Seminar 1

Introduction to the IA-32 assembly language (Intel Architecture)

Contents

Seminar 1 1
1. Elements of the IA32 assembly language 1
1.1. Constants 2
1.2. Variables 2
1.3. Instructions 5
2. Structure of a program 6
3. Representation of integer numbers in the computer's memory 7
4. Signed and unsigned instructions 9
5. Signed and unsigned representation intervals 9

- **The IA-32 is a 32-bit computing architecture (basically meaning that its main elements have 32 bits in size)** and it is based on the previous Intel 8086 computing architecture.
- it is an abstract model of a microprocessor specifying the microprocessor's elements, structure and instruction set.

1. Elements of the IA32 assembly language

An **algorithm** is a sequence of steps/operations necessary to solve a specific problem.
For example, the algorithm for solving the 2^{nd} degree algebraic equation $a*x^2+b*x+c=0$ contains the steps:
1) compute the value of delta
2) if delta is greater or equal to zero, compute the solutions x1 and x2 using the well-known formulas.

An **algorithm** contains two things:

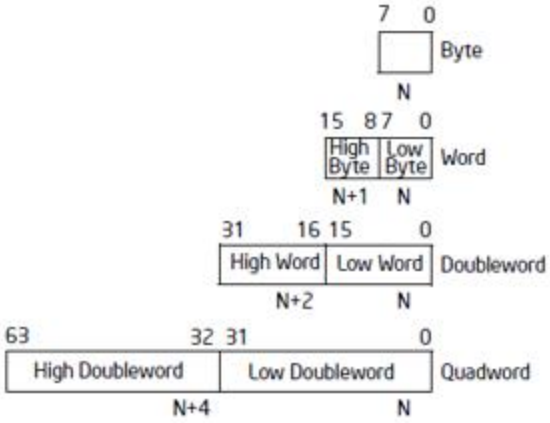
- a set of data/entities a, b, c, we need to compute x1 and x2
- and a sequence of operations/steps

When an algorithm is specified in a programming language, we refer to this algorithm as a **program**.

Throughout the semester we will study the IA-32 assembly language, starting with the **data part** of the assembly language and later the **operational part (instructions)**.

All data used in an IA-32 assembly program is **essentially numerical** (integer numbers) and can have different **types**:

- **Byte** – data is represented on 8 bits
  - **What is a bit?** The bit is **the smallest quantity of information** and it represents a **binary digit (0/1)**.
- **Word** – contains 2 bytes (data is represented on 16 bits)
- **Doubleword** – contains 4 bytes (data is represented on 32 bits)
- **Quadword** – contains 8 bytes (64 bits)



In the IA-32 assembly language we have:

- data that changes its value throughout the execution of the program (i.e. variable data or **variables**)
- data that does not change its value throughout the execution of the program (i.e. constant data or **constants**)

1.1. Constants

We have 3 types of constants in the IA-32 assembly language:

- numbers (natural or integer):
  - written in base 2; ex.: 101b, 11100b
  - written in base 16; ex.: 34ABh, 0ABCDh
  - written in base 10; ex.: 20, -114
- character; ex.: ‘a’, ‘B’, ‘c’ .. – we use **apostrophe / single quote**
- string (sequence of characters); ex.: ‘abcd’, “test” …

Declaring a constant:

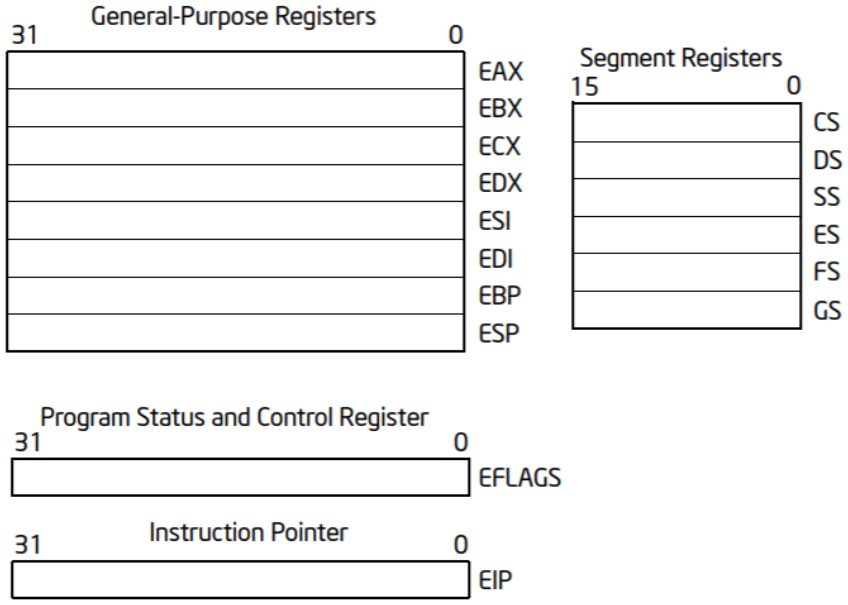
ten **EQU** 10

1.2. Variables

The IA-32 assembly language has 2 kinds of variables: pre-defined variables and user-defined variables. A variable has a **name**, a **data type** (*byte, word* or *doubleword*), a current **value** and a **memory location** (where the variable is stored).

Pre-defined variables (CPU registers):

The CPU registers are memory areas located on the CPU which are used for various computations. The IA-32 CPU registers are:



1) General registers (each register has 32 bits in size):

- **Obs**. **Every register may also be seen as the concatenation of two 16 bits subregisters. The upper register, which contains the most significant 16 bits of the 32 bits register, doesn't have a name and it isn't available separately**

- General-purpose registers are represented on a **doubleword,** but we can also use them as word or byte, for example:

EAX (32 de biti)			
Word high		Word low: **AX** (16 biti)	
Byte high din word high	Byte low din word high	Byte high din word low: **AH** (8 biti)	Byte low din word low: **AL** (8 biti)

! we cannot access the high word of EAX (similarly for EBX, ECX, etc.)

- EAX - **accumulator**. **Used by most instructions as one of their operands**
  - the lower or least significant part of EAX can be referred by AX
  - AX is formed by two 8-bit subregisters, AL and AH
- EBX - **base register**
  - the lower or least significant part of EBX can be referred by BX
  - BX is formed by two 8-bit subregisters, BL and BH
- ECX - **counter register**
  - the lower or least significant part of ECX can be referred by CX
  - CX is formed by two 8-bit subregisters, CL and CH)
- EDX - **data register**
  - the lower or least significant part of EDX can be referred by DX
  - DX is formed by two 8-bit subregisters, DL and DH

- ESP - **stack register**
  - the lower or least significant part of ESP can be referred by SP
- EBP - **stack register**
  - the lower or least significant part of EBP can be referred by BP
- EDI - **index register** (**Destination Index**)
  - the lower or least significant part of EDI can be referred by DI
  - usually used for accessing elements from bytes and words strings
- ESI - **index register** (**Source Index**)
  - the lower or least significant part of ESP can be referred by SI
  - usually used for accessing elements from bytes and words strings

2) Segment registers (each register has 16 bits – represented as word):

Every program is composed by one or more segments of one or more types. At any given moment during run time there is only at most one active segment of any type

Registers **CS (Code Segment)**, **DS (Data Segment), SS (Stack Segment)** and **ES (Extra Segment)** from BIU contain the values of the selectors of the active segments, correspondingly to every type. So registers CS, DS, SS and ES determine the *starting addresses* and *the dimensions of the 4 active segments*: code, data, stack and extra segments.

Registers FS and GS can store selectors pointing to other auxiliary segments without having predetermined meaning.

CS, DS, SS, ES, FS, GS – are not used in a program

3) Other registers (32 bit registers): ***EIP and Flags***.
Register EIP (which offers also the possibility of accessing its less significant word by referring to the IP subregister) contains **the offset of the current instruction inside the current code segment**, this register being managed exclusively by BIU.

User-defined variables:
For these variables, the programmer has to define the **name**, data **type** and initial **value**.
Examples:

- *a* ***DB*** *23* ; defines the variable with the name "a", data type byte (DB-Define Byte) and initial value 23
- *c* ***DB*** *'a'* ; defines the variable with the name "c" with a character as an initial value
- *s* ***DB*** *'Hello, how are you?'* ;define the variable with the name "s" with a string as an initial value
- *a1* ***DW*** *23ABh* ; defines the variable with the name "a1", data type word (DW-Define Word) and initial value 23ABh
- *a12* ***DD*** *-101* ; defines the variable with the name "a12", data type doubleword (DD-Define DoubleWord) and initial value -101.
- b **DQ** 10

Declaring variables with no initial value:

- a **RESB** 1 ; reserve 1 byte
- a **RESB** 64 ; reserve 64 bytes
- a **RESW** 1 ; reserve 1 word
- **RESD, RESQ**

1.3. Instructions

- Usually instructions have at most two operands and can be of the form INSTRUCTION destination, source
- Of the two operands, at least one of the operands should be a register or a constant value (at most one operand can be a memory location)

MOV Instruction

Syntax: **MOV dest, source**

Effect: dest = source

Restrictions:

- dest and source are either registers, variables of type byte, word or dword, or constants;
- dest cannot be a constant
- both operands need to be of the **same size;**
- at least one of the operands should be a register or a constant value

Example:

```
mov ax, 3 ; ax = 3
mov bx, ax ; bx=ax
mov [a], eax
```

Arithmetic expressions

1. ADD dest, source ;

Effect: dest = dest + source

Restrictions:

- dest and source are either registers, variables of type byte, word or dword, or constants;
- dest cannot be a constant
- at least one of the operands should be a register or a constant value
- both operands should have the same type: byte, word or doubleword

Examples:

```
add bx, cx
add word [a], 101b
```

2. SUB dest, source

Effect: dest = dest - source

- dest and source are either registers, variables of type byte, word or dword, or constants;
- dest cannot be a constant
- at least one of the operands should be a register or a constant value
- both operands should have the same type: byte, word or doubleword

Examples:

```
sub ax, 2
sub [a], eax
```

Seminar I. Introduction to the IA-32 assembly language. Converting numbers between numbering bases 2, 10, 16. Representation of integer numbers in the computer's memory. Signed and unsigned instructions.

The IA-32 computing architecture is the microprocessor (CPU) architecture introduced by the Intel Corporation in 1985 for their 80386 microprocessor. It is an abstract model of a microprocessor specifying the microprocessor's elements, structure and instruction set. The IA-32 is a 32-bit computing architecture (basically meaning that its main elements have 32 bits in size) and it is based on the previous Intel 8086 computing architecture.

I.1. The elements of the IA32 assembly language

An **algorithm** is, as you well know, a sequence of steps/operations necessary in order to solve a specific (mathematical or not) problem. For example, the algorithm for solving the 2nd degree algebraic equation $a*x^2+b*x+c=0$ contains the steps:

1) compute the value of *delta*

2) if *delta* is greater or equal to zero, compute the solutions x^1 and x^2 using the well-known formulas.

But besides this sequence of steps/operations, al algorithm also includes a set of data/entities on which those steps/operations operate. For our example, the data of the algorithm is: *a, b, c, delta,* x^1 *and* x^2.

So, an algorithm is two things:

- a set of data/entities
- and a sequence of operations/steps

An algorithm can be described using the natural language (e.g. romanian language, English language etc.) or it can be described in a programming language (e.g. C, Java, python, php etc.). When an algorithm is specified in a programming language, we refer to this algorithm as a **program.** In a similar way, a program contains: a) a set of data/entities and b) a set of operations/instructions.

Throughout the semester we will study the IA-32 assembly language. We will first describe the data part of the assembly language and later the operational part (instructions). All data used in an IA-32 assembly program is essentially numerical (integer numbers) and can have 3 basic types:

- byte – that data is represented on 8 bits
- word – that data is represented on 16 bits
- doubleword – that data is represented on 32 bits

In the IA-32 assembly language we have data that does not changes its value throughout the execution of the program (i.e. constant data or constants) and data that does change its value throughout the execution of the program (i.e. variable data or variables).

I.1.1 Constants

We have 3 types of constants in the IA-32 assembly language:

- numbers (natural or integer):

    - written in base 2; ex.: 101b, 11100b
    - written in base 16; ex.: 34ABh, 0ABCDh
    - written in base 10; ex.: 20, -114
- character; ex.: ‘a’, ‘B’, ‘c’ ..
- string (sequence of characters); ex.: ‘abcd’, “test” …

I.1.2 Variables

The IA-32 assembly language has 2 kinds of variables: pre-defined variables and user-defined variables. A variable has a name, a data type (byte, word or doubleword), a current value and a memory location (where the variable is stored).

Pre-defined variables (CPU registers):

The CPU registers are memory areas located on the CPU which are used for various computations. The IA-32 CPU registers are:

1) General registers (each register has 32 bits in size):
- EAX (the lower or least significant part of EAX can be referred by AX and AX is formed by two 8-bit subregisters, AL and AH)
- EBX (the lower or least significant part of EBX can be referred by BX and BX is formed by two 8-bit subregisters, BL and BH)
- ECX (the lower or least significant part of ECX can be referred by CX and CX is formed by two 8-bit subregisters, CL and CH)
- EDX (the lower or least significant part of EDX can be referred by DX and DX is formed by two 8-bit subregisters, DL and DH)
- ESP (the lower or least significant part of ESP can be referred by SP)
- EBP (the lower or least significant part of EBP can be referred by BP)
- EDI (the lower or least significant part of EDI can be referred by DI)
- ESI (the lower or least significant part of ESP can be referred by SI)

2) Segment registers (each register has 16 bits):
CS, DS, SS, ES, FS, GS – are not used in a program
3) Other registers (32 bit registers): EIP and Flags.

User-defined variables:

For these variables, the programmer has to define the name, data type and initial value.
Examples:
1) *a DB 23* : defines the variable with the name “a”, data type byte (DB-Define Byte) and initial value 23
2) *a1 DW 23ABh* : defines the variable with the name “a1”, data type word (DW-Define Word) and initial value 23ABh
3) *a12 DD -101* : defines the variable with the name “a12”, data type doubleword (DD-Define DoubleWord) and initial value -101.

I.1.3 Instructions

MOV – assignment instruction
Syntax: mov dest, source

(where dest and source are either registers, variables or constants of type byte, word or dword; dest can not be a constant)
Effect: dest := source
Examples: mov ax, 2
mov [a], eax

ADD – addition instruction
Syntax: add dest, source
(where dest and source are either registers, variables or constants of type byte, word or dword; dest can not be a constant)
Effect: dest := dest + source
Examples: add bx, cx
add [a], 101b

SUB – substraction instruction
Syntax: sub dest, source
(where dest and source are either registers, variables or constants of type byte, word or dword; dest can not be a constant)
Effect: dest := dest - source
Examples: sub ax, 2
sub [a], eax

I.2. The 1st 32bit 8086 assembly language program

```
;
; Comments are preceded by the ';' sign. This line is a comment (is ignored by the assembler)
; This program computes the expression: x:= a + b – c = 3 + 4 – 2 = 5.
;
;
bits 32
; declare the EntryPoint (a label defining the very first instruction of the program)
global start

; declare external functions needed by our program
extern exit                ; tell nasm that exit exists even if we won't be defining it
import exit msvcrt.dll     ; exit is a function that ends the calling process. It is defined in msvcrt.dll


; our data is declared here (the variables needed by our program)
segment data use32 class=data
   ; ...
```

```
        a dw 3
        b dw 4
        c dw 2
        x dw 0

; our code starts here
segment code use32 class=code
start:
        mov ax, [a]             ; ax := a = 3
        add ax, [b]             ; ax := ax + b = 3+4 = 7
        sub ax, [c]             ; ax := ax – c = 7 – 2 = 5
        mov [x], ax             ; x := ax = 5

        ; exit(0)
        push    dword 0         ; push the parameter for exit onto the stack
        call    [exit]          ; call exit to terminate the program
```

I.3. Converting numbers between numbering bases 2, 10, 16

A number is converted from a *source base* to a *destination base*. There are two algorithms that are mostly used for converting a natural number between numbering bases: one is based upon successive division operations and the other is based on successive multiplication operations.

