Interfacing with high-level languages: ASM + C multi-module programming

Calling a procedure (subprogram) defined in C based on calling conventions.
A call convention gives answers to the following questions:

- How do we pass parameters to subroutines?
  - Which types of parameters can we pass? and in what order?
  - How many parameters? Any number of parameters?
- What resources are volatile (may be altered by the callee = programul apelat)?
- Where is the result stored?
- What clean-up actions are required after the call?
- Who is responsible to make them?
- Conventions
  - Decided (and documented) by callee, not by caller = programul apelant!
  - Commonly used: **CDECL**, **STDCALL**
  - Less commonly used or obsolete: PASCAL, FORTRAN, SYSCALL, etc.

1. CDECL convention

- Specific to the C programming language
- How do we pass parameters to subroutines? By pushing them on the stack
  - Which types of parameters can we pass? **Any type, but extended at least to DWORD**
  - In what order? From right to the left, that is in the reverse order of declaration
  - How many parameters? Any? Yes, in C is allowed functions with any parameters (ex: printf)
  - What resources are volatile? EAX, ECX, EDX, Eflags (*just flags)
  - Where is the result stored? EAX, EDX:EAX
  - What clean-up actions are required? Freeing up the arguments
  - Who is responsible? The caller! (Programul Apelant)

Parameters			Volatile resources	Results	Cleanup
Storage	**Order**	**Number**			
Stack	Reverse	***Any***	EAX, ECX, EDX, Flags*	**EAX** / EDX:EAX	***Caller***

2. STDCALL convention

- Specific to Windows operating system
  - Also called **WINAPI**
  - Used by Windows system libraries
- Very similar to the CDECL convention
- Differences:
  - *A fixed number of parameters*
  - *The cleanup is performed by callee (Pogramul Apelat)*

Parameters		Results	Cleanup

Storage	Order	Number	Volatile resources		
Stack	Reverse	***Fixed***	EAX, ECX, EDX, Flags*	EAX / EDX:EAX	***Callee***

For calling a procedure defined in C it is MANDATORY to follow these stages:

1. **Entry code**: entering the procedure and preparing its execution
2. **Call code**: preparing and performing the call
3. **Return/exit code**: return from the procedure and free resources that are not necessary anymore

The modern compilers generate AUTOMATICALLY the corresponding code for each stage, but when we call a procedure defined in the C language from a procedure defined in assembly, it is MANDATORY that we write the corresponding code for each stage.

1. Entry code

Purpose: entering the procedure and preparing for execution

Tasks:

- **Create a stack frame**
- Reserve on the stack memory required for storing local variables
- Save on the stack a copy of the modified non-volatile resources

A stack frame is a data structure stored on the stack and it may contain:

- The parameters pushed by the calling program
- The return address (to the instruction that is after the call instruction)
- Copies of the non-volatile resources used by the procedure
- Local variables

Example:

```
; Entry code:
; ...
; - create the stack frame
push ebp
mov ebp, esp
```

2. Call code

Purpose: prepare and perform the call

Tasks:

- Save volatile resources that are used (push EAX, push ECX, pushad, ...)
- Ensure/enforce constrains (ESP is aligned, DF = 0 etc.)

- Prepare arguments sent to the procedure (place on the stack/in registers based on the call convention)
- Perform the call:
  - **call _procedure** - if the procedure will be linked statically (from C imported)
  - **call [procedure]** - if the procedure will be linked dynamically (from msvcrt.dll imported)

Example:

```
; Calling code
; ----------------------------------------------------
; Storing the volatile resources being in use;
; push eax, push ecx, pushad, ...

; - compliance with established constraints
; this is not the case here

; - prepare the arguments for the called procedure (pushing them on the stack)
```

3. Exit/return code

Purpose: return from the procedure and free the unnecessary resources.

Tasks:

- Restoring nonvolatile altered resources;
- Removing local variables of the function;
- Destroying the stack frame;
- Returning to the calling code and removing the parameters.

