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).