The conversion algorithm that uses successive divisions

- This algorithm is useful when converting a number from base 10 to another numbering base (because computations are done in the source base; i.e. base 10)
- The initial number is continuously divided to the destination base (i.e. the initial number is divided to the destination base, then the obtained quotient (romanian: catul) is divided to the destination base and so on..) until we get a zero quotient. The remainders (Romanian: resturile) obtained taken in the reverse order form the representation of the initial number in the destination base

Ex.1: The representation of the number 23 (currently written in base 10) in the new base 2 is: 10111.
Ex.2: The representation of the number 28 (currently written in base 10) in the new base 16 is: 1C (because the digit representing 12 in base 16 is **C**).

The conversion algorithm that uses successive multiplications

- This algorithm is useful when converting a number from base different than 10 to base 10 (because computations are done in the destination base; i.e. base 10)
- Considering that the representation of the number in base s is: $a_n a_{n-1} \dots a_1 a_0$, the representation of this number in the destination base d will be computed like this:

$a_n *s\hat{}n + a_{n-1}*s\hat{}(n-1) + \ldots + a_1 *s\hat{}1 + a_0*s\hat{}0$

where computations are done in base *d*.

Ex.1: The representation of the number 10111 (currently written in base 2) in the new base 10 is: 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 23

Ex.2: The representation of the number 1C (currently written in base 16) in the new base 10 is: 1*16^1 + 12*16^0 = 28

Table useful for performing fast conversions between bases 2, 10 and 16

The following table will be useful for performing fast conversions of a semi-byte (a 4 bits/binary digits number) from base 2 to 10 and 16 and reverse.

Base 2	**Base 10**	**Base 16**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Binary digit = **Bit** (a software concept that represents the smallest quantity of information)

I.4. Representation of integer numbers in the computer's memory

Consider the following instruction:

mov ax, 7

The above instruction instructs the CPU (i.e. microprocessor) to set the value of the ax register (which is a memory zone on the CPU) to 7. A natural question that arises is: how does the CPU *represent integer numbers in the memory (and also on the CPU registers)* ?

The CPU represents an integer number on 1, 2, 4 or 8 bytes on the IA-32 architecture (1 byte = 8 consecutive bits). In fact, there are two kinds of representation of integer numbers in the computer's memory: signed representation and unsigned representation. *The CPU choses one of these two representations depending on the specific instruction it executes*.

Unsigned representation of numbers

- in unsigned representation we can only represent positive natural numbers
- the unsigned representation of a positive number is equal to the representation of that number in base 2
- ex.1: the unsigned representation of 17 on 8 bits is : 0001 0001
- ex.2: the unsigned representation of 39 on 8 bits is : 0010 0111

Signed representation of numbers

- in the signed representation we can represent positive and negative integer numbers
- the signed representation of a positive number is equal to the unsigned representation of that number (i.e. it is equal to the representation of that number in base 2)
- the signed representation of a negative number is equal to the representation in *2's complementary code* (Romanian: codul complementar fata de 2) of that number; in order to obtain the *2's complementary code* of a negative number, we substract the absolute value of the number (Romanian: modulul numarului) from 1 followed by as many zeroes as needed in order to represent the absolute value of the number.
- in the signed representation, the most significant bit (i.e. binary digit) of the representation is the sign bit(1=negative number; 0=positive number).
- ex.1: the signed representation of 17 on 8 bits is : 0001 0001 (the most significant bit is 0, so the number is positive)
- ex.2: the signed representation of -17 on 8 bits is : 1110 1111 (note that the sign bit is 1 in this case, so the number is negative)

  1 0000 0000 –
  1 0001
  1110 1111
- ex.3: the signed representation of -39 on 16 bits is : 1111 1111 1101 1001 (note that the sign bit is 1 in this case, so the number is negative)

  1 0000 0000 0000 0000–
  10 0111
  1111 1111 1101 1001

Now, we can consider the reverse problem of "representation", that is "interpretation". Let's assume that the binary content of the AL register is 1110 1111 and the next instruction to be executed is:

mul bl

The mul instruction just multiplies the value from the AL register with the value from the BL register and stores the result in AX (more details about the mul instruction will be given in seminar no. 2). When the CPU executes the above instruction it needs to ask itself the question (the human programmer also asks himself the same question): *what integer number does the sequence of bits from AL (i.e. 1110 1111) represents in our conventional numbering system (i.e. base 10)*? The CPU must *interpret* the sequence of bits from AL into a number in order to perform the mathematical operation (multiplication).

Just like we have two types of "representations", we also have two corresponding types of "interpretations": signed interpretation and unsigned interpretation.
In our example where we have in the AL register the sequence of 8 bits: 1110 1111, this value can be interpreted:

- unsigned: in this case, we now that in the unsigned representation, only positive numbers are represented, so our sequence of bits represents a positive number and it is the representation in base 2 of that number; so the number in base 10 is:

  1*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1* 2^2 + 1*2^1 + 1*2^0=
  128 + 64 + 32 + 0 + 8 + 4 + 2 + 1 = 239
- signed: in this case, we know that in the signed representation the most significant bit of the representation is the sign bit; our sign bit is 1 which means that this is the signed representation of a negative number; in other words this is the complementary code representation of the negative number; in order to obtain the *direct code* (the representation in base 2 of the absolute value of the number) we use the following rule: *take all the bits of the complementary code representation, from right to left, keep all the bits until the first 1, including this one, and reverse the remaining bits (1 becomes 0, 0 becomes 1)*. So, for our example of 1110 1111, the direct code is: 0001 0001 = 17. So the sequence of 8 bits 1110 1111 from the memory is interpreted signed into the number -17.

I.5. Signed and unsigned instructions

On the IA-32 architecture, related to the unsigned and signed representation of numbers, there are 3 classes of instructions:

- instructions which do not care about signed or unsigned representation of numbers: **mov, add, sub**
- instructions which interpret the operands as unsigned numbers: **div, mul**
- instructions which interpret the operands as signed numbers: **idiv, imul, cbw, cwd, cwde**

It is important to be consistent when developing a IA-32 assembly program: either consider all numerical values in a program to be unsigned (in which case you should use only instructions from class 1 and 2) or consider all numerical values in a program to be signed (in which case you should use only instructions from class 1 and 3).

2. Structure of a program

```
; Comments are preceded by the ';' sign. This line is a comment (is ignored by the assembler)

bits 32 ; assembling for the 32 bits architecture

global start ; we ask the assembler to give global visibility to the symbol called start
             ;(the start label will be the entry point in the program)
extern exit ; we inform the assembler that the exit symbol is foreign
             ;it exists even if we won't be defining it
import exit msvcrt.dll  ;we specify the external library that defines the symbol
      ; msvcrt.dll contains exit, printf and all the other important C-runtime functions

; our variables are declared here (the segment is called data)
segment data use32 class=data
; ...
   a dw 3
   b dw 4

; the program code will be part of a segment called code
segment code use32 class=code
start:
; ...
   mov ax, [a]   ; ax = a
   add ax, [b]    ; ax = a + b = 7
   ; call exit(0) , 0 represents status code: SUCCESS
   push dword 0 ; saves on stack the parameter of the function exit
   call [exit] ; function exit is called in order to end the execution of
the program
```

Exercises

Ex 1: 4+2

segment code:

```
        Mov al,4        ; al=4
        Add al,2        ; al=1+2=3
```

Ex 2: 1-5

segment code:

```
        Mov al,1        ; al=1
        Sub al,5        ; al=-4
```

Ex. 3: a+b, a,b – word

```
segment data
        a dw 5
        b dw 7
segment code
        Mov  AX, [a]; ax=5
        Add AX, [b]; ax=12
```

Ex. 4: (a+b) - c; a,b,c – doublewords

```
segment data
        A  dd 2
        B dd 5
        C dd 12
segment code
        Mov eax, [a]; eax=2
        Add eax, [b]; eax=7;
        Sub eax, [c]; eax=-5
```

3. Representation of integer numbers in the computer's memory

Consider the following instruction:

mov ax, 7

ax=0007h = 00000000 00000111b (you will talk about converting numbers between bases at your first lab)

The above instruction instructs the CPU (i.e. microprocessor) to set the value of the ax register (which is a memory zone on the CPU) to 7. A natural question that arises is: **how does the CPU *represent integer numbers in the memory (and also on the CPU registers)* ?**

The CPU represents an integer number on 1, 2, 4 or 8 bytes on the IA-32 architecture (1 byte = 8 consecutive bits). In fact, there are **two kinds of representation** of integer numbers in the computer's memory: **signed representation and unsigned representation**. *The CPU **chooses** one of these two representations **depending on the specific instruction** it executes*.

Unsigned representation of numbers

- in unsigned representation we can **only represent positive** natural numbers
- the unsigned representation of a positive number is equal to the representation of that number in base 2
- ex.1: the unsigned representation of 17 on 8 bits is : 0001 0001
- ex.2: the unsigned representation of 5 on 8 bits is : 0000 0101

Signed representation of numbers

- in the signed representation we can represent positive and negative integer numbers
- the **signed representation of a positive number is equal to the unsigned representation of that number** (i.e. it is equal to the representation of that number in base 2)
- the **signed representation** of a negative number is equal to the representation **in *2's complementary code*** (Romanian: codul complementar fata de 2) of that number; in order to obtain the *2's complementary code* of a negative number, **we subtract the absolute value of the number (Romanian: modulul numarului) from 1 followed by as many zeroes as needed in order to represent the absolute value of the number**.
- in the signed representation, the most significant bit (i.e. binary digit) of the representation is the sign bit (1=negative number; 0=positive number).
- ex.1: the signed representation of **17** on 8 bits is : 0001 0001 (the most significant bit is 0, so the number is positive)
- ex.2: the signed representation of **-17** on 8 bits is : 1110 1111 (note that the sign bit is 1 in this case, so the number is negative)

```
1 0000 0000 –
      1 0001
-------------
 1110 1111
```

- ex.3: the signed representation of -39 on 16 bits is : 1111 1111 1101 1001 (note that the sign bit is 1 in this case, so the number is negative)

```
1 0000 0000 0000 0000–
           10 0111
-------------------
 1111 1111 1101 1001
```

Now, we can consider the reverse problem of "representation", that is "interpretation". Let's assume that the binary content of the AL register is 1110 1111 and the next instruction to be executed is:

```
mul BL
```

The mul instruction just multiplies the value from the AL register with the value from the BL register and stores the result in AX (more details about the mul instruction will be given in seminar no. 2).

When the CPU executes an instruction (for example a multiplication) it needs to ask itself the question (the human programmer also asks himself the same question): ***what integer number does the sequence of bits from AL (i.e. 1110 1111) represents in our conventional numbering system*** *(i.e. base 10)*? The CPU must ***interpret*** the sequence of bits from AL into a number in order to perform the mathematical operation (multiplication).

Just like we have two **types of "representations**", we also have two corresponding **types of "interpretations**": **signed interpretation and unsigned interpretation.**

In our example where we have in the AL register the sequence of 8 bits: 1110 1111, this value can be interpreted:

- unsigned: in this case, we now that in the unsigned representation, only positive numbers are represented, so our sequence of bits represents a positive number and it is the representation in base 2 of that number; so the number in base 10 is:

  1*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1* 2^2 + 1*2^1 + 1*2^0=

  128 + 64 + 32 + 0 + 8 + 4 + 2 + 1 = 239
- signed: in this case, we know that in the signed representation the most significant bit of the representation is the sign bit; our sign bit is 1 which means that this is the signed representation of a negative number; in other words this is the complementary code representation of the negative number; in order to obtain the *direct code* (the representation in base 2 of the absolute value of the number) we use the following rule: *take all the bits of the complementary code representation, from right to left, keep all the bits until the first 1, including this one, and reverse the remaining bits (1 becomes 0, 0 becomes 1)*. So, for our example of 1110 1111, the direct code is: 0001 0001 = 17. So the sequence of 8 bits 1110 1111 from the memory is interpreted signed into the number -17.

4. Signed and unsigned instructions

On the IA-32 architecture, related to the unsigned and signed representation of numbers, there are 3 classes of instructions:

- instructions which do not care about signed or unsigned representation of numbers: **mov, add, sub**
- instructions which interpret the operands as unsigned numbers: **div, mul**
- instructions which interpret the operands as signed numbers: **idiv, imul, cbw, cwd, cwde**

It is important to be consistent when developing a IA-32 assembly program: either consider all numerical values in a program to be unsigned (in which case you should use only instructions from class 1 and 2) or consider all numerical values in a program to be signed (in which case you should use only instructions from class 1 and 3).

5. Signed and unsigned representation intervals

Number of bytes	Unsigned representation	Signed representation
1	$[0, 2^8-1] = [0, 255]$	$[-2^7, 2^7-1] = [-128, 127]$
2	$[0, 2^{16}-1] = [0, 65535]$	$[-2^{15}, 2^{15}-1] = [-32\,768, 32\,767]$
4	$[0, 2^{32}-1] = [0, 4\,294\,967\,295]$	$[-2^{31}, 2^{31}-1] = [-2\,147\,483\,648, 2\,147\,483\,647]$
8	$[0, 2^{64}-1] = [0, 18\,446\,824\,753\,389\,551\,615]$	$[-2^{63}, 2^{63}-1] = [-9\,223\,412\,376\,694\,775\,808, 9\,223\,412\,376\,694\,775\,807]$

Seminar 2

Signed and unsigned instructions. Arithmetic instructions (multiplications and divisions). Signed and unsigned conversions.

Contents

Seminar 2 1

1. Signed and unsigned instructions 1

2. (Signed and unsigned) multiplication and divisions instructions 2

3. Signed and unsigned conversions 5

1. Signed and unsigned instructions

On the IA-32 architecture, related to the unsigned and signed representation of numbers, there are 3 classes of instructions:

- instructions which do not care about signed or unsigned representation of numbers: **mov, add, sub**
- instructions which interpret the operands as unsigned numbers: **div, mul**
- instructions which interpret the operands as signed numbers: **idiv, imul, cbw, cwd, cwde**

It is important to be consistent when developing a IA-32 assembly program: either consider all numerical values in a program to be unsigned (in which case you should use only instructions from class 1 and 2) or consider all numerical values in a program to be signed (in which case you should use only instructions from class 1 and 3).

Important rules that must be obeyed by arithmetic instructions with 2 operands:

- all operands must have the same size/type (i.e. you can add byte to byte, but not byte to a word)
- at least one of the operands must be a general register or a constant and if it is a constant, this constant can not appear as a destination operand

Related to the above two rules, let's assume we have the following code:

```
a db 10
b db 11
……..
add ax, [a]
add [a], [b]
```

The instruction *add [a], [b]* would fail, meaning that there will be a compile error (i.e. assembly error) and the executable file cannot be built. This is because that instruction does not obey the second rule from above ([a] and [b] are both memory references / variables). On the other hand, although the instruction *add ax, [a]* breaks rule number one from above (since AX is a word and [a] was declared as byte), the compiler will not complain and will build the executable file! But when this instruction gets executed, it will not do what you meant: add the byte "a" to the register "AX"! Instead it will add a *word from the memory that starts where variable "a" starts* (this word is composed from the bytes 10 and 11) to the register AX. This is because although variable "a" was declared as a byte using the Define Byte (DB) directive, the NASM assembler does not link the type (data size) to a memory location in instructions. The declaration "a db 10" just declares/reserves 1 byte at the current memory address. You can then write in your code *mov ax, [a]* and the assembler will tell the CPU to move a word starting in the memory at the address of "a" into AX. Or you can write the code *mov eax, [a]* and the assembler will tell the CPU to move a doubleword starting in the memory at the address of "a" into EAX. *[a]* means just the starting point in the memory of a data – it does not say anything about the type (size) of "a". The type (size) information is inferred from the other operand of the instruction together with the first rule described above: all operands must have the same size/type.