Example:

```
; restoration of the stack frame of the calling program
mov esp, ebp
pop ebp

;   return from the procedure without freeing the space for the parameters
; (it is the caller's responsibility to do so)
ret
```

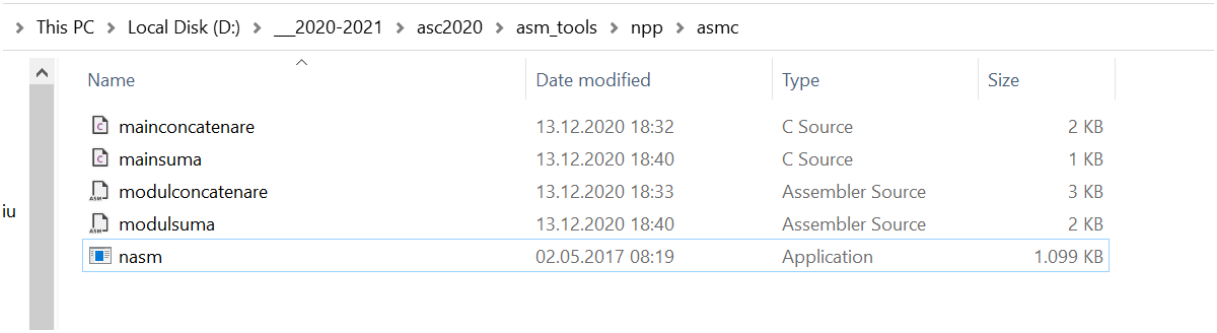
Example:

Read 2 numbers a and b (in .c file). Compute the sum of the numbers (in .asm file) and print the result in the .c file.

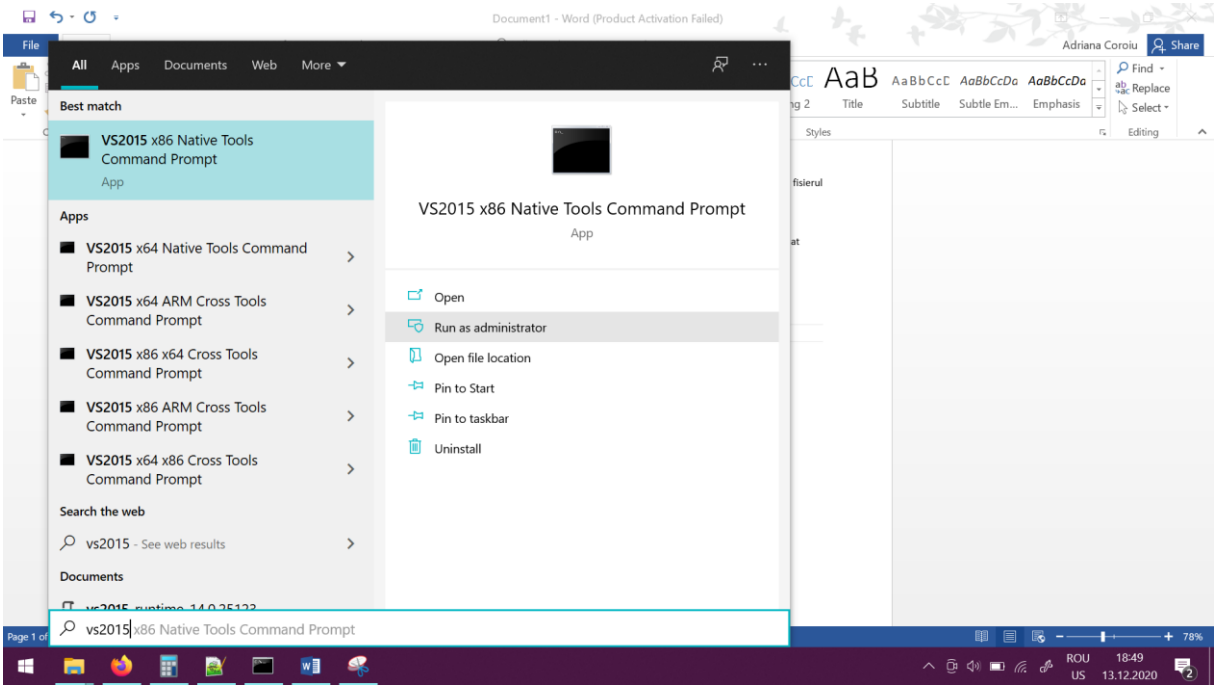
```c
/*++
Se cere un program C care apeleaza functia sumaNumere scrisa in limbaj de asamblare.
Aceasta functie primeste
ca parametri doua numere naturale citite in programul C, calculeaza suma lor si
transmite aceasta valoare ca rezultat.
Programul C va afisa suma calculata de functia sumaNumere
--*/

#include <stdio.h>

// functia declarata in fisierul modulSumaNumere.asm
int sumaNumere(int a, int b);

int main()
{
    // declaram variabilele
    int a = 0;
    int b = 0;
    int sum = 0;

    // citim de la tastatura cele doua numere
    printf("a=");
    scanf("%d", &a);

    printf("b=");
    scanf("%d", &b);

    // apelam functia scrisa in limbaj de asamblare
    sum = sumaNumere(a, b);

    // afisam valoarea calculata de functie
    printf("Suma numerelor este %d", sum);
    return 0;
}
```

```asm
bits 32

; informam asamblorul ca dorim ca functia _sumaNumere sa fie disponibila altor unitati de compilare
global _sumaNumere

; linkeditorul poate folosi segmentul public de date si pentru date din afara
segment data public data use32

; codul scris in asamblare este dispus intr-un segment public, posibil a fi partajat cu alt cod extern
segment code public code use32

; int sumaNumere(int, int)
; conventie cdecl
_sumaNumere:
    ; creare cadru de stiva pentru programul apelat
    push ebp
    mov ebp, esp

    ; obtinem argumentele transmise pe stiva functiei sumaNumere
    ; la locatia [ebp+4] se afla adresa de return (valoarea din EIP la momentul apelului)
    ; la locatia [ebp] se afla valoarea ebp pentru apelant
    mov eax, [ebp + 8]        ; eax <- a

    mov ebx, [ebp + 12]        ; ebx <- b

    add eax, ebx            ; calculam suma
                            ; valoarea de rezulat a functiei este in eax

    ; refacem cadrul de stiva pentru programul apelant
    mov esp, ebp
    pop ebp

    ret
    ; conventie cdecl - este responsabilitatea programului apelant sa elibereze parametrii transmisi

