**Lecture 7**

**Function calls for printing on screen and reading from keyboard**

**(based on CDECL convention)**

* Even if the assembly language directly uses the hardware components of the system, there are frequently used pieces of code and writing them every single time would be impractical (for example, communicating with input/output devices, which often implies complex protocols).
* One of the roles of the operating system is **to abstract the hardware machine for the programmer**, providing multiple libraries of functions that can be called for certain frequent operations, such as printing the data in a certain format, finding a substring in a string or different mathematical functions.
* Using a functions **means transferring control to the procedure address, executing the code corresponding to the function, and returning to the instruction immediately after the call of the function:**

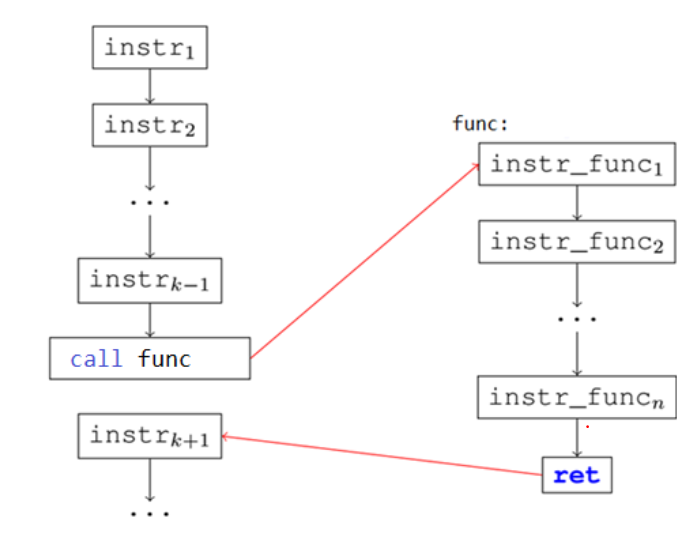


Figure 1

We are using the instruction CALL for extern functions calling.

*Syntax:*

**call [system\_function\_name]**

*Semantics:*

* CALL pushes the address of the instruction immediately after itself (return address) onto the stack and then transfers control to the procedure address (as we can see in Figure 1).
* This allows a return instruction to pop the return address and thus return control to the instruction immediately after the CALL instruction.

Usually functions have parameters and a return value. There are many conventions for passing parameters, but we will use the CDECL (which stands for C declaration) calling convention.

***The calling convention*** is not related to the syntax of the assembly language, but it is a ”contract” between the caller (the code from assembly) and the external function (the function written in C), specifying how parameters are passed and how the value is returned.

**CDECL convention rules**

* Arguments of an external function are placed on the stack from right to left (an element from a stack is a doubleword);
* The result returned by the external functions are stored in EAX register;
* Registers EAX, ECX, EDX are used within the functions (so they can be overwritten) so if we need the values stored in EAX, ECX, EDS we need to store them (in auxiliary variables for instances or using PUSHAD before using the external functions and POPAD after we execute the external functions) before the function call.
* The external functions don’t empty the stack; it is the responsibility of the programmer to pop the arguments out of the stack after the function call.

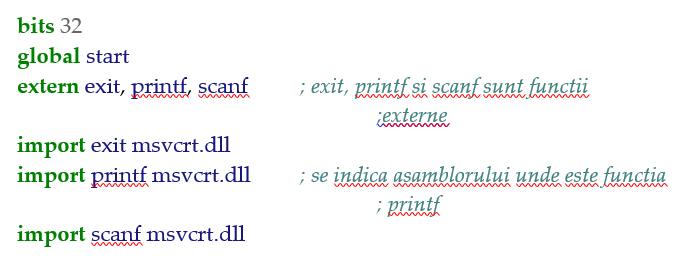
The pop operation from the stack for the parameters of the external functions is a **particular arithmetic operation** as a computation related to Extended Stack Pointer Register and the numbers of bytes that are on the stack:

|  |  |  |
| --- | --- | --- |
| push eax  **call [system\_function\_name]**  add esp, 4\*1 | push eax  push ebx  **call [system\_function\_name]**  add esp, 4\*2 | push eax  push ebx  push ecx  **call [system\_function\_name]**  add esp, 4\*3  push eax  push ebx  push ecx  push edx  call […]  add esp, 4\*4 |
| 4 (because 1dd = 4 bytes) \* numbers of doublewords saved on the stack | | |

The strings that create the formats or messages have to be strings ASCIIZ (last element of the string is 0).

