

## Multi-module programming in assembly

### Purpose

Generally, non-trivial programs (real application programs) tend to be large, consisting of thousands of lines of code, which leads to increased code complexity.

As a consequence of this issue, the following questions arise inherently:

- \* how is it possible to “divide” or to “separate” the given problem into sub-problems of minimal difficulty?
- \* moreover, after the “separation”, which of the identified sub-problems is already known or already a sub-problem has a solution that can be reused?

One of the variants to “separate” the code into sub-problems is **code modularization**.

*(Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.)*

Assembly language programs do not “know” the concept of modularization and subprogram. But we can create a sequence of instructions that can be **called** from an area of a program and after the code sequence is executed to **return** the control to the program who made the call. Such a sequence of instructions is called subprogram or procedure.

If we consider to group a sequence of a code into a label, we can consider the call of a subprogram can be realized with a **jmp** instruction. But these solution is suitable only if we want to create sub-programs into the same .asm source file.

When we are trying to create a solution to a large problem using code from different .asm source files a particular problem appears: the problem is that **the processor does not know where to return when the subprogram** (the sequence from a second .asm file) **is over**. Therefore, **we have to save the return address** when we call a subprogram, and the return from the subprogram will actually be a jump to the return address.

The place where the **return address is stored** is the **execution stack** (basically the stack but because we are using the stack in this dynamic way is called the execution stack).

There are two instructions that allow us to call a procedure and to return from a procedure: call and ret.

firstasmsubprogram.asm	secondasmsubprogram.asm
;code	;code
...	...

<b>call</b> name_of_label	name_of_label
...	...
;code	;code
	<b>ret</b>

The **call instruction** is actually a jump instruction which also save on the stack the return address (the address of the instruction that follows the call instruction, not the jump destination).

The **ret instruction** transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a call instruction, and the return is made to the instruction that follows the call instruction.

The optional operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed.

Remarks:

- \* All data and code labels are visible within the entire program, therefore no duplicate labels should exist.
- \* We have the possibility to start a label with a point (.label). A label beginning with a point is treated as a **local label**.
- \* Example:
  - o .label: ; a label to be used in a procedure (a local label)
  - o label: ; a non-local label

The NASM assembler provides a simple mechanism to build a program from multiple source file through its preprocessor.

Using a similar syntax to the C preprocessor, NASM's preprocessor lets us to include other source files into our code. This is realized by the nasm.exe (from asm\_file/nasm folder).

A program written in assembly language can be split into several source files that can be assembled separately in **.obj files**.

In order to write a multi-module program, the following requirements should be fulfilled:

- \* **all segments should be declared using public qualifier**, because the code segment of the final program is built by concatenation of code segments from each module; the same for data segment
- \* all data and code labels from a module that need to be "exported" to other modules should be declared **using the global directive**
- \* data and code labels that are declared inside a module and that are used within other module should be "imported" using **the extern directive**
- \* a variable should be completely defined inside a single module (not one-half in one module and one-half in another).

- \* Also, the transfer of execution control from one module to another can be done only through jump instructions (**jmp, call, or ret**).

### **The stages (assembling / linking / debugging / running)**

**1. ASSEMBLING** step is realized with the next command:

```
nasm -f obj module.asm
```

The option -f specifies the type of file that will be generated, in this case an .obj file.

**2. LINKING** step is realised using the command alink with the following attributes:

```
alink module_1.obj ... module_n.obj -oPE -subsys console -entry start
```

There is a file called "ALINK.TXT" in the folder nasm of asm\_tools that describes the ALINK possible options.

ALINK options:

-o xxx

xxx:

COM = output COM file

EXE = output EXE file

PE = output Win32 PE file (.EXE)

-subsys xxx:

The option -subsys set the windows subsystem to use (default=windows).

windows, win or gui => windows subsystem

console, con or char => console subsystem

native => native subsystem

posix => POSIX subsystem

-entry name

The option -entry specifies the program entry point (**first instruction to be executed**).

### **3. DEBUGGING:**

OLLYDBG.EXE module.exe

RUNNING:

module.exe

In other words Multi-module programming = building an executable file that is composed from several obj modules.

We will write several source files: `module1.asm`, `module2.asm` ... `module.asm`, compile them separately using the command:

```
nasm.exe -fobj module1.asm
```

.....

```
nasm.exe -fobj moduleN.asm
```

and link them together in an executable file with the command:

```
alink.exe -oPE -subsys console -entry start module1.obj module2.obj ...moduleN.obj
```

We will get one executable file: `module1.exe`.

