**Ministry of Education, Culture, and Research of the Republic of Moldova**

**Technical University of Moldova**

**The Faculty of Computers, Informatics, and Microelectronics**

**REPORT**

Laboratory work no.5

*Computer Architecture*

Executed by:

st. gr. FAF-212 Lupascu Felicia

Verified by:

univ. assist. Voitcovschi Vladislav

Chişinău - 2023

Contents

[Objectives 3](#_Toc134868719)

[Condition 3](#_Toc134868720)

[Theory 4](#_Toc134868721)

[Getting Started 4](#_Toc134868722)

[Implementation 5](#_Toc134868723)

[Main Menu 5](#_Toc134868724)

[Inverting a String 5](#_Toc134868726)

[Converting a String to Uppercase 6](#_Toc134868728)

[Substraction of two numbers 7](#_Toc134868729)

[Even or Odd Number 8](#_Toc134868730)

[The smallest number 9](#_Toc134868731)

[Upercase to Lowercase 10](#_Toc134868732)

[Multiplyer 11](#_Toc134868733)

[Prime or Not 12](#_Toc134868735)

[The largest Number 12](#_Toc134868736)

[Randomizer 14](#_Toc134868737)

[The output: 15](#_Toc134868738)

[Conclusion 16](#_Toc134868739)

# Objectives

1. Understanding the basic concepts of assembly Objectives for the Laboratory Task:
2. Create an NASM assembler program with 10 cyclic processes (functions).
3. Develop an interactive menu that allows the user to select from the 10 processes.
4. Code each of the 10 processes in a cyclical manner, so the program returns to the interactive menu after each process is completed.
5. Ensure that the program is well-commented and structured to make it easy to understand and modify.
6. Test the program to ensure it works correctly and that users can choose any of the 10 processes.
7. Personalize the program in a unique way so that no two programs are identical.
8. Present the program before the class and demonstrate how it works.
9. Implement a generator that produces 10 random numbers between 1 and 55 to serve as the variants for each process.

# Condition

For this laboratory work, each student has to create a NASM assembler program that contains 10 cyclic processes (functions). Additionally, the program must allow the user to choose which of the 10 processes to execute at the program launch

• To accomplish this task, follow the steps below:

• Write an interactive menu that allows the user to choose from the 10 processes.

• Write the code for each of the 10 processes. Each process must be written cyclically so that the program always returns to the interactive menu after a process is completed.

• Ensure that your program is well-commented and structured clearly so that it is easy to understand and modify

• Test the program to ensure that it works correctly and that the user can choose any of the 10 processes

• Ensure that each student personalizes their program in a unique way so that there are no identical programs.

• Prior to presenting the lab, each student must present their program and demonstrate that it can be used to choose any of the 10 processes.

❖ The first program will contain a generator of 10 random numbers from 1 to 55

❖ These will be the variants of each.

# Theory

**Assembly language** is a low-level programming language for a computer or other programmable device specific to a particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.

NASM (Netwide Assembler) is a popular open-source assembler that supports a range of architectures, including x86 and x86-64. It is available for multiple platforms, including Linux, macOS, and Windows.

NASM has a syntax that is like Intel syntax, which makes it easy for developers who are familiar with Intel syntax to transition to NASM. However, NASM also has some differences, such as the use of backticks for string literals and the requirement to specify the size of memory operands.

NASM can be used to create executable files, object files, and shared libraries. It supports several output formats, including ELF, COFF, and Mach-O. NASM also includes preprocessor directives, macros, and conditional assembly, which makes it a powerful tool for writing efficient and optimized assembly code.

Some of the features of NASM include support for multiple addressing modes, support for macros and conditional assembly, support for debugging information, and support for multiple output formats. Additionally, NASM has a large and active community, which provides support and resources for developers who use NASM.

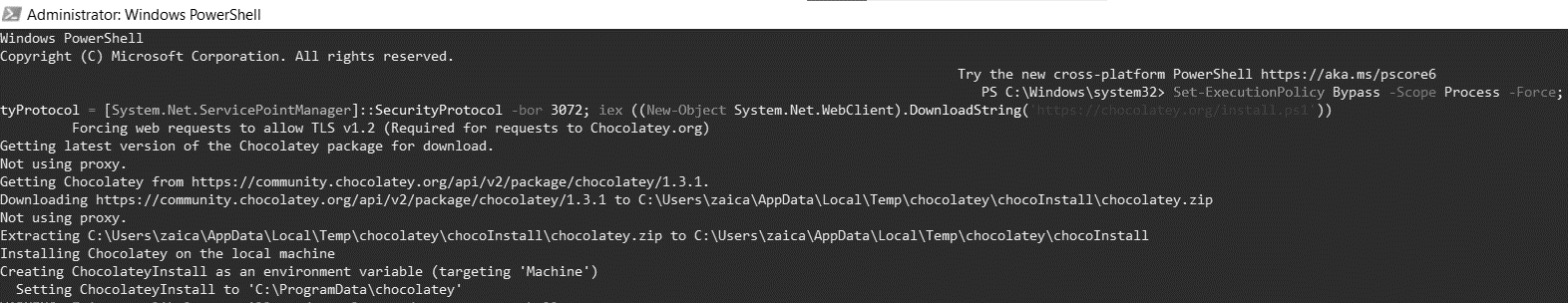
# Getting Started

https://www.jdoodle.com/compile-assembler-nasm-online/

Here is my code and my working place.

The **Netwide Assembler** (**NASM**) is an [assembler](https://en.wikipedia.org/wiki/Assembly_language#_blank) and [disassembler](https://en.wikipedia.org/wiki/Disassembler) for the [Intel](https://en.wikipedia.org/wiki/Intel) [x86](https://en.wikipedia.org/wiki/X86) architecture. It can be used to write [16-bit](https://en.wikipedia.org/wiki/16-bit), [32-bit](https://en.wikipedia.org/wiki/32-bit) ([IA-32](https://en.wikipedia.org/wiki/IA-32)) and [64-bit](https://en.wikipedia.org/wiki/64-bit) ([x86-64](https://en.wikipedia.org/wiki/X86-64)) programs. It is considered one of the most popular assemblers for [Linux](https://en.wikipedia.org/wiki/Linux).

Setting up the environment



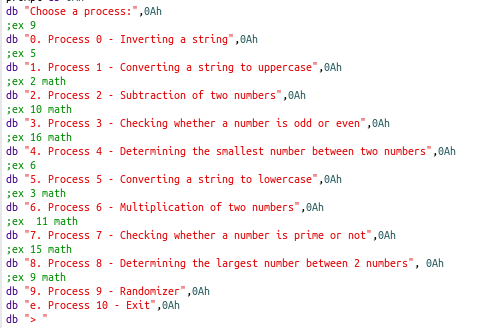
# Implementation

Exercise numbers

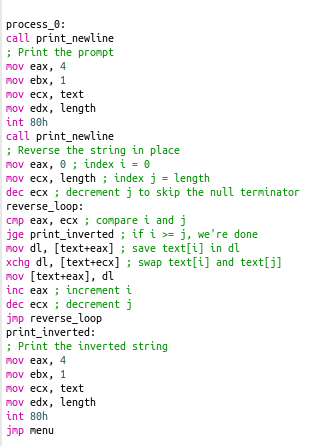
**9 5 2 10 16 6 3 11 15 9,** I mixed them, so I did procedures from both compartiments.

# Main Menu

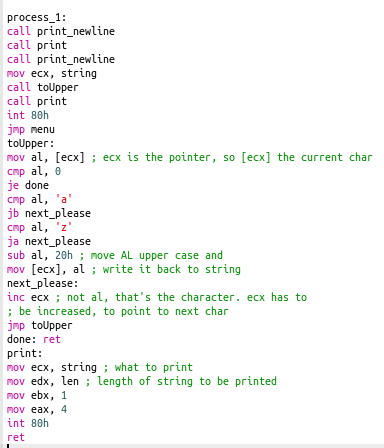
This is a list of processes that a program offers to the user. Each process is numbered and has a brief description next to it. The program prompts the user to choose a process by displaying the message "Choose a process:", followed by a newline character. The user can then enter the number or letter corresponding to the desired process followed by the ">" symbol.



# **Inverting a String**

This code defines a procedure labeled "process\_0" that takes a string and reverses it in place, then prints the inverted string. The procedure begins by calling another procedure labeled "print\_newline" which prints a newline character. The code then prints a prompt asking the user to enter a string, using the system call "int 80h" with function code 4 and passing the necessary arguments in the registers.

# **Converting a String to Uppercase**

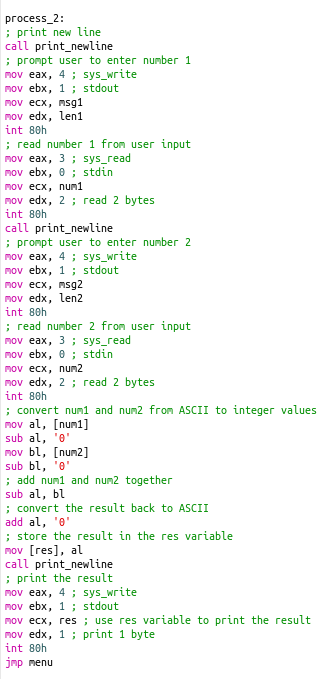
This code is designed to convert a given string to uppercase. The process begins by printing a prompt to the user. It then takes the input string and passes it to the toUpper function.If the current character is not null, the function checks if it is a lowercase letter (between 'a' and 'z' in ASCII). If it is not, it jumps to the next character. If it is a lowercase letter, it subtracts 20h (hexadecimal 20) from the ASCII code of the letter, effectively converting it to its uppercase equivalent.

# **Substraction of two numbers**

This code prompts the user to enter two numbers, reads them in as ASCII characters, converts them to integer values, subtracts the second number from the first, converts the result back to an ASCII character, and then prints out the result.

Here are the individual steps:

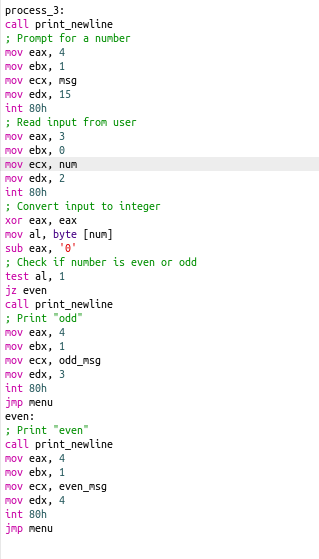
1. The code prints a newline character to standard output.
2. The code prompts the user to enter the first number by printing a message to standard output using the sys\_write system call.
3. The code reads two bytes from standard input into a buffer named num1 using the sys\_read system call.
4. The code prints another newline character to standard output.
5. The code prompts the user to enter the second number by printing a message to standard output using the sys\_write system call.
6. The code reads two bytes from standard input into a buffer named num2 using the sys\_read system call.
7. The code converts the ASCII character in num1 to an integer value by subtracting the ASCII value of '0'.
8. The code converts the ASCII character in num2 to an integer value by subtracting the ASCII value of '0'.
9. The code subtracts the value in num2 from the value in num1 and stores the result as an ASCII character in a variable named res.
10. The code prints a newline character to standard output.
11. The code prints the value in res to standard output using the sys\_write system call.
12. The code jumps back to the menu label to continue processing user input.



# **Even or Odd Number**

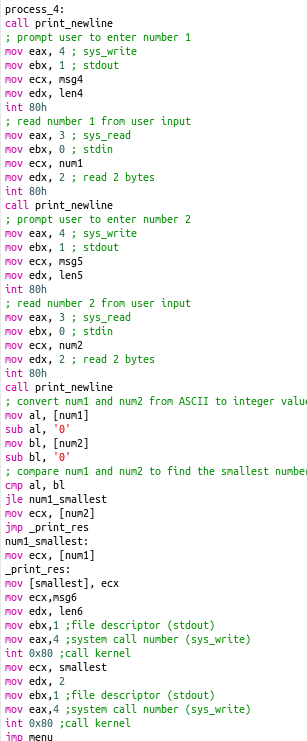
This code prompts the user to enter a number, reads the input, and then checks whether the number is even or odd. It first prints a prompt message using the sys\_write system call. It then reads user input using the sys\_read system call and stores it in a buffer called num.

The code then converts the input to an integer by subtracting the ASCII value of the character '0'. It then uses the test instruction to check the least significant bit of the integer to determine whether it is even or odd. If the least significant bit is 0, the number is even, and the code jumps to the even label, which prints a message indicating that the number is even. If the least significant bit is 1, the number is odd, and the code jumps to the odd label, which prints a message indicating that the number is odd. Finally, the code jumps back to the menu label, which prints the main menu.



# **The smallest number**

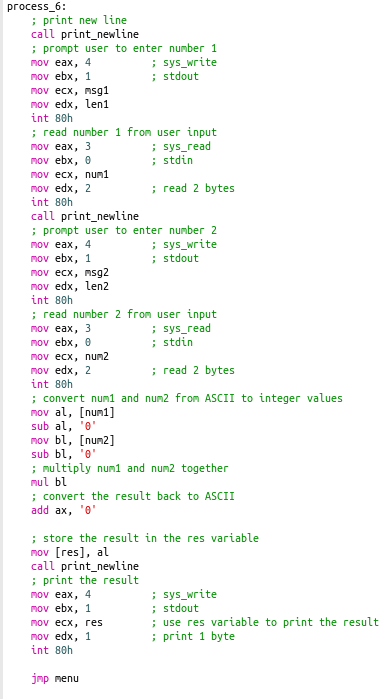
This code prompts the user to enter two numbers and then finds the smallest number among them. It first prompts the user to enter number 1 and reads the input from the user. It does the same for number 2. Then it converts the ASCII representation of the two numbers to integer values and compares them using the "cmp" instruction. If the first number is smaller or equal to the second number, it saves the first number as the smallest number, otherwise it saves the second number. It then prints the smallest number to the console. Finally, it jumps back to the menu.



# **Upercase to Lowercase**

This program takes a string input from the user, converts all uppercase letters to lowercase, and then prints the modified string. The toLower function loops through each character in the string, checks if it is an uppercase letter, and if so, converts it to lowercase. The priint function simply prints the string to the console. Finally, the modified string is printed using the priint function.

# **Multiplyer**

This code prompts the user to enter two single-digit numbers, multiplies them together, and then prints the result. It first prompts the user to enter the two numbers and reads them as ASCII characters. It then converts the ASCII characters to integer values, multiplies them together using the mul instruction, converts the result back to ASCII, and stores the result in the res variable. Finally, it prints the result as a single ASCII character.

# **Prime or Not**

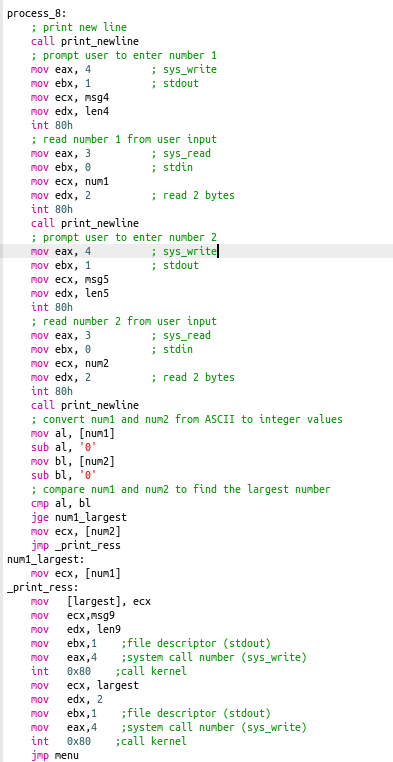
This code checks whether a given number is prime or not. It first prompts the user to input a number, reads the input, and converts it to an integer. It then checks if the number is less than or equal to 2. If so, it is a prime number. If not, it starts a loop to divide the number by all integers from 2 up to half of the number. If there is no remainder when dividing the number by a given integer, it means the number is not prime, and the code jumps to the "not\_prime" label to print a message indicating that. If the loop completes without finding any divisors with no remainder, it means the number is prime, and the code jumps to the "prime" label to print a message indicating that. After printing the message, the code jumps back to the main menu.

# **The largest Number**

This code prompts the user to enter two numbers, reads them as strings, converts them to integers, compares them, and then outputs the largest number.

Here is a breakdown of the code:

* The first line calls the print\_newline subroutine to print a newline character.
* The next several lines prompt the user to enter the first number and then read it from standard input using the sys\_read system call. The number is stored in the num1 buffer.
* After reading the first number, another newline is printed using print\_newline.
* The next several lines prompt the user to enter the second number and then read it from standard input. The number is stored in the num2 buffer.
* Another newline is printed.
* The two numbers are converted from ASCII to integer values using the sub instruction to subtract the ASCII value of '0' from each character.
* The cmp instruction is used to compare the two numbers, and the jge instruction jumps to num1\_largest if the first number is greater than or equal to the second number.
* If the second number is greater than the first number, the ecx register is loaded with the second number instead.
* The largest number is stored in the largest buffer using the mov instruction.
* The largest number is printed to the screen along with a message using the sys\_write system call.
* The program jumps back to the main menu using jmp menu.



# **Randomizer**

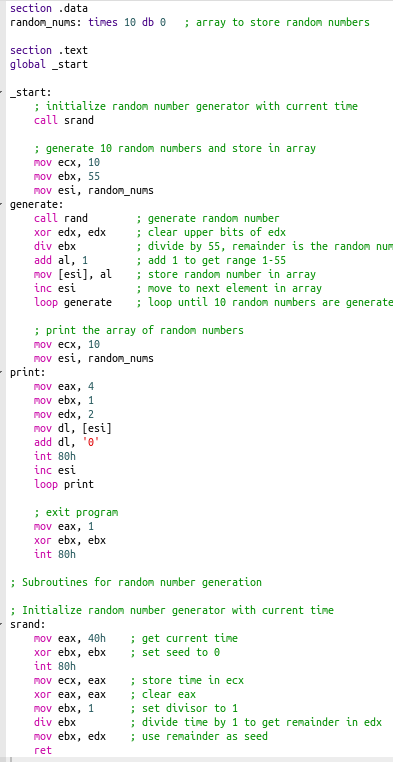
This code is written in NASM and generates an array of 10 random numbers between 1 and 55. It uses the rand and srand subroutines for random number generation.

The data section defines an array called random\_nums with 10 bytes to store the 10 random numbers generated. The text section contains the main code along with the srand and rand subroutines.

The \_start label is the entry point for the program. It calls the srand subroutine to seed the random number generator with the current time. Then it enters a loop to generate 10 random numbers using the rand subroutine, and store them in the random\_nums array.

To generate a random number between 1 and 55, the program divides the random number generated by 55 and takes the remainder. Then it adds 1 to the remainder to shift the range from 0-54 to 1-55.





The output:

Conclusion

After finishing this lab assignment, I developed additional knowledge using Assembly language. Utilizing a different operating system from the previous laboratory work was interesting. Writing an interactive menu that would let the user select from the 10 processes was both difficult and satisfying for me.

My ability to write cyclic programs improved as a result of writing the code for each of the ten processes. Additionally, I discovered how crucial a clear structure and commenting are to the readability and adaptability of code.

Overall, I believe that my understanding of Assembly language and my programming abilities have improved as a result of this lab work.