We will use the external **functions printf and scanf.**

Both functions are from the library **msvcrt.dll** and before using them, we need to declare these functions and after that to import them.



**Printing on the screen**

For printing a text on the screen we will use the printf function, which requires a certain format.

Syntax of the printf function in a high level programming language is:

int printf (const char \* format, variable\_1, constant\_2, ...);

The first argument of the function is a string containing the printing format, followed by the same number of arguments (constant values or variable names) as specified in the format.

The character string representing the format can contain certain formatting, starting with the character %, which will be replaced by the values given in the following arguments.

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Type | Example | Value representation dimension |
| c | Character | a | byte |
| d | Signed decimal integer | 392 | dword |
| u | Unsigned decimal integer | 7235 | dword |
| x | Hexadecimal integer | 7fa | dword |
| s | String (terminated with a 0) | example | string of bytes terminated with 0 |

The printf function respects the cdecl convention and the steps from the cdecl convention are applied to the printf function.

Example:

|  |  |
| --- | --- |
| Printing a message on the screen | |
| In a high level programming language:  printf ("Now we have CSA lecture."); | Equivalent in assembly language:  segment data use32 class=data  text db ("Now we have CSA lecture."), 0 ; definining the message in data segment  segment code use32 class=code  push dword text ; pushing the parameter on the stack  call [printf] ; calling the printf function  add esp, 4 \* 1 ; cleaning the parameters from the stack |
| Printing a signed integer in base 10 | |
| In a high level programming language:  printf("%d", -17); | Equivalent in assembly language:  segment data use32 class=data  format db "%d", 0 ; definining the format  nr dd -17  segment code use32 class=code  push dword [nr] ; pushing the parameters on the stack from right to left  push dword format  call [printf] ; calling the printf function  add esp, 4 \* 2 ; cleaning the parameters from the stack |
| Printing an integer in base 16 | |
| In a high level programming language:  printf("%x", 0xAB); | Equivalent in assembly language:  segment data use32 class=data  format db "%x", 0 ; definining the format  segment code use32 class=code  push dword 0ABh ; pushing the parameters on the stack from right to left  push dword format  call [printf] ; calling the printf function  add esp, 4 \* 2 ; cleaning the parameters from the stack |
| Printing a message that contains an integer in base 10 | |
| In a high level programming language:  printf("It's week %d of school", n); | Equivalent in assembly language:  segment data use32 class=data  n dd 11  format db "It's week %d of school", 0 ; definining the format    segment code use32 class=code  push dword [n] ; pushing the value of n on the stack  push dword format  call [printf] ; calling the printf function  add esp, 4 \* 2 ; cleaning the parameters from the stack |
| Printing a message that contains more integers in base 10 | |
| In a high level programming language:  printf("It's semester %d, week %d of school.", 1, 9); | Equivalent in assembly language:  segment data use32 class=data  format db "It's semester %d, week %d of school.", 0 ; definining the format    segment code use32 class=code  push dword 9 ; pushing parameters on the stack  push dword 1  push dword format  call [printf] ; calling the printf function  add esp, 4 \* 3 ; cleaning the parameters from the stack |

**Reading from keyboard**

For reading input data from keyboard we use the scanf function.

Syntax of the scanf function in a high level programming language is:

int scanf (const char \* format, variable\_address\_1, ...);

The syntax of the scanf function is similar to the syntax of the printf function. The main difference is that its argument do not have to be constants or values of variables, but only addresses of variables, where the values read will be stored.