One module will contain the main program and the other modules describe functions/procedures which are called from the main module.

Using the reserved word `global` we can export a symbol (variable or procedure) defined in the current module, in order to use it in another module; the other module will import the external symbol using the reserved word `import`.

**Obs: Constants/equ cannot be exported since they do not have a memory space.**

### Passing the parameters to a function/procedure defined in another module

There are three alternatives for this:

1. Parameters can be passed **using the registers**; the problem with this is the fact that there is a limited number of registers and some of them can be occupied with data (so they are not available)
2. Parameters can be passed to the function in the other module by **declaring** them **global**; the problem with this is that it breaks an old and important principle of programming: modularization (i.e. a program is better maintained if it is formed by independent modules linked together, e.g. functions, source files etc.) and everything becomes global (part to the same namespace which can cause name clashes – the same symbol is defined in different places); modularization is the reason we have functions with local variables in a program and not the whole code being written in a giant main body/function.
3. Parameters can be passed **using the stack** – this is the most powerful and flexible solution which is used by the majority of compiled programming languages.

Here is an example for each of the three mechanisms for passing the parameters described above.

*All examples are solving a simple problem, that of computing the expression:  $x = a + b$ .*

**Ex.1. Parameters are passed by the main module to the function in the other module **using registers**.**

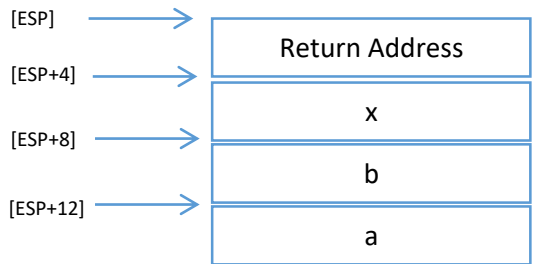
Module main.asm	Module function.asm
<pre> bits 32 global start extern exit import exit msvcrt.dll ; we import the 'addition' function from the ; function.asm module extern addition  segment data use32 class=data public     a db 2     b db 3     x db 0  segment code use32 class=code public start:     ; put the parameters in registers     mov bl, [a]     mov bh, [b]     ; call the function     call addition      ; result is in AL     mov [x], al      ; call exit(0)     push dword 0     call [exit]</pre>	<pre> bits 32 ; we export the 'addition' function in order to be ; used in the main module global addition  segment code use32 class=code public ; the code segment contains only the addition ; function addition:     ; the parameters are in: BL=a, BH=b     ; we will return the result in AL     mov al, bl     add al, bh      ; return from function     ; (it removes the Return Address from the stack     ; and jumps to the Return Address)     ret</pre>

Ex.2. Parameters are passed by the main module to the function in the other module using global variables.

Module main.asm	Module function.asm
<pre> bits 32 global start extern exit import exit msvcrt.dll ; we import the 'addition' function from the ; function.asm module extern addition  ; we export variables a, b and x in order to be used ; in the other module global a global b global x  segment data use32 class=data public</pre>	<pre> bits 32 ; we export the 'addition' function in order to be ; used in the main module global addition  ; import the a, b, x variables from the other module extern a, b, x  segment code use32 class=code public ; the code segment contains only the addition ; function addition:     ; the parameters are directly accessible in global</pre>

<pre> a db 2 b db 3 x db 0  segment code use32 class=code public start:     ; there is no need to do anything with the     ; parameters. They are already accessible to     the     ; other module (because they are global).      ; call the function     call addition      ; the result is already placed in x by the     addition     ; function      ; call exit(0)     push dword 0     call [exit] </pre>	<pre> ; variables a, b and x (which are global) mov al, [a] add al, [b] mov [x], al  ; return from function ; (it removes the Return Address from the stack ; and jumps to the Return Address) ret </pre>
--	---

Ex.3. Parameters are passed by the main module to the function in the other module **using the stack.**