Obs. "[ ]" is the addressing operator and its effect can be explained in the following two examples:

mov EAX, [a] ⇒ moves in EAX a doubleword starting at the address of variable "a" (i.e. it moves in EAX the value of variable "a", assuming "a" was declared as a doubleword)

mov EAX, a ⇒ moves in EAX the starting address (i.e. the offset – you will see later what this means) of variable "a" (NOT the value of variable "a" !)

2. (Signed and unsigned) multiplication and divisions instructions

MUL – unsigned multiplication instruction

Syntax: mul source

(where source is either register or variable of type byte, word or dword)

Effect: - if source is a byte ⇒ AX =AL * source

- if source is a word ⇒ DX:AX = AX * source

- if source is a dword ⇒ EDX:EAX = EAX * source

Example: The instruction *mul BX* stores in two 16-bit registers the result of the multiplication which is a 32-bit number. More specifically, the effect of this instruction is: DX:AX:= AX*BX. The result of the multiplication (a 32-bit number) is stored in the registers DX and AX instead of a proper 32-bit register like EAX for compatibility reasons with the previous Intel 8086 computing architecture. Let's assume that the result of the above multiplication would be the number 12345678h (in the hexadecimal base). The least significant (low) 16 bits of this number would be stored in AX and the most significant (high) 16 bits of this number would be stored in DX. Knowing that a hexadecimal digit is represented on 4 bits, we conclude that AX would store 5678h and DX would store 1234h (the hexadecimal number 1234h occupies 16 bits).

DIV – unsigned division instruction

Syntax: div source

(where source is either register or variable of type byte, word or dword)

Effect: - if source is a byte ⇒ AL =AX / source (quotient/catul) and

AH:=AX % source (remainder/restul)

- if source is a word ⇒ AX =DX:AX / source (quotient) and

DX:=DX:AX % source (remainder)

- if source is a dword ⇒ EAX = EDX:EAX / source (quotient) and

EDX = EDX:EAX % source (remainder)

IMUL – does the same thing as MUL but considers the operands as signed numbers

IDIV – does the same thing as DIV but considers the operands as signed numbers

Examples

Ex1. Compute the value of the expression x = ((a + b) * c) / d where all numbers are unsigned numbers and a, b, c, d are all bytes.

```
;
; BEGIN 32 bits PROGRAM
;
bits 32
; declare the EntryPoint (a label defining the very first instruction of the program)
global start

; declare external functions needed by our program
extern exit                 ; tell nasm that exit exists even if we won't be defining it
import exit msvcrt.dll      ; exit is a function that ends the calling process. It is defined in msvcrt.dll
```

```
; our data is declared here (the variables needed by our program)
segment data use32 class=data
        a db 3
        b db 4
        c db 2
        d db 3
        x db 0

; our code starts here
segment code use32 class=code
start:
        mov al, [a]                     ; AL:=a = 3
        add al, [b]                     ; AL:=AL+b = 3+4 = 7
        mul byte [c]                    ; AX:=AL*c = 7*2 = 14
                                        ; for this mul instruction we had to specify the type of operand [c]
                                        ; (i.e. byte), so that mul knows what to do. Remember, "c" is just
                                        ; a memory reference, it does not have a type associated to it!
        div byte [d]                    ; AL:=AX / d = 14 / 3 = 4      AH:=AX % d = 14 % 3 = 2
                                        ; similar to the above mul, we had to explicitly specify the type of
                                        ; [d] (i.e. byte)
        mov [x], AL                     ; x:=AL = 4

        ; exit(0)
        push    dword 0         ; push the parameter for exit onto the stack
        call    [exit]              ; call exit to terminate the program
```

3. Signed and unsigned conversions

Unsigned conversion: place zeroes in the high part ("manually")

How do we convert AL to AX?

- MOV AH, 0

How do we convert BX to CX:BX?

- MOV CX, 0

How do we convert AX to EAX?

```
MOV BX, AX
MOV EAX, 0   ;because we cannot directly access the high word from EAX!
MOV AX, BX
```

Ex2. Compute the value of the expression x = (a – b * c) / d where all numbers are unsigned numbers and a, b, c, d are all bytes.

```
bits 32
; declare the EntryPoint (a label defining the very first instruction of the program)
global start
; declare external functions needed by our program
extern exit                ; tell nasm that exit exists even if we won't be defining it
import exit msvcrt.dll     ; exit is a function that ends the calling process. It is defined in msvcrt.dll

; our data is declared here (the variables needed by our program)
segment data use32 class=data
        a db 30
        b db 4
        c db 2
        d db 3
        x db 0

; our code starts here
segment code use32 class=code
start:
```

```
mov al, [b]                    ; AL:=b = 4
mul byte [c]                   ; AX:=AL*c = 4*2 = 8
mov bl, [a]                    ; BL:=a=30
; Now we need to substract AX from BL, but we can not do that directly due to the rule that both
; operands of sub must have the same type/size. In such situations we always convert the smallest
; type to the larger one (i.e. we convert BL from byte to word). BL:=30=0001 1110b. The number 30
; represented unsigned on 16 bits looks like this: 0000 0000 0001 1110b. So we see that the only
; difference in the unsigned representation between the representation of 30 on 8 bits and the
; representation of 30 on 16 bits, is the fact that 30 on 16 bits has an additional 8 zero bits in the
; front. This is why the unsigned conversion of a byte/word/dword is realized by adding non
; significant zeroes in front of the number.
mov bh, 0                      ; BX:=0000 0000 0001 1110b
                               ; we converted unsigned BL to BX
sub bx, ax                     ; BX:= BX-AX = 30 – 8 = 22
mov ax, bx                     ; AX:=BX=22
div byte [d]                   ; AL:=AX / d =22 / 3 = 7 (quotient)    AH:=AX % d =22 % 3 = 1 (remainder)
mov [x], AL                    ; x:=AL=7

; exit(0)
push    dword 0                ; push the parameter for exit onto the stack
call    [exit]                 ; call exit to terminate the program
```

In the above example we have seen that in order to convert unsigned BL to BX, we just added 8 non significant zeros to BH (by moving zero to BH). For the signed representation however, there is a different story. If the 8-bit number is positive, signed converting this number to 16 bits is the same as for unsigned representations: just put 8 non significant zeros in front of it (in the high part). Example:

```
mov  al, 12
mov ah, 0
```

Now AX=12= 0000 0000 0000 1100b (AH=0000 0000b and AL=0000 1100b).

But if the number is negative, we have to put 8 digits of 1 in front of it. See below:

```
mov al, -12                    ; AL:=1111 0100b
```

; -12 represented on 16 bits is: 1111 1111 **1111 0100**b

Because for the signed representation, we have two cases (when the number is positive and when the number is negative), we have specialized instructions that perform the signed conversion (i.e. these instructions either add 8 (or 16 or 32) zero bits in front of the number if the number is positive or they add 8 (or 16 or 32) one bit in front of the number if the number is negative); in other words, these instructions add the sign bit of the number 8 (or 16 or 32) times in front of the number. These instructions are presented below.

CBW – (signed) convert byte to word

Syntax: cbw

Effect: converts signed AL to AX

CWD – (signed) convert word to dword (doubleword)

Syntax: cwd

Effect: converts signed AX to DX:AX

CWDE – (signed) convert word to dword extended

Syntax: cwde

Effect: converts signed AX to EAX

CDQ – (signed) convert doubleword to quadword

Syntax: cdq

Effect: converts signed EAX to EDX:EAX

To summarize the unsigned conversions (we do not have specialized instructions for this):

- unsigned convert AL to AX: mov ah, 0
- unsigned convert AX to DX:AX : mov dx, 0
- unsigned convert EAX to EDX:EAX: mov edx, 0

Ex. 3 Compute the value of the expression (a * b) / d – c where all numbers are signed and *a, c, d* are bytes and *b* is a word.

```
bits 32
global start
extern exit
import exit msvcrt.dll

; our data is declared here (the variables needed by our program)
segment data use32 class=data
        a db -3
        b dw 4
        c db 2
        d db 3
        x dw 0

; our code starts here
segment code use32 class=code
start:
        mov al, [a]     ; AL:=a = -3
        cbw             ; AX:=-3    (convert AL to AX signed)
        imul word [b]   ; DX:AX:= AX * B = -3 * 4 = -12
        ; we need to convert “d” from byte to word
        mov bx, ax      ; DX:BX:=-12
        mov al, [d]     ; AL:=d = 3
        cbw             ; AX:=AL=3
        mov cx, ax      ; CX:=3
        mov ax, bx      ; move BX back to AX so that DX:AX=-12
        idiv cx         ; AX:=DX:AX / CX = -12 / 3 = -4       DX:=DX:AX % CX = -12 % 4 = 0
        ; we must convert “c” from byte to word; first clear the AX register
        mov bx, ax      ; BX:=AX=-4
        mov al, [c]     ; AL:=c=2
```

```
 cbw                ; AX:=AL=2
 sub BX, AX         ;BX:=BX-AX = -4 -2 = -6
 mov [x], BX        ; x:=-6


; exit(0)
push    dword 0             ; push the parameter for exit onto the stack
call    [exit]                ; call exit to terminate the program
```

Seminar II. Signed and unsigned instructions. Arithmetic instructions (multiplications and divisions). Signed and unsigned conversions.

II.1. Signed and unsigned instructions

On the IA-32 architecture, related to the unsigned and signed representation of numbers, there are 3 classes of instructions:

- instructions which do not care about signed or unsigned representation of numbers: **mov, add, sub**
- instructions which interpret the operands as unsigned numbers: **div, mul**
- instructions which interpret the operands as signed numbers: **idiv, imul, cbw, cwd, cwde**

It is important to be consistent when developing a IA-32 assembly program: either consider all numerical values in a program to be unsigned (in which case you should use only instructions from class 1 and 2) or consider all numerical values in a program to be signed (in which case you should use only instructions from class 1 and 3).

Important rules that must be obeyed by arithmetic instructions with 2 operands:

- all operands must have the same size/type (i.e. you can add byte to byte, but not byte to a word)
- at least one of the operands must be a general register or a constant and if it is a constant, this constant can not appear as a destination operand

Related to the above two rules, let's assume we have the following code:

```
a db 10
b db 11
……..
add ax, [a]
add [a], [b]
```

The instruction *add [a], [b]* would fail, meaning that there will be a compile error (i.e. assembly error) and the executable file can not be built. This is because that instruction does not obey the second rule from above ([a] and [b] are both memory references / variables). On the other hand, although the instruction *add ax, [a]* breaks rule number one from above (since AX is a word and [a] was declared as byte), the compiler will not complain and will build the executable file! But when this instruction gets executed, it will not do what you meant: add the byte "a" to the register "AX"! Instead it will add a *word from the memory that starts where variable "a" starts* (this word is composed from the bytes 10 and 11) to the register AX. This is because although variable "a" was declared as a byte using the Define Byte (DB) directive, the NASM assembler does not link the type (data size) to a memory location in instructions. The declaration "a db 10" just declares/reserves 1 byte at the current memory address. You can then write in your code *mov ax, [a]* and the assembler will tell the CPU to move a word starting in the memory at the address of "a" into AX. Or you can write the code *mov eax, [a]* and the assembler will tell the CPU to move a doubleword starting in the memory at the address of "a" into EAX. *[a]* means just the

starting point in the memory of a data – it does not say anything about the type (size) of "a". The type (size) information is inferred from the other operand of the instruction together with the first rule described above: all operands must have the same size/type.

Obs. "[ ]" is the addressing operator and its effect can be explained in the following two examples:

mov EAX, [a] => moves in EAX a doubleword starting at the address of variable "a" (i.e. it moves in EAX the value of variable "a", assuming "a" was declared as a doubleword)

mov EAX, a => moves in EAX the starting address (i.e. the offset – you will see later what this means) of variable "a" (NOT the value of variable "a" !)

II.2. (Signed and unsigned) multiplication and divisions instructions

MUL – unsigned multiplication instruction

Syntax: mul source

(where source is either register or variable of type byte, word or dword)

Effect: - if source is a byte => AX:=AL * source

- if source is a word => DX:AX:= AX * source
- if source is a dword =>EDX:EAX:= EAX * source

Example: The instruction *mul BX* stores in two 16-bit registers the result of the multiplication which is a 32-bit numbers. More specifically, the effect of this instruction is: DX:AX:= AX*BX. The result of the multiplication (a 32-bit number) is stored in the registers DX and AX instead of a proper 32-bit register like EAX for compatibility reasons with the previous Intel 8086 computing architecture. Let's assume that the result of the above multiplication would be the number 12345678h (in the hexadecimal base). The least significant (low) 16 bits of this number would be stored in AX and the most significant (high) 16 bits of this number would be stored in DX. Knowing that a hexadecimal digit is represented on 4 bits, we conclude that AX would store 5678h and DX would store 1234h (the hexadecimal number 1234h occupies 16 bits).

DIV – unsigned division instruction

Syntax: div source

(where source is either register or variable of type byte, word or dword)

Effect: - if source is a byte => AL:=AX / source (quotient/catul) and AH:=AX % source (remainder/restul)

- if source is a word => AX:=DX:AX / source (quotient) and DX:=DX:AX % source (remainder)
- if source is a dword =>EAX:= EDX:EAX / source (quotient) and EDX:=EDX:EAX % source (remainder)

IMUL and **IDIV** – does the same thing as MUL and DIV but considers the operands as signed numbers.

Examples

Ex1. Compute the value of the expression x:=((a+b)*c) / d where all numbers are unsigned numbers and a, b, c, d are all bytes.

```
;
; BEGIN 32 bits PROGRAM
;
bits 32
; declare the EntryPoint (a label defining the very first instruction of the program)
global start

; declare external functions needed by our program
extern exit                 ; tell nasm that exit exists even if we won't be defining it
import exit msvcrt.dll      ; exit is a function that ends the calling process. It is defined in msvcrt.dll


; our data is declared here (the variables needed by our program)
segment data use32 class=data
        a db 3
        b db 4
        c db 2
        d db 3
        x db 0

; our code starts here
segment code use32 class=code
start:
        mov al, [a]                     ; AL:=a = 3
        add al, [b]                     ; AL:=AL+b = 3+4 = 7
        mul byte [c]                    ; AX:=AL*c = 7*2 = 14
                                        ; for this mul instruction we had to specify the type of operand [c]
                                        ; (i.e. byte), so that mul knows what to do. Remember, "c" is just
                                        ; a memory reference, it does not have a type associated to it!
        div byte [d]                    ; AL:=AX / d = 14 / 3 = 4      AH:=AX % d = 14 % 3 = 2
                                        ; similar to the above mul, we had to explicitly specify the type of
                                        ; [d] (i.e. byte)
        mov [x], AL                     ; x:=AL = 4

       ; exit(0)
       push    dword 0          ; push the parameter for exit onto the stack
       call    [exit]              ; call exit to terminate the program
```

II.3. Signed and unsigned conversions

Ex2. Compute the value of the expression x:=(a-b*c) / d where all numbers are unsigned numbers and a, b, c, d are all bytes.

```
bits 32
; declare the EntryPoint (a label defining the very first instruction of the program)
```

```
global start
; declare external functions needed by our program
extern exit                ; tell nasm that exit exists even if we won't be defining it
import exit msvcrt.dll     ; exit is a function that ends the calling process. It is defined in msvcrt.dll



; our data is declared here (the variables needed by our program)
segment data use32 class=data
        a db 30
        b db 4
        c db 2
        d db 3
        x db 0

; our code starts here
segment code use32 class=code
start:
        mov al, [b]                     ; AL:=b = 4
        mul byte [c]                    ; AX:=AL*c = 4*2 = 8
        mov bl, [a]                     ; BL:=a=30
        ; Now we need to substract AX from BL, but we can not do that directly due to the rule that both
        ; operands of sub must have the same type/size. In such situations we always convert the smallest
        ; type to the larger one (i.e. we convert BL from byte to word). BL:=30=0001 1110b. The number 30
        ; represented unsigned on 16 bits looks like this: 0000 0000 0001 1110b. So we see that the only
        ; difference in the unsigned representation between the representation of 30 on 8 bits and the
        ; representation of 30 on 16 bits, is the fact that 30 on 16 bits has an additional 8 zero bits in the
        ; front. This is why the unsigned conversion of a byte/word/dword is realized by adding non
        ; significant zeroes in front of the number.
        mov bh, 0               ; BX:=0000 0000 0001 1110b
                                ; we converted unsigned BL to BX
        sub bx, ax              ; BX:= BX-AX = 30 – 8 = 22
        mov ax, bx              ; AX:=BX=22
        div byte [d]            ; AL:=AX / d =22 / 3 = 7 (quotient)    AH:=AX % d =22 % 3 = 1 (remainder)
        mov [x], AL             ; x:=AL=7

       ; exit(0)
       push    dword 0          ; push the parameter for exit onto the stack
       call    [exit]           ; call exit to terminate the program
```

In the above example we have seen that in order to convert unsigned BL to BX, we just added 8 non significant zeros to BH (by moving zero to BH). For the signed representation however, there is a different story. If the 8-bit number is positive, signed converting this number to 16 bits is the same as for unsigned representations: just put 8 non significant zeros in front of it (in the high part). Example:

```
        mov  al, 12
```

mov ah, 0

Now AX=12= 0000 0000 0000 1100b (AH=0000 0000b and AL=0000 1100b).

But if the number is negative, we have to put 8 digits of 1 in front of it. See below:

mov al, -12 ; AL:=**1111 0100**b

; -12 represented on 16 bits is: 1111 1111 **1111 0100**b

Because for the signed representation, we have two cases (when the number is positive and when the number is negative), we have specialized instructions that perform the signed conversion (i.e. these instructions either add 8 (or 16 or 32) zero bits in front of the number if the number is positive or they add 8 (or 16 or 32) one bit in front of the number if the number is negative); in other words, these instructions add the sign bit of the number 8 (or 16 or 32) times in front of the number. These instructions are presented below.

CBW – (signed) convert byte to word
Syntax: cbw
Effect: converts signed AL to AX

CWD – (signed) convert word to dword (doubleword)
Syntax: cwd
Effect: converts signed AX to DX:AX

CWDE – (singned) convert word to dword extended
Syntax: cwde
Effect: converts signed AX to EAX

CDQ – (singned) convert doubleword to quadword
Syntax: cdq
Effect: converts signed EAX to EDX:EAX

To summarize the unsigned conversions (we do not have specialized instructions for this):

- unsigned convert AL to AX: mov ah, 0
- unsigned convert AX to DX:AX : mov dx, 0
- unsigned convert EAX to EDX:EAX: mov edx, 0

Ex. 3 Compute the value of the expression (a*b)/d – c where all numbers are signed and *a, c, d* are bytes and *b* is a word.

```
bits 32
global start
extern exit
import exit msvcrt.dll

; our data is declared here (the variables needed by our program)
segment data use32 class=data
        a db -3
        b dw 4
        c db 2
```

```
        d db 3
        x dw 0

; our code starts here
segment code use32 class=code
start:
        mov al, [a]         ; AL:=a = -3
        cbw                 ; AX:=-3    (convert AL to AX signed)
        imul word [b]       ; DX:AX:= AX * B = -3 * 4 = -12
        ; we need to convert “d” from byte to word
        mov bx, ax          ; DX:BX:=-12
        mov al, d           ; AL:=d = 3
        cbw                 ; AX:=AL=3
        mov cx, ax          ; CX:=3
        mov ax, bx          ; move BX back to AX so that DX:AX=-12
        idiv cx             ; AX:=DX:AX / CX = -12 / 3 = -4       DX:=DX:AX % CX = -12 % 4 = 0
        ; we must convert “c” from byte to word; first clear the AX register
        mov bx, ax          ; BX:=AX=-4
        mov al, c           ; AL:=c=2
        cbw                 ; AX:=AL=2
        sub BX, AX          ;BX:=BX-AX = -4 -2 = -6
        mov [x], BX         ; x:=-6


       ; exit(0)
       push    dword 0          ; push the parameter for exit onto the stack
       call    [exit]             ; call exit to terminate the program
```

Seminar 3

Little-endian representation of numbers in the memory. Conditional and unconditional jumps.

String operations

Contents

Seminar 3 1

1. Little-endian representation in the memory and non-destructive conversions 1

2. Conditional and unconditional jump instructions 4

3. Working with strings of bytes 6

1. Little-endian representation in the memory and non-destructive conversions

In the computer memory (not on the CPU registers!) numbers represented on more than 1 byte are represented in little-endian format (i.e. the least significant byte of the representation is placed at the smallest memory address).

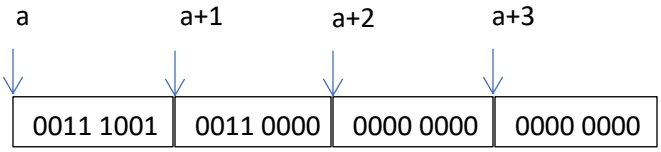
We also said in the previous seminar that numbers are represented in the memory and on the CPU registers in the unsigned representation or the signed representation. And now we are saying numbers are represented in little-endian format in the memory. So which is it?

Well, every number in assembly is represented on 1, 2, 4 or 8 bytes (depending on the context in which it is declared; if we have this declaration: "a dw 2", then 2 will be represented on 2 bytes). It is first represented in binary using signed or unsigned representation. If the number is represented on more than 1 byte and in the memory (not on the CPU registers), the bytes of the signed or unsigned representation are then placed in the memory in little-endian format.

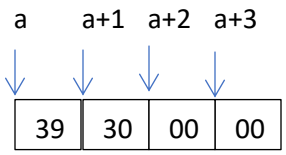
Example:

```
a  DD  0

....

mov dword [a], 12345
```

In the above instruction, the number 12345 is positive so it will be represented in the unsigned representation on 4 bytes as 0000 0000 0000 0000 0011 0000 0011 1001. Let's write this binary number in the hexadecimal base (as we know from seminar 1 that a group of 4 bits is a hexadecimal digit): 00003039h. This double word number will then be represented in memory in little-endian representation on 4 bytes as:



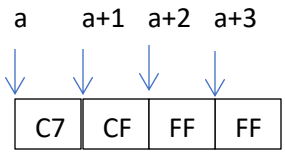
The memory addresses: “a”, “a+1”, “a+2”, “a+3” represent the memory addresses of the 4 bytes of variable “a”; each address is the address of 1 byte. The byte from memory address “a” is the least significant byte of the number, the byte from memory address “a+1” is the next least significant byte of the number and so on. We usually use the hexadecimal base when we represent a number in the memory (because there are less digits) and we also know that 4 bits is a hexadecimal digit. The above little-endian representation is:



If we consider the next example:

```
mov dword [a], -12345
```

we first represent -12345 in the signed representation as 1111 1111 1111 1111 1100 1111 1100 0111 and we consider this sequence on 32 bits in hexadecimal as FFFFCFC7h so this will be represented in the memory in little-endian format as:

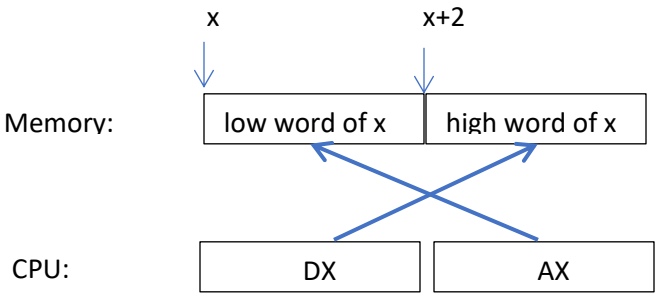


The little-endian representation is important when we want to do “non-destructive conversions” like: refer to the low word of a double word from the memory, refer to the high byte of a word from the memory, etc.

Ex. 1. Write a program that computes the expression: x:=a*b+c*d where all numbers are unsigned and represented on a word.

```
bits 32
global start
extern exit
import exit msvcrt.dll
```

```
segment data use32 class=data
        a dw 2
        b dw 5
        c dw 10
        d dw 40
        x dd 0
segment code use32 class=code
start:
        mov ax, [a]
        mul word [b]            ; DX:AX := AX * b = 2*5 = 10
        ; we save the doubleword DX:AX into x
```




```
        mov word [x], ax        ; save AX into the low word of "x" (i.e. the word from the address "x")
        mov word [x+2], dx      ; save DX into the high word of "x" (i.e. the word from the address "x+2")

        mov ax, [c]
        mul word [d]                    ; DX:AX := AX*d = c*d = 400

        ; add DX:AX to x
        add word [x], ax        ; we first add AX to the low word of "x"
                                ; if there is an overflow, the CF (Carry Flag) will be set to 1
                                ; otherwise CF will be set to zero
        adc word [x+2], dx      ; add DX to the high word of "x", but add also the CF
        ; call exit(0)
        push dword 0
        call [exit]
```

New instructions:

ADC – Add with Carry

adc dest, source => dest:=dest+source+CF

SBB – Substract with borrow

sbb dest, source => dest:=dest+source-CF

2. Conditional and unconditional jump instructions

Conditional jump instructions are just like the "IF" instruction in a high-level programming language. In assembly language, an "IF" is composed of 2 instructions: the compare instruction and the conditional jump instruction.

The compare instruction:

cmp a, b ; performs a non-distructive *sub a, b* (meaning that *a* does not change) and sets the flags accordingly

A conditional jump instruction checks the value of specific flags and depending on the value, performs a "jump in the program" (i.e. moves the execution to a different part of the program – a part of the program is identified by a label). A conditional jump instruction makes sense and should follow a **cmp** instruction.

Jump Instructions are composed from a few key words:

- E = equal;
- N = not;
- J = jump
- A = above; B = below ; for unsigned numbers
- G = greater L = less ; for signed numbers

Conditional jump instructions that consider the numbers unsigned

jb label ; (Jump if below) jumps to label if a < b (CF = 1)

jbe label ; (Jump if below or equal) jumps to label if a ≤ b (CF = 1 or ZF = 1)

jnb label ; (Jump if not below) jumps to label if a ≥ b

jnbe label ; (Jump if not below or equal) jumps to label if a > b

ja label ; (Jump if above) jumps to label if a > b (CF = 0 & ZF = 0)

jae label ; (Jump if above or equal) jumps to label if a ≥ b

jna label ; (Jump if not above) jumps to label if a ≤ b

jnae label ; (Jump if not above or equal) jumps to label if a < b

The *a* and *b* values from above are the *a* and *b* operands of the **cmp** instruction that was issued before the conditional jump instruction.

Conditional jump instructions that consider the numbers signed

jl label ; (Jump if less) jumps to label if a < b (SF ≠ OF)

jle label ; (Jump if less or equal) jumps to label if a a ≤ b

jnl label ; (Jump if not less) jumps to label if a a ≥ b

jnle label ; (Jump if not less or equal) jumps to label if a > b

jg label ; (Jump if greater) jumps to label if a > b (SF = 0 & ZF = 0)

jge label ; (Jump if greater or equal) jumps to label if a a ≥ b (SF = OF)

jng label ; (Jump if not greater) jumps to label if a a ≤ b

jnge label ; (Jump if not greater or equal) jumps to label if a < b

The *a* and *b* values from above are the *a* and *b* operands of the **cmp** instruction that was issued before the conditional jump instruction.

Conditional jump instructions that consider one flag

jc label ;Jump if CF=1 (carry flag)

jnc label

je label ;Jump if ZF=1 (zero flag)

jz label ;Jump if ZF=1

jne label

jnz label

js label ;Jump if SF = 1 (sign flag)

jns label

jp label ;Jump if PF = 1 (parity flag)

jnp label

jo label ;Jump if OF =1 (overflow flag)

jno label

Conditional jump instructions that check the value of CX or ECX

Jcxz label ;Jump if CX = 0

Jecxz label ;Jump if ECX = 0

Unconditional jump instruction

jmp label ; always jumps to the specified label

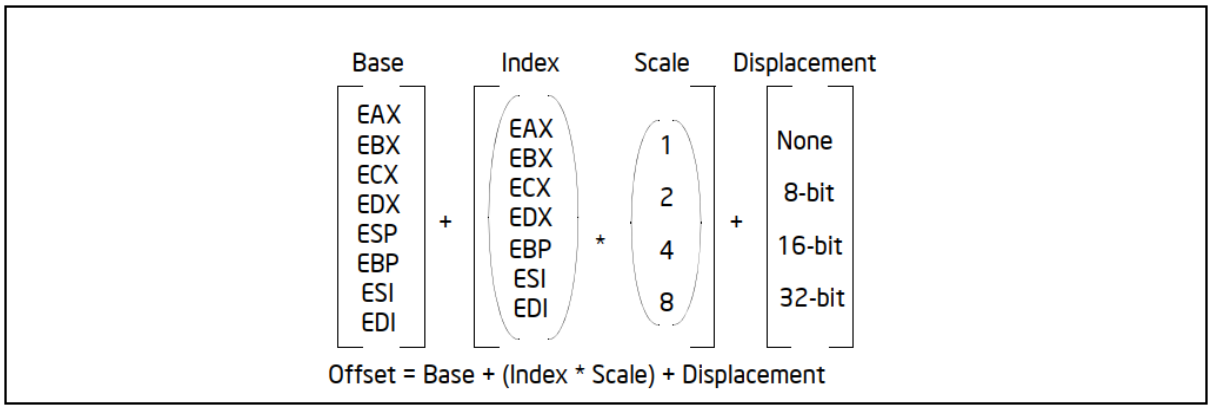
3. Working with strings of bytes

Memory addresses and offsets

In order to work with strings of bytes/words/dwords we need to know more about memory addresses. So far, we have used expressions like:

mov ax, [a]

where *a* is a variable and the instruction above moves a word value from the memory starting at the memory address "*a*". We have seen so far that a variable name is just a constant address – the address of that variable in the memory. To be more precise, a variable name is a constant offset. A full address specification comes in the form of two numbers: **segment_selector : offset**. A **segment_selector** is a 16-bit number and specifies a pointer in a segment descriptor table (GDT – global descriptor table) which defines a memory zone. For today's seminar you do not really need to understand what a memory segment is (you will find more details about memory segments and memory management at the course), just know that a memory segment is a continuous memory zone and its address is already stored in the appropriate segment register (CS, DS, ES or SS) by the operating system before your program starts and you are not allowed to change it. The **offset** is a 32-bit number which specifies a pointer inside the segment specified by the **segment_selector**. The name of a variable, like I have said before, represents just the offset of that variable (the segment to which this offset refers to is the data segment whose starting address is already placed in the DS register by the operating system). The full format of an offset includes 4 quantities : a base, an index, a scale and a displacement and is presented below in the following figure:



In an offset specification any combination of the above 4 quantities can appear (inside "[ ]"), including each component individually. The displacement is just a constant number. Below you can find some examples of offset specifications (as the second operand of the instruction):

mov ax, [a] *; only displacement*

mov ax, [eax] *; only base or index*

mov ax, [a+eax+ebx] *; base, index and displacement*

mov ax, [eax+eax+a+2] *; base, index and displacement*

*mov ax, [a+4+ebx*2]* *; index, scale (i.e. 2) and displacement*

*mov ax, [eax + ebx*4 + 20]; base, index, scale (i.e. 4) and displacement*

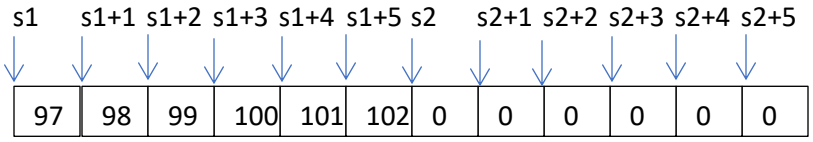
Ex.2. Being given a string of bytes containing lowercase letters, build a new string of bytes containing the corresponding uppercase letters.

