

**MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA**  
**TECHNICAL UNIVERSITY OF MOLDOVA**  
**FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS**  
**DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS**

# **Computer Architecture**

## ***Laboratory work 4 : Simple problems in Assembly***

Elaborated:

st.gr. FAF-211

Corețchi Mihai

Verified:

asist.univ.

Vladislav Voitcovich

Chișinău, 2023

## Content

<b>Introduction</b> . . . . .	<b>3</b>
<b>Objectives</b> . . . . .	<b>4</b>
<b>Task 1</b> . . . . .	<b>5</b>
<b>Task 2</b> . . . . .	<b>5</b>
<b>Task 3</b> . . . . .	<b>8</b>
<b>Task 4</b> . . . . .	<b>15</b>
<b>Conclusions</b> . . . . .	<b>17</b>

## **Introduction**

The programming language known as "asm" or assembly language is a low-level language used to write instructions for a computer's processor. It is a symbolic representation of the machine code that the processor can understand and execute. Assembly language is classified as a low-level language due to its proximity to the hardware of the computer, which makes it fast and efficient but difficult to write and read compared to high-level languages.

Each instruction in assembly language represents a specific operation that the processor can perform, such as adding two numbers or moving data from one memory location to another. The instructions are written using mnemonics, which are brief codes representing the machine code instructions. Assembly language is typically utilized in system programming, device drivers, and embedded systems, where performance and efficiency are essential. It necessitates a good understanding of computer architecture and can be challenging to write and debug, but it can also be highly rewarding and potent for the programmer who masters it.

Assembly language is a vital tool for programmers working with low-level hardware and software interfaces, allowing them to write code that interacts directly with the computer's hardware, such as the processor, memory, and input/output devices. Because assembly language is so closely tied to computer hardware, it is frequently used in operating system development and system-level software, such as firmware and device drivers. It is also used to optimize performance in high-performance computing applications like scientific simulations or video game engines.

In addition to system programming, assembly language is widely used in embedded systems development. These are small computers or microcontrollers that are integrated into larger systems like appliances, vehicles, and medical devices. Because these systems often have limited processing power and memory, assembly language can be an efficient and compact code writing tool. However, assembly language programming requires a strong understanding of computer architecture and the instruction set of the target processor. It can be challenging to write and debug, requiring a great deal of attention to detail. Nevertheless, it can also be a powerful and rewarding programming language, enabling programmers to write highly optimized code for specific hardware and software environments.

## Objectives

1. Low-level hardware programming: Assembly language allows developers to write code that interacts directly with the hardware of a computer or device. This is useful for low-level programming tasks such as device drivers or system-level software.
2. Embedded systems programming: Assembly language is commonly used for programming small computers or microcontrollers that are integrated into larger systems, such as vehicles or medical devices.
3. Performance optimization: Assembly language provides a high degree of control over how code is executed, which can be used to optimize performance for specific hardware or software environments.
4. Reverse engineering: Assembly language is often used for reverse engineering, which involves analyzing software or hardware to understand how it works. This can be useful for security research or to gain a better understanding of legacy software or hardware.
5. Academic research: Assembly language can be a useful tool for academic research in fields such as computer science or engineering, where low-level programming tasks are required.
6. PPersonal learning: Learning assembly language can be a valuable personal development goal for programmers who want to gain a deeper understanding of computer architecture and low-level programming concepts.
7. Porting software: Assembly language can be used to port software from one platform to another, particularly in cases where the original source code is not available or is difficult to modify.
8. Debugging and optimization of high-level code: Assembly language is often used to debug and optimize high-level code by examining the machine code generated by the compiler and making adjustments to improve performance or fix bugs.
9. Operating system development: Assembly language is a key tool for operating system development, particularly for low-level tasks such as interrupt handling and memory management.
10. Real-time systems programming: Assembly language is commonly used in real-time systems programming, where precise timing and fast execution are critical. This includes applications such as control systems, robotics, and aviation systems.

## **Task 1**

Familiarize yourself with the basics of Assembly .

Assembly Language is a low-level programming language that is used to write programs for computers, microcontrollers, and other devices. In Assembly Language, the programmer writes code that corresponds directly to the machine code instructions that the computer can execute. This makes it a powerful language for writing highly optimized and efficient code, but also makes it more complex and difficult to work with than higher-level programming languages.

At the heart of Assembly Language are registers and memory. Registers are small, fast, on-chip storage locations that can be used to store data or perform arithmetic and logical operations. Memory, on the other hand, is a larger, slower storage area that can hold much more data than registers, but is more expensive to access.

