	CF, OF
RCR <s>, CL	Rotate right with carry with a number of positions indicated by CL.	CF, OF

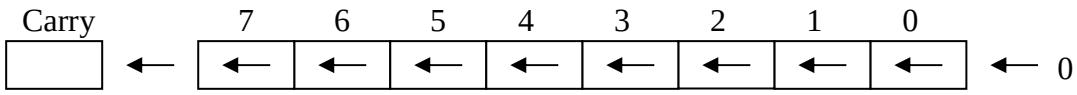
The format for all shift and rotate instructions is identical:

OPCODE {register | memory}, {CL | 1|nr}

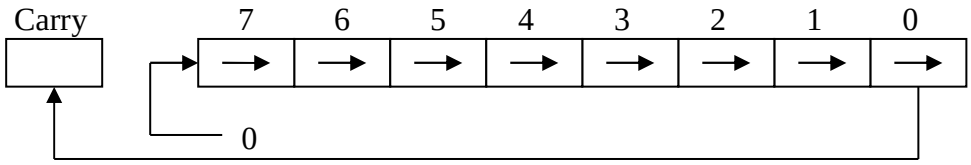
The result overwrites the source operand. The number of shift/rotate positions can be, number stored previously in register CL or nr for later processors.

The following figures show the result of these instructions on a byte operand for one position shifting/rotation.

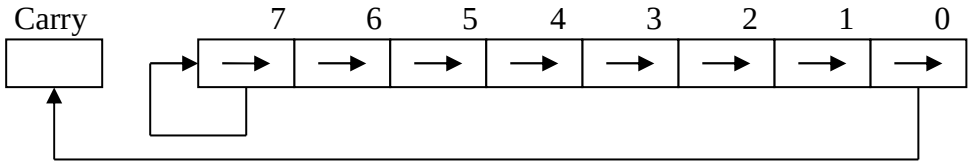
SHL, SAL



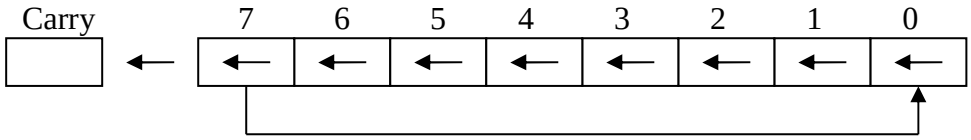
SHR



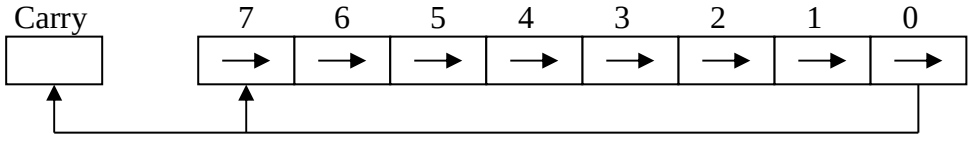
SAR



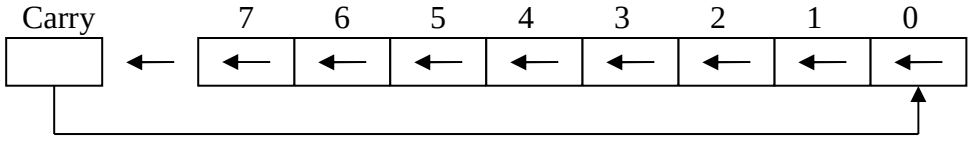
s ROL



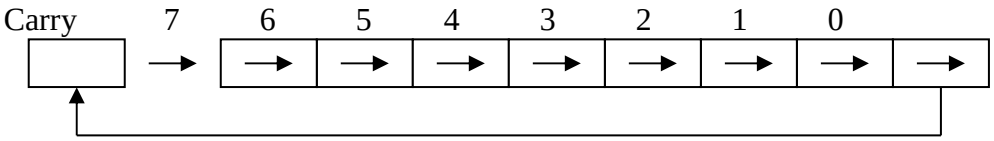
ROR



RCL



RCR



Example:

```
        ;a number stored in ax
        ;is multiplied by 10
SHL AX, 1           ;*2
MOV BX, AX          ;
SHL AX, 1           ;*4
SHL AX, 1           ;*8
ADD AX, BX          ;*10

        ;an unsigned number stored in ax
        ;is divided by 512
SHR AX, 1           ;/2
XCHG AH, AL         ;
XOR AH,AH                   ;/512

        ;a number stored in ax represented in C2 with sign
        ;is divided by 2
MOV AX, -16         ;
SAR AX, 1           ;/2

        ;a 32 bits unsigned number
        ;is divided by 2

DATA    SEGMENT
        MEM32 DD 500000
DATA    ENDS
CODE SEGMENT
… …
        SHR WORD PTR MEM32[2], 1    ;shifting in CF
        RCR WORD PTR MEM32[0], 1    ;rotation with CF
… …
CODE ENDS
```

3. Lab tasks

1. Study the examples.
2. Trace the examples with Turbo Debugger.
3. Write a program that generates an integer in byte representation and stores it to a REZ location after the formula:
   REZ = AL*NUM1+(NUM2*AL+BL)
   All parameters are byte size.
4. Implement the following operations using arithmetic and shift instructions:
   AX = 7*AX–2*BX–BX/8
   Parameters are byte size.
5. (complementary) Design an algorithm to multiply two 4 bytes numbers in C2 representation.

LABORATORY WORK NO. 6
DATA TRANSFER INSTRUCTIONS

1. Object of laboratory

Study of data transfer instructions for the I8086 microprocessor, including the input-output instructions.

2. Theoretical considerations

Data transfer is one of the most common tasks when programming in an assembly language. Data can be transferred between registers or between registers and the memory. Immediate data can be loaded to registers or to memory. The transfer can be done on bye, word or double word size. The two operands must have the same size. Data transfer instructions don't affect the flags (excepting the ones that have this purpose). They are classified as follows:

- „classical" data transfer instructions
- address transfer instructions
- flag transfer instructions
- input/output instructions.

2.1. „Classical" transfer instructions

Include the following instructions:

MOV <d>, <s>
XCHG <d>, <s>
XLAT
PUSH <s>
POP <d>

Data is copied from source to destination with the MOV instruction. The syntax of this is instruction is:

MOV {register | memory}, {register | memory | immediate data}

This instruction copies the source operand to the destination. Right after a MOV instruction is executed, the source operand and the destination have the same value. The old content of the destination is overwritten.

Example:

```
DATA    SEGMENT
MEM     LABEL BYTE          ;byte and
MEMW    DW ?                ;word
VCT  DB         100 DUP (?) ;vector
DATA ENDS
CODE SEGMENT
        ASSUME CS:CODE, DS:DATA
        … …
        MOV AX, 7           ;immediate data to register
        MOV MEM, 7          ;immediate byte to directly
                            ;addressed memory
        MOV MEMW, 7         ;immediate word to directly
                            ;addressed memory
        MOV VCT[BX], 7      ;immediate byte to indirectly
                            ;addressed memory
        MOV MEMW, DS        ;segment register to memory
        MOV MEMW, AX        ;general register to directly addressed memory
        MOV VCT[BX], AL     ;general register to indirectly
                            ;addressed memory
        MOV AX, MEMW        ;directly addressed memory to general register
        MOV AL, VCT[BX]     ;indirectly addressed memory to
                            ;general register
        MOV DS, MEMW        ;directly addressed memory to
                            ;segment register
        MOV AX, BX          ;general register to general register
        MOV DS, AX          ;general register to segment register
        MOV CX, ES          ;segment register to general register
        … …
CODE ENDS
```

The following MOV instructions are not permitted: immediate data to segment register, memory location to memory location, segment register to segment register and MOV to the CS segment register.

MOV instructions that require two instructions are presented below.

Example:

```
                ;immediate data to segment register
        MOV AX, 1000H
        MOV DS, AX
                ;memory location to memory location
        MOV AX, MEM1
        MOV MEM2, AX
                ;segment register to segment register
        MOV AX, DS
        MOV ES, AX
```

Newer processors allow these instruction, however early versions of assemblers will not recognize them as valid instructions.

Data, respectively source and destination operands interchange is done with the XCHG instruction. Its syntax is presented below:

XCHG {register | memory}, {register | memory}

Example:

```
        XCHG AX, BX         ;interchanges ax with bx
        XCHG MEM16, AX      ;interchanges the memory
                            ;word mem16 with the ax register
        XCGH DL, MEM8       ;interchanges the memory byte mem8
                            ;with register dl
        XCGH AH, CL         ;interchanges ah with cl
```

The XLAT instruction coverts the content of register AL, using a translation table. A pointer to the start of the table should be in register BX. The content of register AL is interpreted as an index in the table. The result of the conversion is given by the value of the byte that is placed at this address in the table. The syntax is as follows:

XLAT [segment register : offset]

Using a reference to an address in the XLAT instruction is necessary when the table is not located in the data segment, which is the only implicit segment for this instruction. It allows the assembler to determine the segment register that has to be used for the execution of the instruction.

Here is an example that translates a Hexadecimal digit in a printable ASCII code:

Example

```
                        ;hexadecimal to ASCII conversion
                        ;input : al = hexadecimal digit
                        ;output : al = the corresponding ASCII code
CONV PROC NEAR
        MOV BX,  OFFSET TABEL
        XLAT CS:TABEL
        RET
CONV    ENDP
TABEL   DB '0123456789ABCDEF'        ;ASCII code table
```

The PUSH and POP instructions are used for data transfer to and from a stack.

The stack is a memory location used for temporary data storage. The top of the stack address is managed automatically, by hardware, through a register that points to the top of the stack, namely SP register. This is why these instructions, PUSH and POP, only allow access to the top of the stack. The data that is placed on the stack can be accessed in reverse order of the placement (LIFO system- Last In First Out). Initially the stack contains no data. As data is being placed, during the execution of the program, the stack grows in size, towards smaller addresses. As data is being extracted from the stack, its size is decreasing, by successively freeing the locations that have the smallest address.

The instructions for subroutine call, namely CALL, INT and return from subroutines, RET and IRET, automatically use the stack for saving and restoring the return addresses.

The PUSH instruction is used to put a 2 byte operand on the stack. The POP instruction is used to extract the last value from the stack. The syntaxes for these instructions are:

```
PUSH {register | memory}
POP  {register | memory}
```

When pushing an operand on the stack, the first thing that is done is decrementing the stack pointer SP by 2 and copy the operand to this memory location. When extracting from the stack, first the value on the top of the stack is copied and the SP is incremented by 2.

The PUSH and POP instructions are usually used in pairs. Normally, the number of pushes has to be equal to the number of pops to/from the

stack to bring the stack to its initial state. The words are popped in the reverse order of the pushes.

```
        Example
INT     PROC FAR
        PUSH DS
        PUSH AX
        PUSH CX
        PUSH SI
        PUSH BP
        … …
        POP BP
        POP SI
        POP CX
        POP AX
        POP DS
        IRET
INT     ENDP
```

If there is no need to restore the values pushed on the stack , e.g. parameter transfer to a procedure, the stack can be freed by adding a number to the SP registers (unloading the stack).

Example:

```
        PUSH AX
        PUSH BX
        PUSH CX
        … …
        ADD SP, 6
```

The values that are not on the top of the stack can still be accessed by indirect addressing, using the BP register as base register:

Example:

```
PUSH AX
PUSH CX
PUSH DX
MOV BP, SP
… …
MOV AX, [BP+4]
MOV CX, [BP+2]
MOV DX, [BP+0]
… …
ADD SP, 6
```

Here is an example of a loop that is included in another loop, using the CX register as a counter in for both loops.

Example:

```
MOV CX, 10              ;init counter for outer loop
ET1:                    ;start of outer loop
        ;… …
        PUSH CX         ;saving counter outer  loop
        MOV CX, 20      ;init counter inner loop
        ET2:            ;start of inner loop
        ;… …
        LOOP ET2
        POP CX          ;restore counter outer loop
                        ;outer loop
        ;… …
LOOP ET1
```

2.2. Instructions for address transfer

They are used for loading effective addresses (16 bits) or physical ones (32 bits) into registers or register pairs. There are 3 such instructions:

```
LEA <d>, <s>
LDS <d>, <s>
LES <d>, <s>
```

The LEA instruction loads the effective address of the source operand, that has to be a memory location, to the general register that is specified as the destination. Its syntax is as follows:
LEA {register}, {memory}

The LDS and LES instructions load the physical address that is contained by the source operand, which has to be a double memory word, to the segment register that is specified by the instruction mnemonic, DS and ES, and to the general register that is specified as destination. The instruction mnemonic is:

LDS {register}, {memory}
LES {register}, {memory}
LFS {register}, {memory}
LGS {register}, {memory}

The LEA instruction can be used for loading the effective address of an operand that is placed in the memory, by direct or indirect addressing.