Module main.asm	Module function.asm
<pre> bits 32 global start extern exit import exit msvcrt.dll ; we import the 'addition' function from the ; function.asm module <b>extern addition</b>  segment data use32 class=data public a db 2 b db 3 x db 0  segment code use32 class=code public start:     ; put the parameters a, b and x on the stack     ; we can not put bytes on the stack, we will put     ; dwords     mov eax, 0     mov al, [a]     push eax     mov al, [b]     push eax     mov al, [x]     push eax </pre>	<pre> bits 32 ; we export the 'addition' function in order to be ; used in the main module <b>global addition</b>  segment code use32 class=code public ; the code segment contains only the addition ; function addition:     ; the parameters are on the stack     ; the stack looks like this: </pre> <div style="text-align: center;">  </div> <pre> ; remember that a stack element is 4 bytes ; (dword) and the stack grows toward smaller ; addresses (meaning that the dword from the top </pre>

<pre> ; call the function call addition ; the result is in the dword from the top of the stack pop eax mov [x], al      ; x := a + b ; we still have to remove 2 dwords from the stack ; (the dwords corresponding to 'a' and 'b') add esp, 4*2 ; instead of the above instruction we could have ; used two 'pop eax' instructions  ; call exit(0) push dword 0 call [exit] </pre>	<pre> ; of the stack is placed at the smallest memory ; address). ; the Return Address was placed on the stack by ; the 'call addition' instruction in the main module.  mov eax, dword [esp+12] mov bl, al      ; bl = a mov eax, dword [esp+8] add bl, al      ; bl = a + b mov al, bl mov dword [esp+4], eax ; place a+b on the stack ; for the main module  ; return from function ; ('ret' removes the Return Address from the ; top of the stack and jumps to the Return Address) ret </pre>
--	--

**Working with multiple .asm files for solve a problem:**

**Problem 1.** Read two numbers a and b from the keyboard into the first file (mainsuma.asm) and then to compute the sum into the second file (secondsuma.asm)

**Step1:** Create a folder containing ALINK.exe si NASM.exe (both files nasm and alink are in nasm folder from asm tools folder)

Also both .asm files should be in the same folder with the alink and nasm application files

**Step 2:** Solve problems using stack as a method to parse the parameters from the main .asm file into second .asm file

mainsuma.asm – main .asm file contain the next sequence of code lines:

bits 32

global start

import printf msvcrt.dll

import exit msvcrt.dll

import scanf msvcrt.dll

extern printf, exit, scanf

extern calculsuma ; declarare the label using extern directive

```
format_afisare db "suma=%d", 10, 0
```

```
mesaja db "introduceti a=", 10, 0
```

```
mesajb db "introduceti b=", 10, 0
```

```
format db "%d", 0
```

```
a resd 1
```

```
b resd 1
```

```
segment code use32 public code
```

```
start:
```

```
;in main we will read both variables a and b
```

```
;display an input message on the screen for variable a
```

```
push dword mesaja
```

```
call [printf]
```

```
add esp, 4
```

```
;read a
```

```
push dword a
```

```
push dword format
```

```
call [scanf]
```

```
add esp, 4*2
```

```
;display an input message on the screen for variable b
```

```
push dword mesajb
```

```
call [printf]
```

```
add esp, 4
```

```
;read b
```

```
push dword b
```

```
push dword format
```

```
call [scanf]
```

```
add esp, 4*2
```

```
;both numbers will be sent in second file using the stack
```

```
push dword [a] ;save on the stack variable a
```

```
push dword [b] ;save on the stack variable b
```

```
call calculsuma ; call the subprogram calculsuma from the second .asm file
```

Now the stack contains the following:

Adresa de retur	Esp+0
[b]	Esp+4
[a]	Esp+8

; into the main file we also print the result

```
push ebx
```

```
push format_afisare
```



```

    call [printf]
    add esp, 2*4
    push dword 0
    call [exit]

```

The file secondsuma.asm contains the code:

```

bits 32
segment code use32 public code
global calculsuma ; we define a global label for calculsuma
calculsuma:

```

```

    mov eax, [esp + 4] ;we take the first element from the stack - b

```

```

    mov ebx, [esp + 8] ; we take the second element from the stack - a

```

```

    add ebx, eax ; calcul

```

```

ret 4*2 ; in this case 8 represent also the number of bytes that should be taken out from the stack

```

After the code is written we wil start cmd (start -> cmd)

