Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 4:

Elaborat:
st. gr. FAF-221 Cuzmin Simion

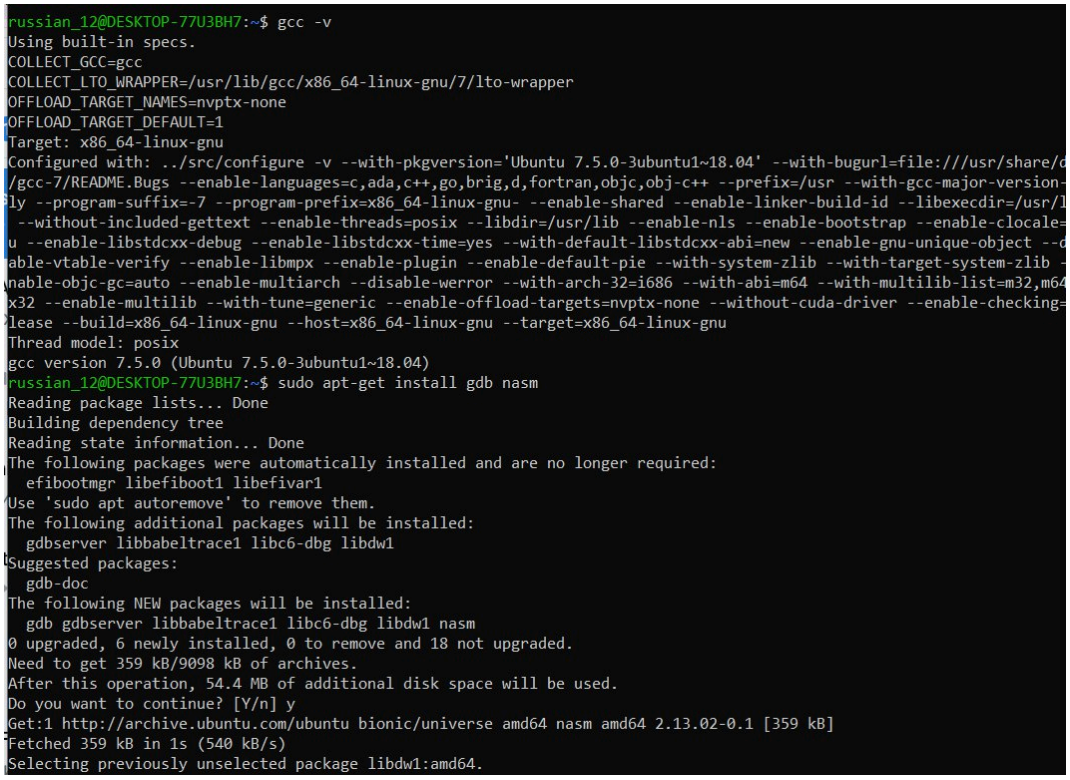
Verificat:

asist. Univ. Voitcovschi Vladislav

Introduction:

In the realm of software development, particularly at the intersection of hardware interaction and performance optimization, assembly language holds a pivotal role. Among the tools available for programming in assembly, the Netwide Assembler (NASM) stands out for its versatility, efficiency, and wide acceptance. NASM is an essential tool for developers looking to harness the full potential of the Intel x86 architecture, offering a rich set of features designed to facilitate the development of complex software systems. Developed initially in 1996 by Simon Tatham and Julian Hall, NASM was conceived out of a necessity for a high-quality assembler that could cater to the needs of both novice and experienced programmers. Over the years, it has evolved, supported by a vibrant community of developers, to include a robust set of functionalities that make it the assembler of choice for a broad spectrum of applications in the fields of operating system development, embedded systems, and high-performance computing, to name a few.

Installation of gdb and nasm :



Exercise 1:

Firstly, I implemented a simple hello-world to console in windows environment. I installed nasm and the linker GoLink. Using this command i created the object:

nasm -f win32 helloworld.asm -o helloworld.obj

And after the .exe:

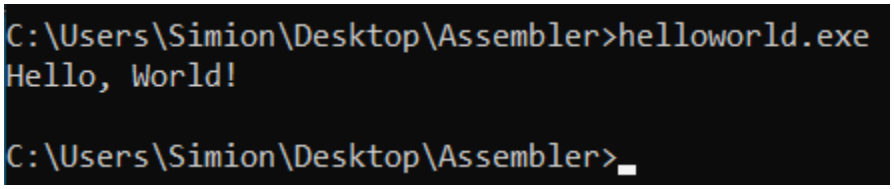
golink /entry:Start /console kernel32.dll user32.dll helloworld.obj

The code:

```
NULL EQU 0
STD_OUTPUT_HANDLE EQU -11

extern _GetStdHandle@4
extern _WriteFile@20
extern _ExitProcess@4

global Start

section .data

    Message db "Hello, World!", 0Dh, 0Ah
    MessageLength EQU $-Message

section .bss

    StandardHandle resd 1
    Written resd 1

section .text

    Start:
        push STD_OUTPUT_HANDLE
        call _GetStdHandle@4
        mov dword [StandardHandle], EAX
```

```
    push NULL
    push Written
    push MessageLength
    push Message
    push dword [StandardHandle]
    call _WriteFile@20

    push NULL
    call _ExitProcess@4
```

Result:



This NASM code is a minimal Windows assembly program that prints "Hello, World!" to the console and exits. It defines constants for null pointers and the standard output handle, then declares external references to Windows API functions for handling console output and process termination. In the .data section, it defines the message and its length. The .bss section reserves space for variables to store the standard output handle and the number of bytes written. The .text section contains the program's logic: it retrieves a handle to the standard output, writes the message to it, and then terminates the process. The code demonstrates basic assembly language features, system call usage, and interaction with the Windows API for output and process control.

Exercise 2:

For the next 2 exercises I met some problems and that's why I decided to install a Windows Subsystem for Linux (WSL) and run the code there. First exercise is an addition of 2 numbers problems.

The code:

```
section .data

section .bss
    x_buffer: resd 1
    x: resd 8
    y: resd 8
    p: resd 1
    cnt: resd 1
    ok: resd 1
    zero: resd 1

section .text
    global _start
_start:

    xor eax, eax

    mov [x], eax
    mov [cnt], eax
```

```
    mov [ok], eax

    mov eax, 10
    mov [p], eax

    mov eax, 0x30
    mov [zero], eax

et_citire:
    xor edx, edx
    mov [x_buffer], edx

    mov eax, 0x3
    xor ebx, ebx
    mov ecx, x_buffer
    mov edx, 0x1
    int 0x80

    mov eax, 0x30

    cmp eax, [x_buffer]
    jle verif_mai_mic
    jmp et_citire_y

    verif_mai_mic:
        mov eax, 0x39
        cmp eax, [x_buffer]
        jge et_construire_numar

    jmp et_citire_y

    et_construire_numar:
        mov ecx, [p]
        xor edx, edx
        xor eax, eax

        mov eax, [x_buffer]
        sub eax, 0x30
        mov ebx, eax
        mov eax, [x]
```

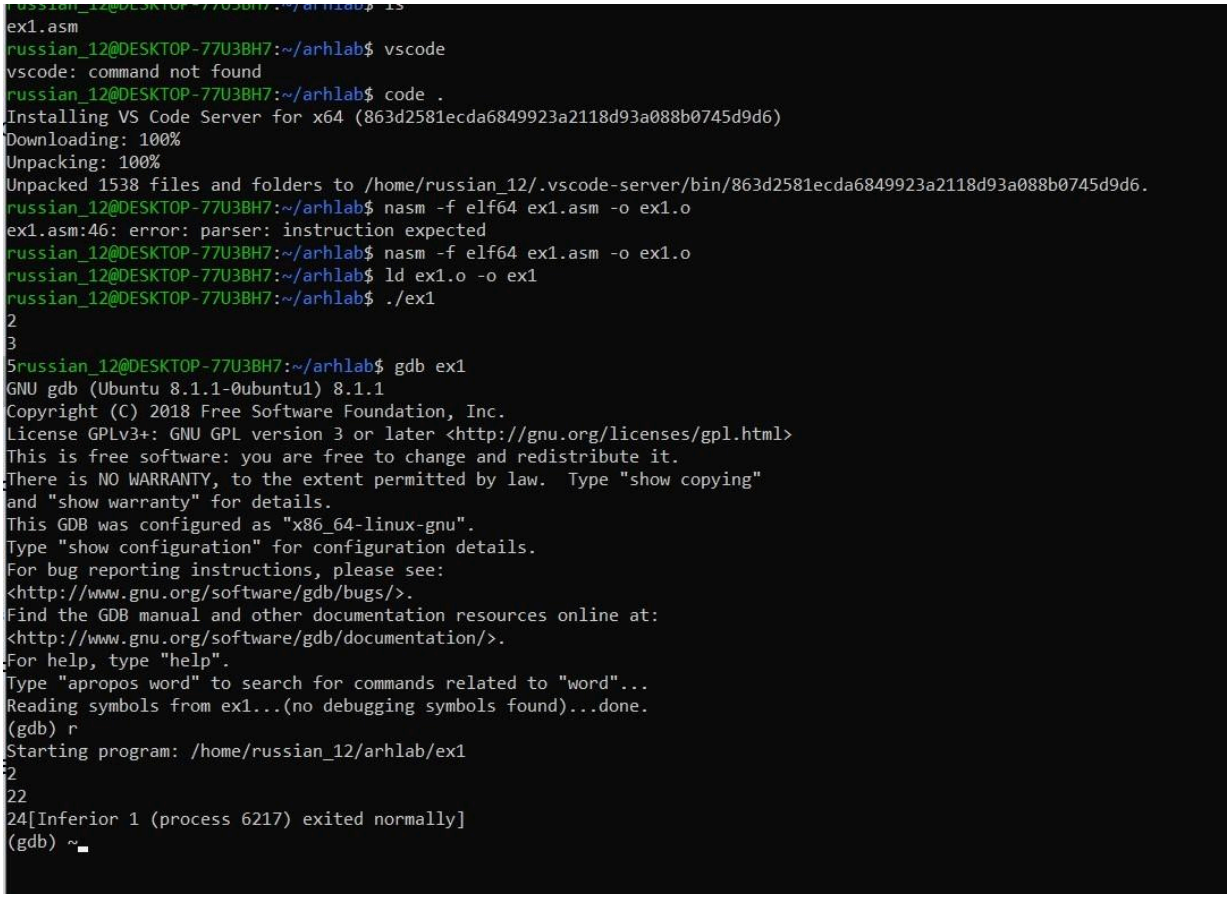
```
        mul ecx
        add eax, ebx

        mov [x], eax

        jmp et_citire

et_citire_y:

xor eax, eax
mov [y], eax

et_citire_y_1:
    xor edx, edx
    mov [x_buffer], edx

    mov eax, 0x3
    mov ebx, 0x0
    mov ecx, x_buffer
    mov edx, 0x1
    int 0x80

    mov eax, 0x30

    cmp eax, [x_buffer]
    jle verif_mai_mic_y
    jmp continue

    verif_mai_mic_y:
        mov eax, 0x39
        cmp eax, [x_buffer]
        jge et_construire_numar_y

    jmp continue

    et_construire_numar_y:

        mov ecx, [p]
        xor edx, edx
        xor eax, eax
```

```
        mov eax, [x_buffer]
        sub eax, 0x30
        mov ebx, eax
        mov eax, [y]
        mul ecx
        add eax, ebx

        mov [y], eax

        jmp et_citire_y_1

continue:

mov eax, [x]
add [y], eax

xor eax, eax
mov [x], eax

et_oglindit:

    xor edx, edx
    mov [x_buffer], edx

    mov eax, [y]
    mov ecx, 10
    div ecx

    cmp edx, 0
    mov ebx, 1
    mov [ok], ebx
    je zero_in_coada
    jmp resume

    zero_in_coada:
        xor ebx, ebx
        cmp ebx, [ok]
        je inc_cnt
```

```
        jmp resume

        inc_cnt:
            mov ebx, [cnt]
            inc ebx
            mov [cnt], ebx

    resume:

    mov [x_buffer], edx
    mov [y], eax

    xor edx, edx
    mov eax, [x]
    mul ecx
    add eax, [x_buffer]

    mov [x], eax


    mov eax, [y]
    cmp eax, 0
    jne et_oglindit


et_print:

    xor edx, edx
    mov [x_buffer], edx

    mov eax, [x]
    mov ecx, 10
    div ecx

    add edx, 0x30
    mov [x_buffer], edx

    mov [x], eax
```

```
    mov eax, 0x4
    mov ebx, 0x1
    mov ecx, x_buffer
    mov edx, 0x1
    int 0x80

    mov eax, [x]
    cmp eax, 0
    je afis_zero

    jmp et_print

afis_zero:

    xor ebx, ebx
    cmp [cnt], ebx
    je terminate

    mov eax, 0x4
    mov ebx, 0x1
    mov ecx, zero
    mov edx, 0x1
    int 0x80

    mov eax, [cnt]
    dec eax
    mov [cnt], eax

    jmp afis_zero

terminate:
    mov eax, 0x1
    xor ebx, ebx
    int 0x80
```

Result:



This NASM assembly code for Linux reads and processes numeric input from the user, performs calculations, and outputs the results to the console. It involves direct system calls for input/output operations and showcases manipulation of numeric data at a low level. The program reserves space for various variables in the .bss section for operations like storing input numbers, counters, and flags. It begins by initializing variables and setting up a loop to read characters from standard input, converting them into numeric values by assembling digits into numbers. These numbers undergo further processing, including addition and reversal of digits. The core logic involves carefully controlled loops, conditional checks, and arithmetic operations to construct and manipulate these numbers. System calls are used extensively for reading from stdin and writing to stdout, with specific attention to handling numeric data as ASCII characters. The program concludes by outputting the processed number, handling special cases like leading zeros, and finally exits cleanly by invoking the Linux system exit call. This code exemplifies fundamental assembly language programming techniques, including direct interaction with the operating system's kernel via system calls, for low-level data processing and output.

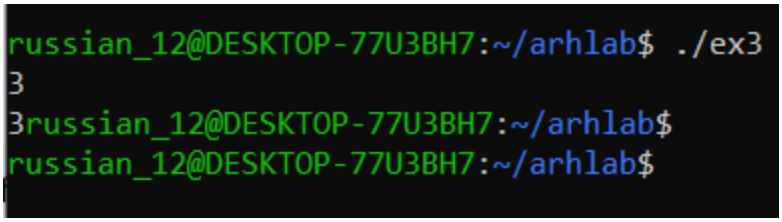
Exercise 3:

The 3rd exercise is a subtraction of 2 numbers.

The code:

```
section .bss
    x_buffer: resd 1

section .text
    global _start
_start:
    ; Read a character from stdin
    mov eax, 0x3      ; syscall number for sys_read
    xor ebx, ebx      ; file descriptor 0 (stdin)
    mov ecx, x_buffer ; buffer to store input
    mov edx, 0x1      ; number of bytes to read
    int 0x80          ; make syscall

    ; Write the read character to stdout
    mov eax, 0x4      ; syscall number for sys_write
    mov ebx, 0x1      ; file descriptor 1 (stdout)
    mov ecx, x_buffer ; buffer from which to write
    mov edx, 0x1      ; number of bytes to write
    int 0x80          ; make syscall

terminate:
    mov eax, 0x1      ; syscall number for sys_exit
    xor ebx, ebx      ; status 0
    int 0x80          ; make syscall
```

Result:



This code is an assembly program for Linux on the Intel x86 architecture that reads a single character from standard input (stdin) and writes it to standard output (stdout) using system calls. It first reserves a 4-byte space (x_buffer) for input data, then uses the sys_read system call (number 3) to read a character into x_buffer. Following this, it employs the sys_write system call (number 4) to write the character from x_buffer to stdout. The program concludes by invoking the sys_exit system call (number 1) to exit gracefully. System calls are made via the int 0x80 interrupt, with parameters passed in registers: eax for the system call number, ebx for the file descriptor (0 for stdin, 1 for stdout), ecx for the buffer's memory address, and edx for the number of bytes to read or write.

Conclusion:

In conclusion, through the exercises detailed in this report, we've embarked on a practical journey exploring the capabilities of NASM (Netwide Assembler) for developing assembly language programs on both Windows and Linux systems. Starting with a simple "Hello, World!" program and advancing through numerical operations including addition, subtraction, and manipulation of long numbers, this work illustrates the fundamental principles of low-level programming. These exercises have not only reinforced the importance of understanding hardware interactions and optimization at the assembly level but also highlighted the versatility of NASM as a tool for such endeavors. By tackling the challenges of arithmetic operations on data exceeding native register sizes and performing basic input/output operations via system calls, we've gained insights into the precision and control assembly language affords. This experience underscores the enduring relevance of assembly language in areas where performance and efficiency are paramount. Moreover, it serves as a

reminder of the value of mastering these foundational skills in the evolving landscape of computer architecture and software development.