Example:

```
LEA DX, ALFA
LEA DX, ALFA[SI]
```

The effect of the first instruction can be also obtained by using the next instruction:

```
MOV DX, OFFSET ALFA
```

MOV DX, OFFSET ALFA[SI] is an incorrect instruction

This option is quicker, but can only be obtained in the case of operands specified by direct addressing.

Example:

```
DATA SEGMENT
STRING          DB    "THIS IS A STRING"
FPSTRING        DD    STRING        ; FAR POINTER TO STRING
POINTERS        DD    100 DUP (?)
DATA ENDS
CODE SEGMENT
        … …
LES DI, FPSTRING        ;the address contained in the source location is
                        ;loaded to
                        ; the pair es:di
LDS SI, POINTERS[BX]    ;the address contained in the source location is
                        ;loaded to
                        ;the pair ds:si
        … …
CODE ENDS
```

2.3. Flag Transfer instructions

In the I8086 microprocessor's set of instructions there are instructions for loading and storing the flags. The syntax is:

LAHF
SAHF
PUSHF
POPF

The least significant byte of the flag register can be loaded to the AH register using the LAHF register, and also the content of the AH register can be stored to the low byte with the SAHF instruction. The structure of the low byte is:

bit	7	6	5	4	3	2	1	0
	SF	ZF	x	AF	x	PF	x	CF

The whole flag register can be pushed and restored only to the stack register, the instructions to are PUSHF and POPF. The flag register's structure is:

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	x	x	x	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

2.4. Input/output instructions

I/O ports, are constituent elements of interfaces. They connect central units with peripheral devices.

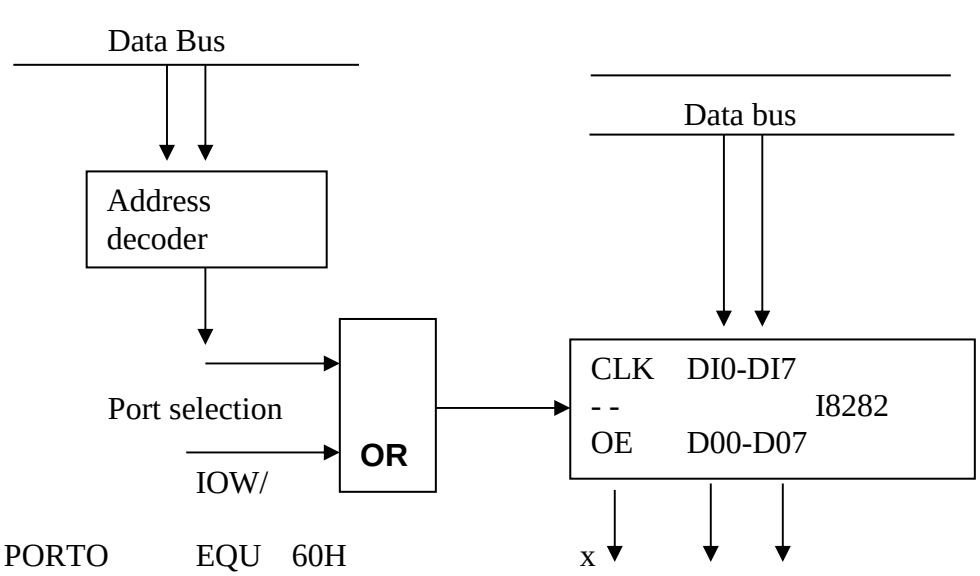
Each peripheral device has its own address through which it can be selected by the central unit. From the central unit's point of view, the peripheral registers can be either input registers or output ones. For transfers of data to these registers, we use the OUT instruction, and for getting, reading data we use the IN instruction. Their syntaxes are:

IN {AX | AL}, {peripheral immediate address | DX}
OUT {peripheral immediate address | DX }, {AX | AL}

The peripheral register's address can be specified by an immediate 8 bit data or by previously storing the I/O address in the DX register. Using DX allows the usage of a larger address than 255.

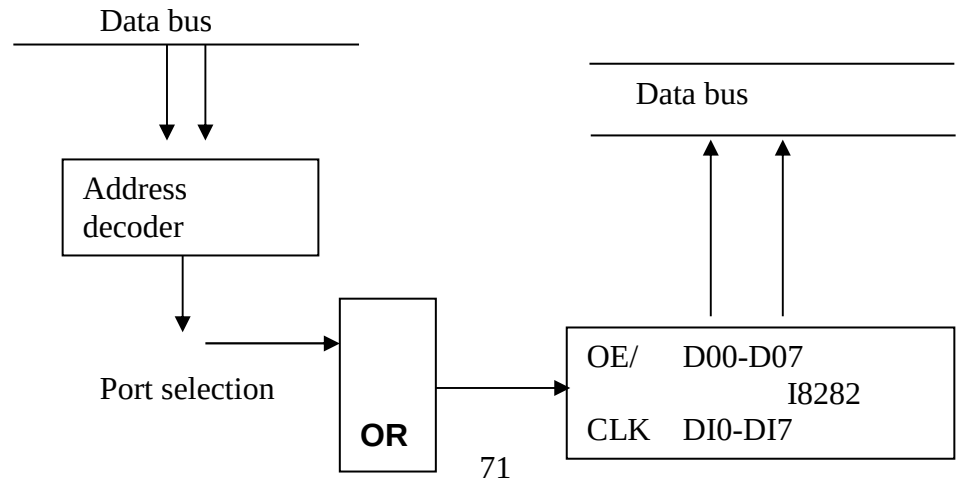
Data transfer is made between the central unit's accumulator and the peripheral registers. This transfer can be of 8, 16 ore 32 bits, depending on the register one uses, either AL, AX or EAX.

Example 1:




```
PORTO       EQU   60H
            …
            MOV   AL, 50
            OUT   PORTO, AL
```

Example 2:




```
PORTI EQU  80H
       …
       IN   AL, PORTI
```

The IN and OUT instructions are the only instructions that allow interaction between the processor and other devices. Some computer architectures have their memory organized in such a way that the areas from the memory space are dedicated to some peripheral equipments and not to data space e.g. Video RAM memory. Access to these memory areas will actually mean access to a peripheral equipment. Such input/output systems are called „memory-mapped" (inputs/outputs organized as memory areas).

Let's consider that a peripheral equipment requires a state port and a data port, both on 8 bits. In a regular input/output system, there are two input ports, for instance 0F8H and 0F9H, dedicated to that equipment. In a memory-mapped system there are two addresses, usually adjacent, for instance C800:0000 and C800:0001, corresponding to the state and data ports. The state-read and data-read sequences, in the two input/output types are:

```
IN    AL, 0F8H          ;read state
IN    AL, 0F9H          ;read data
MOV   ES, 0C800H
MOV   AL, ES:[0]        ;read state
MOV   AL, ES:[1]        ;read data
```

Example: in a PC-AT system, the first serial port uses other ports, starting with 3F8H, but at the same time, the access to the port can be done through the memory, at the address 40:0000. For COM2: ports starting with 2F8H or through the memory, at 40:0002.

3. Lab tasks

1. Study of the shown examples.
2. The students will write a program, which copies a string of values from consecutive memory locations to another location, placed in a different data segment.

3. The students will write a program that duplicates the last two elements of a stack without using push or pop instructions. They will only access the stack using the BP and SP registers.
4. The PC speaker is programmed as follows:
    a) the frequency of the sound is programmed in the next sequence:

```
MOV   AL, 36H              ;the 8253's circuit mode word
OUT   43H, AL
MOV   AX, FRECVENTA        ;the frequency is loaded to ax
OUT   42H, AL        ;the least significant byte is sent
MOV   AL, AH
OUT   42H, AL        ; the most significant byte is sent
b) the sound is being validated:
IN    AL, 61H
OR    AL, 3  ;logical or between al and immediate data
                   ;the validation bits are positioned
OUT   61H, AL
c) the sound is invalidated:
IN    AL, 61H
AND   AL, 0FCH       ;logical and between al and
                     ;immediate data
                     ;the validation bits are erased
OUT   61H, AL
```

N.B. Previous example may not work on arbitrary PC configuration. In time I/O port addresses may change.

5. Write a program that fills a 5 byte memory area, located at consecutive addresses with a value that is loaded by direct addressing to al. They will write more programs, using different addressing modes. Which program is the most efficient?
6. Write a program that transfers two memory words that are placed at successive addresses to another address, using the stack instructions.
7. The students have to write the shortest program that duplicates the last 10 words that were put on the stack, to the stack.

Solved problems:

Modify the content of two words from the memory, using their far addresses (32 bit address). Hint: use the LDS and LES instructions.

Solution:

```
_DATA SEGMENT PUBLIC 'DATA'
        X           DW 10
        Y           DW 15
        ADR_X       DD X
        ADR_Y       DD Y
_DATA ENDS
_CODE SEGMENT PARA PUBLIC 'CODE'
 ASSUME CS:_CODE
 START       PROC FAR
        PUSH DS
        XOR AX,AX
        PUSH AX
        MOV  AX, _DATA              ;initializing the segment register
        MOV  DS, AX
        LDS   SI, ADR_X    ;load address of x to DS:SI -> far address
                                    ; 32 bits
        LES   DI, ADR_Y    ;load address of y to ES:DI -> far address
                                    ;32 bits
        MOV  WORD PTR [SI], 20  ;the x variable is modified, by indexed
                                    ;addressing
        MOV  WORD PTR ES:[DI], 30       ;the y variable is modified, by
                                        ;indexed addressing
        RET                 ;exiting to DOS
START ENDP
_CODE ENDS
        END START
```

The program reads all the keys from the keyboard, until 0 is pressed. It will display the ASCII codes of these keys. Use the XLAT instruction.

```
_DATA SEGMENT
        TAB_CONV DB '0123456789ABCDEF'      ;conversion table
        MESAJ       DB     '-HAS THE ASCII CODE'
        TASTA       DB     2 DUP (?) , 0DH, 0AH, '$'
_DATA ENDS
```

```
_COD SEGMENT PARA PUBLIC 'CODE'

 ASSUME CS:_COD, DS:_DATA
 START  PROC FAR
        PUSH DS
        XOR AX,AX
        PUSH AX
        MOV  AX, _DATA          ;initializing the data segment register
        MOV  DS, AX

 AGAIN:
        MOV  AH, 1          ;echo reading of a key
        INT  21H
        CMP  AL, '0'
        JZ   FINISHED
        MOV  AH,AL              ;saving key code
        LEA  BX, TAB_CONV       ;the conversion table's offset to BX
        AND  AL, 0FH            ;only the first 4 bits (nibble) are taken
        XLAT TAB_CONV           ;convert the second nibble
        MOV  TASTA+1, AL        ;it is the code's second digit
        MOV  AL, AH             ;the initial code of the key
        MOV  CL, 4              ;we shift to the right with 4 positions
        SHR  AL, CL             ;shift
        XLAT TAB_CONV           ;converting the first nibble
        MOV  TASTA, AL          ;the ASCII code of the first nibble
                                ;
        LEA  DX, MESAJ
        MOV  AH, 9H             ;display the key's code
        INT  21H
        JMP AGAIN

FINISHED :
        RET
START ENDP
 _COD ENDS
        END START
```

LABORATORY WORK NO. 7
FLOW CONTROL INSTRUCTIONS

1. Object of laboratory

The x86 microprocessor family has a large variety of instructions that allow instruction flow control. We have 4 categories: jump, loop, calling and return instructions.

2. Theoretical considerations

2.1. Jump instructions.

Jumping is the most direct method of modifying the instruction flow. The jump instructions change the value of the IP register and sometimes of the CS register (for intersegment jump), so the IP and CS registers will be loaded with the address of the target.