```

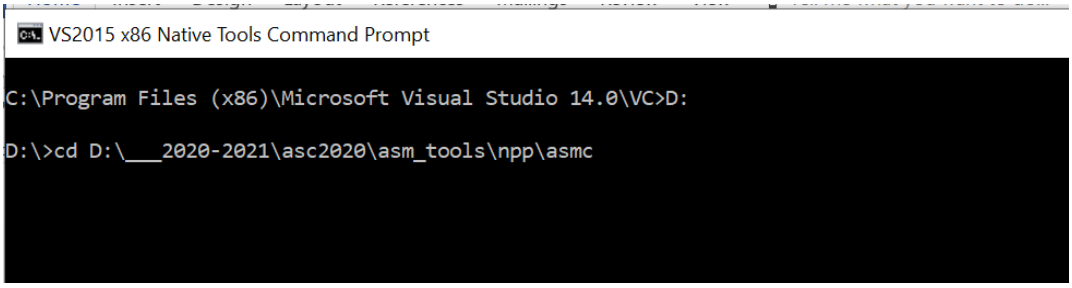
In order to execute the examples, we save the both files .asm and .c into the same folder with the nasm.exe:



Then open the Command Line (for instance cmd from from Visual Studio 2015 – or other variants containing C modules): Start - > vs2015 (atentie sa fie x86)