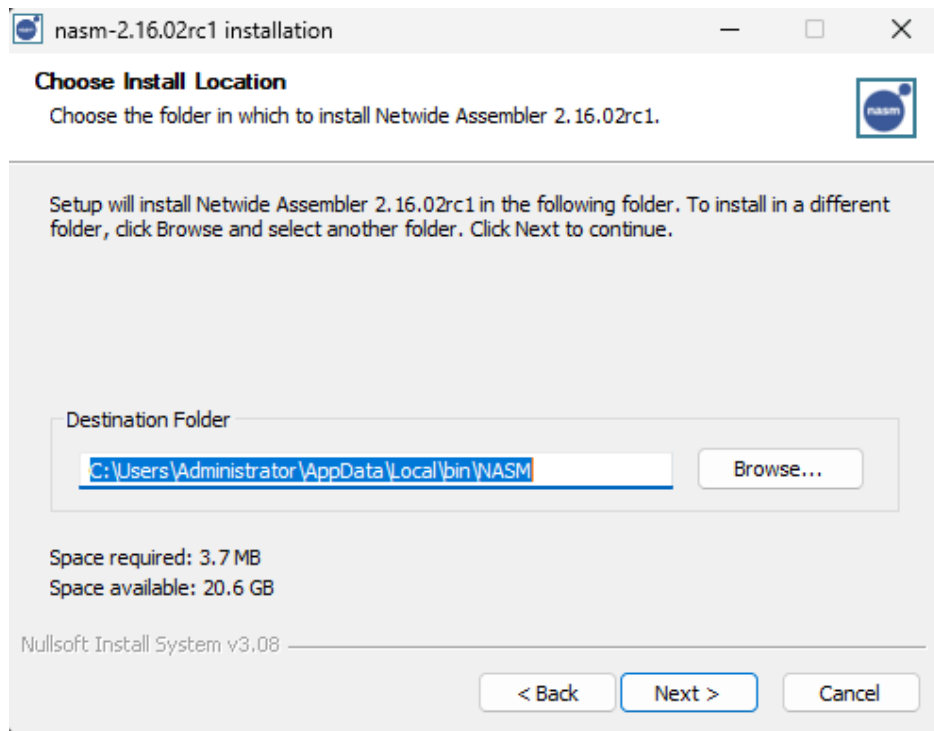
Assembly Language instructions are written using a specific syntax that varies depending on the processor architecture and the specific assembly language being used. Instructions typically consist of an operation code (opcode) that specifies the operation to be performed, along with one or more operands that provide the data for the operation.

To write Assembly Language code, a programmer must have a deep understanding of the underlying hardware architecture, including the registers, memory, and instruction set of the processor being used. They must also be able to read and write machine code in hexadecimal format.

Overall, Assembly Language is a powerful and important tool for programming low-level systems, but it requires a high level of technical expertise and a deep understanding of computer hardware.

## **Task 2**

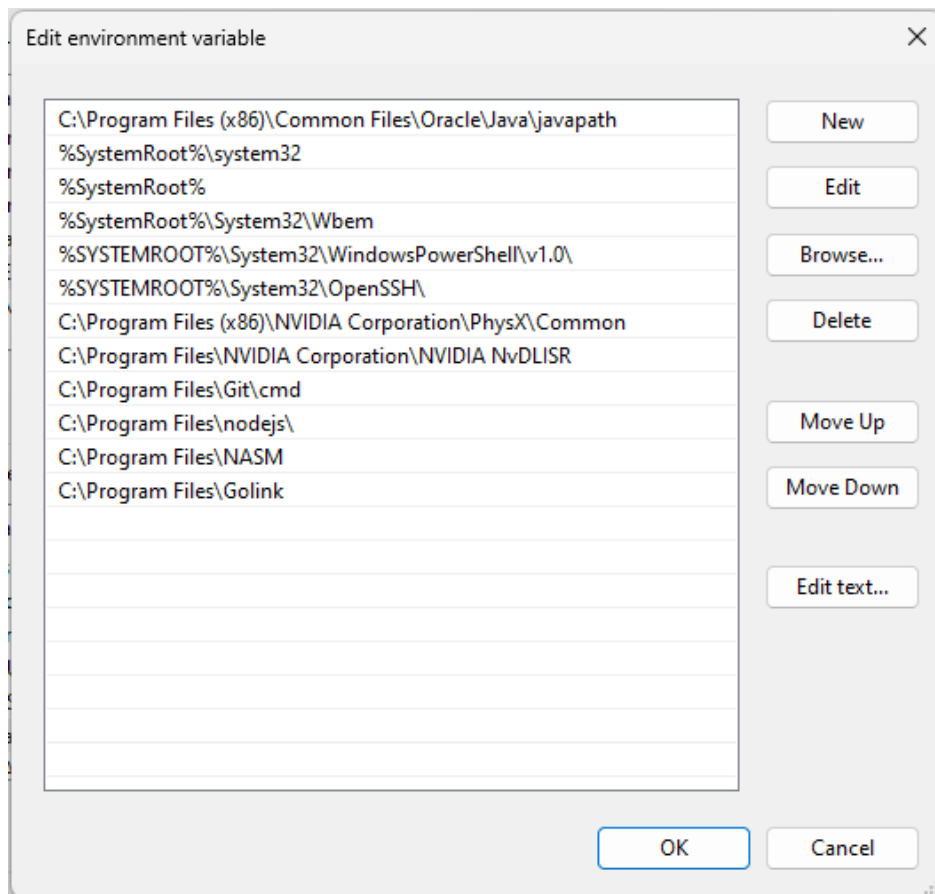
NASM is a popular assembler used to write code in assembly language. Sublime Text is a versatile text editor that is commonly used for programming, and Golink is a linker used to create executable files from object files. Here are the steps to install these tools on a Windows system:



### *NASM installing*

The first step is to download the NASM installer for Windows from the official website. Run the installer and follow the instructions to install NASM on your system. Once installed, you can check if it's properly installed by opening a command prompt and typing "nasm -v". If NASM is installed correctly, it will display the version number.

To install Sublime Text, visit the official website and download the installer for your version of Windows. Run the installer and follow the instructions to install Sublime Text on your system. Once installed, you can launch Sublime Text from the Start menu.

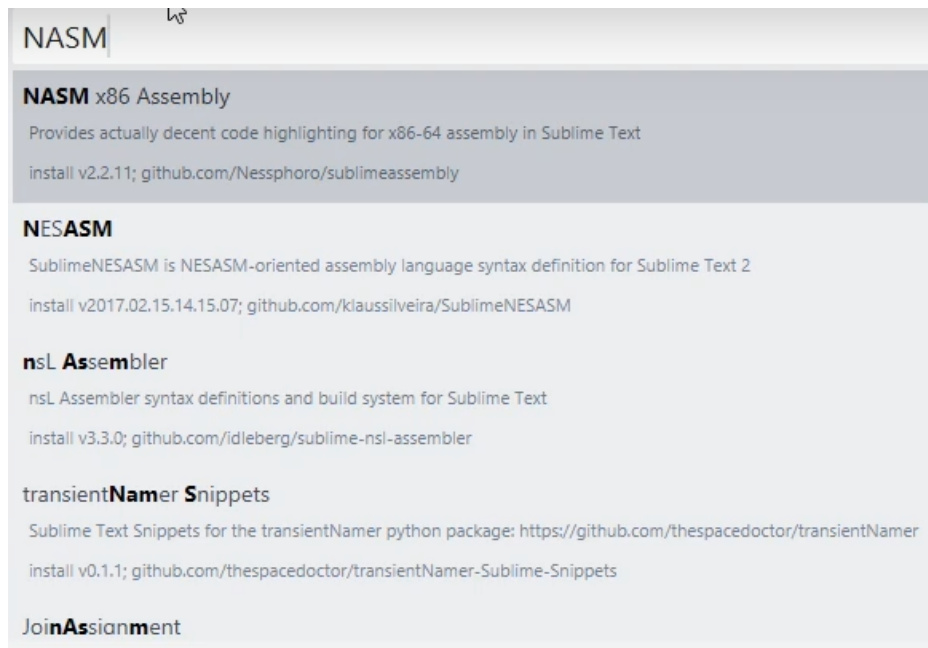


### *Setting administrator rights for both GoLink and NASM*

To install Golink, download the zip file from the official website and extract it to a folder on your system. You can add the folder to your system PATH to make it easier to use. To do this, right-click on "This PC" in Windows Explorer and select "Properties". Click on "Advanced system settings", then click on "Environment Variables". Under "System Variables", select "Path" and click "Edit". Add the path to the Golink folder to the list of paths and click "OK" to save your changes.

### *Configuring Sublime Text for Assembly Language Programming*

To configure Sublime Text for assembly language programming, you can install a package called "Easy Assembly". To do this, open Sublime Text and go to "Preferences" > "Package Control". In the search bar, type "Assembly x86" and select it from the list. Click "Install" to install the package. Once installed, you can use it to write and edit assembly language code.



### *NASM installing into Sublime Text*

#### **Problem 1**

This code is written in assembly language and runs on the x86 architecture. It defines two constants using the EQU directive, NULL and STDOUTHANDLE, which are set to 0 and -11 respectively. It also declares three external functions, GetStdHandle@4, WriteFile@20, and ExitProcess@4, that will be used later in the code.

The Start label is defined as the entry point of the program. It calls the GetStdHandle@4 function to retrieve the standard output handle and stores the value in the StandardHandle variable. It then adds the values of num1 and num2 and stores the result in the result variable.