2.1.1. The unconditional jump.

The JMP instruction is used for making an unconditional jump to a specified address. The jump in the same segment can be short/relative or near, the destination address that can be between -126..129 bytes relative to the jump instruction for short jump or in the same segment for near jump. A far jump is a jump to a different segment.

From the destination address specification point of view there are direct and indirect jumps. In the direct jumps, the destination address is specified through a label. The syntax is:

```
JMP     label
```

For the **short** jumps in the same segment, the addressing is IP **relative**.

After the instruction code there is a displacement on a byte that represent the distance from the current address to the destination.

Example:

```
ALFA:
        …
        JMP    ALFA
```

If the distance to the label is under 126 bytes and has been defined **before** the jump instruction, than a short jump type is encoded.

If the label is defined **after** the jump instruction than a near jump type is encoded, indifferent if the distance between the jump instruction and label is lesser or not than 129 bytes. We can force a short jump by using the SHORT operator.

Example:

```
        JMP    SHORT BETA
        …
BETA:
```

Observation: Using the SHORT operator in an improper situation will generate an assembly error.

For a near jump the target address is encoded in the instruction on 2 bytes.

In case of using the direct addressing for jumping between different segments the instruction code is followed by a displacement of four bytes that representing the destination address segment : offset.

If the destination label has been defined before, the encoding is correct. If the label is defined afterwards, it is necessary to specify the FAR type for this label.

Example:

```
        JMP    FAR PTR GAMA
        …
GAMA:
```

In case of indirect jumping, the destination address is specified through an operand, the syntax is:

```
        JMP {register| memory}
```

Example:

```
        JMP    AX
        JMP    [BX]
        JMP    ALFA          ; ALFA is a var. word or double word
```

If the variable is defined after the jump instruction in the case of far jumps we must use DWORD PTR operator.

Example:

```
        JMP    DWORD PTR ALFA
```

Example:

```
CODE SEGMENT
        JMP          PROCES

CTL_TBL     LABEL        WORD
        DW           EXTENDED        ; the key with extended
                                     ; code (2 car.)
        DW           CTRLA           ; the key CTRL/A
        DW           CTRLB           ; the key CTRL/B

PROCES:
        MOV          AH, 8H          ; reading the key in AL
        INT          21H
        CBW
        MOV          BX, AX
        SHL          BX, 1           ; the address calculation
in the table
        JMP          CTL_TBL [BX]
        …

EXTENDED:
        MOV          AH, 8H          ; takes the second cod
        INT          21H
        …

CTRLA:                               ; routine for CTRL/A
        …
        JMP          NEXT
CTRLB:                               ; routine for CTRL/B
        …
        JMP          NEXT
NEXT:                                ; continue
        …
CODE ENDS
```

2.1.2. Conditional jumps

The conditional jump is the most frequent method of modifying the instruction flow. It consists of a process in two steps. In the first step the condition is tested and in the second the jump is done if the condition is true or the next instruction in executed if the condition is false

The jump instruction syntax is:

```
Jcc     label
```

The conditional jumps are short type, so that the distance at the destination address must be in the -126..129 range. Otherwise an error is signaled. The destination address is specified through a displacement on a signed byte relative to the current address. The conditional jumps use as a condition the flags or logical combination of the flags.

The flags can be set by any of the instructions that affect the flags. The most frequent WAY IS TO use the CMP or TEST instructions.

The conditional jump is made using one of the 13 conditional jump instructions.

If the target of the conditional jump is out of range, it must be replaced through a conditional jump of reverse condition followed by an unconditional jump.

Example:

```
        CMP   AX, 7
        JE    NEAR
        CMP   AX, 6       ; if AX is 6 and the jump is greater than 129 bytes
                          ; the instruction of conditional jump is replaced
        JNE   NEAR
        JMP   FAR

NEAR:                     ; less than 128 bytes
                          ; from the jump instruction
        …
FAR:                      ; more than 128 bytes
                          ; from the jump instruction
```

2.1.3. Compare and jump

The CMP instruction compares 2 operands by subtracting the source operand from the destination operand without affecting the destination and setting the flags. The syntax is:

CMP {register | memory}, {register | memory | immediate value}

The conditional jump instruction used after the compare instruction has the flow chart accordance with the tested relation, generated form the next letters:

LETTER	MEANING	
J	Jump	
G	Greater than	(for signed value)
L	Less than	(for signed value)
A	Above	(for unsigned values)
B	Below	(for unsigned values)
E	Equal	
N	Not	

In the next table there are represented the conditional jump instruction according to each relation:

Jump condition	Compare with sign	Jump condition	Compare without sign	Jump condition
Equal =	JE	ZF=1	JE	ZF=1
Not equal <>	JNE	ZF=0	JNE	ZF=0
Greater than >	JG or JNLE	ZF=0 and SF=OF	JA or JNBE	ZF=0 and CF=0
Less than <	JL or JNGE	SF<>OF	JB or JNAE	CF=1
Greater than or equal >=	JGE or JNL	SF=OF	JAE or JNB	CF=0
Less than or equal <=	JLE or JNG	ZF=1 or SF=OF	JBE or JNA	CF=1 or ZF=1

Example

```
; IF (CX< -20) THEN DX=30 ELSE DX=20
        CMP   CX, -20
        JL    LESS
        MOV   DX, 20
        JMP   CONT
LESS:
        MOV   DX, 30
CONT:
```

Example:

```
; IF (CX>= -20) THEN DX=30 ELSE DX=20
        CMP   CX, -20
        JNL   NOTLESS
        MOV   DX, 20
        JMP   CONT
NOTLESS:
        MOV   DX, 30
CONT:
```

Jumps based on flag value are:

INSTRUCTIONS	JUMP CONDITIONS
JO	OF=1
JNO	OF=0
JC	CF=1
JNC	CF=0
JZ	ZF=0
JNZ	ZF=1
JS	SF=1
JNS	SF=0
JP	PF=1
JNP	PF=0
JPE	PF=1
JPO	PF=0
JCXZ	CX=0

As it can be observed JCXZ is the only conditional jump instruction that does not test the flags but the content of the CX register.

Example:

```
        ADD   AX, BX
        JO    OVERFLOW
        …
OVERFLOW:
```

2.2. Loop instructions

The cycling instructions allow an easy programming of the control structures of the final test cycle type.

The syntax of these instructions is:

```
LOOP      label     ; CX is decremented and if CX is not
                    ; zero the loop is done.
LOOPE     label     ; CX is decremented and if CX is not
                    ; zero and ZF=1 the loop is done.
LOOPZ     label     ; identical with LOOPE
LOOPNE    label     ; CX is decremented and if CX is not
                    ; zero and ZF=1 the loop is done.
LOOPNZ    label     ; identical with LOOPNE
```

The loop instructions decrement the content of the CX register and if the jump condition is fulfilled the loop is done.

The distance between the looping instruction and the destination address must be between the range -126..129 bytes

Example:

```
        MOV   CX, 200           ; initialize counter
NEXT:
        …
        LOOP  NEXT              ; repeat if CX in not null
                                ; continue after the cycle
```

This loop has the same effect as the one in the next example:

Example:

```
        MOV   CX, 200
NEXT:
        …
        DEC   CX
        CMP   CX, 0
        JNE   NEXT
```

The first version is more efficient.

Using the JCXZ instruction allows us to avoid executing a loop for CX=0.

Example:

```
NEXT:
        JCXZ  CONT
        …
        LOOP  NEXT
CONT:
```

2.3. Using procedures

The procedures are code units that fulfill specific functions. They represent a way of dividing the code in functional parts or blocks so that a specific function can be executed from any other point in the program without having to insert the same code again and again.

The procedures from the assembler language are comparable with the C functions..

For defining and using procedures there are two pseudo-instructions and two instructions. The PROC and ENDP directives mark the beginning and the end of the procedure. The CALL instruction is used to call the defined procedures, and the RET instruction is used for returning to the calling point.

The CALL and RET instructions use the stack to store and restore the return address. The CALL instruction pushes on stack the return address (the address after the CALL instruction) and then a jump to the address at the beginning of the procedure is done.

The RET instruction extracts from the stack the address introduced by the CALL instruction and returns to the instruction after the call.

The procedures can be found or not in the same segment with the calling instructions.

From this point of view there are NEAR and FAR type procedures. When declaring the procedures their type is declared too. The NEAR type is implicit.

The procedure definition syntax is:

```
Label       PROC  [NEAR | FAR]
            …
            RET   [constant]
Label       ENDP
```

The RET instruction allows one constant operand that specify a number of bytes that will be added to the content of the SP register after returning from the procedure. This operand can be used for deleting from the stack the arguments that were transmitted to the procedure through the stack.

The call procedure syntax is:

```
CALL  {register | memory}
```

3. Lab tasks

1. Study the instructions and the examples presented before.
2. Write a program sequence that transforms the ASCII code of a small letter in the ASCII code of the capital letter. The code will be taken from a memory location and saved in the same memory location.
3. Write a program that calculates the average of the numbers from an array of unsigned values. Write the average obtained on the display and the message "The average is: ". The average will be calculated as a integer number. Use DOS system function calls to print messages.
4. Compute the average only for the number between [5..10]
5. Write a program that displays the content of AX register in decimal. HINT: divide AX several times with 10, print the results in reverse order
6. Write a program that reads an integer without sign from the keyboard until the enter key is pressed. HINT: every digit you read will be converted to its numeric value. Compute like this: 145=(((1*10)+4)*10)+5
7. Write a procedure that converts a hex digit (0 to F) in an ASCII character. Send the hex digit to the procedure in the AL register, and the procedure returns the ASCII character in the same register.

LABORATORY WORK NO. 8
WORKING WITH MACROS AND LIBRARIES

1. Object of laboratory

Getting used to defining and using macros, procedure defining and using LIB library librarian.

2. Theoretical considerations

2.1. Working with macros

The macros, procedures and libraries are the programmer tools, which allow the call and the using of previously written and debugged code.

The macros are facilities for assembly language programmers. A macro is a pseudo-operation that allows repeated including of code in the program. The macro, once defined, his call by name allows his insertion any time is needed. When meeting a macro name, the assembler **expands** his name in corresponding code of the macro body. For this reason, it is said the macros are executed in-line because the sequential execution flow of the program in not interrupted.

Macros can be created as a part of user program or grouped into another file as a macro library. A macro library is a usual file, which contains a series of macros and which is referred during program assembly, at the first pass of the assembler over the source program. It has to be specified that a macro library contain unassembled source lines. Because of that, macro libraries have to be included in the user source program using the INCLUDE pseudo-instruction – see Annex 12 example. This is the major difference between the macros library and a procedure library in object code that contains assembled procedures as object code and which is referred to the link-edit.

Firms offer this kind of macro libraries, for example DOS.INC and BIOS.INC by IBM.

For defining a macro it is used the sequence beneath:

```
name    MACRO       {macro parameters}
        LOCAL       local label list of the macro
                    these are expanded with different names at the
                    repeated call of the macro
                    {macro body}
        ENDM
```

Example:

```
INTIR MACRO     TIME
      LOCAL             P1,P2       ;p1 and p2 are local labels
      PUSH      DX                  ;saves the dx and cx registers
      PUSH      CX                  ;cx
      MOV       DX, TIME            ;loads a delay in dx
P1:   MOV       CX, 0FF00H          ;loads cx with 0FF00H
                                      ;counts
P2:   DEC       CX                  ;delays decrementing cx
      JNZ       P2                  ;if cx!=0 continue
      DEC       DX                  ;if cx=0 decrements dx
      JNZ       P1                  ;if dx!=0 loads again cx
      POP       CX                  ;if dx=0 remake cx
      POP       DX                  ;and dx
      ENDM                          ;end macrou
```

P1 and P2 are the local labels of the macro.

2.2. Pre-defined macros

TASM recognizes pre-defined macros. Those are IRP, IRPC and REPT. They are used for repeated defining.

