Seminar VI. Multi-module programming in assembly

- Multi-module programming = building an executable file that is composed from several obj modules.
- You 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
  You will obtain 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.
- At the lab you will only write 2-module programs (one module containing the main program and the other containing a function that is 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 ***extern***.
  Obs: Constants/equ can not 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:

- 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)
- 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.
- 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.

Below we will give an example for each of the three mechanisms for passing the parameters described above, all examples 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
bits 32 **global start** **extern exit**	**bits 32** **; we export the ‘addition’ function in order to be** **; used in the main module**

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]
	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

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

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
    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
	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
	    ; 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

```
; 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]
```

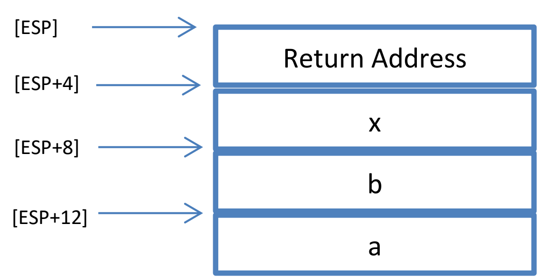
Ex.3. Parameters are passed by the main module to the function in the other module using the stack.

Module main.asm

```
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 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

    ; 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
```

Module function.asm

```
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 on the stack
    ; the stack looks like this:
```




```
    ; remember that a stack element is 4 bytes
    ; (dword) and the stack grows toward smaller
    ; addresses (meaning that the dword from the top
    ; 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
```

; call exit(0) **push dword 0** **call [exit]**	**; return from function** **; ('ret' removes the Return Address from the** **; top of the stack and jumps to the Return Address)** **ret**

Ex. 4. Write a program that concatenates 2 strings by calling a function from another module and then prints the resulted string on the screen.

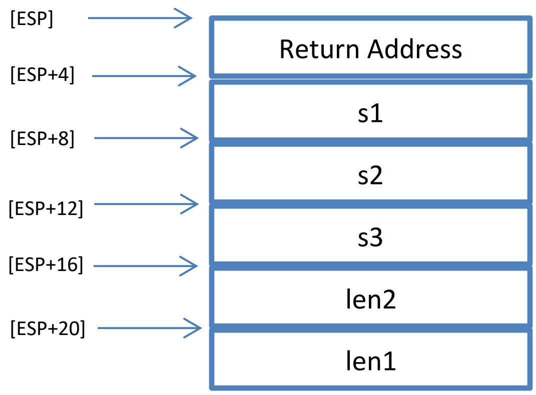
Main.asm module:

```
bits 32
global start
extern exit, printf
extern concatenare          ; import 'concatenare' from the other module
import printf msvcrt.dll
import exit msvcrt.dll


segment data use32 class=data public
    s1 db 'abcd'
    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
    add esp, 4*5

    push dword s3
    call [printf]

    push dword 0
    call [exit]
```

Function.asm module:

```
bits 32
global concatenare              ; export concatenare
```

```
segment code use32 class=code public
    concatenare:
        ; the stack looks like this:
```




```
        ; first copy s1 in s3
        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
        rep movsb               ; rep repeats movsb ECX times

        ; then, copy s2 at the end of s3
        mov esi, [esp+8]        ; ESI = the offset of the source string (s2)
                                ; EDI already contains the offset of the destination string
        mov ecx, [esp+16]       ; ECX = len2

        rep movsb               ; rep repeats movsb ECX times

        ret
```

Ex. 5. Write a multi-module programming which prints the value of AL in binary on the screen.

Main module:

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

extern printBinary

segment code use 32 class=code
```

```
start:
        ; call printBinary(integer AL)
        mov al, 11000111b
        push eax
        call printBinary
        add esp, 4*1

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

Secondary module:

```
bits 32
global printBinary
extern printf
import printf msvcrt.dll

segment data use 32
            format db "%c", 0
            savedECX dd 0

segment code use 32

printBinary:
    ; print the low byte from dword [esp+4] in binary on the screen
    mov eax, [esp+4]        ; take the parameter from the stack and store it in EAX
                            ; (we only need the least significant byte)

    ; We obtain first the binary digits of AL by continuously dividing AL
    ; to 2 and then the obtained quotient to 2 and so on until we get the
    ; quotient zero. We keep the remainders which are the digits of AL in
    ; base 2, but they are in reverse order.
    ; Example: assume we want to obtain the digits of 6 in base 2:
    ;      6 div 2 = 3, 6 mod 2 = 0
    ;      3 div 2 = 1, 3 mod 2 = 1
    ;      1 div 2 = 0, 1 mod 2 = 1
    ; The digits of 6 in base 2, in reverse order are: 011.
    ; The digits of 6 in base 2 in the correct order are: 110.
    ; Because we obtain the digits in the reverse order, we print them in the correct
    ; order by placing them on the stack and then poping them, one at a time,
    ; from the stack and printing them.
    mov ecx, 0          ; ECX stores the number of digits placed on the stack
    mov dx, 0
    mov bl, 2
    repeat:
```

```
                 mov ah, 0
                div bl          ; AH contains the remainder (the digit)
                mov dl, ah
                push dx         ; place the digit on the stack as a word; it would have been
                        ; better if we pushed a dword on the stack, but it works the same this way
                 inc ecx        ; increment the number of digits on the stack
                 cmp al, 0
                 ja repeat      ; when we get to quotient zero, we stop

mov eax, 0
; know all we have to do is pop digits from the stack and print them one at a time
popDigit:
        mov [savedECX], ecx         ; save ECX so that it is not modified inside the printf function
        pop ax                      ; pop the digit from the stack
        add al, '0'                 ; obtain the ASCII code of the digit
        ; call printf("%c", byte c)  - print the digit on the screen
        push eax                    ; even if we print a character (%c), we put a dword on the stack
                                    ; but only the least significant byte of this dword is used
        push dword format
        call [printf]
        add esp, 4*2
        mov ecx, [savedECX]         ; restore ECX
loop popDigit

ret
```