In cmd, we will set the path to the our working directory (we can use Change Directory – cd command)

```

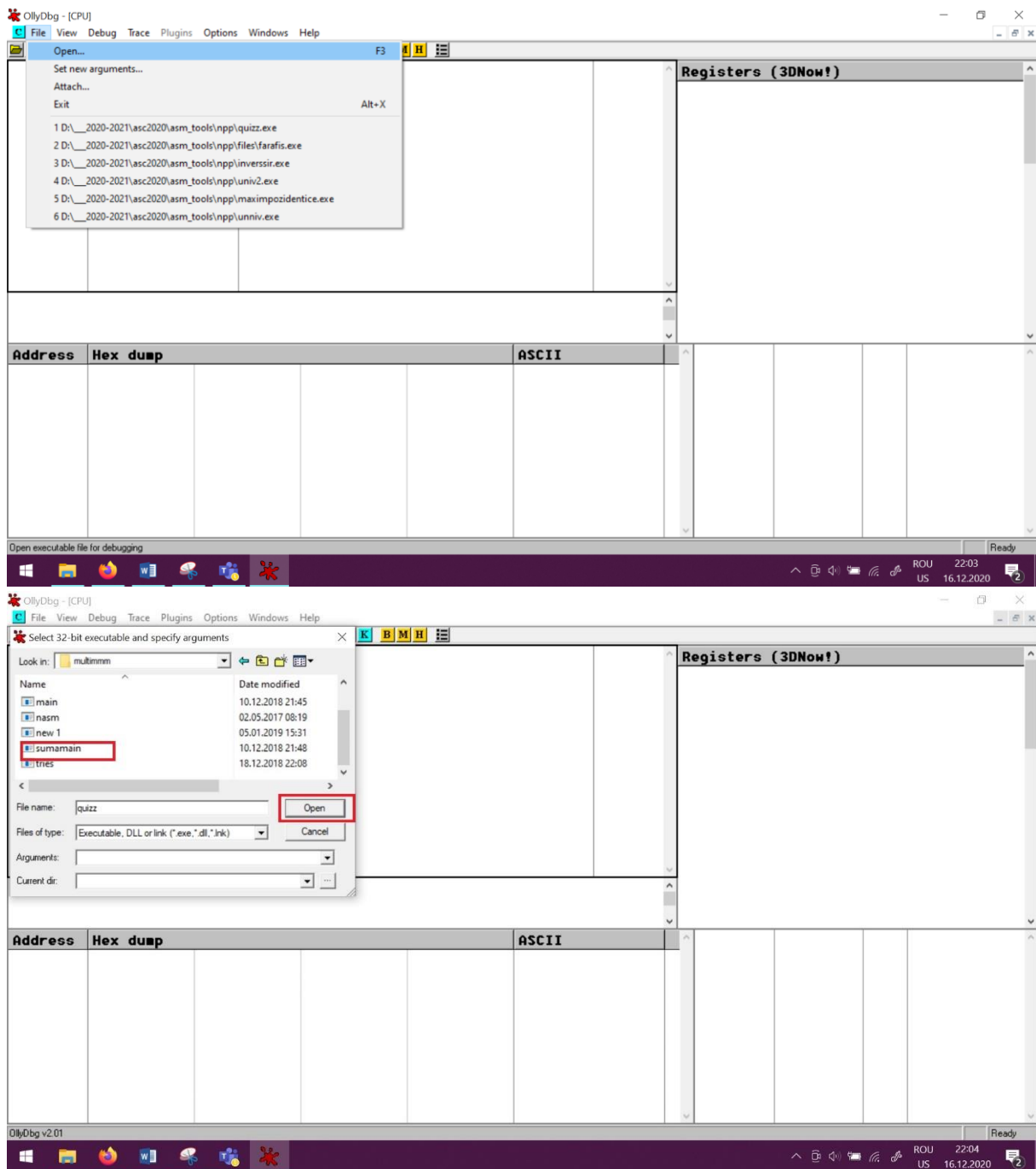
D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>nasm -f obj mainsuma.asm
D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>nasm -f obj secondsuma.asm
D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>alink mainsuma.obj secondsuma.obj -oPE -subsys console -entry start