Example:

```
        IRP     VAL, <2,4,6,8,10>
        DB      VAL
        DB      VAL*2
        ENDM
```

In some cases, the formal parameter substitution with actual parameters creates some problems. Let’s follow the macroinstruction, which suggests interchanging two 16 bites quantities.

```
TRANS  MACRO  X, Y
        PUSH AX
        PUSH BX
        MOV  BX, X
        MOV  AX, Y
        MOV  X, AX
        MOV  Y, BX
        POP   BX
        POP   AX
ENDM
```

Apparently, every thing is ok. However, unexpected situation can appear, like in following sequence:

```
        TRANS       AX, SI              ;interchange ax with SI
```

This referred macroinstruction will be expanded in:

```
        PUSH  AX
        PUSH  BX
        MOV   BX, AX
        MOV   AX, SI
        MOV   AX, AX
        MOV   SI, BX
        POP    BX
        POP    AX
```

and it is obviously the AX register is not modifying. Worst thing can happen, like beneath:

```
        TRANS       SP, DI              ;interchange SP with DI
```

which is expanded in:

```
        PUSH  AX
        PUSH  BX
        MOV   BX, SP
        MOV   AX, DI
        MOV   SP, AX                    ;SP is modified here
        MOV   DI, BX                    ;POPs are compromised
        POP    BX
        POP    AX
```

Danger appears, therefore, in situation in which actual parameters are conflicting with some variables or registers being used in the macroinstruction. Situations like this must be avoided.

2.3. UsingTLIB librarian

The syntax for launching TLIB librarian is:

TLIB *library_name [/C] [/E] [/P] [/O] command, listing_file_name*

where:

- *library_name* represents the path and the library file name
- *command* represents commands sequence that will be executed on the library
- *listing_file_name* represents the path and the name of the file in which you want the crossed references to be generated for PUBLIC symbols and for the library modules names. The listing is generated after the processing in the library is finished.

A command is like:

<symbol> module_name

where <*symbol*> represents:

+	:	adds module_name to the library
-	:	deletes module_name from the library
*	:	extracts module_name from the library without deleting it
-+ or +-	:	replaces module_name in the library
-* or *-	:	extracts module_name from the library and deletes module_name from the library
/C	:	case-senzitive library
/E	:	creates extended dictionary
/P size	:	sets library page dimension to size

For moving to the next line, use ‘&’ character.

2.4. Examples of programs that are using macros and libraries

2.4.1. Program EXEMMAC.ASM

```
;PROGRAM EXAMPLE FOR USING A SIMPLE MACRO

TITLE Program with macro call

STACK SEGMENT PARA 'STACK'
        DB              64 DUP ('STACK')
STACK   ENDS

DATA SEGMENT    PARA 'DATA'
TAMP DB         2000 DUP (' ')
DATA    ENDS

INTIR MACRO     TIME
        LOCAL   P1, P2  ;;p1 and p2 are local labels
        PUSH  DX        ;;saves dx and cx registers
        PUSH  CX
        MOV   DX, TIME          ;; loads a delay in dx
P1:     MOV   CX, 0FF00H  ;;loads cx with 0FF00h
                        ;;counts
P2:     DEC   CX                ;;delays by decrementing cx
        JNZ   P2                ;;if cx!=0 continue
        DEC   DX                ;;if cx=0 decrements dx
        JNZ   P1                ;;if dx!=0 loads again cx
        POP   CX                ;;if dx=0 remake cx and dx
        POP   DX                ;;
        ENDM                    ;;end macro

MYCOD SEGMENT PARA 'CODE'       ;defines code segment
PROCED  PROC    FAR             ;procedure with proced name
        ASSUME  CS:MYCOD, ES:DATA, DS:DATA, SS:STACK
        PUSH  DS
        XOR AX,AX
        PUSH AX
        MOV   AX, DATA          ;puts data segment in ax
        MOV   ES, AX
```

```
        MOV   DS,AX                     ;loads es with data segment
;program will clear the display writing 25*80 spaces on the screen
;writing those with different values in bl the screen color will change
;intir macro will maintain this color for a time
        MOV   CX, 08H                   ;loops 8 times
        MOV   BL, 00H                   ;sets background color
LOOP1:  LEA     BP, TAMP                ;writes black string
        MOV   DX, 0000H                 ;sets the cursor to the upper
                                        :left
        MOV   AH, 19                ;writes attribute string
        MOV   AL, 1                 ;writes a character and moves
                                        ;the cursor
        PUSH CX                         ;saves cx
        MOV   CX, 07D0H                 ;writes 2000 spaces
        INT     10H                     ;call 10h
        INTIR 10000                 ;delays 10 units
        ADD   BL, 10H                   ;changes background color
        POP     CX                      ;restores  cx
        LOOP  LOOP1                     ;loops 8 times
        RET                             ;hands over the control to
                                        ;dos
PROCED      ENDP                        ;end procedure
MYCOD ENDS                              ;end code segment
        END   PROCED                    ;end program
```

2.4.2. Program EXBIMAC.ASM

```
TITLE Example of macro library using
IF1                                     ;includes a previously created
        INCLUDE C:\TASM\MLAB.MAC   ; macro library
; available on ftp.utcluj.ro/pub/users/cemil/asm/labs
ENDIF

STACK SEGMENT PARA 'STACK' ;defines a stack segment
        DB 64  DUP ('STACK')
STACK ENDS

SEGDATA SEGMENT PARA 'DATA' ;data segment definition
MESSAGE DB          'I am a simple counting program$'
TAMP  DB 2000 DUP (‘ ‘)
SEGDATA ENDS
```

```
COD1  SEGMENT PARA 'CODE'          ;code segment definition
MYPROC PROC      FAR               ;procedure with myproc name
ASSUME              CS:COD1, DS:SEGDATA, SS:STACK

        PUSH  DS                   ;saves ds
        SUB   AX, AX               ;0 in ax
        PUSH  AX                   ;0 on the stack
        MOV   AX, SEGDATA          ;adr segdata in ax
        MOV   DS, AX               ;adr segdata in ds
        DELETE                     ;clear screen macro call
        CURSOR 0019H               ;pos cursor macro call
        TYPECAR    MESSAGE         ;message type macro call
        MOV   AX, 00H              ;0 in ax for counting
REPEAT:     CURSOR      0C28H ;in middle of the screen
        TYPENUM                    ;number type macro call
        INTIR 1000          ;delay macro call
        ADD   AL, 01H              ;increment al
        DAA                        ;decimal adjustment
        CMP   AL, 50H              ;test final
        JE    SFIR                 ;after 9 executions
        JMP   REPEAT               ;else repeat
SFIR:   DELETE              ;clear screen macro call
        RET                        ;back to dos
MYPROC ENDP                        ;end procedure
COD1  ENDS                        ;end segment
        END   MYPROC               ;end program
```

2.4.3 Calling a procedure defined in a different source file

Main program:

```
;Program example for procedure use procedure defined in a different source
;file
TITLE Program with procedure call

STACK       SEGMENT    PARA 'STACK'
        DB      64 DUP ('STACK')
STACK       ENDS

DATA SEGMENT PARA `DATA`
TAMP DB     2000 DUP ('  ')
DATA        ENDS
```

```
COD1 SEGMENT PARA 'CODE'          ;code segment definition
PROCED    PROC FAR                ;procedure with proced name
        ASSUME    CS:COD1, ES: DATA, DS:DATA, SS:STACK
        EXTRN     INTIRP:NEAR ;extern declaration for INTIRP
                              ;procedure
        PUSH DS                   ;saves ds
        SUB  AX, AX               ;0 in ax
        PUSH AX                   ;puts 0 on the stack
        MOV  AX, DATA             ;puts seg data in ax
        MOV DS,AX
;main program
        MOV  AX, 100              ;parameter in ax
        CALL INTIRP               ;intirp procedure call
        RET                       ;gives the control to dos
PROCED    ENDP                    ;procedure end
COD1 ENDS                         ;code segment end
        END  PROCED               ;end program
;End of first source file _____________   _____   _______

;Start of second sourse file_____________________________
;called procedure
COD1 SEGMENT PARA 'CODE'    ;defines code segment
             PUBLIC INTIRP ;public declaration for INTIRP
                           ;procedure
        ASSUME      CS:COD1
INTIRP      PROC NEAR       ;intirp procedure name
        PUSH DX             ;saves dx şi cx registers
        PUSH CX             ;
        MOV  DX, AX         ;loads a delay in dx
P1:     MOV  CX, 0FF00H  ;loads 0FF00h in cx
                            ;counts
P1:     DEC  CX             ;delays decrementing cx
        JNZ  P2             ;if cx!=0 continue
        DEC  DX             ;if cx=0 decrements dx
        JNZ  P1             ;if dx!=0 loads again cx
        POP  CX             ;if dx=0 restore cx and
        POP  DX             ;dx
        RET                 ;return to the main procedure
```

```
INTIRP ENDP                    ;procedure end
COD1 ENDS
END
```

3. Lab tasks

1. Study the given example and exemmac.asm program.
2. Assemble this program with TASM and create EXEMMAC.LST file, study the way INTIR macro has been expanded.
3. Edit the links with LINK and execute exemmac.exe generated program.
4. Modify INTIR macro TIME parameter with different values with an edit program and repeat the steps from 1 to 3.
5. Study the case in which the macro is written into a separate file and it is included with INCLUDE directive (see previously example); notice the difference from a module included before compilation (with INCLUDE), a macro (which is similar) and a library (which contains compiled modules) – point out the similarity with .h files from C which are being compiled in the same time with the program.
6. Study the example of using a macro library MLIB.MAC in the exbimac.asm program.
7. Study the expand mode of PUSHALL and POPALL macros in TASM created listing of the program from the step 5.
8. Edit the links with TLINK program and execute EXBIMAC.EXE program.
9. Write a procedure with the same function as INTIR macro with INTIRP name. Include this procedure into a library with BIBLIO.LIB name. TIME parameter will be passed to the procedure in AX register.
10. Modify exmmac.asm to axmlib.asm and replace macros with procedure calls to INTIRP procedure, which initially has been included in BIBLIO.LIB.
11. Trace the program from steps 3 and 10 and follow the differences of generated code and change in instruction flow.

Lab work no. 9

Programs with multiple segments

Object of laboratory

Procedure definition, procedure call from the same segment and from different segments; working with programs written in more, separately assembled modules.

Theoretical considerations

Procedures may be defined as FAR or NEAR type. The procedure's type determines the way in which the call is made and the information that is saved on the stack at calling.

When calling a NEAR type procedure, IP register and the state is saved on the stack. CS register remains unmodified and is not saved on the stack. This implies the belonging of the two procedures, the called one and the one that makes the call, to the same code segment. If the two procedures are defined in different program modules or files, the fact that they belong to the same segment is defined in concordance with the names of code segments in which the procedures were defined. The code segment needs to have the **same name**. The link-editor knows to concatenate in a single segment code segments with the same name from different modules.

The declaration of a procedure that is defined in another program module than the one that makes the call (uses the procedure) is made through the EXTRN directive. The called procedure has to be declared with the PUBLIC directive in the module in which it is defined. EXTRN and PUBLIC declarations must be written **inside** the segment and not outside for near procedure

When calling a FAR type procedure CS, IP are saved on the stack. In this case the two procedures must belong to **different** segments. EXTRN declaration is made **outside** the segment and the PUBLIC inside the segment. FAR type calling is used only when the NEAR type calling is not possible, because this type of call is slower due to the more references made to the stack both at calling time and return time. A FAR type call is necessary when the length of the two procedures might exceed 64K, this being the maximum admitted dimension for a segment.

Procedure definition example: NEAR type procedures with procedures in different modules/files:

The calling, main, procedure:

```
DATE SEGMENT   PARA PUBLIC 'DATA'        ; data segment definition
        ;...
DATE ENDS
STAC SEGMENT PARA STACK  'stack'         ;stack segment definition
                        db 64           dup ('MY_STACK')
STAC ENDS

COD1   SEGMENT       PARA PUBLIC 'CODE'   ; cod segment definition

EXTRN PROCED: NEAR
PRPRINC         PROC   FAR                          ; main procedure definition
ASSUME CS: COD1, DS: DATE, SS: STAC, ES: NOTHING
        PUSH   ds                           ;prepare stack
        SUB    ax, ax                       ;to return
        PUSH   ax                           ; to DOS
        MOV    AX, DATE                             ; load register
        MOV    DS, AX                       ; DS with data segment

; The instructions of the main procedure
        CALL   PROCED                       ; call procedure
; Other instructions

        RET                                 ; coming back to DOS
PRPRINC         ENDP                        ; end procedure
COD1   ENDS                                 ; segment's end
END PRPRINC                             ; end of the first module
```

----------end of first file

The called procedure defined in another program module:

```
COD1 SEGMENT     PARA 'CODE'        ;       segment       code
definition
PUBLIC    PROCED                    ;   declare   proced   as
PUBLIC
ASSUME     CS: COD1
PROCED    PROC      NEAR            ; procedure definition

;The instructions of the called procedure

     RET                 ;  coming  back  to  the  procedure,
which made the call
```

```
PROCED    ENDP                ; end procedure
COD1 ENDS                     ; end segment
     END                           ; end of second module
```
---------- end of second file

FAR type procedure call example, procedure in different segments with the procedures in two different modules/files:

```
EXTRN PROCED2:FAR
STAC SEGMENT PARA STACK  'stack'   ;stack            segment
definition
     db 64          dup ('MY_STACK')
STAC ENDS

DATE SEGMENT   PARA PUBLIC 'DATA'  ;       data       segment
definition
     ;…                                    data definition
DATE ENDS

COD2 SEGMENT   PARA PUBLIC 'CODE'  ;       code       segment
definition
ASSUME CS: COD2, DS: DATE, SS:STAC, ES:NOTHING

PRPRINC2 PROC       FAR                  ;  main  procedure
definition
     PUSH DS                  ; prepare stack
     SUB  AX, AX              ; to return
     PUSH AX                  ; to DOS
     MOV  AX, DATE            ; load register
     MOV  DS, AX              ; DS with data segment
; The main procedure's instructions
     CALL PROCED2             ; procedure call

; Other instructions

     RET                      ; coming back to DOS
PRPRINC2 ENDP                 ; end procedure
COD2 ENDS                     ; end segment
     END PRPRINC              ; end of the first module
```

---------- end of first file

The called procedure defined in another program module:

```
COD3 SEGMENT     PARA 'CODE'       ;       code       segment
definition
PUBLIC    PROCED2                 ;  procedure  declaration
as public
```

```
ASSUME    CS: COD3
PROCED2   PROC FAR       ; procedure definition
; The instructions of the called procedure

          RETF                                          ; back
     to the procedure which made the call
PROCED2   ENDP                ; end procedure
COD3      ENDS                ; end segment
          END                 ; end of second module
```
---------- end of second file

Passing parameters to procedures

There are three known types of parameter transfers to procedures in assembly language: through registers, through pointers and data structure and through the stack.

Transfer through registers

The advantage of this solution is that that in the procedure, the actual parameters are immediately available. For register conservation, these are saved on the stack before calling the procedure and are restored after returning from the procedure. There are 2 disadvantages of this:

- the limited number of available registers
- non-uniformity of the method – there is no ordered modality of transferring, each procedure having it's own rules for transfer

Another advantage is speed, many operations with the memory (stack) not needed.

Transfer through memory

In this transfer type a data zone is prepared previously and the address of this data zone is transmitted to the procedure.

To ease access to the parameters it is recommended to define a structure, which describes the structure of the parameters:

```
_ZONA STRUC
        VAL1        DD    ?
        VAL2        DD    ?
        RETURN      DD    ?
_ZONA ENDS

DAT SEGMENT PARA PUBLIC 'data'
        ZONE _ZONA    <10, 20, ?>
dat ends

COD SEGMENT PARA PUBLIC 'code'
Assume cs:cod, ds:dat
```

```
extrn   proce:near
        LEA    BX, ZONE
        CALL  PROCe
cod ends
end
```

Parameter transfer through stack

Transferring parameters through the stack is the most uniform transfer modality. The transfer through stack is compulsory if the applications contain both ASM modules and modules in high level languages. The standard access technique to the parameters procedure is based on based addressing using BP register, which uses by default SS register as segment register to access the data. The access is achieved through the following operations, executed when entering the procedure:

- BP register is saved on the stack
- SP is copied to BP
- the registers used by the procedure are saved on the stack
- the parameters are accessed through indirect addressing using BP

When ending the procedure, the following operations are executed:

- the saved registers are restored
- BP is restored
- Return to the program which made the call through RET

Lab tasks

1. Study the given examples, noticing the differences between the two procedure call types: FAR and NEAR.
2. Write a program which calculates the sum of a string of numbers using a NEAR and then a FAR type procedure, written in another code segment, first both segments being written in the same file and then in different files. The procedure will be called *sum* and it will get as input parameters: the address and length of the string from DS: BX and CX registers. The procedure will return the sum in AX register.

Observations:

- The procedures which are to be included in a library will be defined of the same type, FAR or NEAR, in segments with the same name (if possible), in order not to complicate any more the call and the link edition.
- It is also recommended to group procedures of the same type (mathematical, display, etc) in different libraries having suggestive names.

Solved problem: Write a recursive procedure to display a number stored in AX

Solution:

```
TIP     STRUC                    ; pattern for parameters
   _BP  DW ?
   _CS  DW ?
   _IP  DW ?
   N    DW ?
TIP     ENDS


MYSTACK SEGMENT STACK 'stack'
        DB 4096 DUP (?)     ; stack segment declaration
MYSTACK ENDS


COD SEGMENT PARA PUBLIC 'CODE'
   ASSUME CS:COD, SS:MYSTACK
   DISPL PROC FAR
        PUSH BP         ; standard access
        MOV BP, SP           ; sequence
        PUSH DX
        PUSH AX         ;   we   will   work   with   these
registers in the
                        ; procedure, so we save them
        PUSH BX

        MOV AX,[BP].N
        CMP AX, 10              ; if n<10, dl=n
        MOV DL, AL
        JB DISPLAY_1        ;  jump  to  display  (we  have
only one number)
        MOV BX, 10              ; general case
        MOV DX, 0           ;  calculates  n/10  and  n  mod
10
        DIV BX              ; AX=n/10;
                            ; dl=n mod 10
        PUSH AX             ;  recursive  call  with  n/10
parameter
        CALL FAR PTR DISPL
DISPLAY_1:

        ADD DL, '0'             ; +’0’
        MOV AH, 02H         ; Dos function for display
```

```
        INT  21H            ; display

        POP  BX             ; restore
        POP  AX                       ; registers
        POP     DX
        POP  BP
        RETF 2              ;  FAR type return
   DISPL     ENDP

   START:


        MOV  AX, 65535 ; prepare register with number to
display
        PUSH AX        ;  we  put  it  on  the  stack  as
parameter
        CALL FAR PTR DISPL  ; procedure call

        MOV  AX, 4C00H ; return  to
        INT  21H       ; DOS

COD     ENDS
        END START
```

LAB WORK NO. 10
USING SYSTEM FUNCTIONS IN ASSEMBLY LANGUAGE

1. Object of laboratory

Study of OS's functions (BIOS and DOS functions) and their use in assembly language. An example for using the terminal is presented.

2. Theoretical considerations

The OS is a collection of routines (procedures) useful for efficiently manage system's resources. These routines are divided in 2 categories: BIOS routines and DOS routines. For IBM-PC compatible systems a series of system routines are available for the user. They are written as procedures accessed through software interrupts.

BIOS functions are written to allow access for the user to system resources (hardware) DOS functions are written for OS's specific purposes. DOS functions facilitate usage of the file system, the user doesn't need to have a full knowledge of the hardware of the file system in order to create a file.

Access to BIOS and DOS functions from user programs is done through software interrupts (INT instruction).

The main groups of BIOS functions available for the user are:

INT 10h - use of video terminal
INT 11h - determine system's configuration
INT 12h - determine RAM's capacity
INT 13h - access to HDD and FDD
INT 14h - use of serial interface
INT 15h - APM
INT 16h - use of keyboard
INT 17h - use of parallel interface
INT 19h - loader of resident system on disk
INT 1Ah - real time clock controller(RTC)

During a subroutine call (by INT) other functions can be specified. The function is specified "by convention", placing it's number in AH

register. The call for a certain BIOS function is possible through a generic sequence:

```
MOV  AH, function_nr        ; specify function
INT  int_nr                 ; specify interruption
```

According to a function's complexity a series of parameters can be specified by following it's pattern.

DOS functions are mainly referring to files, but there is a large scale of functions. All DOS functions are called by INT 21h and by specifying the desired function (eventually the parameters) in AH register.

Some BIOS functions will be presented:

INT 10h

This function facilitates the use of video terminal. Inside INT 10h function there are many sub-functions which allow character posting and use of various graphic modes. For graphic modes with bigger resolutions this interrupt it's not recommended because it's slow; it is recommended to write directly into video memory. For example posting a character on screen: calling 10h interrupt implies a big amount of code to be executed (interpreting parameters transferred into registers, setting sub-function etc), while for writing directly into video memory only one MOV instruction is needed.

Still, this interrupt is very practical for programs that are not posting big amount of information on screen at certain moment. By using this interrupt the programmer doesn't have to calculate video memory addresses in which to write every character.

(AH)	Function	Input parameters	Output parameters
00h	Select functioning mode for graphic terminal	(AL) =0 alphanumeric 40 col. x25 lin. b/w =1 alphanumeric 40 col. x25 lin. Col =2 alphanumeric 80 col. x25lin. b/w =3 alphanumeric	

		80 col. x25lin. col. =4 graphic 320 colx200 lin col. =5 graphic 320 colx200 lin b/w =6 graphic 640 colx200 lin b/w	
01h	Select shape and size for cursor	(CH) 0-4 bites for cursor's start line (CH) bites 5-7=0 (CL) 0-4 bites for cursor's finish line (CL) bites 5-7=0	
02h	Cursor positioning	(DH, DL) lin., col. (0, 0) –left upper corner (BH) – page nr. =0 for graphic mode	
03h	Read cursor's coordinates	(BH) – page nr =0 for graphic mode	(DH, DL) lin. , col. (CH, CL) shape
04h	Read position for optic indicator		(AH) =0 light pen inactive =1 light pen active (DH, DL) lin., col. Cursor (CH) line pixel (0-199) (BX) col pixel (0-319636)
05h	Select active display pages	(AL) – page nr. 0-7 for mode 0 and 1 0-3 for mode 2 and 3	
06h	Execute operation "scroll up"	(AL) – nr. of lines (AL) =0 delete	

		window (CH, CL) – lin., col. for left upper corner (DH, DL) – lin., col. for right down corner (BH) – the attribute of a white line	
07h	Execute operation "scroll down"	(AL) – nr. of lines (AL) =0 delete window (CH, CL) – lin. , col. for left upper corner (DH, DL) – lin. , col. for right down corner (BH) – the attribute of a white line	
08h	Read character from screen and determine it's attribute (the character is read from cursor's current position).	(BH) – referred number of page (only for alphanumeric mode)	(AL) – read character (AH) – character's attribute
09h	Show character on screen.	(BH) – referred number of page (only for alphanumeric mode) (BL) –character's attribute (for alphanumeric mode) - color (for graphic mode) (CX) – nr. of	

		characters to show (AL) – character	
0Ah	Replace characters on screen maintaining color characteristics	(BH) –the number for reference page (CX) – nr. of characters to show (AL) (AL) – character	
0Bh	Set color characteristics(only for 320x200 pixels graphic mode)	(BH) – palette's index (conf doc IBM-PC) (BL)-value of color within palette	
0Ch	Show point on screen (for graphic mode)	(DX) – line's number (CX) – column's number (AL) –color's value (0, 1, 2, 3)	
0Dh	Read point's color (for graphic mode)	(DX) – line's number (CX) – column's number	(AL) – read color
0Eh	Show character on screen and update cursor's position	(AL) – character to show (BL) –background color (for graphic mode) (BH) – page (for alphanumeric mode)	
0Fh	Read characteristics for current mode		(AL) – current module (AH) – character's number of columns (BH) – page number

INT 13h

Direct access to HDD and FDD is possible through this interrupt. INT 13 allows writing and reading disk sectors directly without considering the existent system of files on disk. That's why this function is not recommended when working with files, for these operations DOS functions are more appropriate. But there are situations when these functions are the only solution: reading a disk with a different file system, other than FAT.