Method 1:

```
bits 32
global start
extern exit
import exit msvcrt.dll
```

```
segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0
```

; the data segment looks in the memory like this:



; the top row in the above figures represents offsets. For example, in OllyDebugger, the first byte ; from the data segment and thus the offset of *s1* is always 0x00401000. The offset of variable *s2* ; is equal to the offset of *s1* plus 6 (i.e. 0x00401006). The bottom row represents the actual

; values from the memory in base 10 (97 is the ASCII code of 'a', 98 is the ASCII code of 'b', 99 ; is the ASCII code of 'c' …). The bytes from string s2 are initialized with zero.

; *lenS1 equ $-s1* defines a constant (equ = constant, no memory space is reserved)

; $ is the location counter, the current offset in the data segment up to the code line in which it

; appears. So, before the code line *lenS1 equ $-s1* , there were 6 bytes generated in the string *s1*

; so the current offset is $=s1+6. *s1* in the above instruction is just the offset of variable *s1*, so

; len1 = $ - s1 = s1+6 – s1 = 6 bytes (the length in bytes of string *s1*).

; *s2 times lenS1 db 0* defines a variable s2 which contains lenS1=6 bytes, all initialized with

; the value zero.

```
segment code use32 class=code
start:
    ; we solve this problem by considerring in ESI the current index in string s1 and string s2
    ; and we do a loop with lenS1=6 iterations and at each iteration we move the byte s1[ESI]
    ; into s2[ESI], after we change it to an uppercase letter. So, in this loop, ESI will have the
   ; values: 0, 1, 2, 3, 4, 5.
```

```
    mov esi, 0

    repeat:
        mov al, [s1+esi]    ; AL <- the byte from the offset s1+esi
        sub al, 'a' -'A'    ; obtain the corresponding uppercase letter in AL
        mov [s2+esi], al   ; AL -> the byte from the offset s2+esi

        inc esi             ; esi:=esi+1; move to the next index in strings s1 and s2
        cmp esi, lenS1
        jb repeat           ; IF (esi < lenS1) jump to repeat,
                            ; otherwise continue below


    push dword 0
    call [exit]
```

Method 2:

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0

segment code use32 class=code
start:
    ; this time we solve this problem by considerring in ESI the offset of the current byte from
```

; string s1 and in EDI the offset of the current byte from string s2. We do a loop with
; lenS1=6 iterations and at each iteration we move the byte from offset ESI into the byte
; from offset EDI, after we change it to an uppercase letter. So, in this loop, ESI will have
; the values: s1+0, s1+1, s1+2, s1+3, s1+4, s1+5 and EDI will have the values: s1+6, s1+7,
; s1+8, s1+9, s1+10, s1+11.

```
mov esi, s1      ; initialize esi
mov edi, s2      ; initialize edi
mov ecx, lenS1   ; ecx will store the number of iterations in the loop

repeat:
    mov al, [esi]          ; AL <- the byte from the offset s1+esi
    sub al, 'a' -'A'       ; obtain the corresponding uppercase letter in AL
    mov [edi], al          ; AL -> the byte from the offset s2+edi

    inc esi      ; esi:=esi+1; move to the next byte in strings s1
    inc edi      ; edi:=edi+1; move to the next byte in strings s2
    dec ecx      ; exc:=ecx-1
    cmp ecx, 0   ; IF (ecx > 0) jump to repeat
    jb repeat    ; otherwise exit the loop

push dword 0
call [exit]
```

Instructions for loops

LOOP label

- Continues the loop as long as the value of ECX ≠ 0
- It is equivalent to these 3 instructions:

```
;   dec ecx
;   cmp ecx, 0
;   ja label
```

LOOPE / LOOPZ

- Continues the loop as long as the value of ECX ≠ 0 and ZF = 1
- So the cycle terminating either if ECX = 0 or if ZF = 0.

LOOPNE / LOOPNZ

- Continues the loop as long as the value of ECX ≠ 0 and ZF = 0
- So the cycle terminating either if ECX = 0 or if ZF = 1.

Method 3:

```
bits 32
global start
extern exit
import exit msvcrt.dll
segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0
segment code use32 class=code
start:
    mov esi, s1      ; initialize esi
    mov edi, s2      ; initialize edi
    mov ecx, lenS1   ; ecx will store the number of iterations in the loop
    jecxz finished
    repeat:
        mov al, [esi]          ; AL <- the byte from the offset s1+esi
        sub al, 'a' -'A'       ; obtain the corresponding uppercase letter in AL
        mov [edi], al          ; AL -> the byte from the offset s2+edi

        inc esi      ; esi:=esi+1; move to the next byte in strings s1
        inc edi      ; edi:=edi+1; move to the next byte in strings s2
        loop repeat
    finished:
    push dword 0
    call [exit]
```

Ex.3. How many times is the loop executed?

```
MOV ECX,5
repeat:
DEC ECX
LOOP repeat
```

Ex.4. How many times is the loop executed?

```
MOV EBX, 0
MOV ECX,0
repeat:
INC EBX
LOOP repeat
```

Seminar III. Little-endian representation of numbers in the memory. Conditional and unconditional jumps. String operations

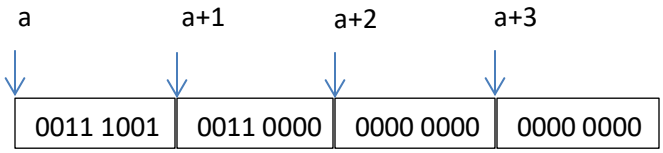
III. 1 Little-endian representation in the memory and non-destructive conversions

In the computer memory (not on the CPU registers!) numbers represented on more than 1 byte are represented in little-endian format (i.e. the least significant byte of the representation is placed at the smallest memory address).
We also said in the previous seminar that numbers are represented in the memory and on the CPU registers in the unsigned representation or the signed representation. And now we are saying numbers are represented in little-endian format in the memory. So which is it?
Well, every number in assembly is represented on 1, 2, 4 or 8 bytes (depending on the context in which it is declared; if we have this declaration: "a dw 2", then 2 will be represented on 2 bytes). It is first represented in binary using signed or unsigned representation. If the number is represented on more than 1 byte and in the memory (not on the CPU registers), the bytes of the signed or unsigned representation are then placed in the memory in little-endian format.

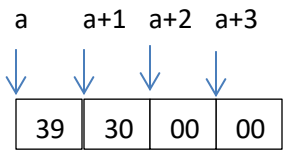
Example:

```
a DD 0
....
mov dword [a], 12345
```

In the above instruction, the number 12345 is positive so it will be represented in the unsigned representation on 4 bytes as 0000 0000 0000 0000 0011 0000 0011 1001. Let's write this binary number in the hexadecimal base (as we know from seminar 1 that a group of 4 bits is a hexadecimal digit): 00003039h. This double word number will then be represented in memory in little-endian representation on 4 bytes as:

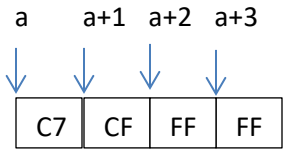


The memory addresses: "a", "a+1", "a+2", "a+3" represent the memory addresses of the 4 bytes of variable "a"; each address is the address of 1 byte. The byte from memory address "a" is the least significant byte of the number, the byte from memory address "a+1" is the next least significant byte of the number and so on. We usually use the hexadecimal base when we represent a number in the memory (because there are less digits) and we also know that 4 bits is a hexadecimal digit. The above little-endian representation is:



If we consider the next example:
mov dword [a], -12345

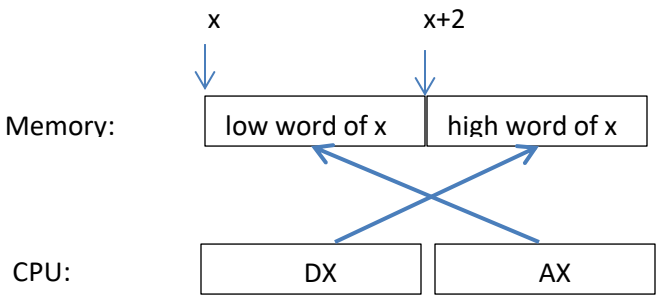
we first represent -12345 in the signed representation as 1111 1111 1111 1111 1100 1111 1100 0111 and we consider this sequence on 32 bits in hexadecimal as FFFFCFC7h so this will be represented in the memory in little-endian format as:



The little-endian representation is important when we want to do "non-destructive conversions" like: refer to the low word of a double word from the memory, refer to the high byte of a word from the memory, etc.

Ex. 1. Write a program that computes the expression: x:=a*b+c*d where all numbers are unsigned and represented on a word.

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
        a dw 2
        b dw 5
        c dw 10
        d dw 40
        x dd 0


segment code use32 class=code
start:
        mov ax, [a]
        mul word [b]            ; DX:AX := AX * b = 2*5 = 10
        ; we save the doubleword DX:AX into x
```



```
mov word [x], ax        ; save AX into the low word of "x" (i.e. the word from the address "a")
mov word [a+2], dx      ; save DX into the high word of "x" (i.e. the word from the address "a+2")

mov ax, [c]
mul word [d]                    ; DX:AX := AX*d = c*d = 400

; add DX:AX to x
add word [x], ax        ; we first add AX to the low word of "x"
                        ; if there is an overflow, the CF (Carry Flag) will be set to 1
                        ; otherwise CF will be set to zero
adc word [x+2], dx      ; add DX to the high word of "x", but add also the CF

; call exit(0)
push dword 0
call [exit]
```

New instructions:
ADC – Add with Carry
adc dest, source => dest:=dest+source+CF

SBB – Substract with borrow
sbb dest, source => dest:=dest+source+CF

III.2 Conditional and unconditional jump instructions

Conditional jump instructions are just like the "IF" instruction in a high-level programming language. In assembly language, an "IF" is composed of 2 instructions: the compare instruction and the conditional jump instruction.

The compare instruction:
cmp a, b : performs a non-distructive *sub a, b* (meaning that *a* does not change) and sets the flags accordingly

The conditional jump instructions check the value of specific flags and depending on this value, performs a "jump in the program" (i.e. moves the execution to a different part of the program – a part of the program is identified by a label). A conditional jump instruction makes sense and should follow a **cmp** instruction.

Conditional jump instructions that consider the numbers unsigned

jb label : (Jump if below) jumps to label if a<b
jbe label : (Jump if below or equal) jumps to label if a<=b
jnb label : (Jump if not below) jumps to label if a>=b
jnbe label : (Jump if not below or equal) jumps to label if a>b

ja label : (Jump if above) jumps to label if a>b
jae label : (Jump if above or equal) jumps to label if a>=b
jna label : (Jump if not above) jumps to label if a<=b
jnae label : (Jump if not above or equal) jumps to label if a<b

The *a* and *b* values from above are the *a* and *b* operands of the **cmp** instruction that was issued before the conditional jump instruction.

Conditional jump instructions that consider the numbers signed

jl label : (Jump if less) jumps to label if a<b
jle label : (Jump if less or equal) jumps to label if a<=b
jnl label : (Jump if not less) jumps to label if a>=b
jnle label : (Jump if not less or equal) jumps to label if a>b
jg label : (Jump if greater) jumps to label if a>b
jge label : (Jump if greater or equal) jumps to label if a>=b
jng label : (Jump if not greater) jumps to label if a<=b
jnge label : (Jump if not greater or equal) jumps to label if a<b

The *a* and *b* values from above are the *a* and *b* operands of the **cmp** instruction that was issued before the conditional jump instruction.

Unconditional jump instruction

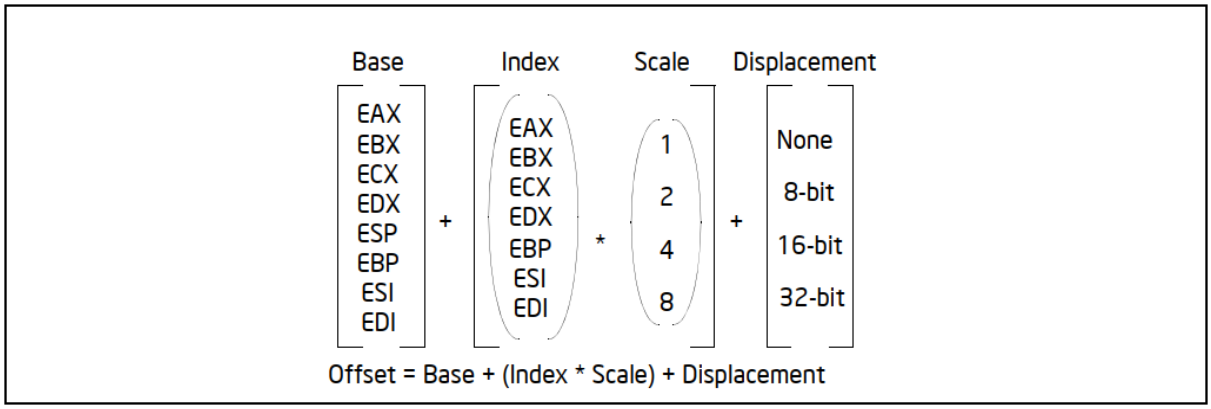
jmp label : always jumps to the specified label

III.3 Working with strings of bytes

Memory addresses and offsets

In order to work with strings of bytes/words/dwords we need to know more about memory addresses. So far, we have used expressions like:
mov ax, [a]
where *a* is a variable and the instruction above moves a word value from the memory starting at the memory address "*a*". We have seen so far that a variable name is just a constant address – the address of that variable in the memory. To be more precise, a variable name is a constant offset. A full address specification comes in the form of two numbers: **segment_selector: offset**. A **segment_selector** is a 16-bit number and specifies a pointer in a segment descriptor table (GDT – global descriptor table) which defines a memory zone. For today's seminar you don not really need to understand what a memory segment is (you will find more details about memory segments and memory management at the course), just know that a memory segment is a continuous memory zone and its address is already stored in the appropriate segment register (CS, DS, ES or SS) by the operating system before your program starts and you are not allowed to change it. The **offset** is a 32-bit number which specifies a pointer inside the segment specified by the **segment_selector**. The name of a variable, like I have said before, represents just the

offset of that variable (the segment to which this offset refers to is the data segment whose starting address is already placed in the DS register by the operating system). The full format of an offset includes 4 quantities : a base, an index, a scale and a displacement and is presented below in the following figure:



In an offset specification any combination of the above 4 quantities can appear (inside "[ ]"), including each component individually. The displacement is just a constant number. Below you can find some examples of offset specifications (as the second operand of the instruction):

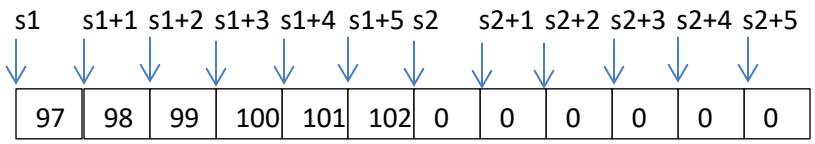
mov ax, [a] ; *only displacement*
mov ax, [eax] ; *only base or index*
mov ax, [a+eax+ebx] ; *base, index and displacement*
mov ax, [eax+eax+a+2] ; *base, index and displacement*
*mov ax, [a+4+ebx*2]* ; *index, scale (i.e. 2) and displacement*
*mov ax, [eax + ebx*4 + 20]; base, index, scale (i.e. 4) and displacement*

Ex.2. Being given a string of bytes containing lowercase letters, build a new string of bytes containing the corresponding uppercase letters.

Variant 1:

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0
```

; the data segment looks in the memory like this:



; the top row in the above figures represents offsets. For example, in OllyDebugger, the first byte
; from the data segment and thus the offset of *s1* is always 0x00401000. The offset of variable *s2*
; is equal to the offset of *s1* plus 6 (i.e. 0x00401006). The bottom row represents the actual
; values from the memory in base 10 (97 is the ASCII code of 'a', 98 is the ASCII code of 'b', 99
; is the ASCII code of 'c' …). The bytes from string s2 are initialized with zero.

; *lenS1 equ $-s1* defines a constant (equ = constant, no memory space is reserved)
; $ is the location counter, the current offset in the data segment up to the code line in which it
; appears. So, before the code line *lenS1 equ $-s1* , there were 6 bytes generated in the string *s1*
; so the current offset is $=s1+6. *s1* in the above instruction is just the offset of variable *s1*, so
; len1 = $ - s1 = s1+6 – s1 = 6 bytes (the length in bytes of string *s1*).

; *s2 times lenS1 db 0* defines a variable s2 which contains lenS1=6 bytes, all initialized with
; the value zero.