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved

Loading file mainsuma.obj
Loading file secondsuma.obj
matched Externs
matched ComDefs
Generating PE file mainsuma.exe

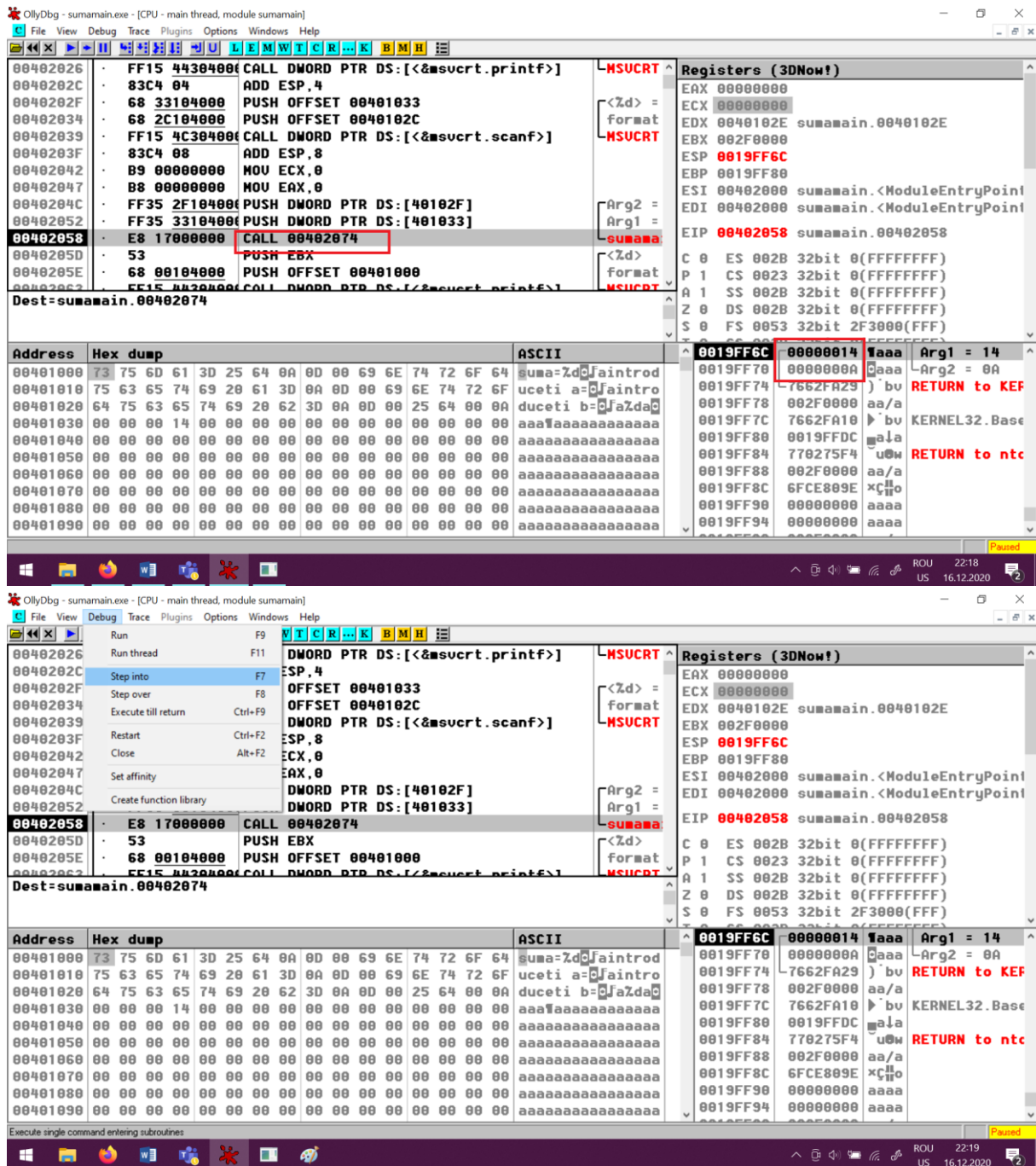
D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>mainsuma.exe
introduceti a=
2
introduceti b=
4
suma=6