INT 14h

This function facilitates access to system's serial interface. There are 4 available functions:

AH	the function
00h	initialize port (4 ports are supported)
01h	send one character
02h	read one character
03h	check port's status

INT 16h

This function is used both for reading characters from keyboard and for obtaining keyboard's current state (Caps Lock, Ctrl, Shift etc.) . Function 00h (in AH) is used to read a character from the keyboard, the character will be returned in AL.

INT 19h

After POST the processor executes the code for this interruption by trying to read a code named bootstrap from the floppy or from the hard disk. So, this interruption loads to memory at address 0000:7C00h the first sector from floppy or hard disk and makes a JMP to this address in this way the bootstrap gets the control. The execution of this code determines which partition is active and then loads into memory and executes the boot sector from this partition. Because of this we can have multiple operating systems on a single hard disk and we can choose one of them at start up.

INT 1Ah

This function allows access to system's clock, for reading and setting the time.

AH function
00h read time
01h set time

The hour is represented as units; one unit has 55ms. When the system is on the number of units increases every 55ms.

For AT class computers through this interruption BIOS gives access to system's RTC. The RTC runs even when the computer is off because of the battery on mainboard It uses a memory segment from CMOS to store the time. This memory segment is rewritten every 55ms without using the microprocessor.

DOS functions are mainly referring to files but there is a large variety of functions. Every DOS function is called with INT 21h and by specifying the desired function (and other parameters) in AH register.

DOS – INT21h functions, for keyboard and monitor.

(AH)	Function	Input parameters	Output parameters
00h	End program's execution		
01h	Read character from keyboard and send it in echo to screen. If CTRL-BREAK are pressed INT 23h executes		(AL) –inserted character
02h	Show character on screen. If CTRL-BREAK are pressed INT 23h executes	(DL) –the character	
05h	Print character	(DL) – the character	
06h	Direct read/write - from keyboard - to screen	(DL) = 0FFh (DL) – the character	(AL) – the character (AL) = 0 (no character)
07h	Read character from keyboard without echo and without interpretation		(AL) – the character
08h	Read character from keyboard without		(AL) – the

	echo If CTRL-BREAK are pressed INT 23h executes		character
09h	Show a row from memory ending with $ (24h)	(DS:DX) – row address	
0Ah	Read from keyboard and place into a memory buffer a row of characters, until <CR> is pressed	(DS:DX) – buffer's address	
0Bh	Determines keyboard's situation. If CTRL-BREAK are pressed INT 23h executes		(AL) =Off one character is available (AL) =0 no character
0Ch	Initialize keyboard's for buffer and then call a function. The system waits for a character.	(AL) – requested function (01h, 06h, 07h, 08h, 0Ah) .	

Observations and recommendations for using these functions:

1. BIOS functions save registers CS, SS, DS, ES, BX, CX, DX, and destroy the others, so the user must save them and restore them.
2. Before modifying the monitor's functioning regime it is recommended to save the current attribute and reset it at the end.
3. To erase the screen (window) it is recommended the use of function 06h with 0 displacement (in AL) and not 25!
4. For 0Eh function (write in teletype mode) the attribute is "inherited" from the previous character. But this is a slow process. It can be accomplished much faster by MOVS repetition!
5. DOS functions offer less facilities than BIOS functions when working with the screen.

3. Lab tasks:

1. Study DOS and BIOS functions for using the screen.
2. Write a program which draws on the graphic screen a sequence of squares of constant dimensions but with different colors and positioning.

LAB WORK NO. 11
THE USAGE OF THE MATHEMATICAL COPROCESSOR

1. Object of laboratory

The purpose of this lab is to familiarize the user with the mathematical coprocessor's functions, its instructions for real numbers and other coprocessor operations.

2. Theoretical considerations

Even tough the 8086,80286,80386 and 80486 SX processors have a series of powerful integer arithmetical they do not support floating point arithmetical operations or integer numbers represented on multiple bytes.

Because of these impediments INTEL developed the INTEL 8087 (80287, 80387) arithmetic coprocessor. As its name says the co-processor is made up of several processors that cooperate with the computer's main processor. The coprocessor can not extract by himself the instructions from the memory, this job that is done by the main processor.

2.1. Working principle

The coprocessor is activated at the same time as the system's general RESET signal. This signal brings the coprocessor in its initial state (with error masking, register erase, stack initialization, number rounding, etc.). After the main processor executes the first instruction the coprocessor can detect with which type of processor it has to work. Depending on the processor type the coprocessor will reconfigure itself accordingly. Obsoleted in more modern designs bye monolithic hardware structure of the two processors.

The coprocessor connects to the processors local bus through several lines: data/address, state, clock, ready, reset, test and request/grant. Being connected to the microprocessor's local bus allows the coprocessor access to all memory resources through the request/grand buss request.

The two processors are working in parallel, which implies synchronization between the code running on them.

Usually the error and instruction synchronization is done by compilers and assemblers, while the data synchronization has to be done by the user of the assembling language.

The responsibility of controlling the program belongs to the main processor. Instructions for the coprocessor start with a special ESCAPE code. The coprocessor monitors the instruction flow of the main processor. By decoding the escape code the coprocessor knows when the processor's instruction queue is loaded with instructions for the coprocessor and stores these instructions in its own queue. The ESC instruction code is the following:

1101 1xxx	mod xxx r/m

X meaning don care. Thus all instructions that have the operation code between D8 and DF will be considered as ESC instructions. Together with the three bits from the second byte there are a total of 64 allowed instructions for the coprocessor.

The microprocessor executes the ESC instruction by computing a memory address (from mod and r/m) and executes a bus cycle, reading the data from the computed address (if mod is 11 more bus cycles will be executed). The data is not really read from memory instead a bus cycle is generated (it shows a NOP instruction) the coprocessor being activated by the ESC instruction decodes the six bits from the ESC instruction and can capture the address and/or the data from the memory selected by the instruction. This mechanism allows the programmer to treat the ESC instruction (defined by the coprocessor) as a normal instruction with all the addressing methods. If the coprocessor requires more data from memory it can request he control from the microprocessor. The main processor's registers are not accessible to the coprocessor. The coprocessor keeps the TEST line on high as long as he executes to tell the processor that he is busy.

Inside the coprocessor we have an 80 bytes memory organized as a stack of eight 10 byte elements. On these 10 bytes the floating point numbers are represented in IEEE temporary real format. The coprocessor can access the computer's memory with any addressing mode and any legal format of data. The data brought from memory is converted into the coprocessor's internal format and put on top of the stack. When writing to the main memory the internal format is converted into the format specified by the user.

The condition for executing floating point operations in the coprocessor is: the operand has to be at the top of the stack (for the

operations with two operands, at least one of them). So, with the aid of the coprocessor we can do the following operations:

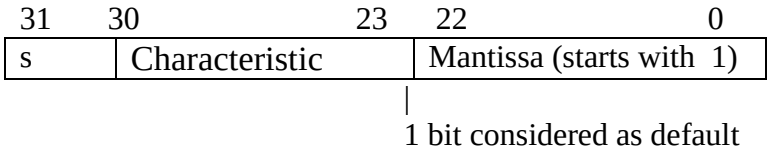
- load data into the coprocessor's internal stack from the computer's memory
- execute the necessary arithmetical operations
- store the results into the computer's memory

2.2. INTEL 8087 recognized data types

The great advantage of the coprocessor is that he works not only with floating point numbers but with integer numbers also and recognizes packed decimal data types. So if we have to execute a complicated integer operation and it has to be very fast it can be done with the help of the coprocessor, without having to do a time consuming conversion from integer to floating point and backwards just so that the coprocessor can work with them.

2.2.1. Floating point data types

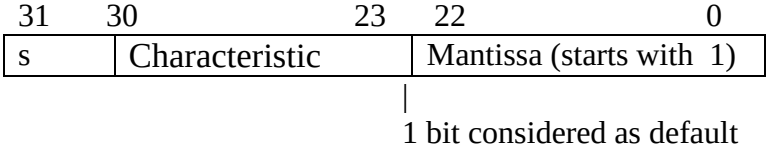
Short Real a 32 bit number represented in floating point. The number is decomposed in mantissa and characteristic. The characteristic is represented on 8 bits from which the most significant is the sign bit and it's treated differently. The physical length of the mantissa is 23 bits. The sigh of the real number is given by the most significant bit:



This floating point number representation always works with normalized numbers, meaning that the mantissa's first bit is always 1, and thus this bit is never written being considered by default. Thus the mantissa's real size is 24 bits.

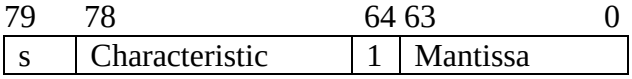
It is important for us to no the actual precision, The mantissa represents 6-7 digits, while the characteristic with its 8 bits raises their number to the order of $\sim 10^{38}$ (the exact number can not be determined because it depends on the mantissa). The highest number is approximately of $1.7 * 10^{38}$ and the lowest positive real number in around 10^{-38}.

Long Real a 64 bit number represented in floating point. The number is decomposed in mantissa and characteristic. The characteristic is represented on 11 bits from which the most significant is the sign bit and it's treated differently. The physical length of the mantissa is 52 bits. The sigh of the real number is given by the most significant bit:



As in the previous case here also he have a default bit, thus the actual length of the mantissa in 53 bits. And these 53 bits we can represent approximately 16-17 digits, the representation of the smallest number in very precise.

Temporary real numbers an 80 bit number represented in floating point. The number is decomposed in mantissa and characteristic. The characteristic is represented on 15 bits from which the most significant is the sign bit. The physical length of the mantissa is 64 bits. The sigh of the real number is given by the most significant bit:



The temporary floating point numbers are not always normalized. The mantissa does not have to start with 1. Thus, in this case the size of the mantissa in of only 64 bits. The high precision floating point number's 64 bits normalized mantissa represents approximately 19 decimal digits. The length of the characteristic is 15 bits. Because the number is not always normalized the lowest possible number that can be represented is much smaller than we would expect: cca. 10^{-4932}.
This representation method is highly sensitive to the possibility of the number being different or equal to 0.

2.2.2. Signed Integer in 2's Complement

Word, double word and quad word representations are accepted.
DW, DD and DQ.

2.2.3. Signed Integer in Packed BCD representation

Signed integer values represented on ten bytes can be used. DT. Numbers up to 18 decimal digits can be represented.

2.3. Operation errors (exceptions)

When using floating point operations we can encounter countless errors, starting from trivial logarithmic errors, to errors caused by representation limitations. These we will call exceptions. We will study these types of errors and the ways in which we can manipulate them.

When an error appears the coprocessor can manifest two behavior types. It signals the error using an interrupt if the user validates this. If not, the coprocessor will analyze the error internally and according to the signaled errors will do the following tasks. The coprocessor's designers categorized all errors in the following 6 classes:

2.3.1. Invalid operation

This can be: an upper or lower overflow of the coprocessor's internal stack. The lower overflow can appear when we try to access an element that doesn't exist on the stack. These are usually severe algorithmic errors; the coprocessor does not execute the operation.

We have an undefined result if we try to divide 0.0 by 0.0; the coprocessor is not prepared for this. Similar situations appear when we try to subtract infinite from infinite, etc. These errors (even tough they can be avoided by proper algorithms) are not as severe errors as stack overflows.

The same result will be obtained if a coprocessor function is called with wrong parameters.

If an undefined result appears the coprocessor puts into the characteristic a reserved value (bits of 0).

2.3.2. Overflow

The result is bigger than the largest number that can be represented. The coprocessor writes the infinite value instead of the result and moves on.

2.3.3. Division by zero

The divider is zero while the number to be divided is different from 0 or infinite. The coprocessor writes the infinite value instead of the result and moves on.

2.3.4. Underflow

The value of the result is smaller then the smallest number that can be represented. The result will be 0 and the coprocessor will move on.

2.3.5. De-normalized Operand

Appears if one of the operands is not normalized or the result can not be normalized (for example if it's so small that it normalization can not be done). The coprocessor moves on (the values that are not 0 will be lost, will be turned into 0).

2.3.6. Inexact result

The result of the operation is inaccurate due to necessary or prescribed rounding. This kind of results cad be obtained when dividing 2.0 by 3.0 and the result can be represented only as an infinite fraction. The coprocessor does the rounding and moves on.

The above were described in order of their severity. If a stack overflow appears the program is flawed and it will not be continued.