```
segment code use32 class=code
start:
     ; we solve this problem by considerring in ESI the current index in string s1 and string s2
     ; and we do a loop with lenS1=6 iterations and at each iteration we move the byte s1[ESI]
     ; into s2[ESI], after we change it to an uppercase letter. So, in this loop, ESI will have the
    ; values: 0, 1, 2, 3, 4, 5.

    mov esi, 0

    repeat:
        mov al, [s1+esi]    ; AL <- the byte from the offset s1+esi
        sub al, 'a' -'A'    ; obtain the corresponding uppercase letter in AL
        mov [s2+esi], al   ; AL -> the byte from the offset s2+esi

        inc esi             ; esi:=esi+1; move to the next index in strings s1 and s2
        cmp esi, lenS1
        jb repeat           ; IF (esi < lenS1) jump to repeat,
                           ; otherwise continue below


    push dword 0
    call [exit]
```

Variant 2:

```
bits 32
global start
extern exit
```

```
import exit msvcrt.dll

segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0

segment code use32 class=code
start:
     ; this time we solve this problem by considerring in ESI the offset of the current byte from
    ; string s1 and in EDI the offset of the current byte from string s2. We do a loop with
    ; lenS1=6 iterations and at each iteration we move the byte from offset ESI into the byte
     ; from offset EDI, after we change it to an uppercase letter. So, in this loop, ESI will have
     ; the values: s1+0, s1+1, s1+2, s1+3, s1+4, s1+5 and EDI will have the values: s1+6, s1+7,
    ; s1+8, s1+9, s1+10, s1+11.

    mov esi, s1      ; initialize esi
    mov edi, s2      ; initialize edi
    mov ecx, lenS1   ; ecx will store the number of iterations in the loop

    repeat:
        mov al, [esi]          ; AL <- the byte from the offset s1+esi
        sub al, 'a' -'A'       ; obtain the corresponding uppercase letter in AL
        mov [edi], al          ; AL -> the byte from the offset s2+edi

        inc esi      ; esi:=esi+1; move to the next byte in strings s1
        inc edi      ; edi:=edi+1; move to the next byte in strings s2
        dec ecx      ; exc:=ecx-1
        cmp ecx, 0   ; IF (ecx > 0) jump to repeat
        jb repeat    ; otherwise exit the loop


    push dword 0
    call [exit]
```

Seminar 4

String instructions. String problems

Contents

Seminar 4 1
1. String instructions for data transfer 1
1.1 String instructions for data transfer 1
1.2. String instructions for data comparison 2
2. String problems 3

1. String instructions for data transfer

The string instructions have only default operands and they work in the following pattern: they process the current element of the string(s) and they move to the next element in the string(s). In order to work with string instructions, we must initially:

- set the offset of the source string in ESI (the source string is the one we do not modify)
- set the offset of the destination string in EDI (the destination string is the one we modify)
- set the parsing direction (rom. directia de parcurgere) of strings; if the Direction Flag DF=0 strings are parsed from left to right and if DF=1 strings are parsed from right to left

Some string instructions work only with the source string, some others work only with the destination string and some others work with both.

1.1 String instructions for data transfer

(Load String of Bytes)
1. LODSB AL← <DS:ESI>
if DF=0 inc(ESI) else dec(ESI)

(Load String of Words)
2. LODSW AX← <DS:ESI>
if DF=0 ESI←ESI+2 else ESI←ESI-2

(Load String of Doublewords)
3. LODSD EAX← <DS:ESI>
if DF=0 ESI←ESI+4 else ESI←ESI-4

(Store String of Bytes)
4. STOSB <ES:EDI>← AL
if DF=0 inc(EDI) else dec(EDI)

(Store String of Words)
5. STOSW <ES:EDI>← AX

if DF=0 EDI←EDI+2 else EDI←EDI-2

(Store String of Doublewords)

6. STOSD <ES:EDI>← EAX
if DF=0 EDI←EDI+4 else EDI←EDI-4

(Move String of Bytes)

7. MOVSB <ES:EDI>← <DS:ESI> (a byte)
if DF=0 {inc(ESI); inc(EDI)} else {dec(ESI); dec(EDI)}

(Move String of Words)

8. MOVSW <ES:EDI>← <DS:ESI> (a word)
if DF=0 {ESI←ESI+2; EDI←EDI+2}
else {ESI←ESI-2; EDI←EDI-2}

(Move String of Doublewords)

9. MOVSD <ES:EDI>← <DS:ESI> (a doubleword)
if DF=0 {ESI←ESI+4; EDI←EDI+4}
else {ESI←ESI-4; EDI←EDI-4}

1.2. String instructions for data comparison

(Scan String of Bytes)

10. SCASB CMP AL, <ES:EDI>
if DF=0 inc(EDI) else dec(EDI)

(Scan String of Words)

11. SCASW CMP AX, <ES:EDI>
if DF=0 EDI←EDI+2 else EDI←EDI-2

(Scan String of Doublewords)

12. SCASD CMP EAX, <ES:EDI>
if DF=0 EDI←EDI+4 else EDI←EDI-4

(Compare String of Bytes)

13. CMPSB CMP <DS:ESI>, <ES:EDI> (a byte)
if DF=0 {inc(ESI); inc(EDI)} else {dec(ESI); dec(EDI)}

(Compare String of Words)

14. CMPSW CMP <DS:ESI>, <ES:EDI> (a word)
if DF=0 {ESI←ESI+2; EDI←EDI+2}
else {ESI←ESI-2; EDI←EDI-2}

(Compare String of Doublewords)

15. CMPSD CMP <DS:ESI>, <ES:EDI> (a doubleword)
if DF=0 {ESI←ESI+4; EDI←EDI+4}
else {ESI←ESI-4; EDI←EDI-4}

2. String problems

Ex.1. Being given a string of bytes containing lowercase letters, build a new string of bytes containing the corresponding uppercase letters.

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
   s1 db 'abcdef'
   lenS1 equ $-s1                ; defines the length in bytes of string "s1", i.e. 6
   s2 times lenS1 db 0           ; reserve lenS1 bytes for string "s2"

segment code use32 class=code
start:
   mov esi, s1             ; set the offset of the source string s1 in ESI
   mov edi, s2             ; set the offset of the destination string s2 in EDI
   mov ecx, lenS1          ; we will use a loop/cycle with lenS1 iterations
   cld

   repeat:
      lodsb            ; mov al, [esi]    +    inc esi
      sub al, 'a' -'A'
      stosb            ; mov [edi], al    +    inc edi

      loop repeat     ; is equivalent to these 3 instructions:
                              ;       dec ecx
                             ;       cmp ecx, 0
                              ;       ja repeat

   push dword 0
   call [exit]
```

Ex.2. Being given a string of bytes, write a program that obtains the mirrored string of bytes.

Example: Being given the string of bytes:
s db 17, 20, 42h, 1, 10, 2ah
the corresponding mirrored string of bytes will be
t db 2ah, 10, 1, 42h, 20, 17.

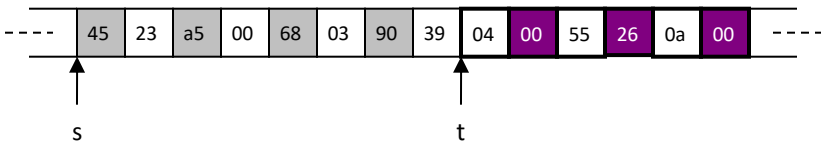
In order to solve the problem, we will parse the initial string "s" in a loop and copy each byte in string "t". While string "s' will be parsed from left to right (i.e. DF=0), string "t" will be parsed from right to left (i.e. DF=1). Thus, the first byte of string "s" will be copied in the last byte of string "t", the second byte of string "s" will be copied in the last but one byte of string "t" and so on..

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
        s   db   17, 20, 42h, 1, 10, 2ah
        len_s   equ   $-s
        t   times  len_s  db 0

segment code use32 class=code
start:
        mov esi, s          ; set the starting offset of the source string “s” in ESI
                            ; ESI now contains the offset of the first byte in string “s”
        mov edi, t          ; set the starting offset of the destination string “t” in EDI
                            ; but because string “t” needs to be parsed from right to left, the starting offset
                            ; of string “t” should be the offset of the last byte in string “t” (i.e. EDI=t + len_s - 1)
        add edi, len_s-1
        mov ecx, len_s
        jecxz theend        ; if ECX==0 jump to “theend”
repeat:
        cld                 ; DF=0 (parse strings from left to right)
        lodsb               ; mov al, [esi]    +    inc esi

        std                 ; DF=0 (parse strings from right to left)
        stosb               ; mov [edi], al    +    dec edi

        loop repeat         ;

theend:
        push dword 0
        call [exit]
```

Ex.3. Two strings of words are given. Concatenate the string of low bytes of the words from the first string to the string of high bytes of the words from the second string. The resulted string of bytes should be sorted in ascending order in the signed interpretation.

Example: Having the strings of words:
s dw 2345h, 0a5h, 368h, 3990h
t dw 4h, 2655h, 10

these strings will be represented in the memory in little-endian format as (the colored bytes are the ones required by the text of the problem):



The result string should be:
u: 90h, a5h, 0h, 0h, 26h, 45h, 68h

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
        s   dw   2345h, 0a5h, 368h, 3990h
        len_s   equ   ($-s)/2                ; the length (in words) of string “s”
        t   dw   4h, 2655h, 10
        len_t   equ   ($-t)/2                ; the length (in words) of string “t”
        len   equ   len_s+len_t              ; the length of the result string

        u   times len  db  0                 ; the result string

segment code use32 class=code
start:
        ; first we copy the low bytes of the words from string “s” into the resulted string “u”

        mov esi, s      ; set the offset of the source string (i.e. the offset of the 1st byte from string “s”)
        mov edi, u      ; set the offset of the dest string (i.e. the offset of the 1st byte from string “u”)

        cld             ; DF=0

        mov ecx, len_s          ; use a loop with len_s iterations
        jecxz theend
        repeat:
                lodsw           ; mov ax, [esi]    +    esi:=esi+2
                                ; AL will store the low byte of the current word from string “s”
                                ; AH will store the high byte of the current word from string “s”

                stosb           ; mov [edi], al    +    edi:=edi+1
                                ; we only need to copy the low byte (i.e. AL) into the “u” string
                loop repeat

        ; next, we need to copy the high bytes of the words from string “s” into the string “u”
        mov esi, t              ; set the offset of the source string “t”
        mov ecx, len_t          ; use a loop with len_t iterations
        jecxz theend
                        .
        repeta1:
                lodsw           ; mov ax, [esi]    +    esi:=esi+2
                                ; AL will store the low byte of the current word from string “s”
                                ; AH will store the high byte of the current word from string “s”

                xchg al, ah     ; interchange AL with AH
                                ; we need to put the high byte in AL in order to use stosb below
                stosb           ; mov [edi], al    +    edi:=edi+1

        loop repeta1    ;the loop block could have also been written like this:
                        ; repeta1:
                                ; lodsb
```

```
                        ; lodsb
                        ; stosb
                        ; loop repeta1

    ; We now begin the second part of the program, that is sorting the string "u" in ascending order (in
    ; the signed interpretation). In order to perform the sorting, we use a variant of bubble sort
    ; algorithm which is depicted below :
    ;       // u is a vector of length "len"
    ;       changed = 1;
    ;       while (changed = =1) {
    ;               changed = 0;
    ;               for (i=1; i<=len-1; i++) {
    ;                       if (u[i+1]<u[i]) {
    ;                               aux = u[i];
    ;                               u[i] = u[i+1];
    ;                               u[i+1] = aux;
    ;                               changed = 1;
    ;                       }
    ;               }
    ;       }

    mov dx, 1           ; the equivalent of "changed=1" from the algorithm.

repeat2:
    cmp dx, 0
    je theend           ; if DX=0 then it means that there was no change in the last parse of the
                        ; string, so we exit the loop because the string is sorted ascending
    mov esi, u          ; prepare the parsing of string "u"; set the starting offset in ESI
    mov dx, 0           ; initialize DX
    mov ecx, len-1      ; parse string "u" in a loop with len-1 iterations (the equivalent of the "for"-
                        ; loop from the above algorithm).

    repeat3:
        mov al, byte [esi]      ; al = u[i].
        cmp al, byte [esi+1]    ; compare al=u[i] cu u[i+1]
        jle next                ; if u[i]<=u[i+1] move to the next iteration (i++). Otherwise
                                ; interchange u[i]  (byte [esi]) with u[i+1] (byte [esi+1]) in the
                                ; following 3 instructions. We used the "jle" instruction because
                                ; we want to do signed comparison.
        mov ah, byte [esi+1]
        mov byte [esi], ah
        mov byte [esi+1], al
        mov dx, 1               ; set DX to 1 in order to signalize that an interchange happened

        next:
            inc esi             ;we move to the next byte in the "u" string (equivalent to " i++").
            loop repeat3        ; resume repeat3 if we did not reach the end of string "u"
            jmp repeat2         ; otherwise resume the repeat2 cycle.

theend:
    push dword 0
    call [exit]
```

Seminar IV. String instructions. Complex string problems.

The string instructions have all default operands and they work in the following pattern: they do something with the current element of the string(s) and they move to the next element in the string(s). In order to work with string instructions, we must initially:

- set the offset of the source string in ESI (the source string is the one we do not modify)
- set the offset of the destination string in EDI (the destination string is the one we modify)
- set the parsing direction (rom. directia de parcurgere) of strings; if the Direction Flag DF=0 strings are parsed from left to right and if DF=1 strings are parsed from right to left

Some string instructions work only with the source string, some others work only with the destination string and some others work with both.

String instructions for data transfer

(Load String of Bytes)
1. LODSB AL← <DS:ESI>
if DF=0 inc(ESI) else dec(ESI)

(Load String of Words)
2. LODSW AX← <DS:ESI>
if DF=0 ESI←ESI+2 else ESI←ESI-2

(Store String of Bytes)
3. STOSB <ES:EDI>← AL
if DF=0 inc(EDI) else dec(EDI)

(Store String of Words)
4. STOSW <ES:EDI>← AX
if DF=0 EDI←EDI+2 else EDI←EDI-2

(Move String of Bytes)
5. MOVSB <ES:EDI>← <DS:ESI>
if DF=0 {inc(ESI); inc(EDI)} else {dec(ESI); dec(EDI)}

(Move String of Words)
6. MOVSW <ES:EDI>← <DS:ESI>
if DF=0 {ESI←ESI+2; EDI←EDI+2}
else {ESI←ESI-2; EDI←EDI-2}

String instructions for data comparisons

(Scan String of Bytes)
7. SCASB CMP AL, <ES:EDI>
if DF=0 inc(EDI) else dec(EDI)

(Scan String of Words)

8. SCASW CMP AX, <ES:EDI>
if DF=0 EDI←EDI+2 else EDI←EDI-2

(Compare String of Bytes)

9. CMPSB CMP <DS:ESI>, <ES:EDI>
if DF=0 {inc(ESI); inc(EDI)} else {dec(ESI); dec(EDI)}

(Compare String of Words)

10. CMPSW CMP <DS:ESI>, <ES:EDI>
if DF=0 {ESI←ESI+2; EDI←EDI+2}
else {ESI←ESI-2; EDI←EDI-2}

There also exist instructions LODSD, STOSD, MOVSD, SCASD, CMPSD that work with strings of doublewords, use EAX and always increment/decrement ESI and EDI by 4 bytes.

We solve the last problem from the previous seminar using string instructions.

Ex.1. Being given a string of bytes containing lowercase letters, build a new string of bytes containing the corresponding uppercase letters.

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
   s1 db 'abcdef'
   lenS1 equ $-s1                ; defines the length in bytes of string “s1”, i.e. 6
   s2 times lenS1 db 0           ; reserve lenS1 bytes for string “s2”

segment code use32 class=code
start:
   mov esi, s1          ; set the offset of the source string s1 in ESI
   mov edi, s2          ; set the offset of the destination string s2 in EDI
   mov ecx, lenS1       ; we will use a loop/cycle with lenS1 iterations
   cld

   repeat:
      lodsb          ; mov al, [esi]    +   inc esi
      sub al, 'a' -'A'
      stosb          ; mov [edi], al    +   inc edi

      loop repeat    ; is equivalent to these 3 instructions:
```

```
                ;       dec ecx
                ;       cmp ecx, 0
                ;       ja repeat

    push dword 0
    call [exit]
```

Ex.2. Being given a string of bytes, write a program that obtains the mirrored string of bytes.

Example: Being given the string of bytes:
s db 17, 20, 42h, 1, 10, 2ah
the corresponding mirrored string of bytes will be
t db 2ah, 10, 1, 42h, 20, 17.

In order to solve the problem, we will parse the initial string "s" in a loop and copy each byte in string "t". While string "s' will be parsed from left to right (i.e. DF=0), string "t" will be parsed from right to left (i.e. DF=1). Thus, the first byte of string "s" will be copied in the last byte of string "t", the second byte of string "s" will be copied in the last but one byte of string "t" and so on..

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
        s   db   17, 20, 42h, 1, 10, 2ah
        len_s   equ   $-s
        t   times  len_s  db 0

segment code use32 class=code
start:
        mov esi, s      ; set the starting offset of the source string "s" in ESI
                        ; ESI now contains the offset of the first byte in string "s"
        mov edi, t      ; set the starting offset of the destination string "t" in EDI
                        ; but because string "t" needs to be parsed from right to left, the starting offset
                        ; of string "t" should be the offset of the last byte in string "t" (i.e. EDI=t + len_s - 1)
        add edi, len_s-1
        mov ecx, len_s
        jecxz theend    ; if ECX==0 jump to "theend"
repeat:
        cld             ; DF=0 (parse strings from left to right)
        lodsb           ; mov al, [esi]    +    inc esi

        std             ; DF=0 (parse strings from right to left)
        stosb           ; mov [edi], al    +    dec edi
```

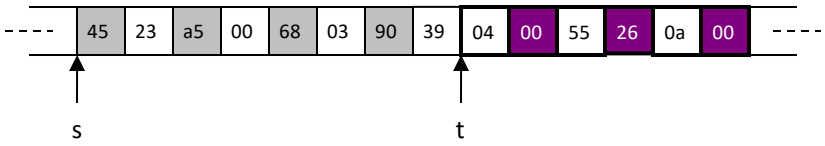
```
        loop repeat     ;

theend:
        push dword 0
        call [exit]
```

Ex.3. Two strings of words are given. Concatenate the string of low bytes of the words from the first string to the string of high bytes of the words from the second string. The resulted string of bytes should be sorted in ascending order in the signed interpretation.

Example: Having the strings of words:

```
s   dw   2345h, 0a5h, 368h, 3990h
t   dw   4h, 2655h, 10
```

these strings will be represented in the memory in little-endian format as (the colored bytes are the ones required by the text of the problem):



The result string should be:

u: 90h, a5h, 0h, 0h, 26h, 45h, 68h

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
        s   dw   2345h, 0a5h, 368h, 3990h
        len_s   equ   ($-s)/2           ; the length (in words) of string "s"
        t   dw   4h, 2655h, 10
        len_t   equ   ($-t)/2           ; the length (in words) of string "t"
        len   equ   len_s+len_t         ; the length of the result string

        u   times len  db  0            ; the result string

segment code use32 class=code
start:
        ; first we copy the low bytes of the words from string "s" into the resulted string "u"

        mov esi, s      ; set the offset of the source string (i.e. the offset of the 1st byte from string "s")
```

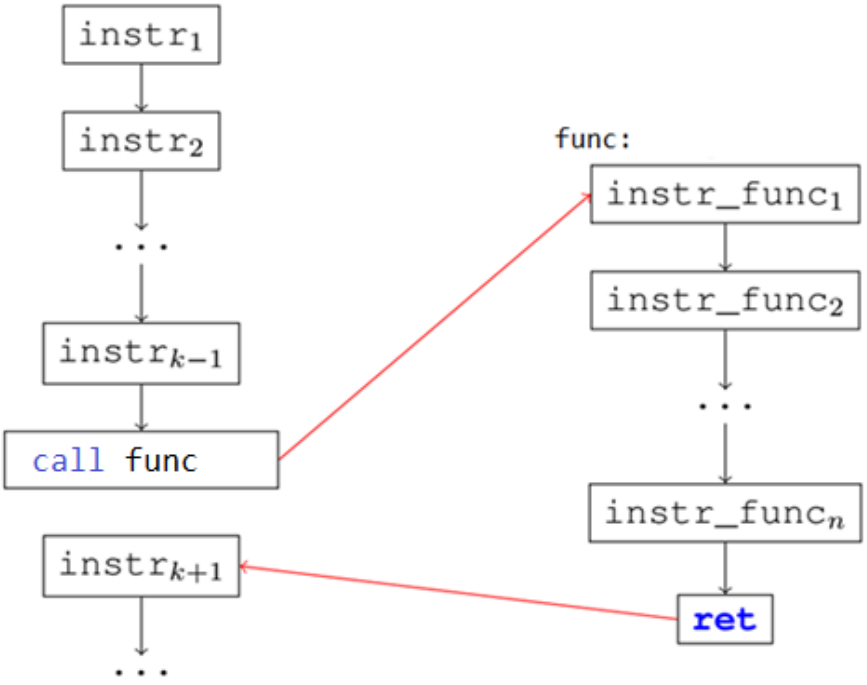
```
mov edi, u      ; set the offset of the dest string (i.e. the offset of the 1st byte from string “u”)

cld             ; DF=0

mov ecx, len_s          ; use a loop with len_s iterations
jecxz theend
repeat:
        lodsw           ; mov ax, [esi]    +    esi:=esi+2
                        ; AL will store the low byte of the current word from string “s”
                        ; AH will store the high byte of the current word from string “s”

        stosb           ; mov [edi], al    +    edi:=edi+1
                        ; we only need to copy the low byte (i.e. AL) into the “u” string
        loop repeat

; next, we need to copy the high bytes of the words from string “s” into the string “u”
mov esi, t              ; set the offset of the source string “t”
mov ecx, len_t          ; use a loop with len_t iterations
jecxz theend
                .
repeta1:
        lodsw           ; mov ax, [esi]    +    esi:=esi+2
                        ; AL will store the low byte of the current word from string “s”
                        ; AH will store the high byte of the current word from string “s”

        xchg al, ah     ; interchange AL with AH
                        ; we need to put the high byte in AL in order to use stosb below
        stosb           ; mov [edi], al    +    edi:=edi+1

loop repeta1    ;the loop block could have also been written like this:
                ; repeta1:
                        ; lodsb
                        ; lodsb
                        ; stosb
                        ; loop repeta1

; We now begin the second part of the program, that is sorting the string “u” in ascending order (in
; the signed interpretation). In order to perform the sorting, we use a variant of bubble sort
; algorithm which is depicted below :
;       // u is a vector of length “len”
;       changed = 1;
;       while (changed = =1) {
;               changed = 0;
;               for (i=1; i<=len-1; i++) {
;                       if (u[i+1]<u[i]) {
;                               aux = u[i];
;                               u[i] = u[i+1];
```

```
        ;                               u[i+1] = aux;
        ;                               changed = 1;
        ;                       }
        ;               }
        ;       }

        mov dx, 1               ; the equivalent of “changed=1” from the algorithm.

repeat2:
        cmp dx, 0
        je theend               ; if DX=0 then it means that there was no change is the last parse of the
                                ; string, so we exit the loop because the string is sorted ascending
        mov esi, u              ; prepare the parsing of string “u”; set the starting offset in ESI
        mov dx, 0               ; initialize DX
        mov ecx, len-1          ; parse string “u” in a loop with len-1 iterations (the equivalent of the “for”-
                                ; loop from the above algorithm).

        repeat3:
                mov al, byte [esi]      ; al = u[i].
                cmp al, byte [esi+1]    ; compare al=u[i] cu u[i+1]
                jle next                ; if u[i]<=u[i+1] move to the next iteration (i++). Otherwise
                                        ; interchange u[i]  (byte [esi]) with u[i+1] (byte [esi+1]) in the
                                        ; following 3 instructions. We used the “jle” instruction because
                                        ; we want to do signed comparison.
                mov ah, byte [esi+1]
                mov byte [esi], ah
                mov byte [esi+1], al
                mov dx, 1               ; set DX to 1 in order to signalize that an interchange happened

                next:
                        inc esi         ;we move to the next byte in the “u” string (equivalent to “ i++”).
                        loop repeat3    ; resume repeat3 if we did not reach the end of string “u”
                        jmp repeat2     ; otherwise resume the repeat2 cycle.

theend:
        push dword 0
        call [exit]
```

Seminar 5

Library functions call

In order to call functions from a library (e.g. a .dll or .lib library), we need to use the
`call [functionname]`
instruction which pushes the current memory address (i.e. the Return Address) on the stack and performs a jump to the starting address of `functionname`. This allows a `RET` instruction to pop the return address into either CS:IP or IP only (depending on whether it is a Near or Far call) and thus return control to the instruction immediately after the `CALL` instruction.



Before we call the function we need to pass the actual parameters to the function. The parameters are passed to the function using the stack using the cdecl calling convention (although there are other calling conventions that can be used). This calling convention has the following rules:

- The parameters are passed on the stack from right to left; an element of the stack is a dword
- The default result is returned by the function in EAX
- The EAX, ECX, EDX registers can be modified in the body of the function (there is no warranty that they keep their initial value, i.e. the value they had before entering the function, when exiting the function).
- The function will not free the parameters from the stack; it is the responsibility of the calling code

Pay attention! If we try to keep a counter value in **ECX** while executing a **loop** that calls a function, the call could destroy the counter value in ECX. We must save ECX on the stack before each call to a library function, and restore it after the library call returns.

A list of C run-time library functions (i.e. functions of the msvcrt.dll library) can be found here: *https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/crt-alphabetical-function-reference?view=vs-2017*

For printing something on the screen, we will use the function *printf()*. The syntax of this function is:
int printf(const char *format, <value1>, <value2>, …)
where *format* is a string that specifies what is printed on the screen and *value1, value2 ...* are values (bytes, words, doublewords, strings). Every character that appears in *format* is printed on the screen exactly as it is, except the characters that are preceded by '%' which will be replaced by values from the *value1, value2 ...* list. The first character preceded by '%' from *format* will be replaced when printed with *value1*, the second character preceded by '%' from *format* will be replaced when printed with *value2*, … In assembly, any value from the values list can be a constant or a variable. If it is a constant or a variable different than string, its value will be placed on the stack. If the value is a variable of type string, its offset will be placed on the stack. Because we only pass the offset of a string, we must also mark the ending of a string. Therefore strings must be ASCIIZ strings, i.e. in the data segment they must end with a ZERO.

There are multiple type specifications:
%d – signed decimal number
%u – unsigned decimal number
%s – string
%c – character (Although a character is a byte, you need to place a doubleword on the stack!)
%x – unsigned hexadecimal number
etc.

Below there are some examples:
printf("a=%d", x) - prints on the screen "a=[value of x]"
printf("%d + %d=%d",a,b,c) - prints on the screen "[value of a] + [value of b] = [value of c]"
printf("%s %d", s, a) - prints on the screen "[string s] [value of a]".

Function call	**Assembly code**
printf("Seminar 5");	segment data use32 class=data text db "Seminar 5", 0 segment code use32 class=code push dword text call [printf] add esp, 4 * 1

printf("Seminar %u ASC ", 5);	segment data use32 class=data format db "Seminar %u ASC", 0 segment code use32 class=code push dword 5 push dword format call [printf] add esp, 4 * 2

Conversly, we use the function *scanf()* for reading from the keyboard. The syntax is:
int scanf(const char *format, <variable1>, <variable2>, …)
where *format* is a string that specifies what is read from the keyboard and *variable1, variable2 ...* are offsets of variables in assembly (of types bytes, words, dwords, strings). The *format* string should only contain '%' characters followed by a type specification like %d - decimal, %s - string, %c – character. The first '%' expression describes the type of the first value that is read and set to *variable1*. The second '%' expression describes the type of the second value that is read and set to *variable2*. Etc.

Some examples below:
scanf("%d %d", a, b) - reads two integer/decimal values and sets them to a and b
scanf("%s", s) - reads a string into variable s

Function call	**Assembly code**
scanf("%d", &n);	segment data use32 class=data n dd 0 format db "%d",0 segment code use32 class=code push dword n push dword format call [scanf] add esp, 4 * 2
scanf("%s%d", &day,&degrees);	segment data use32 class=data day times 10 db 0 degrees dd 0 format db "%s%d", 0 segment code use32 class=code push dword degrees push dword day push dword format call [scanf] add esp, 4 * 3

Ex.1. The code below will print the message ”n=” on the screen and then will read from the keyboard the value for the signed number n.

```
bits 32
global start
extern exit, printf, scanf          ; exit, printf and scanf are external functions
import exit msvcrt.dll
import printf msvcrt.dll            ; tell the assembler that function printf is in msvcrt.dll
import scanf msvcrt.dll             ;

segment data use32 class=data
        n dd  0
        message  db "n=", 0         ; strings for C functions must end with ZERO (ASCIIZ strings)
        format  db "%d", 0          ; strings for C functions must end with ZERO (ASCIIZ strings)

segment code use32 class=code
   start:

     ; calling printf(message) => "n=" will be printed on the screen
     push dword message             ; we store the offset of message (not its value) on the stack
     call [printf]                  ; call printf
     add esp, 4*1                   ; free parameters from the stack; 4 = dword size in bytes
                                    ; 1 = number of parameters
    ; remember that the stack grows towards small addresses and the elements of the stack are dwords.
    ; that is, assuming the dword from the top of the stack  is at address ADR, by pushing another dword
    ; on top of the stack, the new dword is on address ADR-4. ESP always points to the top of the stack.
    ; we clear/free 4 bytes from the top of the stack by “add ESP, 4”

     ; call scanf(format, n) => read a decimal number in variable n
     ; parameters are placed on the stack from right to left
     push dword n                   ;  push the offset of n
     push dword format              ; push the offset of format
     call [scanf]                   ;
     add esp, 4 * 2                 ; free 2 dwords from the stack

     ; call exit(0)
     push dword 0
     call [exit]
```

Ex.2. A program that reads 2 numbers, a and b, computes their sum and prints it on the screen.