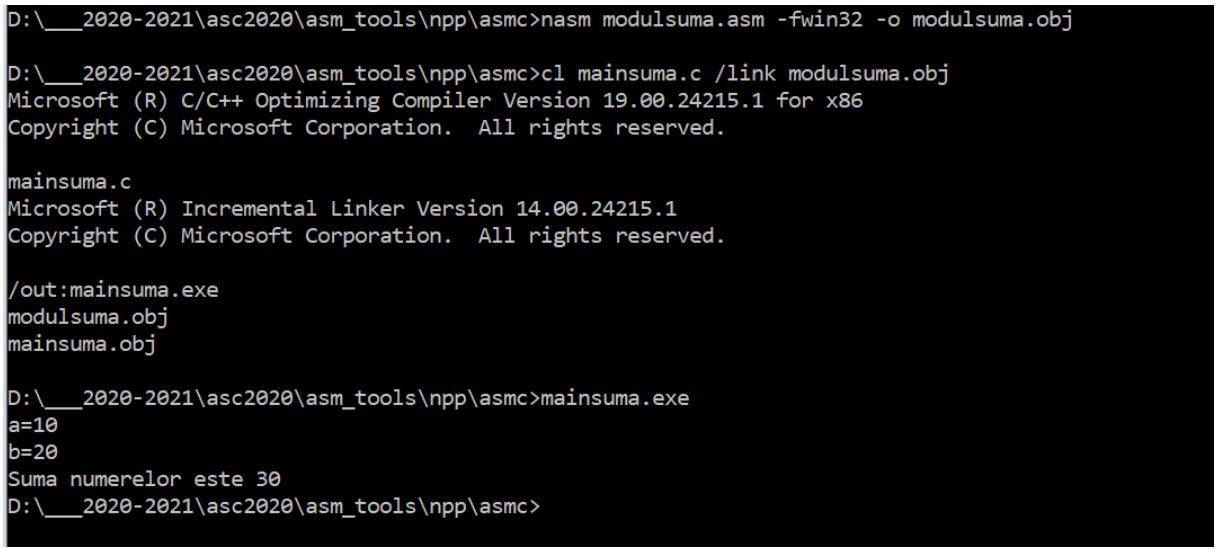


Now, in this cmd, we will open the folder with the codes (we can use command Change Directory CD and the path to the folder):



Now the commands for converting the asm into obj and to link the modules:

1. nasm modulsuma.asm -fwin32 -o modulsuma.obj
2. cl mainsuma.c /link modulsuma.obj
3. mainsuma.exe



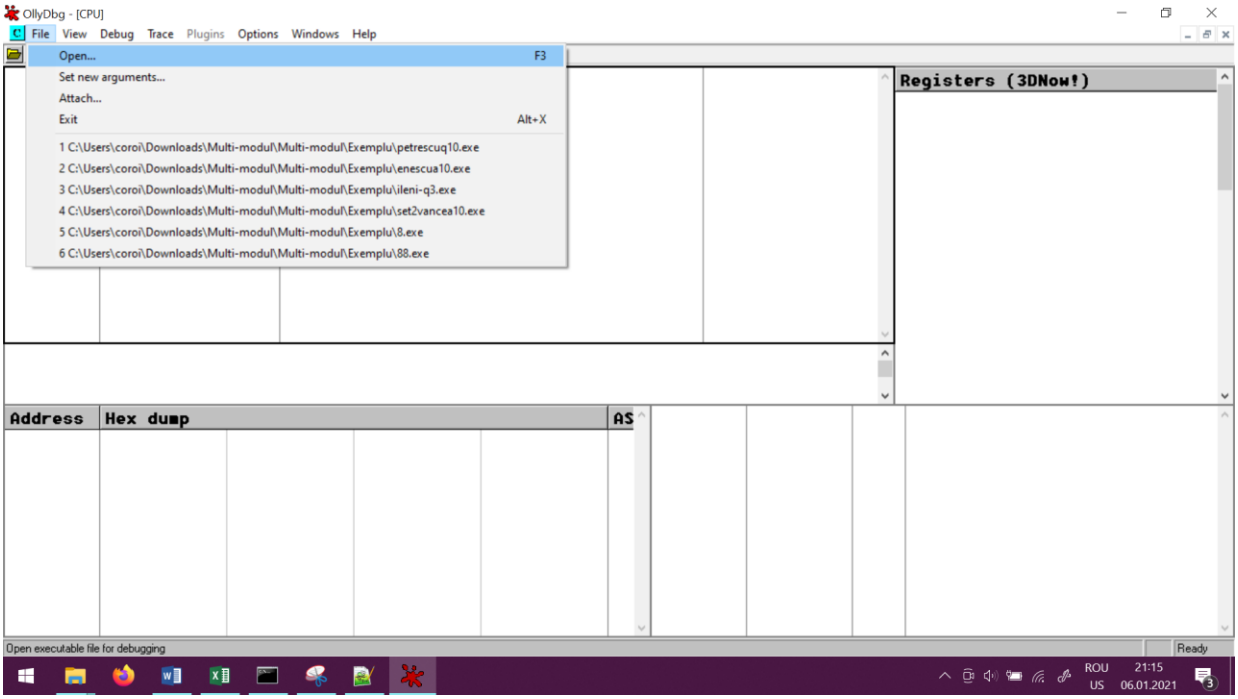
If we want to check the code into the debugger, the commands are:

nasm modulsuma.asm -fwin32 -g -o modulsuma.obj

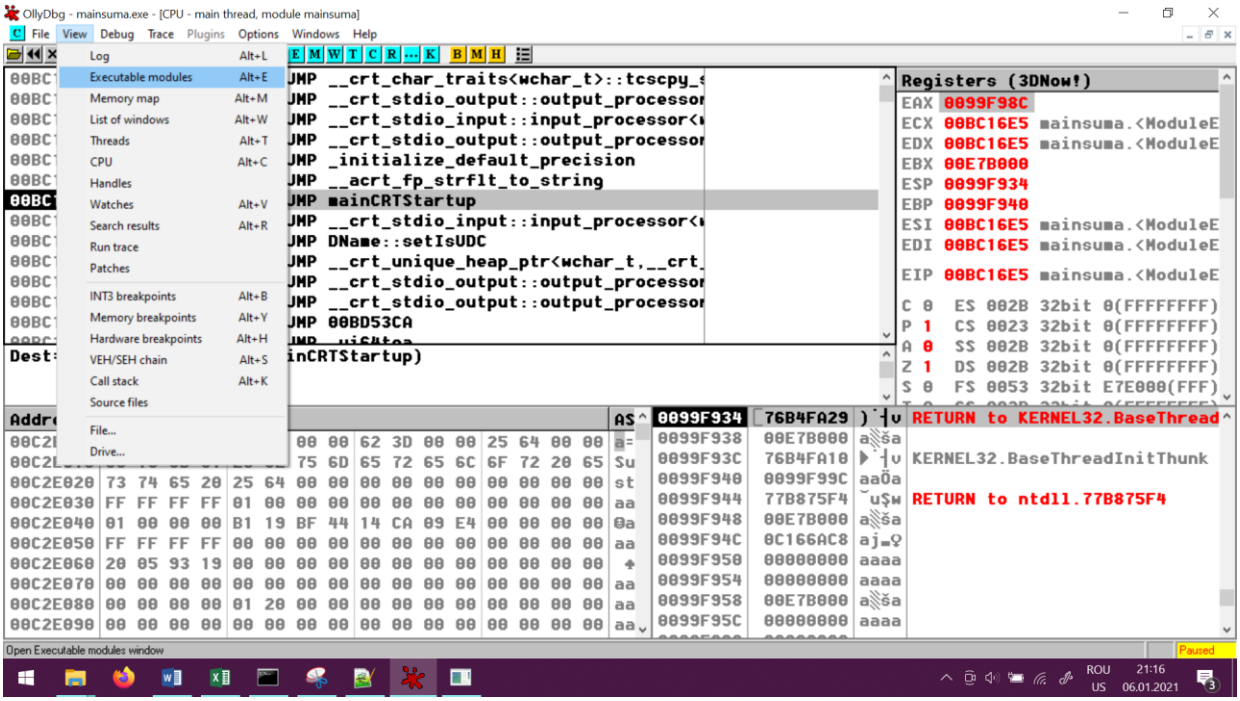
cl **/Z7** mainsuma.c /link modulsuma.obj