At the same time a rounding error needs not to be treated. Not even on paper can we use infinite fractions or irrational numbers as we would like. Practically speaking it is of no importance to us if we lose or not the 20th decimal of the fraction, because this is not the element that caries the important information. To solve this problem a thorough analysis of the situation and results that can appear, the representation's precision, running time, and memory size must be done. As we have seen when representing numbers, the precision from representing short real numbers is not enough for many practical applications. The precision of long real numbers is more than sufficient but occupies double memory space.

2.4. The coprocessors internal architecture

The coprocessor has two distinct components:

- Numerical execution unit: does the arithmetical and transfer instructions common to the coprocessor, and has an internal execution unit and a block of registers;
- Control unit: extracts from memory the instructions and operands and executes the control instructions, has a logical block, pointer and control registers;

2.4.1. The numerical execution unit

From the user's point of view the most important component are the general block registers that are organized as an **internal stack**. All the registers (stack elements) have 80 bits. Each operation is addressed to the element that is on top of the stack. That's why the stack's elements are named ST (0), ST (1)… etc., where ST (0) is the top of the stack, ST (1) the next element, and so on. It represents an inconvenient in assembling language programming as we have to save the stack position for each value, and when inserting a new element all previous elements' stack position is incremented.

2.4.2. Control unit

2.4.2.1. Control Word

The control register is a 16 bit register. The user can set the value of the register and thus access a series of the coprocessors "finer" mechanisms like the rounding method, etc.

As we can se the register is divided in two, that's because the first 8 most significant bits control the processor's working strategy the other 8 bits control the interrupts when an error occurs.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	X	X	ip	rc	rc	pc	pc	x	x	pm	um	om	zm	dm	im

X marks an unused bit
The others are:

IC – Infinite control- the way in which infinite numbers are treated

When division by 0 occurs the coprocessor puts infinite in the result. Mathematically speaking we have to ways of ending the number's axis: projective and affine. The difference between the two is that the latter knows two types of infinite (positive and negative). None of the two methods is better than the other. Bit coding:

0- projective
1- affine

RC- Rounding control

00-round to the closest element that can be represented
01-round downwards
10-round upwards
11-trunkation

PC- Precision Control

In some cases we do not want to work with the result in the internal precision even tough it is always represented as such. If we use previously written procedures designed for IEEE short format, error propagation with other precision is unpredictable. So we can force results to a given precision.

Values of the PC pair of bits

00 -24 bit-short real
01 -NA
10 -53 bit-long real precision
11 -63 bit-high precision

M. Mask – validates or invalidates the coprocessors interrupt. When an error occurs during a floating point operation, the coprocessor sends an interrupt to the processor. On IBM-PC the interrupt created is NMI, an unmasked processor interrupt

Values on MASK

0– validates an interrupt request
1- Invalidates an interrupt request

With the following bits we specify which exception (error) really calls the interrupt. This can be useful when we are not interested in a particular exception, or if we want to control the problem by reading the coprocessors state from the program. The following bits validate the interrupt on 0 and invalidate it on 1.

PM Precision Mask -signals rounding interrupt
UM Underflow Mask - signals underflow interrupt
OM Overflow Mask - signals overflow interrupt
ZM Zero Divide Mask - signals divide by zero interrupt
DM De-normalized OM - signals de-normalizing interrupt
IM Invalid Operation Mask - signals invalid operation interrupt

2.4.2.2. Status Word

It's a 16 bit register. Its content is set according to the last executed operation. We can obtain form it vital information for the user. Two of the first most significant bits correspond with the zero carry bits from the I8086 processor. Because we can load any value on the last 8 STATUS bits using the instruction SAHF, after a coprocessor's operation we can read and use these bits with simple control instructions.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	St	St	St	St	C2	C1	C0	Es	X	Pe	Ue	Ze	De	Ie

In this case we have a 16 bit register split in two. The first 8 bits tell us if the operation done by the coprocessor generates an interrupt, if it does then we are told what exception generated the interrupt. The first 8 most significant bits represent the coprocessors arithmetical status.
IR Interrupt request- If its value is one then it means that the coprocessor requests an interrupt. In this case one of the following six bits will also be 1, and will show what class the error belongs to.
IE Invalid Operation Error-invalid operation
DE Not normalized Operand Error – error caused by an operand that was not normalized or the result can not be normalized
ZE Zero Divide Error – division by zero caused error
OE Overflow Error - overflow caused error
UE Underflow Error - underflow caused error
PE Precision Error – precision error, the result was rounded
C0 (conditional bit 0)
C1 (1st conditional bit)
C2 (2nd conditional 2)
C3 (3rd conditional 3)
SP – the three bits point to the top of the stack. The value 000 signals an empty stack and the first element to be loaded will be the element 0 from the stack, while the value 111 signals that the stack is full.
B Busy – tells us if the coprocessor is working or not. It is active on 1, so in this case we are not allowed to send other instructions to the coprocessor. This bit allows us to synchronize our program with the coprocessor.

The meaning of C0, C1, C2, and C3 is displayed in the following table. As we can see these bits are not easy to define. Practically we only need to check one or two bits, which is fairly simple if we rely on the remarks made at the start of this chapter: we can use the fact that the zero STATUS and Carry flags have the same meaning.

C3	C2	C1	C0	Sign	Meaning
0	0	0	0	+	Not normalized
0	0	0	1	+	Not a number
0	0	1	0	-	Not normalized
0	0	1	1	-	Not a number
0	1	0	0	+	Normalized positive
0	1	0	1	+	Positive infinite
0	1	1	0	-	normalized
0	1	1	1	-	Negative infinite

1 1 1 1	0 0 0 0	0 0 1 1	0 1 0 1	+ empty - empty	Zero (positive) ..... zero (negative) .....
1 1 1 1	1 1 1 1	0 0 1 1	0 1 0 1	+ empty - empty	Invalid, Not normalized ..... Invalid, Not normalized .....

The value of C3 is 0 if the result of the operation (normalized of not normalized) is not 0. If the bit's value is 1 then the result is either 0 or invalid or the corresponding stack element is empty. It can be said that C3 greatly resembles the zero STATUS bit. The value of C2 depends on C3. If the value of C3 in 0, C2 points out the normalized result on 1, and a non-normalized result on 0. If the value of C33 is 1 (the result is 0 or the element is empty) then it's the opposite, C2 set on 1 points to a invalid number, while 0 points to zero.

As we have seen in the previous table C1 points to the number's sign. If the result is negative then C1 is 1 if not C1 is 0. C0 tells us if the result is valid or not. If C) is 0 then there are no severe errors, but if C0 is 1 the result is invalid (the result is infinite or other special value).

If other types of operations are being done, like comparisons then their meaning changes:

C3	C2	C1	C1	Meaning
0 0 1 1	0 0 0 0	X X X X	0 1 0 1	ST (0) >"op" ST (0) <"op" ST (0) ="op" ST (0) and "op" can not be compared

If C3 is 0 then the result of the comparison will be read from C0. If C3 is 1 then C2 will point out if the numbers are equal or not. Or ST (0) can not be compared (void or infinite).

After the partial remainder operation these bits have other meaning. In this case C0, C1, C2 (from top to bottom in this order) will keep the one, two or three bits of the result when the division has a remainder. The value of C2 is 0 after the creation of a partial remainder and 1 in case of error. The meaning is (about the creation of a partial remainder will discus at the instruction description):

Divider/Divided	C3	C1	C0
Divide>Divided /2	X	X	Bit 0
Divider>Divided /4	X	Bit 1	Bit 0
Divider<=Divided /4	Bit 2	Bit 1	Bit 0

X signifies that in that case the bits keep their previous value. For example, if the number to be divided is smaller than the number to be divided by then the remainder is equal to the divider and the result is 0. In this case C3 and C1 keep their value, and C0 will be 0 to signal that the result will be zero. And if the divider is smaller than half the number to be divided, but larger then the quarter of the number to be divided then the result will be 2 or 3; we have this number in C1, C0 and C3 keeps his previous value.

2.4.2.3. Tag Word

15 14	13 12	11 10	9 8	7 6	5 4	3 2	0 1
R7	R6	R5	R4	R3	R2	R1	R0

The bit pairings describe from top to bottom the stack registers 0, 1... Etc.
A bit pairing can have the following meanings:

Value	**Stack registers bits meaning**
00	The corresponding element contains a valid data
01	The corresponding element contains zero
10	The corresponding element contains a special value
11	The corresponding element is empty

The term special value means that the stack element contains infinite of for some reason the result of an operation is invalid.

2.4.2.4. Instruction Pointer

The instruction pointer contains the physical address and the coprocessor's last operation code.

This register helps us when we are writing interrupt routines for catching errors that appear during coprocessor operations. In these cases it is useful to know the operation's code and the physical address (the internal memory location where it can be found). We can easily see its importance when we realize that the program does not have to wait for the coprocessor, while this is working the processor can execute other tasks. We must consider the coprocessor only when its result is needed or when we need to do another operation.

Now, (because our program has passed the instruction that caused an error) we can't find out which was the last instruction sent to the coprocessor. The space reserved for the instruction code is larger then the true size of the code so the code appears as aligned to the right.

2.4.2.5. Data Pointer

The data pointer contains the physical address of the last external data utilized in the last floating point operation.

Like the previous this register uses interrupts for catching errors that may appear during the execution of coprocessor instructions. In this case we must know the data's physical address (external for the coprocessor) used by the last instruction that caused the error.

For the user that that write programs in assembling language it is important to know the coprocessor's condition, more exactly the coprocessor's environment that defines the working conditions at a particular time. This environment is defined by known elements.

2.5. Coprocessor's environment

The internal registers that the user can be access. The mathematical coprocessor has a set of registers of 14 bits organized like this:

	COMAND REGISTER
	STATUS REGISTER
	STACK REGISTER
A19-16	INSTRUCTIONS REGISTER (A15-0) | 0 | OPERATION CODE (bits 10-0)
A19-16	DATA REGISTER (A15-0) | 0. 0

2.6. Coprocessor's instruction set

The coprocessor can be programmed in assembly language using the ESC instruction. This instruction sends a 6 bit operation code on the data bus and if necessary also sends on the data bus a memory address. The coprocessor sees and takes the instruction sent to him and executes it. There are two ways of a new synchronization between the coprocessor and the processor, both attributed to the processor:

- the processor tests the coprocessor's status
- the processor calls a WAIT instruction

2.6.1. Data transfer instructions

Data transfer instructions ensure the exchange of data between the computers memory and the coprocessor's stack. They can be classified like this:

2.6.1.1. LOAD Instructions

FILD adr - Loads on the stack the integer variable located at address „adr". The variable stored in memory of type (DB, DW, and DD) is converted to the coprocessors internal format at load.

FLD adr - Loads on the stack the real variable (long or short) located at address „adr". The variable stored in memory of type (DD, DQ, and DT) is converted to the coprocessors internal format at load

FBLD adr - Loads on the stack the packed decimal variable located at address „adr". The variable stored in memory of type (DT) is converted to the coprocessors internal format at load.

2.6.1.2. STORE Instructions

FIST adr -Stores at the address „adr" the value located on the stack (ST (0)) as a number. The stored value can be only an integer represented on one byte or a short integer, corresponding to the data stored at address „adr" (DW or DD). The stack pointer remains unchanged after the data is stored. The conversion is done during the store process.

FISTP adr - Stores at the address „adr" the value located on the stack (ST (0)) as an integer number. The stored value can be any integer (byte integer, short integer, long integer, corresponding to the data stored at address „adr" (DW, DD or DQ). The conversion is done during the store process. The instruction changes the stack: ST (0) is deleted by decrementing the stack pointer.

FST adr - Stores at the address „adr" the value located on the stack (ST (0)) as an integer number. The stored value can be an integer short integer or in double precision, corresponding to the data stored at address „adr" (DD or DQ). The stack pointer and the data on the stack remain unchanged after the data is stored. The conversion is done during the store process.

FSTP adr - Stores at the address „adr" the value located on the stack (ST (0)) as a floating point number. The value can be a short real with double or extended precision, corresponding to the data stored at address „adr" (DD, DQ or DT). The conversion is done during the store process from the internal format. The instruction changes the stack: ST (0) is deleted by decrementing the stack pointer.