Example:

|  |  |
| --- | --- |
| Reading an integer in decimal and storing it in the variable n | |
| In a high level programming language:  scanf("%d", &n);    &n represent the address of the variable n where the function scanf stores the value read from keyboard | Equivalent in assembly language:  segment data use32 class=data  n dd 0 ; defining the variable n  format db "%d", 0 ; definining the format  segment code use32 class=code  push dword n ; pushing the parameters on the stack from right to left  push dword format  call [scanf] ; calling the scanf function for reading  add esp, 4 \* 2 ; cleaning the parameters from the stack |
| Reading two integers in decimal and storing the values in the variable n and m | |
| In a high level programming language:  scanf("%d %d", &n, &m ); | Equivalent in assembly language:  segment data use32 class=data  n dd 0 ; defining the variable n  m dd 0  format2 db "%d %d", 0 ; definining the format  segment code use32 class=code  push dword m ; pushing the parameters on the stack from right to left  push dword n  push dword format2  call [scanf] ; calling the scanf function for reading  add esp, 4 \* 3 ; cleaning the parameters from the stack |
| Reading three integers in decimal and storing the values in the variable a, b and c | |
| In a high level programming language:  scanf("%d %d %d", &a, &b, &c ); | Equivalent in assembly language:  segment data use32 class=data  a dd 0 ; defining the variable n  b dd 0  c dd 0  format3 db "%d %d %d", 0 ; definining the format  segment code use32 class=code  push dword c ; pushing the parameters on the stack from right to left  push dword b  push dword a  push dword format3  call [scanf] ; calling the scanf function for reading  add esp, 4 \* 4 ; cleaning the parameters from the stack |

If we need to create an interactive mechansism of reading and writing, we need to combine the both functions printf and scanf.

Write a solution to read 2 numbers a and b. Compute the sum of the numbers a and b and print the result on the screen.

**bits** 32

**global** start

**extern** exit, printf, scanf

**import** exit msvcrt.dll

**import** printf msvcrt.dll

**import** scanf msvcrt.dll

**segment** data use32 class=data

a dd 0

b dd 0

result dd 0

format1 db ‘Insert a value for a=’, 0 *; format este un sir C*

format2 db ‘Insert a value for b=’, 0

readformat db ‘%d’, 0

printformat db ‘Rezultat: %d + %d = %d’, 0

**segment** code use32 class=code

start:

*; apel printf(“a=”)*

push dword format1

call [printf]

add esp, 4\*1

*; apel scanf(“%d”, a)*

push dword a *; se pune pe stiva offset-ul variabilei!!*

push dword readformat

call [scanf]

add esp, 4\*2

*; apel printf(“b=”)*

push dword format2

call [printf]

add esp, 4\*1

*; apel scanf(“%d”, b)*

push dword b *; se pune pe stiva offset-ul variabilei!!*

push dword readformat

call [scanf]

add esp, 4\*2

mov eax, [a]

add eax, [b]