Then we open the Olydbg from file

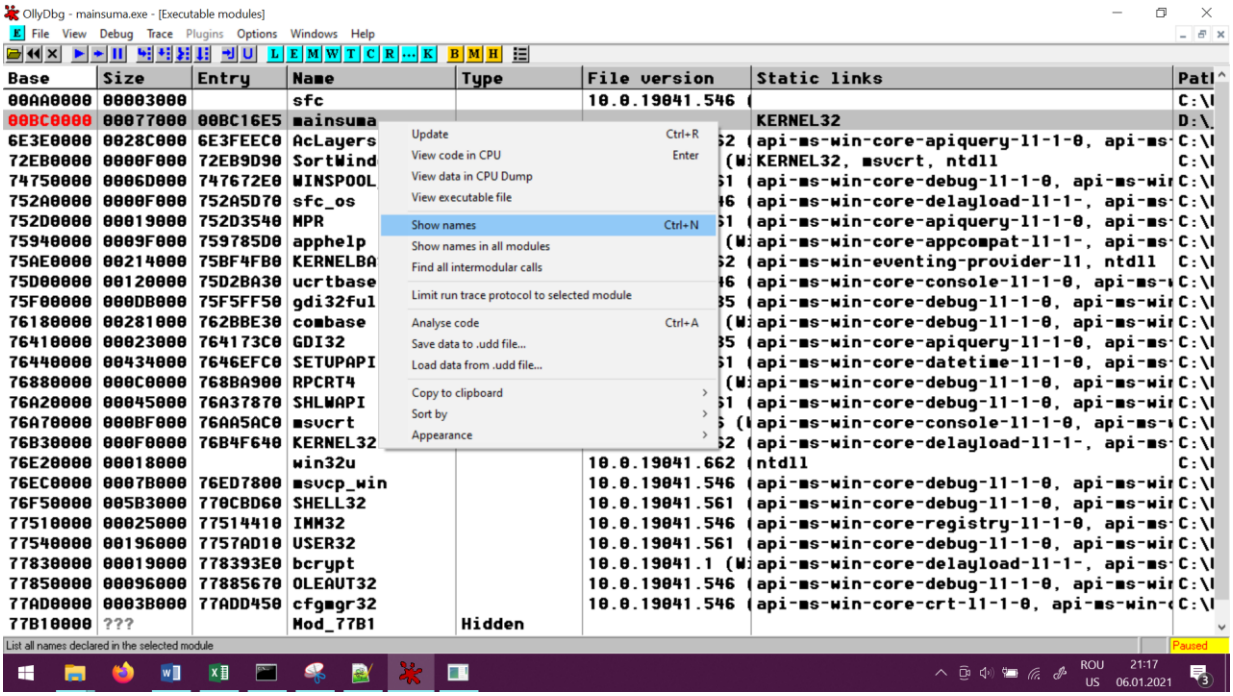
File -> Open



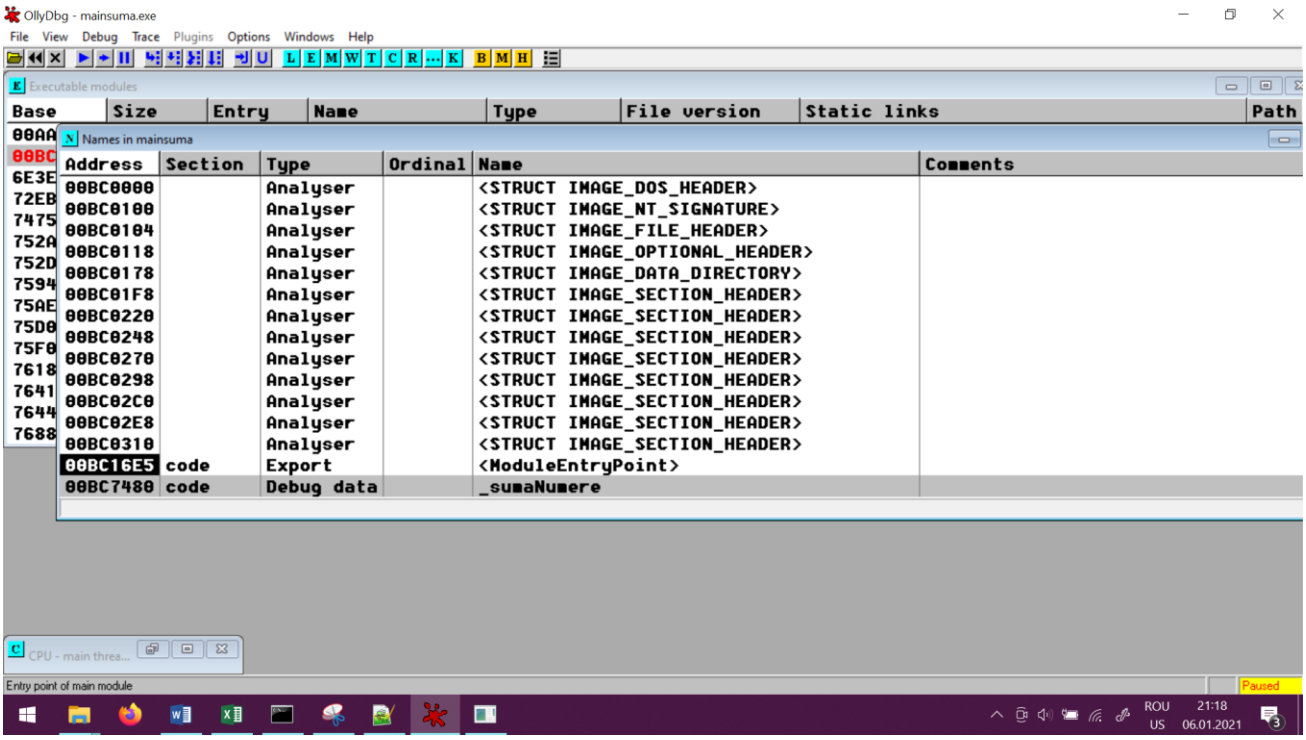
select View -> Executable modules



Right click on the name of the function and select -> Show names



Then double click on the name of the function in asm:



Now we can check the code step-by-step: