**Lab 1 - Basic Elements of Assembly Language**

Contents

[Constants 1](#_Toc83809901)

[Variables 1](#_Toc83809902)

[User-defined variables 1](#_Toc83809903)

[Pre-defined variables (CPU (central processor unit) registers): 2](#_Toc83809904)

[Instructions (mov, add, sub) 3](#_Toc83809905)

[Tools for laboratories 4](#_Toc83809906)

[Instructions: 4](#_Toc83809907)

[Useful infos in OllyDBG 6](#_Toc83809908)

[Conversions in Unsigned Representation (for natural numbers) 7](#_Toc83809909)

[Data in memory 8](#_Toc83809910)

[Homework: 11](#_Toc83809911)

## Constants

We have 3 types of constants in the IA-32 (*IA-32 – a* short for "Intel Architecture, *32*-bit", is the *32*-bit version of the x86 instruction set architecture, designed by Intel) 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, 101
* character; ex.: ‘a’, ‘B’, ‘c’, “a”, “B”
* string (sequence of characters); ex.: ‘abcd’, “test, “arhitectura sistemelor”

## Variables

A variable has a **name**, a **data type** (byte, word or doubleword), a **current value** and a m**emory location** (where the variable is stored).

The IA-32 assembly language has 2 kinds of variables:

* pre-defined variables
* user-defined variables

### User-defined variables

To define variables are used the directives:

A **directive**is a command embedded in the source code that is recognized and acted upon by the assembler.

Directives can define variables, macros, and procedures. They can assign names to memory segments and perform many other housekeeping tasks related to the assembler.

D – directive = DEFINE

Examples:

a db 2

b dw 4

c dd 5

For example, let’s take variable *a,* the name of this variable give us it’s adrress in the memory location.

If we want to acces the value from the *variable a,* the [] is used to acces the value from a:

[a] – the value from variable a

a without [] means the address of a

To reserve space for a variable in memory we are using the directive **RES** followed by the letter corresponding to the data type.

a resb 1 ; reserve 1 byte for variable a

b resw 1; reserve 1 word for variable b

c resd 1; reserve 1 doubleword for variable c

### Pre-defined variables (CPU (central processor unit) registers):

The CPU **registers** are memory areas located on the CPU which are used for various computations.

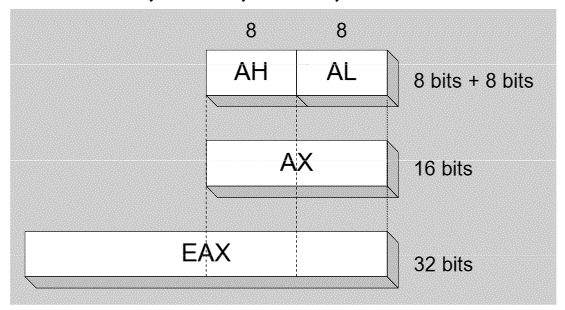
The IA-32 CPU registers (part 1) are:

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

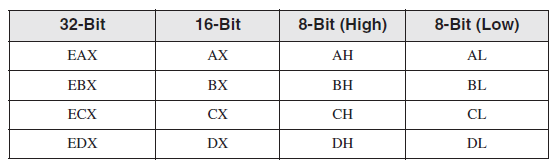
* **EAX** (the lower or least significant part of EAX can be referred by AX and AX is formed by two 8-bit subregisters, AL and AH) *– accumulator register.*

Eax [16 biti-partea high din eax|16 biti low din eax=ax]

**Eax [16 biti-partea high din eax|8 biti high din ax=ah|8 biti low din ax=al]**



* **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) – *base register.*
* **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) – *counter register*.
* **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) – *data register.*



The most useful table:

|  |  |  |
| --- | --- | --- |
| **Number of bits** | **Data type** | **Coressponding dimension registers** |
| 8 bits | Byte | AL, AH, BL, BH, CL, CH, DL, DH |
| 16 bits | Word | AX, BX, CX, DX |
| 32 bits | Doubleword | EAX, EBX, ECX, EDX |
| 64 bits | quadword | EDX:EAX, ECX:EBX |

## Instructions (mov, add, sub)

**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 ; ax will have value 2

mov [a], eax ; variable a will take the value from eax

mov 2, [a] – nu

mov 12, bx – nu

**dest, source – same data type**

a – db 10,

mov al, [a] and not mov ax, [a]

**ADD – addition instruction**

Syntax: add dest, source

(dest and source represent the two operands of the instructions)

(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 (de aceasi dimensiune, tip)

Examples: add bx, cx

add [a], 101b

add [a], cx

add 5, cx nu!

Add 10, [a] nu!

**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

sub ecx, [d] ; d-doubleword

sub 5, [d]

sub 101b, edx

## Tools for laboratories

* Editor: Notepad++
* Assembler: NASM
* Linker: ALINK
* Debugger: Olly DBG

The set of tools can be downloaded from Files (General Channel)

The set of tools already contains the editor Notepad++ which has an integrated plugin which allows you to perform most operations with a few key combinations.

Students that use other operating systems besides Windows can use an emulator (the tools have been succesfully tested with Wine), or run the tools in Windows virtual machine.

* [Students that use other operating systems besides Windows can use the Windows virtual machine available at: https://193.231.20.33/~forest/.](https://193.231.20.33/~forest/)

Instructions:

* Download the set of tools and unzip it (in a place where you will work in this semester
* Start the editor Notepad++ from the directory npp (don’t use your own version of Notepad++)
* Use the editor to implement the given laboratory problems. You can use the following key combinations that can also be found in the menu Plugins -> ASM Plugin:

|  |  |  |
| --- | --- | --- |
| ASM Code Template | Ctrl+Shift+N | Fills in the editor a minimal ASM program |
| Build ASM | Ctrl+F7 | Assemblies the current program |
| Run program | Ctrl+F6 | Runs the current program |
| Debug program | F6 | Debugs the current program, using Olly Debugger (the debugger documentation can be found in the ollydb directory from the archive) |

An **assembler** is a utility program that converts source code programs from assembly language into machine language.

A **linker** is a utility program that combines individual files created by an assembler into a single executable program.

A related utility, called a **debugger** lets you to step through a program while it’s running and examine registers and memory.

**The Assemble-Link-Execute Cycle**

The process of editing, assembling, linking, and executing assembly language programs is summarized as follows:

**Step 1:** A programmer uses a **text editor** to create an ASCII text file named the source file.

**Step 2:** The **assembler** reads the source file and produces an object file, a machine-language

translation of the program.

[Optionally, it produces a listing file.]

If any errors occur, the programmer must return to Step 1 and fix the program.

**Step 3:** The **linker** reads the object file and checks to see if the program contains any calls to

procedures in a link library. The **linker** copies any required procedures from the link library,

combines them with the object file, and produces the executable file.

**Step 4:** The operating system **loader** utility reads the executable file into memory and branches

the CPU to the program’s starting address, and the program begins to execute.

**Next**, open the notepad++ from npp folder from asm\_tools folder, create a new file.

In the menu, choose Pluggins -> ASM Pluggins -> ASM Code Template.

Save the file with .asm extension.

**Structure of a program in assembly**

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

; ...

; the program code will be part of a segment called code

segment code use32 class=code

start:

; ...

;here we will write the instructions to solve the problem

; 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

## Useful infos in OllyDBG

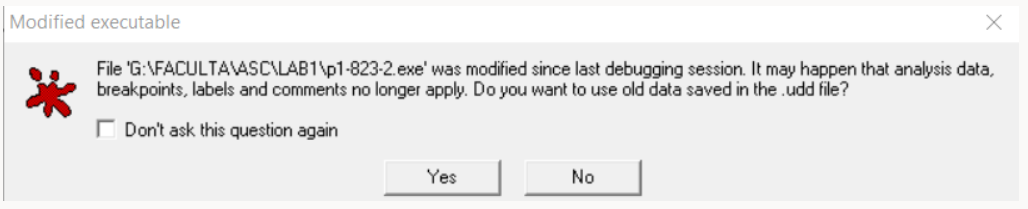
**When you are in the debugger:**

**F8 – to execute the code step by step**

**Ctrl + F2 – to restart the debugger (if doesn’t contain your written code)**

**Double-click on the numbers near registers to see the values inside of each register**

**Attention to this message:**



**The answer will be no all the time, because we need the last version of our code**

# Conversions in Unsigned Representation (for natural numbers)

Most of the instructions have the constraints that operands shoud have the same dimension (the same data type).

Therefore, is necessary to convert numbers from a data type to another data type.

In unsigned representation, we just add value 0 to the left until the new data type is completed.

**MOVZX instruction (*move with zero-extend*)**

* + copies the contents of a source operand into a destination operand and zero-extends the value to 16 or 32 bits

MOVZX *reg32, reg/mem8*

MOVZX *reg32, reg/mem16*

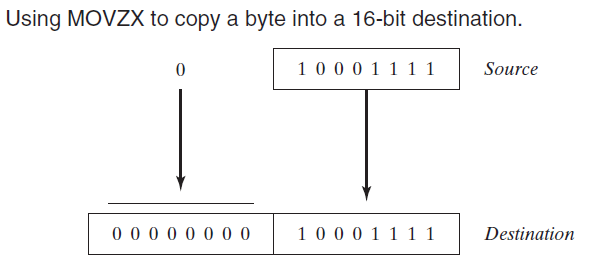
MOVZX *reg16, reg/mem8*

***Example:***

Ex1:

byteVal db 10001111b

movzx ax,[byteVal] ; AX = 0000000010001111b



Ex2:

* mov bx, 0A69Bh ;(in bl is 9B and in bh is A6)
* movzx eax, bx ; EAX = 0000A69Bh
* movzx edx, bl ; EDX = 0000009Bh
* movzx cx, bl ; CX = 009Bh

# Data in memory

The **smallest unit** of data in a computer is called **Bit** (Binary Digit) so the computer stores instructions and data in memory as collections of 0 and 1

A collection of 8 bits form a **byte**

2 bytes create a **word (16 bits)**

4 bytes create a **doubleword (32 bits)**

8 bytes create a **quadword (64 bits)**

* Each variable has an address – their address in memory

Example:

if in data segment we have: **a db 8**

if we want

* to access the value, we use **mov al, [a]**
* to access the address, we use **mov ebx, a** – because an address is all the time represented on 32 bits

An *address* refers to a single location in memory, and x86 processors permit each byte location to have a separate address

**In memory values of the variables are saved in base 16.**

**The manner in which is saved in memory is dependent on their data type:**

1 byte - consists in 2 hexadecimal digits and 1 byte is represented as it is in memory

1 word - consists in 4 hexadecimal digits (2 bytes) and 1 word is represented in memory according little-endian mechanism

1 doubleword - consists in 8 hexadecimal digits (4 bytes) and 1 doubleword is represented in memory according little-endian mechanism

1 quadword – consists in 16 hexadecimal digits (8 bytes) and 1 quadword is represented in memory according little-endian mechanism

The little-endian mechanism ***is based on the* order of the bytes: from low to high bytes in memory:**

* + The least significant byte is stored at the first memory address allocated for the data.
  + The remaining bytes are stored in the next consecutive memory positions.

Examples:

|  |  |  |
| --- | --- | --- |
| How is defined in data segment | How is value in its corresponding register | How looks in memory |
| b db 10 | MOV AL, [b] ; al =10=0Ah | 0A |
| w dw 10 | MOV AX, [w]; ax=10=000Ah | 0A 00 |
| d dd 10 | MOV EAX, [d]; eax=0000000Ah | 0A 00 00 00 |
| q dq 10 | ; q -> EDX:EAX  mov EAX, dword[q+0] ; eax=0000000Ah  mov EDX, dword[q+4]; edx=00000000h | 0A 00 00 00 00 00 00 00 |

X86 processors store and retrieve data from memory using ***little endian* order (low to high bytes)**.

A visual example in Olly DBG for data representation in memory:

***in data segment:***

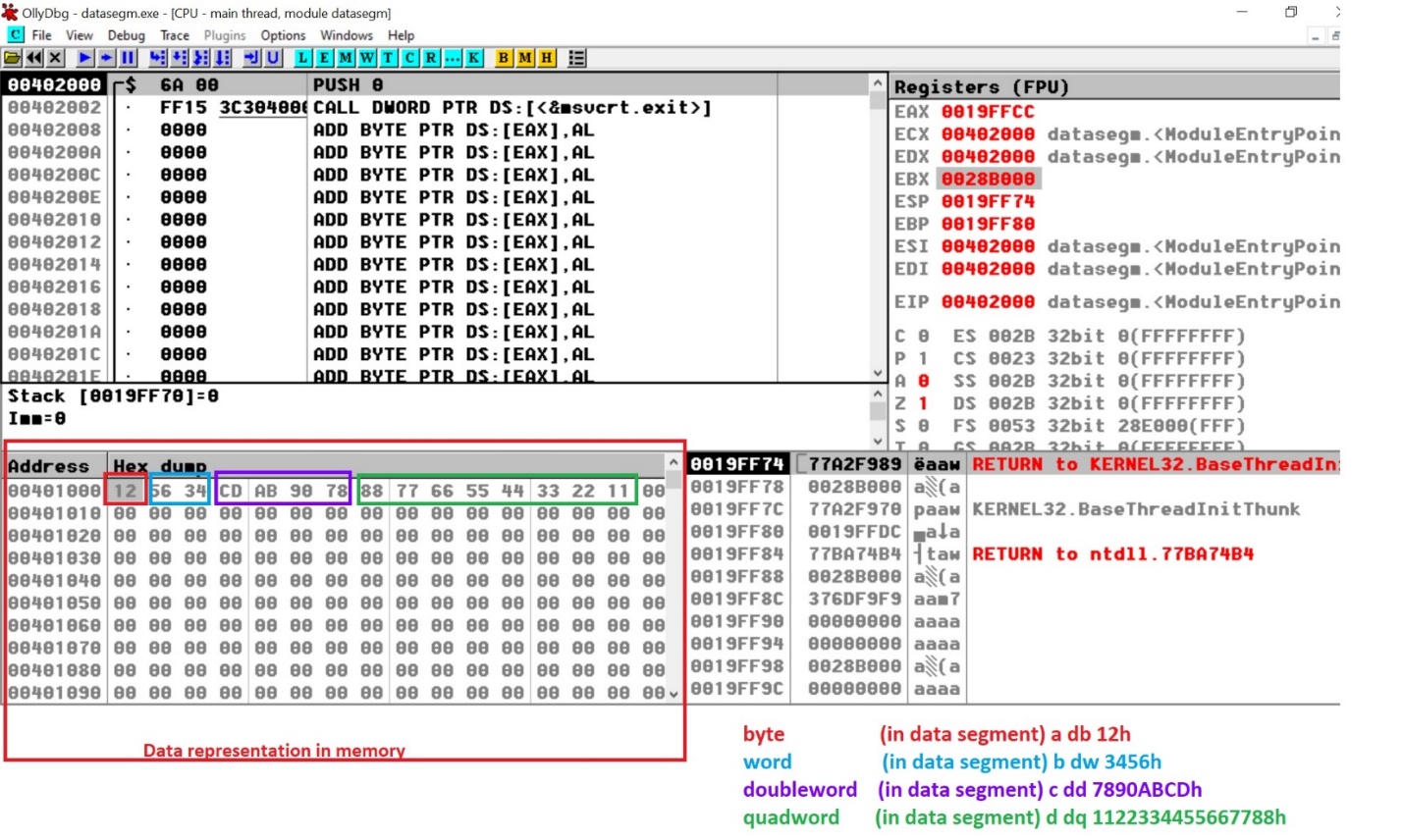
a db 12h

b dw 3456h

c dd 7890ABCDh

d dq 1122334455667788h

in memory from Olly DBG:



## Homework:

Each student will solve one assigned problem:

**Write a program to compute the E expression with the following variables:**

a –word, b – byte, c - word, d – byte

**All the final results will be saved in variable R of doubleword data type**

1. a+b-7-(c+d)

2. (10-d-a+4) + c

3. a+4+b-(5-d)

4. (b-a)+3+d-c

5. (2+a-b)-c+d

6. (6+d-b)-2+a

7. (3+b-a)-5+c

8. a-b+7-(c+d)

9. (d+3-a+4) + c

10. 4+b-(5-d)+a

11. c-(b-a)+3+d

12. (a-b-4)+c-d

13. (6+d+b)-2+a

14. (3-a+b)+5+c

15. (4-b-a)+5-d

16. (12+b-(2+c)-a)-b