mov [result], eax

*; apel printf(“%d + %d = %d\n”, a, b, result)*

push dword [result] *; pune pe stiva valoarea rezultatului*

push dword [b] *; pune pe stiva valoarea lui b*

push dword [a] *; pune pe stiva valoarea lui a*

push dword printformat

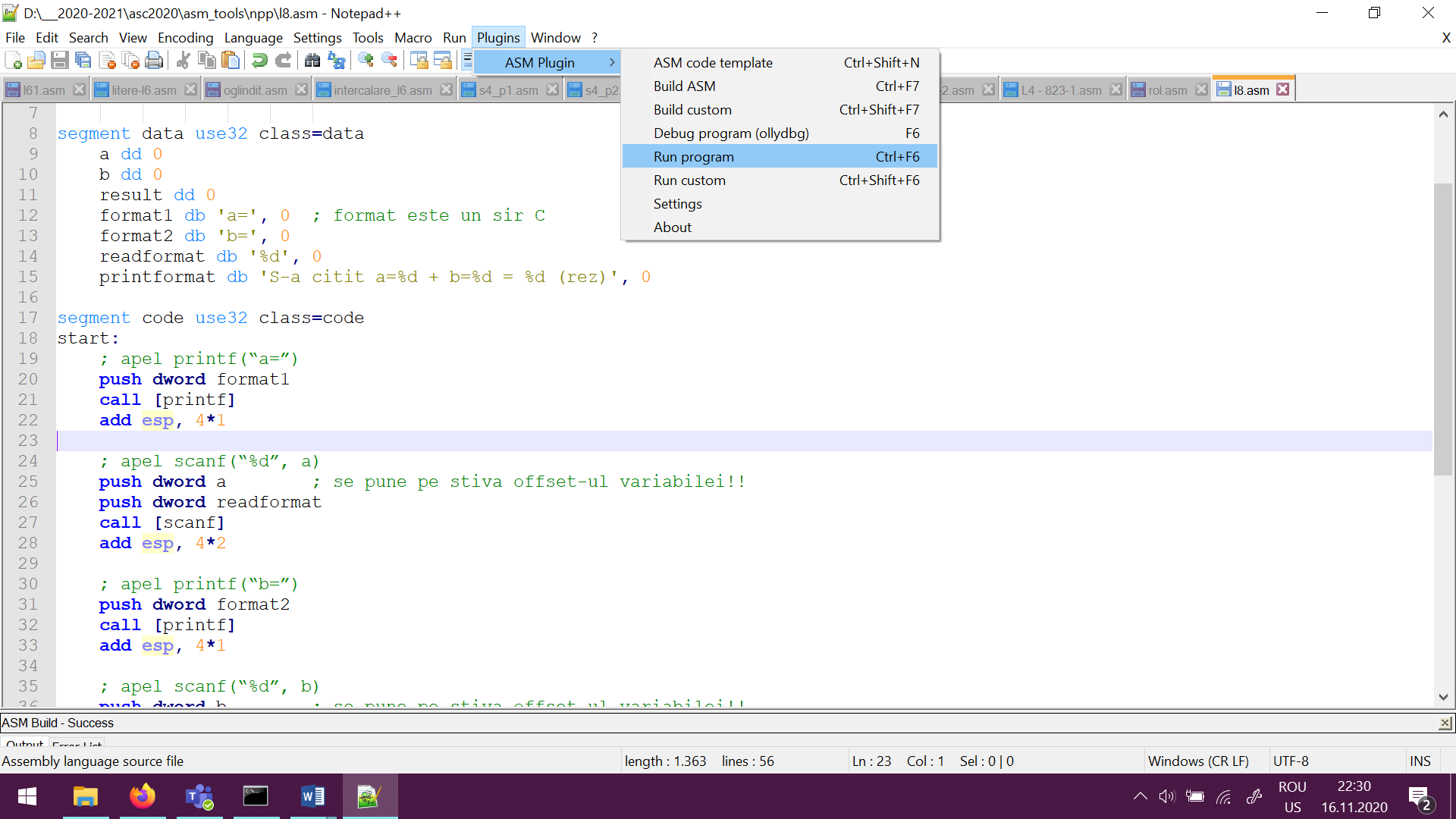
call [printf]

add esp,4\*4

push dword 0

call [exit]

To execute the code we select RUN Program from ASM Plugins.



We cannot use in assembly \n (new line) as in others languages, therefore we need to find a solution: the ascii code for newline:



ex:

data segment

newline db 10, 0

code segment

push dword newline

call [printf]

add esp, 4\*1

Other useful functions:

gets (string) – allow us to read a string from the keyboard with multiple spaces:

bits 32 ; assembling for the 32 bits architecture

; 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

; msvcrt.dll contains exit, printf and all the other important C-runtime specific functions

extern exit**,** printf**,** gets

import exit msvcrt.dll

import printf msvcrt.dll

import gets msvcrt.dll

; our data is declared here (the variables needed by our program)

segment data use32 class**=**data

s resb 20

format\_sir db '%s'**,** 0

mesaj db 'S-a citit acest sir:'**,** 0

; our code starts here

segment code use32 class**=**code

start**:**

**push** **dword** s

**call** **[**gets**]**

**add** **esp,** 4**\***1

;print pe ecran mesajul

**push** **dword** mesaj

**call** **[**printf**]**

**add** **esp,** 4**\***1

; print pe ecran sirul citit

**push** **dword** s

**push** **dword** format\_sir

**call** **[**printf**]**

**add** **esp,** 4**\***2

; exit(0)

**push** **dword** 0 ; push the parameter for exit onto the stack

**call** **[**exit**]** ; call exit to terminate the program