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

COMPUTER ARCHITECTURE

Laboratory work 4:

Introduction in Assembly Language

Elaborated:
st. gr. FAF-213 Konjevic Alexandra

Verified:
asist. Univ Vladislav Voitcovschi

Chişinău – 2023

Objective:

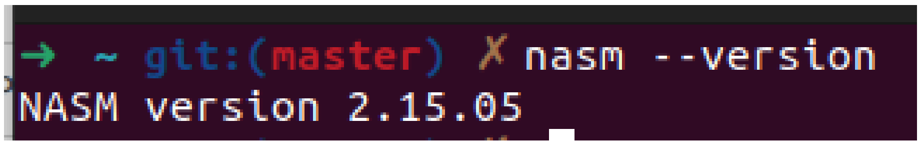
Getting familiar with the basic concepts of Assembly Language – how it works, its syntax; understanding concepts like registers, memory, instructions etc.

Introduction:

Assembly language is a low-level programming language that allows direct control over a computer's hardware. It is a vital tool in computer engineering and is widely used for developing software in various applications. In this report, we explore the fundamentals of assembly language programming, its syntax, and its application in computer systems. Specifically, this report presents beginner programs in assembly language, with the goal of providing a comprehensive understanding of the language. The laboratory work accompanying this report involves practical exercises aimed at reinforcing the theoretical concepts discussed.

Tasks:

1. Familiarize yourself with the basics of Assembly Language: Before diving into NASM, it's important to understand the fundamentals of Assembly Language. Start by learning the basic concepts such as registers, memory, instructions, and the syntax used to write Assembly code.
2. Install NASM: The first step towards learning NASM is to install it on your machine. NASM is available for multiple platforms like Windows, Linux, and macOS. Download and install the version that is compatible with your system.

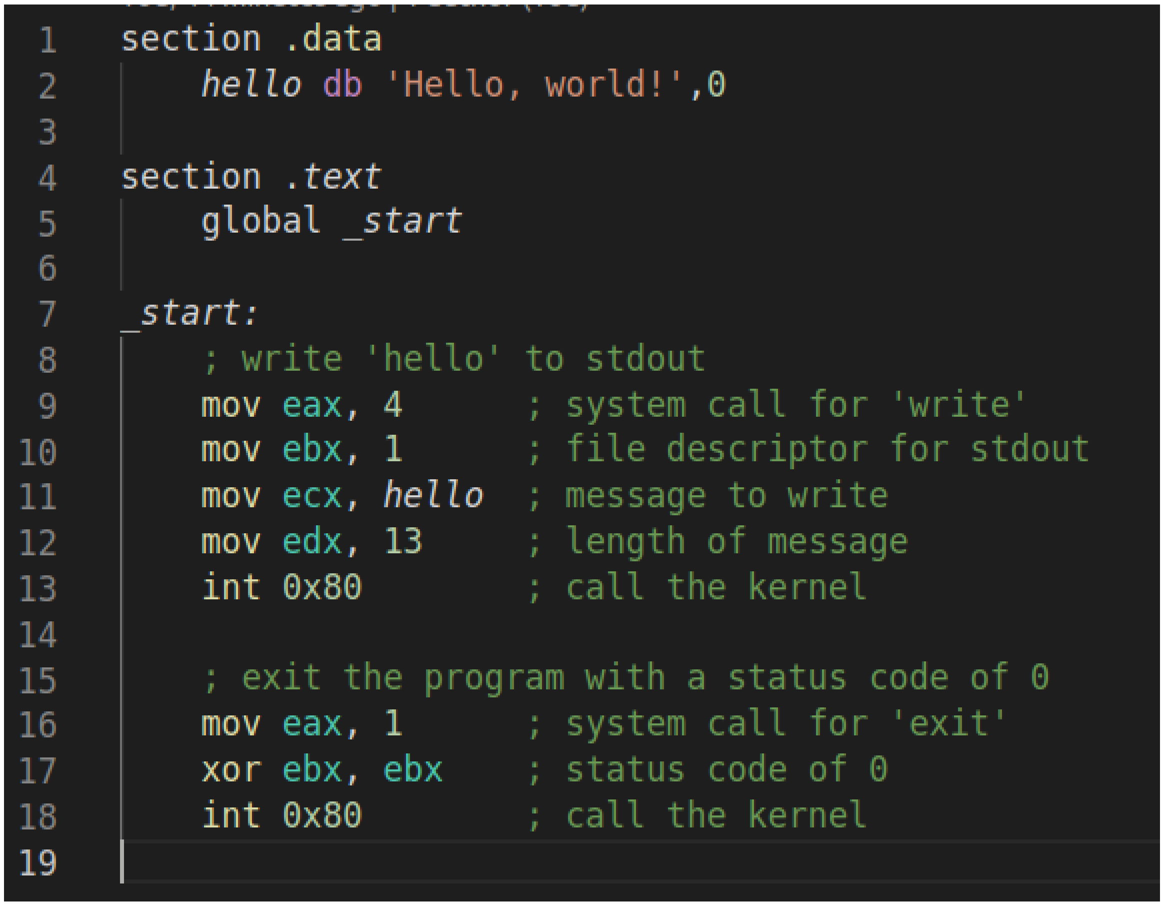


```
→  ~ git:(master) ✗ nasm --version
NASM version 2.15.05
```

NASM installed

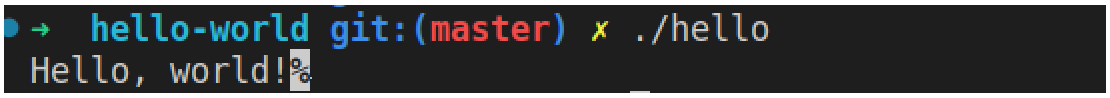
3. Write simple programs: Start by writing simple programs in NASM to get a feel for the language. Start with basic programs like printing messages on the screen, reading input from the user, and performing arithmetic operations. This will help you understand how NASM works and get comfortable with the syntax.

   a. Hello world program



```
section .data
    hello db 'Hello, world!',0

section .text
    global _start

_start:
    ; write 'hello' to stdout
    mov eax, 4      ; system call for 'write'
    mov ebx, 1      ; file descriptor for stdout
    mov ecx, hello  ; message to write
    mov edx, 13     ; length of message
    int 0x80        ; call the kernel

    ; exit the program with a status code of 0
    mov eax, 1      ; system call for 'exit'
    xor ebx, ebx    ; status code of 0
    int 0x80        ; call the kernel

```

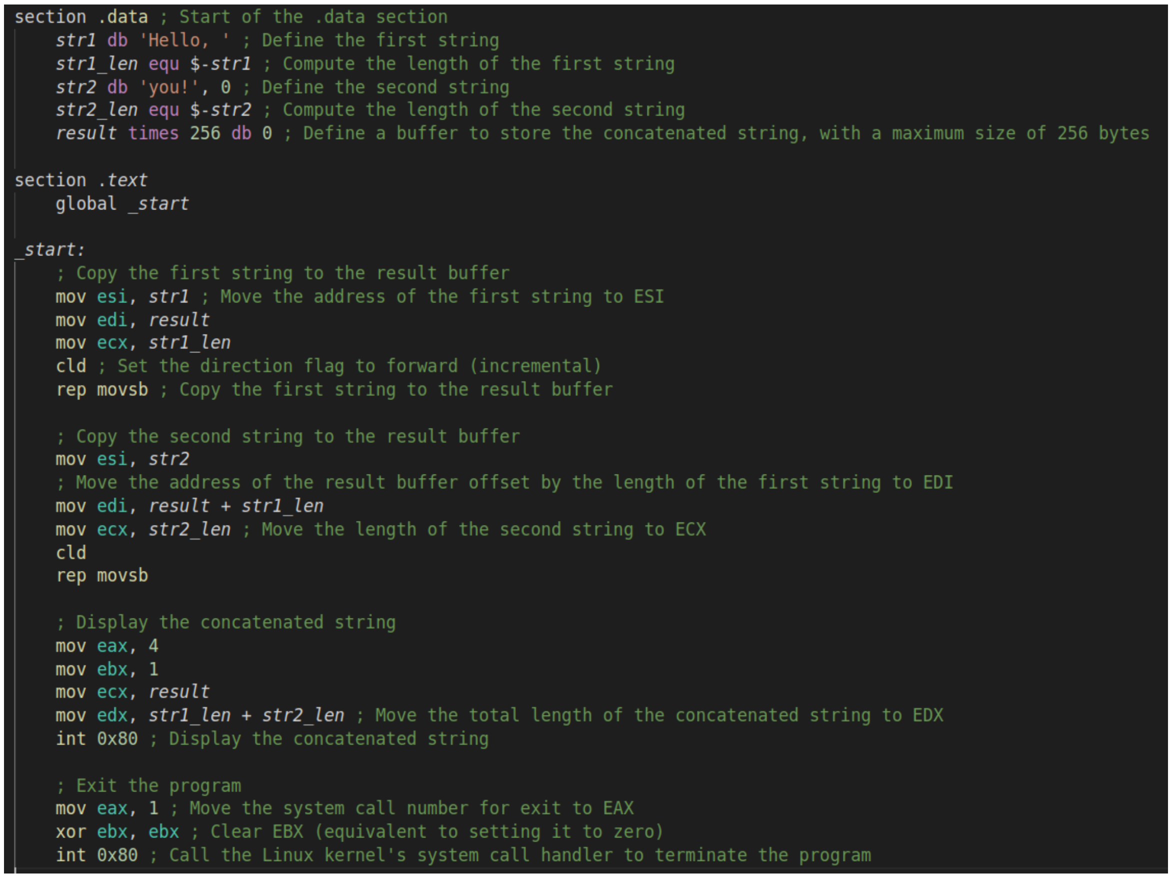
Hello world program in Assembly



```
→  hello-world git:(master) ✗ ./hello
Hello, world!%
```

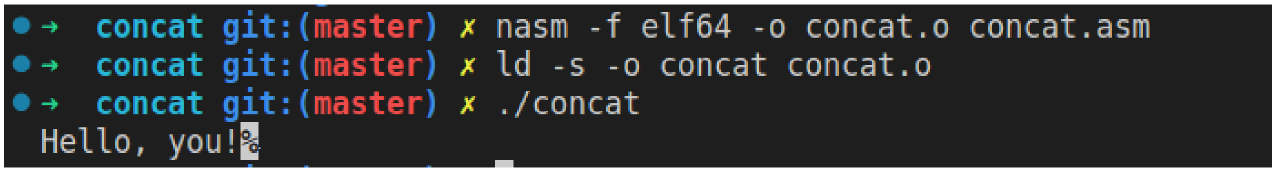
Hello world program output

b. Concatenate strings program



```
section .data ; Start of the .data section
    str1 db 'Hello, ' ; Define the first string
    str1_len equ $-str1 ; Compute the length of the first string
    str2 db 'you!', 0 ; Define the second string
    str2_len equ $-str2 ; Compute the length of the second string
    result times 256 db 0 ; Define a buffer to store the concatenated string, with a maximum size of 256 bytes

section .text
    global _start

_start:
    ; Copy the first string to the result buffer
    mov esi, str1 ; Move the address of the first string to ESI
    mov edi, result
    mov ecx, str1_len
    cld ; Set the direction flag to forward (incremental)
    rep movsb ; Copy the first string to the result buffer

    ; Copy the second string to the result buffer
    mov esi, str2
    ; Move the address of the result buffer offset by the length of the first string to EDI
    mov edi, result + str1_len
    mov ecx, str2_len ; Move the length of the second string to ECX
    cld
    rep movsb

    ; Display the concatenated string
    mov eax, 4
    mov ebx, 1
    mov ecx, result
    mov edx, str1_len + str2_len ; Move the total length of the concatenated string to EDX
    int 0x80 ; Display the concatenated string

    ; Exit the program
    mov eax, 1 ; Move the system call number for exit to EAX
    xor ebx, ebx ; Clear EBX (equivalent to setting it to zero)
    int 0x80 ; Call the Linux kernel's system call handler to terminate the program
```

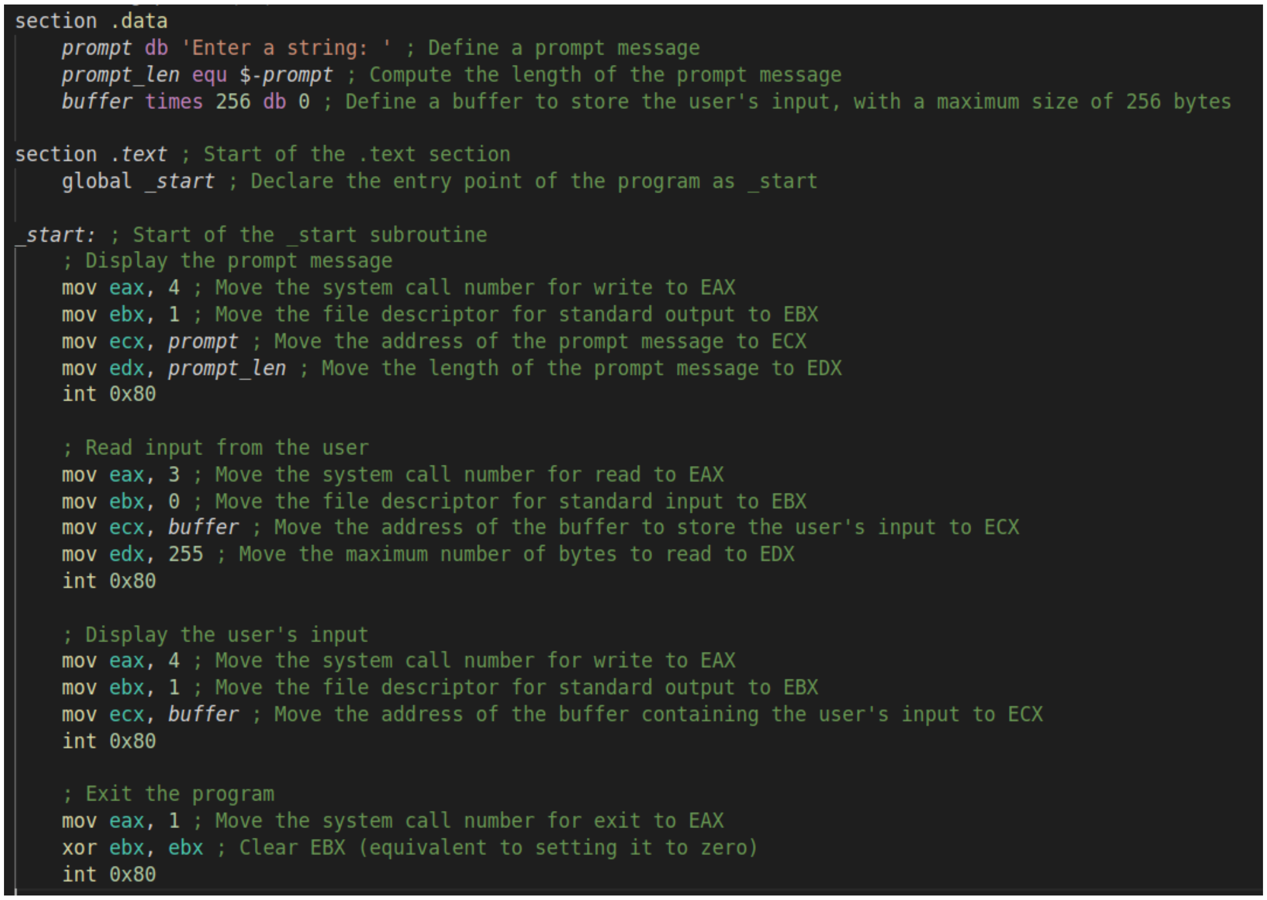
Concatenate characters program



```
● ➜  concat git:(master) ✗ nasm -f elf64 -o concat.o concat.asm
● ➜  concat git:(master) ✗ ld -s -o concat concat.o
● ➜  concat git:(master) ✗ ./concat
  Hello, you!%
```

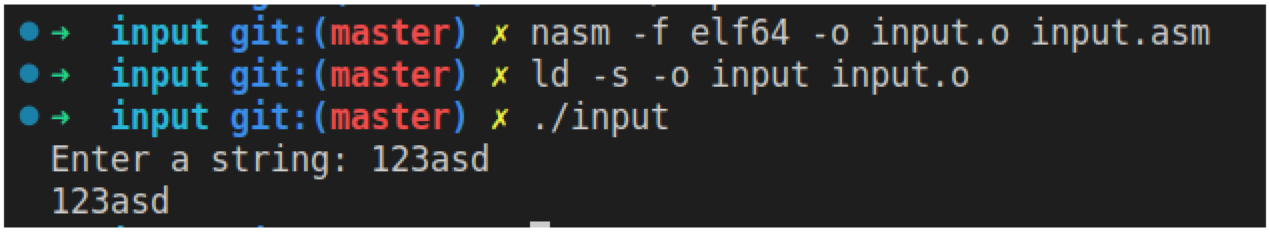
Concatenate program output

c. Read input from user



```
section .data
    prompt db 'Enter a string: ' ; Define a prompt message
    prompt_len equ $-prompt ; Compute the length of the prompt message
    buffer times 256 db 0 ; Define a buffer to store the user's input, with a maximum size of 256 bytes

section .text ; Start of the .text section
    global _start ; Declare the entry point of the program as _start

_start: ; Start of the _start subroutine
    ; Display the prompt message
    mov eax, 4 ; Move the system call number for write to EAX
    mov ebx, 1 ; Move the file descriptor for standard output to EBX
    mov ecx, prompt ; Move the address of the prompt message to ECX
    mov edx, prompt_len ; Move the length of the prompt message to EDX
    int 0x80

    ; Read input from the user
    mov eax, 3 ; Move the system call number for read to EAX
    mov ebx, 0 ; Move the file descriptor for standard input to EBX
    mov ecx, buffer ; Move the address of the buffer to store the user's input to ECX
    mov edx, 255 ; Move the maximum number of bytes to read to EDX
    int 0x80

    ; Display the user's input
    mov eax, 4 ; Move the system call number for write to EAX
    mov ebx, 1 ; Move the file descriptor for standard output to EBX
    mov ecx, buffer ; Move the address of the buffer containing the user's input to ECX
    int 0x80

    ; Exit the program
    mov eax, 1 ; Move the system call number for exit to EAX
    xor ebx, ebx ; Clear EBX (equivalent to setting it to zero)
    int 0x80
```

Input program output



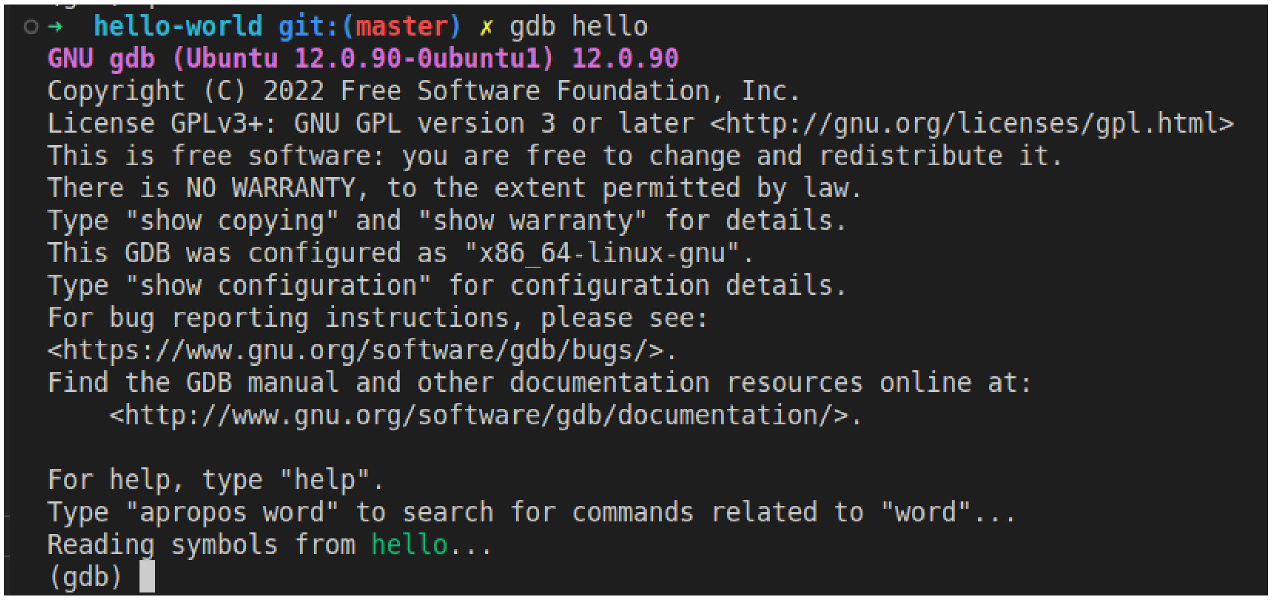
```
● ➜  input git:(master) ✗ nasm -f elf64 -o input.o input.asm
● ➜  input git:(master) ✗ ld -s -o input input.o
● ➜  input git:(master) ✗ ./input
  Enter a string: 123asd
  123asd
```

Input program output

4. Debug your programs: Debugging is an essential part of programming, and NASM is no exception. Learn how to use debugging tools like GDB to identify and fix errors in your code. This will help you become more efficient in your programming and also give you a better understanding of how your code works:

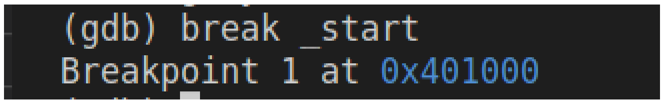
The ability to effectively use a debugger is a crucial skill for any software developer. NASM supports several debuggers, such as GDB and DDD. To get familiar with debugging, I used GDB and followed the steps below (I debugged the hello-world program):

- Compiled the program with the -g flag, to include debugging information in the object file.
- Started the GDB by running the command: `gdb hello`



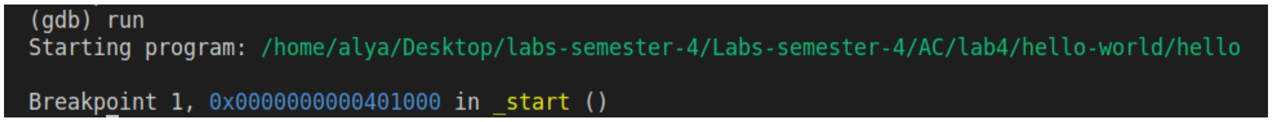
```
○ →  hello-world git:(master) ✗ gdb hello
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...
(gdb) 
```

- I set a breakpoint at the begging of the program:



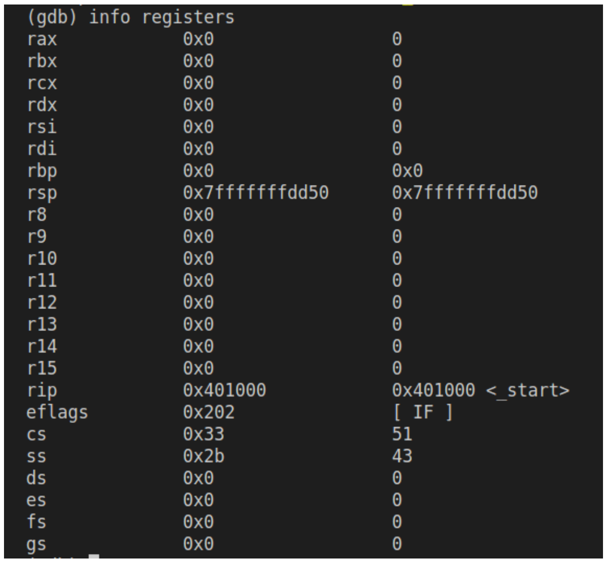
```
(gdb) break _start
Breakpoint 1 at 0x401000
```

- I ran the program:

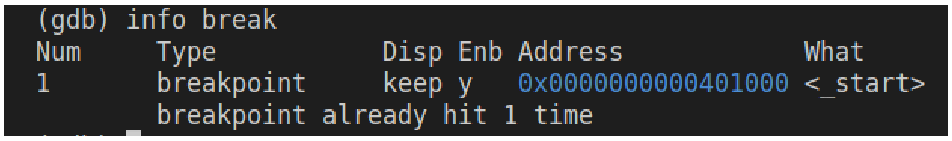


```
(gdb) run
Starting program: /home/alya/Desktop/labs-semester-4/Labs-semester-4/AC/lab4/hello-world/hello

Breakpoint 1, 0x0000000000401000 in _start ()
```

- I used `info registers` and `info break`



```
(gdb) info registers
rax            0x0                 0
rbx            0x0                 0
rcx            0x0                 0
rdx            0x0                 0
rsi            0x0                 0
rdi            0x0                 0
rbp            0x0                 0x0
rsp            0x7fffffffdd50      0x7fffffffdd50
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x401000            0x401000 <_start>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```



```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000401000 <_start>
        breakpoint already hit 1 time
```

Conclusion:

In conclusion, this report has provided an overview of assembly language programming, including its syntax, structure, and application in computer systems. Through practical exercises, I have gained hands-on experience in writing programs using assembly language, providing a foundation for further exploration in this field.

By working with assembly language, I have gained insight into the underlying operations of a computer system, and the role of low-level programming in controlling hardware. This knowledge is critical in the development of software and applications for various domains, including embedded systems, operating systems, and game development.

Assembly language programming requires a thorough understanding of computer architecture and hardware, as well as a keen attention to detail. However, with practice and dedication, it is a powerful tool for developers to optimize performance and implement functionality that may not be possible using higher-level languages.

In conclusion, this laboratory work has provided a solid foundation for further exploration of assembly language programming.