```
bits 32
global start
extern exit, printf, scanf
import exit msvcrt.dll
import printf msvcrt.dll
import scanf msvcrt.dll

segment data use32 class=data
        a dd 0
        b dd 0
        result dd 0
        format1 db 'a=', 0                  ; all formats used for scanf/printf are required to be ASCIIZ strings
        format2 db 'b=', 0                  ; all formats used for scanf/printf are required to be ASCIIZ strings
        readformat db '%d', 0               ; all formats used for scanf/printf are required to be ASCIIZ strings
        printformat db '%d + %d = %d\n', 0  ; all formats are required to be ASCIIZ strings



segment code use32 class=code
start:
        ; call printf("a=")
        push dword format1
        call [printf]
        add esp, 4*1

        ; call scanf("%d", a)
        push dword a                ; push the offset of a for reading (not its value)
        push dword readformat
        call [scanf]
        add esp, 4*2

        ; call printf("b=")
        push dword format2
        call [printf]
        add esp, 4*1

        ; call scanf("%d", b)
        push dword b                ; push the offset of a for reading (not its value)
        push dword readformat
        call [scanf]
        add esp, 4*2

        mov eax, [a]
        add eax, [b]
        mov [result], eax
```

```
; call printf("%d + %d = %d\n", a, b, result)
push dword [result]             ; push the value of result for printing
push dword [b]                  ; push the value of b for printing
push dword [a]                  ; push the value of a for printing
push dword printformat
call [printf]
add esp,4*4

push dword 0
call [exit]
```

Working with files

1. Open a file
FILE * fopen(const char* filename, const char * access_mode)

ARGUMENTS

Mode	Meaning	Description
r	read	- Open file for reading. - The file must exist.
w	write	- If the file does not exist, it creates a new file and opens it for writing. - If a file with the given name exists, it opens it for writing. It overwrites the content of the file.
a	append	- If the file does not exist, it creates a new file and opens it for writing. - If a file with the given name exists, it opens it for writing. It does not overwrite the content, it continues writing at the end of the file.
r+	Read+write for existing files	- Open file for reading and writing. - The file must exist.
w+	Read+write	- If the file does not exist, it creates a new file and opens it for reading and writing. - If a file with the given name exists, it opens it for reading and writing. It overwrites the content of the file
a+	Read+append	- If the file does not exist, it creates a new file and opens it for reading and writing. - If a file with the given name exists, it opens it for reading and writing. It does not overwrite the content, it continues writing at the end of the file.

RESULTS

If the file is successfully opened, **EAX will contain the file descriptor** (an identifier) which can be used for working with the file (reading and writing). **If an error occurs, fopen will set EAX to 0**

2. Write into a file **int fprintf(FILE * stream, const char * format, <variable_1>, < variable _2>, <...>)**
RESULT
In case of an **error**, **EAX** contains a value **< 0**

2. Write into a file **int fwrite(const void * str, int size, int count, FILE * stream)**
• First argument is the string containing the elements to be written • Second argument represents the size of each element to be written • Third argument represents the number of elements to be written • Last argument is the file descriptor
RESULT
EAX will contain the number of elements written. If this number is below **count**, it means that there was an error.

3. Read from a file **int fscanf (FILE * stream, const char * format, <variable1>, <variable2>, ...)**
• First argument is the file descriptor • *Format* is a string specifying what is read from the file and it should only contain '%' followed by a type specification • *variable1*, *variable2*, .. are offsets of the variables in assembly
RESULT
EAX will contain the number of fields successfully converted and assigned.

3. Read from a file **int fread(void * str, int size, int count, FILE * stream)**
• First argument is the string where the bytes that are read from the file are stored • Second argument represents the size of the elements that are read from the file • Third argument represents the maximum number of elements to be read • Last argument is the file descriptor
RESULT
EAX will contain the number of elements read. If this number is below **count**, it means either that there was an error, or that the function got to the end of the file. Obs. When reading the content of a file in smaller chunks, we can check the value of EAX to know when we reached the end of the file.

4. Closing an open file **int fclose(FILE * descriptor)**
RESULT
EAX will contain the value 0 if the file is succesfully closed.

Observations:

- The name of the file must include the extension. (ex. test.txt)
- The files that we are working with must be in the current directory.

Ex. 3

```
; This program reads the content of a text file (a.txt), adds 1 to each byte and then writes
; these bytes to a new file (b.txt) and then renames this new file to be the old file name (a.txt).
bits 32

global start

; declare external functions needed by our program
extern exit, perror, fopen, fclose, fread, fwrite, rename, remove
import exit msvcrt.dll
import fopen msvcrt.dll
import fread msvcrt.dll
import fwrite msvcrt.dll
import fclose msvcrt.dll
import rename msvcrt.dll
import remove msvcrt.dll
import perror msvcrt.dll

segment data use32 class=data
   inputfile db 'a.txt', 0
   outputfile db 'b.txt', 0
   modread db 'r', 0
   modwrite db 'w', 0
   c db 0
   handle1 dd -1
   handle2 dd -1
   eroare db 'error:', 0


segment code use32 class=code
start:

   ; fopen(string path, string mode)   - opens the file path in the specified mode. mode can be “r”
   ; for reading the file or “w” for writing the file
   push dword modread  ; for strings, the offset is pushed on the stack
   push dword inputfile    ; for strings, the offset is pushed on the stack
   call [fopen]
   add esp, 4*2
```

```
   ; fopen returns in EAX the file handle (or zero in case of error)
   ; this file handle is just a dword used by the operating system and is required for all subsequent
   ; function calls that work with this file.
   mov [handle1], eax            ; store the handle in a local variable
   cmp eax, 0
   je theend                     ; if error, move to the end of the program


; fopen(string path, string mode)
   push dword modwrite ; open the outputfile for writting
   push dword outputfile
   call [fopen]
   add esp, 4*2

   ; fopen returns in EAX the file handle or zero (in case of error)
   mov [handle2], eax            ; store the second handle in a local variable
   cmp eax, 0
   je theend

   repeat:
      ;fread(string ptr, integer size, integer n, FILE * handle)    - reads n times size bytes from the
      ; file identified by handle and place the read bytes in the string ptr.
      ; we read 1 byte from the file handle1
      push dword [handle1]              ; read from handle1
      push dword 1                      ; read 1 time
      push dword 1                      ; read 1 byte
      push dword c                      ; store the byte in c
      call [fread]
      add esp, 4*4
      cmp eax, 0                ; the function returns zero in EAX in case of error
      je error

      add byte [c], 1


      ;fwrite(string ptr, integer size, integer n, FILE * handle)    - writes n times size bytes from
      ; the string ptr into the file identified by handle.
      ; write 1 byte in file handle2
      push dword [handle2]              ; write into file handle     2
      push dword 1                      ; write 1 time
      push dword 1                      ; write 1 byte
      push dword c                      ; from c
      call [fwrite]
      add esp, 4*4

      cmp eax, 0
```

```
    je error

    jmp repeat

error:
    ; fclose(FILE* handle)          - close the file identified by handle
    push dword [handle1]
    call [fclose]
    add esp, 4*1

    ; fclose(FILE* handle)          - close the file identified by handle
    push dword [handle2]
    call [fclose]
    add esp, 4*1

; remove( string path )          - remove the file path
push dword inputfile
call [remove]
add esp, 4*1

; rename( string oldname, string newname )    - rename the fine oldname into newname
push dword inputfile
push dword outputfile
call [rename]
add esp, 4*2

cmp eax, 0              ; returns 0 if it is successful. On an error, the function returns a nonzero value
je theend               ; and an error message which can be printed using the “perror()” function

; call perror(eroare) in case of error so that we see a more detailed error message.
push dword eroare
call [perror]
add esp, 4*1

theend:
; exit(0)
push dword 0
call [exit]
```

Ex. 4 Read from the keyboard a number **n** in base 16 that can be represented on a word (no validation needed). Open the file *in.txt*, which contains exactly 16 bytes, and print on the screen only those bytes that correspond to a position of a bit with the value 1 in the binary representation of the number **n**.

Example:

- n = F2A1h = 1111 0010 1010 0001b
- *in.txt* contains: 0123456789abcdef => on the screen: 0579cdef

Seminar V. Library functions call

In order to call functions from a library (e.g. a .dll or .lib library), we need to use the
`call [functionname]`
instruction which pushes the current memory address (i.e. the Return Address) on the stack and performs a jump to the starting address of `functionname`. Before we call the function we need to pass the actual parameters to the function. The parameters are passed to the function using the stack using the cdecl calling convention (although there are other calling conventions that can be used). This calling convention has the following rules:

- The parameters are passed on the stack from right to left; an element of the stack is a dword
- The default result is returned by the function in EAX
- The EAX, ECX, EDX registers can be modified in the body of the function (there is no warranty that they keep their initial value (i.e. the value they had before entering the function) when exiting the function.
- The function will not free the parameters from the stack; it is the responsibility of the calling code

A list of C run-time library functions (i.e. functions of the msvcrt.dll library) can be found here: *https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/crt-alphabetical-function-reference?view=vs-2017*

For printing something on the screen, we will use the function *printf()*. The syntax of this function is:
printf (string format, value1, value2, ...)
where *format* is a string that specifies what is printed on the screen and *value1, value2 ...* are values (bytes, words, dwords, strings). Every character that appears in *format* is printed on the screen exactly as it is, except the characters that are preceeded by '%' which will be replaced by values from the *value1, value2 ...* list. The first character preceeded by '%' from *format* will be replaced when printed with *value1*, the second character preceeded by '%' from *format* will be replaced when printed with *value2*, … In assembly, any value from the values list can be a constant or a variable. If it is a constant or a variable different than string, its value will be placed on the stack. If the value is a variable of type string, its offset will be placed on the stack. Below there are some examples:
printf("a=%d", x) - prints on the screen "a=[value of x]"
printf("%d + %d=%d",a,b,c) - prints on the screen "[value of a] + [value of b] = [value of c]"
printf("%s %d", s, a) - prints on the screen "[string s] [value of a]".

Conversly, we use the function *scanf()* for reading from the keyboard. The syntax is:
scanf (string format, variable1, variable2, ...)
where *format* is a string that specifies what is read from the keyboard and *variable1, variable2 ...* are offsets of variables in assembly (of types bytes, words, dwords, strings). The *format* string should only contain '%' characters followed by a type specification like %d - decimal, %s - string, %c – character. The first '%' expression describes the type of the first value that is read and set to *variable1*. The second '%' expression describes the type of the second value that is read and set to *variable2*. Etc. Some examples below:

scanf("%d %d", a, b) - reads two integer/decimal values and sets them to a and b
scanf("%s", s) - reads a string into variable s

Ex.1. The code below will print the message "n=" on the screen and then will read from the keyboard the value for the number n.

```
bits 32
global start
extern exit, printf, scanf          ; exit, printf and scanf are external functions
import exit msvcrt.dll
import printf msvcrt.dll            ; tell the assembler that function printf is in msvcrt.dll
import scanf msvcrt.dll             ;

segment data use32 class=data
        n dd  0
        message  db "n=", 0         ; strings for C functions must end with ZERO (ASCIIZ strings)
        format  db "%d", 0          ; strings for C functions must end with ZERO (ASCIIZ strings)

segment code use32 class=code
  start:

    ; calling printf(message) => "n=" will be printed on the screen
    push dword message              ; we store the offset of message (not its value) on the stack
    call [printf]                   ; call printf
    add esp, 4*1                    ; free parameters from the stack; 4 = dword size in bytes
                                    ; 1 = number of parameters
    ; remember that the stack grows towards small addresses and the elements of the stack are dwords.
    ; that is, assuming the dword from the top of the stack  is at address ADR, by pushing another dword
    ; on top of the stack, the new dword is on address ADR-4. ESP always points to the top of the stack.
    ; we clear/free 4 bytes from the top of the stack by "add ESP, 4"

    ; call scanf(format, n) => read a decimal number in variable n
    ; parameters are placed on the stack from right to left
    push dword n                    ;  push the offset of n
    push dword format               ; push the offset of format
    call [scanf]                    ;
    add esp, 4 * 2                  ; free 2 dwords from the stack

    ; call exit(0)
    push dword 0                    ; punem pe stiva parametrul pentru exit
    call [exit]                     ; apelam exit pentru a incheia programul
```

Ex.2. A program that reads 2 numbers, a and b, computes their sum and prints it on the screen.

```
bits 32
global start
extern exit, printf, scanf
```

```
import exit msvcrt.dll
import printf msvcrt.dll
import scanf msvcrt.dll

segment data use32 class=data
        a dd 0
        b dd 0
        result dd 0
        format1 db 'a=', 0                  ; all formats used for scanf/printf are required to be ASCIIZ strings
        format2 db 'b=', 0                  ; all formats used for scanf/printf are required to be ASCIIZ strings
        readformat db '%d', 0               ; all formats used for scanf/printf are required to be ASCIIZ strings
        printformat db '%d + %d = %d\n', 0  ; all formats are required to be ASCIIZ strings

segment code use32 class=code
start:
        ; call printf("a=")
        push dword format1
        call [printf]
        add esp, 4*1

        ; call scanf("%d", a)
        push dword a                ; push the offset of a for reading (not its value)
        push dword readformat
        call [scanf]
        add esp, 4*2

        ; call printf("b=")
        push dword format2
        call [printf]
        add esp, 4*1

        ; call scanf("%d", b)
        push dword b                ; push the offset of a for reading (not its value)
        push dword readformat
        call [scanf]
        add esp, 4*2

        mov eax, [a]
        add eax, [b]
        mov [result], eax

        ; call printf("%d + %d = %d\n", a, b, result)
        push dword [result]                 ; push the value of result for printing
        push dword [b]                      ; push the value of b for printing
        push dword [a]                      ; push the value of a for printing
        push dword printformat
        call [printf]
```

```
        add esp,4*4

        push dword 0
        call [exit]
```

Ex. 3

; This program reads the content of a text file (a.txt), adds 1 to each byte and then writes
; these bytes to a new file (b.txt) and then renames this new file to be the old file name (a.txt).

```
bits 32

global start

; declare external functions needed by our program
extern exit, perror, fopen, fclose, fread, fwrite, rename, remove
import exit msvcrt.dll
import fopen msvcrt.dll
import fread msvcrt.dll
import fwrite msvcrt.dll
import fclose msvcrt.dll
import rename msvcrt.dll
import remove msvcrt.dll
import perror msvcrt.dll

segment data use32 class=data
   inputfile db 'a.txt', 0
   outputfile db 'b.txt', 0
   modread db 'r', 0
   modwrite db 'w', 0
   c db 0
   handle1 dd -1
   handle2 dd -1
   eroare db 'error:', 0

segment code use32 class=code
start:
   ; fopen(string path, string mode)   - opens the file path in the specified mode. mode can be “r”
   ; for reading the file or “w” for writing the file
   push dword modread  ; for strings, the offset is pushed on the stack
   push dword inputfile    ; for strings, the offset is pushed on the stack
   call [fopen]
   add esp, 4*2

   ; fopen returns in EAX the file handle or zero (in case of error)
   ; this file handle is just a dword used by the operating system and is required for all subsequent
   ; function calls that work with this file.
   mov [handle1], eax                ; store the handle in a local variable
```

```
cmp eax, 0
je theend                       ; if error, move to the end of the program

; fopen(string path, string mode)
push dword modwrite ; open the outputfile for writting
push dword outputfile
call [fopen]
add esp, 4*2

; fopen returns in EAX the file handle or zero (in case of error)
mov [handle2], eax              ; store the second handle in a local variable
cmp eax, 0
je theend

repeat:
   ;fread(string ptr, integer size, integer n, FILE * handle)    - reads n times size bytes from the
   ; file identified by handle and place the read bytes in the string ptr.
   ; we read 1 byte from the file handle1
   push dword [handle1]                 ; read from handle1
   push dword 1                         ; read 1 time
   push dword 1                         ; read 1 byte
   push dword c                         ; store the byte in c
   call [fread]
   add esp, 4*4

   cmp eax, 0                   ; the function returns zero in EAX in case of error
   je error

   add byte [c], 1

   ;fwrite(string ptr, integer size, integer n, FILE * handle)    - writes n times size bytes from
   ; the string ptr into the file identified by handle.
   ; write 1 byte in file handle2
   push dword [handle2]                 ; write into file handle     2
   push dword 1                         ; write 1 time
   push dword 1                         ; write 1 byte
   push dword c                         ; from c
   call [fwrite]
   add esp, 4*4

   cmp eax, 0
   je error

   jmp repeat


error:
```

```
    ; fclose(FILE* handle)          - close the file identified by handle
    push dword [handle1]
    call [fclose]
    add esp, 4*1

    ; fclose(FILE* handle)          - close the file identified by handle
    push dword [handle2]
    call [fclose]
    add esp, 4*1

; remove( string path )          - remove the file path
push dword inputfile
call [remove]
add esp, 4*1

; rename( string oldname, string newname )    - rename the fine oldname into newname
push dword inputfile
push dword outputfile
call [rename]
add esp, 4*2

cmp eax, 0              ; returns 0 if it is successful. On an error, the function returns a nonzero value
je theend               ; and an error message which can be printed using the “perror()” function

; call perror(eroare) in case of error so that we see a more detailed error message.
push dword eroare
call [perror]
add esp, 4*1

theend:
; exit(0)
push dword 0
call [exit]
```