FBSTP adr - Stores at the address „adr" the value located on the stack (ST (0)) as a packed decimal number (defined at "adr" with DT). The stack pointer is decremented. The conversion is done during the store process from the internal format.

NOTE: You must remember that any type of data can be loaded. When we try to store we have two possibilities: If the data from the stack is to be eliminated we can use the 7 data types. But if we want to keep the stored value on the stack only the 4 basic typed are allowed.

2.6.2. Internal data transfer instructions

FLD ST (i) Put value from ST (i) on the stack. Thus the value from ST (i) will be found twice: in ST (0) and ST (i+1).

FST ST (i) The value from ST (0) is copied in the stack's "i" element. The old ST (i) is lost.

FSTP ST (i) The value from ST (0) is copied in the stack's "i" element. The old ST (i) is lost. ST (0) is eliminated by decrementing the stack pointer.

FXCH ST (i) swap between ST (0) and ST (i).

2.6.3. Constants loading instruction

FLDZ Loads 0 at the top of the stack

FLD1 Loads 1.0 at the top of the stack

FLDPI Loads"pi" at the top of the stack

FLDL2T Loads log (10) at the top of the stack

FLDL2E Loads log (e) at the top of the stack

FLDLG2 Loads log (2) at the top of the stack

FLDLN2 Loads ln (2) at the top of the stack

2.7. Arithmetical and comparison instructions

Arithmetical instructions usually have 2 operands. One of them is always at the top of the stack, and usually this is also the place where the result is be generated. Basic operations can be executed without restrictions with the following methods
-the instruction's mnemonic is being written without an operand. In this case the operands are ST (0) and ST (1).
-the instruction's mnemonic and the operand. The operand can be a memory address or a stack element (ST (1) is also usable but it's not very use full)
-the instruction's mnemonic is written and 2 operands: the first is a stack element (not ST (0)) and the second is ST (0). In this case the result will be put in the place of the first operand and ST (0) will be deleted from the stack. (In the instruction's mnemonic the letter P appears).

2.7.1. Arithmetical instructions

FADD ST (0) ←ST (0) +ST (1) FADD **op** ST (0) ←ST (0) +"op" from memory or stack.
Floating point operation.
FADD op ST (0) ←ST (0) +"op" from memory or stack.
Integer operation.
FADD ST (i), ST (0) ST (i) ←ST (i) +ST (0); ST (0) eliminated
FSUB ST (0) ←ST (0) -ST (1) FSUB **op** ST (0) ←ST (0) –„op" from memory or stack.
Floating point operation.
FISUB op ST (0) ←ST (0) -"op" from memory or stack.
Integer operation.
FSUB ST (i), ST (0) ST (i) ←ST (i) -ST (0); ST (0) eliminated
FSUBR ST (i) ST (i) ←ST (i) -ST (0) ; FSUB ST (i) opposite instruction
FMUL ST (0) ←ST (0) XST (1)
FMUL op ST (0) ←ST (0) x "op" from memory or stack.

Floating point operation.
FIMUL op ST (0) ←ST (0) x ”op” from memory or stack.
Integer operation.
FMULP ST (i), ST (0) ST (i) ←ST (i) x ST (0); ST (0) eliminated
FDIV ST (0) ←ST (0): ST (1) FDIV **op** ST (0) ←ST (0):”op” from memory or stack.
Floating point operation.
FDIV op ST (0) ←ST (0):”op” from memory or stack.
Integer operation.
FDIVP ST (i), ST (0) ST (i) ←ST (i): ST (0); ST (0) eliminated
FDIVR ST (i) ST (i) ←ST (i): ST (0); FDIV ST (i) opposite instruction.

2.8.2. Number comparison instructions

FCOM	The values from ST (0) and ST (1) are compared and C3, C2 and C0 indictors are set
FCOM op	The values ST from (0) and memory or stack (floating point variable) are compared and C3, C2 and C0 indictors are set
FICOM op	The values ST from (0) and memory or stack (floating point variable) are compared and C3, C2 and C0 indictors are set.
FCOMP	The values from ST (0) and ST (1) are compared and C3, C2 and C0 indictors are set. ST (0) is deleted from the stack
FICOMP op	The values ST from (0) and memory or stack (floating point variable) are compared and C3, C2 and C0 indictors are set ST (0) is deleted from the stack.
FCOMPP	The values from ST (0) and ST (1) are compared and C3, C2 and C0 indictors are set. ST (0) and ST (1) are deleted from the stack.
FTST	C3, C2 and C0 indicators are set according to the result of the comparison between the values ST (0) with 0.
FXAM	The condition bits are set according to the value of ST (0).No comparison is being done.

Remarks:

- FCOMP and FCOMPP allow us the easiest ways of eliminating one or two elements from the stack

- FXAM is used for analyzing the special conditions that result where computing errors occur.

2.8. Floating point functions

FSQRT Square root – ST (0) 's square root is put in ST (0). The number has to be positive, or the result will not make sense.

FSCALE 2's power. Puts in ST (0) the ST (0)' s value multiplied with $2^{ST(1)}$:
ST (0) ←ST (0) *2**ST (1) ST (1) has to be an integer, and ST (0)'s absolute value ha to be smaller then 2**15.

FPREM Partial remainder. ST (0) is divided by ST (1) and stored in ST () ←ST (0) -ST (1) * (the biggest lower integer for ST (0) /ST (1)).

FRMDINT Round. ST (0) is replaced with ST (0) rounded. The rounding method is set in the command line.

FXTRACT The value stored in ST (0) is split into Characteristic (in ST (0)) and mantissa (in ST (1)).

FABS ST (0) is replaced with its absolute value.

FCHS ST (0) sign is changed.

FPTAN Partial tangent. The tangent of the angle contained in ST (0) is determined as a ST (1) /ST (0) fraction. The initial value of the angle contained in ST (0) must be between 0 and "pi"/4.

FPATAN Partial Arctangent. The arctangent of the value contained in ST (0) is determined as a ST (1) /ST (0) fraction. The initial value contained in ST (0) must be positive, while ST (1) must be larger ST (0).

F2XM1 2's power. ST (0) will be replaced by 2**ST (0) -1. Initially ST (0) must be between 0 and 0.5.

FYL2X Logarithm. ST (0) ←ST (1) *LOG2 (ST (0)). ST (0) has to be a positive number, while ST (1) has to be a finite number.

FYL2XP1 Logarithm. ST (0) ←ST (1) *LOG2 (ST (0) +1). ST (0) has to be a positive number lower then 0.3, while ST (1) has to be a finite number.

Remarks:

- Sin and Cosine can be determined by using the tangent
- Any exponent can be computed using F2XM1
- For determining the exponent ST (0) $^{ST(1)}$ it is recommended to use the functions FYL2X and then F2XM1!

2.9. Control Instructions

Control instructions have the task of coordinating the microprocessors actions. Usually they have no arithmetic meaning, but some of them do influence drastically the actions of the coprocessor because they save or load the coprocessor's state, more exactly all of its work registers. Among these registers is the stack thus, these can be regarded as gigantic load and save instructions.

FINIT Initialization- the coprocessor is brought in an initial status known as software reset". After the FINIT instruction all of the coprocessors registers will be in their initial status and the stack will be empty.

FENI Interrupt accept- if the coprocessor needs to generate an interrupt when an error is detected, besides the correct positioning of the command register it needs to explicitly accept the interrupt.

FDISI Interrupt ignores- this instruction ignores all interrupts regardless of the command register's bits; to accept a new interrupt the instruction FENI must be called.

FLDCW adr The command register is loaded from the memory location indicated by **adr**

FSTCW adr The command register is saved in a word located at the memory location indicated by **adr**

FSTSW adr The status register is saved in a word located at the memory location indicated by **adr**.

FCLEX The bits that define the exceptions are erased- the instruction erases the corresponding bits regardless of the status of the error bits

FSTENV adr Environment save- the coprocessor's internal registers are saved in a memory location starting at **adr** that has a size of 14 bytes.

FLDENV adr Environment load- the coprocessor's internal registers are loaded from a memory location starting at **adr** that has a size of 14 bytes.

FSAVE adr Status save- the coprocessor's internal registers and its stack are saved in a memory location starting at **adr** that has a size of 94 bytes.

FRSTOR adr Status load- the coprocessor's internal registers and its stack are loaded from a memory location starting at **adr** that has a size of 94 bytes.

FINCSTP Stack indicators increment- after the instruction's action it is incremented with a stack indicator; thus the element that became ST (0) remains unchanged (fact pointed out by the stack description register's bits).

FDECSTP Stack indicators decrement- after the stack indicator is decremented by one; thus the stack's elements remain unchanged (fact pointed out by the stack description register's bits).

FFREE ST (i) the "i" ranked element from the stack is eliminated. The operation does not influence the stack pointer.

FNOP No operation executed.

FWAIT Waits for the current action to finish (similar to the 8086 WAIT instruction)

A simple program that uses the mathematical coprocessor

```
; Program that determines the area of a circle with the radius R
; And volume of a sphere with radius R

DATE      SEGMENT   PARA `DATA`    ;SEGMENT

RAZA      DQ        8. 567
ARIE      DQ        ?              ; RESERVE SPACE
VOLUM     DQ        ?              ; RESULTS
PATRU     DD        4. 0
TREI      DD        3. 0
DATE      ENDS

COD       SEGMENT   PARA `CODE`
.8087
CALCUL    PROC FAR  ;
          ASSUME CS:COD, DS: DATE

          PUSH DS              ; PREPARE
          XOR  AX, AX          ; STACK FOR
          PUSH      AX         ; DOS RETURN
          MOV  AX, DATE        ; LOADING DS
          MOV  DS, AX          ; WITH DATA SEGMENT
```

```
            FINIT           ;COPROCESOR INITIALIZATION
            FLD  RAZA       ;LOAD RAZA ON COPROC STACK
            FMUL RAZA       ;COMPUTE R x R
            FLDPI           ;LOAD PI TO COPROC STACK
            FMUL            ;COMPUTE R x R x PI
            FSTP ARIE       ;SAVING RESULT
            FWAIT           ;SYNCHRONIZATION


            LEA  SI, VOLUM ; VOLUM ADDRSS IN SI
            FINIT           ;COPROCESOR INITIALIZATION
            FLD  RAZA            ; COMPUTATION
            FMUL RAZA            ; R x R
            FMUL RAZA            ; R x R x R
            FLDPI                    ; LOAD PI
            FMUL                 ; MULTYPLY WITH PI
            FMUL PATRU           ; MULTIPLY WITH FOUR
            FDIV TREI                ; DIVISION BY 3
            FSTP QWORD PTR [SI]      ; SAVING RESULT
            FWAIT                ; SYNCHRONIZATION

            RET
CALCUL      ENDP                     ; END PROCEDURE
COD         ENDS                 ; END CODE SEGMENT

            END  CALCUL              ; PROGRAM END
```

3. Lab tasks

1. Run the given example

2. Write a program that determines $\sqrt[3]{2}$. Hint: Use the instructions F2XM1 and FYL2X.

Exam topics for Assembly language programming III E

1. **assembler operation**
2. **main assembler pseudo instruction**
3. **integer number representation in C1, C2 and Sign and Value**
4. **real number representation in IEEE short format**
5. **I-8086 addressing modes**
6. **physical and effective address**
7. **CPU registers and their functions**
8. **arithmetical and logical instructions**
9. **shift and rotate instructions**
10. **data transfer instructions**
11. **address transfer and stack instructions**
12. **input output instructions**
13. **jump, conditional jump and loop instructions**
14. **software interrupts**
15. **procedures**
16. **macro definitions**
17. **macro and procedure libraries**
18. **single and multiple segment programs**
19. **single and multiple module programs**
20. **protected mode operation of I-80x86 processor**
21. **memory management**
22. **main data structures used in protected mode operation**
23. **system function calls**
24. **math coprocessor structure and operation**
25. **math coprocessor data types**
26. **math coprocessor data transfer and constant load instructions**
27. **math coprocessor arithmetic and control instructions**
28. **math coprocessor mathematical functions**
29. **MMX extensions, data types and operating principles**
30. **multimedia calculus and main instruction families**
31. **program optimization, general issues, optimization levels**
32. **non optimal code determination methods**
33. **optimization techniques in ALP**