```

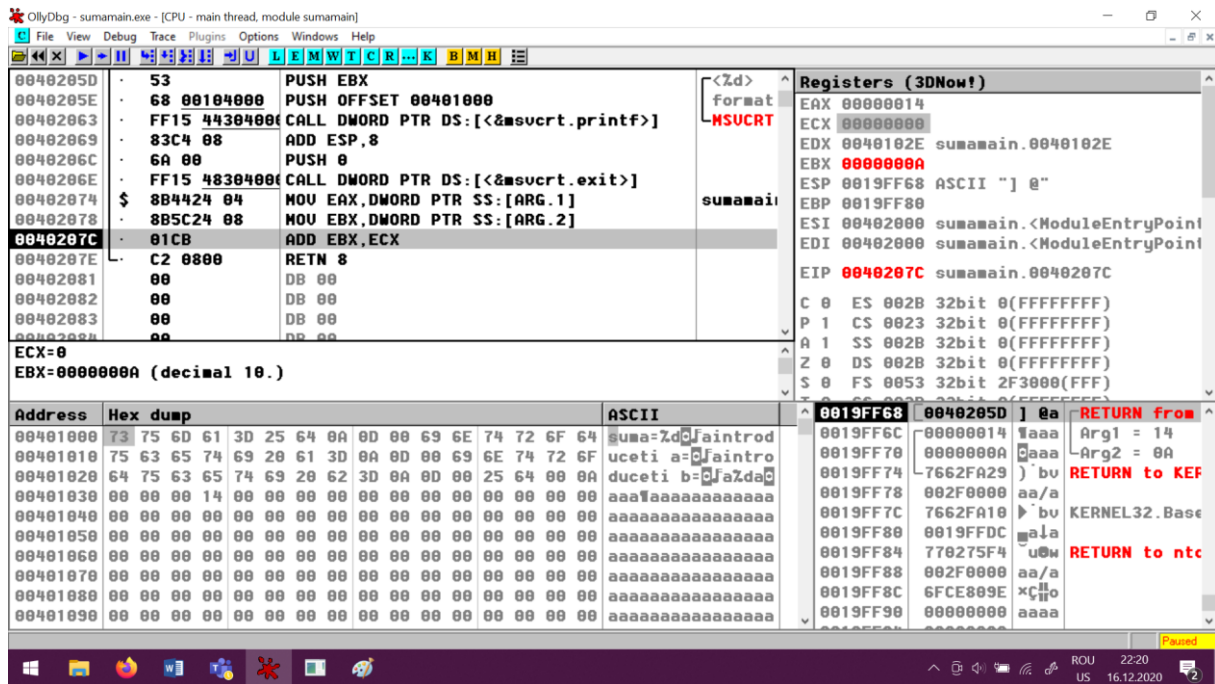
If we want to open the debugger, we have to open the ollydbg.exe (from ollydbg folder from asm tools) and the open file mainsuma.exe:



We execute the code line by line using F8 and when we are on the line with the call for the function from the second file we execute the code using F7 (step into):



After F7 we can see the content from the second file:



## Problem 2. Concatenate two strings s1 and s2. Both strings s1 and s2 are defined in data segment.

In first file sirurimain.asm we just parse the strings and the lengths in the second file using the stack and also we will print the result.

In second file secondmain.asm we will concatenate the strings.

sirurimain.asm has the code:

bits 32

global start

extern exit, printf

extern concatenate

import printf msvcrt.dll

import exit msvcrt.dll

segment data use32 class=data public

s1 db 'abcdef'

len1 equ \$-s1

s2 db '1234'

len2 equ \$-s2

s3 times len1+len2+1 db 0

segment code use32 class=code public

start:

; we place all the parameters on the stack

push dword len1

push dword len2

```

push dword s3
push dword s2
push dword s1

```

```
call concatenare
```

```

push dword s3
call [printf]
add esp, 4*1

```

```

push dword 0
call [exit]

```

Structura stivei:

Adresa de retur	Esp+0
S1	Esp+4
S2	Esp+8
S3	Esp+12
Len2	Esp+16
Len1	Esp+20

sirurisecond.asm

```

bits 32
segment code use32 class=code public
global concatenare ; export concatenare
concatenare:

```

```

mov esi, [esp+4] ;ESI = the offset of the source string (s1)
mov edi, [esp+12] ;EDI = the offset of the destination string(s3)
mov ecx, [esp+20] ; ECX = len1
cld
repeta:
    lodsb
    stosb
loop repeta

```

```

mov esi, [esp+8] ;s2
mov ecx, [esp+16] ; len2

```

```
repeta2:
    lodsb
    stosb
loop repeta2
```

```
ret 4*5
```

```
D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>sirurimain.exe
abcd1234
D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>nasm -f obj sirurimain.asm
D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>nasm -f obj sirurisecond.asm
D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>alink sirurimain.obj sirurisecond.obj -oPE -subsys console -entry start
ALINK v1.6 (C) Copyright 1998-9 Anthony A.J. Williams.
All Rights Reserved

Loading file sirurimain.obj
Loading file sirurisecond.obj
matched Externs
matched ComDefs
Generating PE file sirurimain.exe

D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>sirurimain.exe
abcdef1234
D:\_2018 didactic\ASM_tools\asm_tools_adriana 2018\npp\multimmm\exemplu>
```