The DecimalToString function is called to convert the result value to a string and stores it in the resultstr variable. The DecimalToString function converts the integer value to a string of decimal digits by repeatedly dividing the value by 10 and storing the remainder in a character buffer until the value becomes zero.

The program then uses the WriteFile@20 function to write the message "The result is:" and the result string to standard output. Finally, it exits the program using the ExitProcess@4 function.

The DecimalToString function uses the EBP register as a frame pointer to store the function's local variables on the stack. It uses the DIV instruction to divide the value by 10 and the ADD instruction to convert the remainder to an ASCII character. The result is stored in the character buffer, and the loop continues until the value becomes zero. The function returns the length of the resulting string.

```
NULL EQU 0
```

```
STD_OUTPUT_HANDLE EQU -11
```



```

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

global Start

section .data
    Message db "The result is:", 0
    MessageLength EQU $-Message
    num1 dd 4
    num2 dd 17
    result dd 0

section .bss
    StandardHandle resd 1
    Written resd 1
    result_str resb 11

section .text
Start:
    ; Get the standard output handle
    push STD_OUTPUT_HANDLE
    call _GetStdHandle@4
    mov dword [StandardHandle], EAX

    ; Add the two numbers and store in result
    mov EAX, [num1]
    add EAX, [num2]
    mov [result], EAX

    ; Convert the result to a string
    mov EAX, [result]
    call DecimalToString

```

```

; Write the message and the result to standard output
push NULL
push Written
push MessageLength
push Message
push dword [StandardHandle]
call _WriteFile@20
push NULL
push Written
push 11
push result_str
push dword [StandardHandle]
call _WriteFile@20

; Exit the program
push NULL
call _ExitProcess@4

```

DecimalToString:

```

push EBP
mov EBP, ESP
sub ESP, 8
mov dword [EBP-4], 0
mov dword [EBP-8], 10

```

.Loop:

```

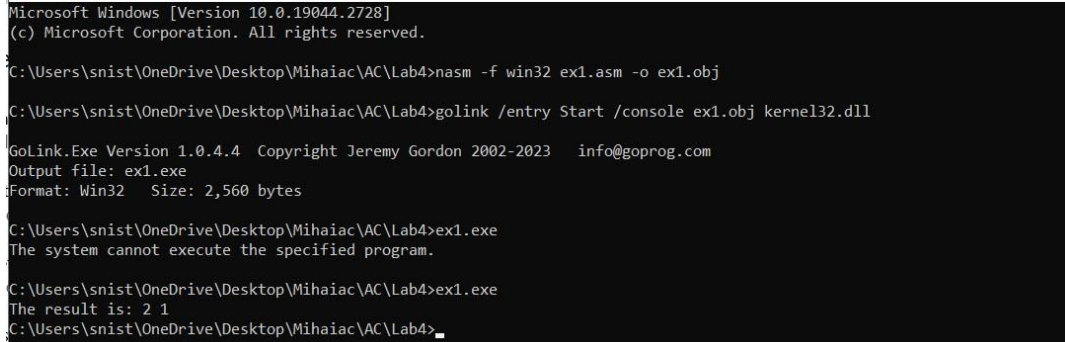
xor EDX, EDX
div dword [EBP-8]
add DL, '0'
mov byte [result_str+eax], DL
inc dword [EBP-4]
test EAX, EAX
jnz .Loop

```

```

mov EAX, dword [EBP-4]
add ESP, 8
pop EBP
ret

```



```

Microsoft Windows [Version 10.0.19044.2728]
(c) Microsoft Corporation. All rights reserved.

C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>nasm -f win32 ex1.asm -o ex1.obj
C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>golink /entry Start /console ex1.obj kernel32.dll
GoLink.Exe Version 1.0.4.4 Copyright Jeremy Gordon 2002-2023 info@goprog.com
Output file: ex1.exe
Format: Win32 Size: 2,560 bytes
C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>ex1.exe
The system cannot execute the specified program.
C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>ex1.exe
The result is: 2 1
C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>_

```

*Output for the addition program*

## Problem 2

This code writes a message to the standard output and then exits. It defines two constants, `NULL` and `STD_OUTPUT_HANDLE`, using the `EQU` directive. It declares three external functions, `GetStdHandle@4`, `WriteFile@20`, and `ExitProcess@4`, and sets the `Start` label as the entry point of the program. The program pushes `STD_OUTPUT_HANDLE` onto the stack and calls the `GetStdHandle@4` function to retrieve the standard output handle. It then writes the `Message` string to standard output using the `WriteFile@20` function. Finally, it exits the program using the `ExitProcess@4` function.

```

NULL EQU 0
STD_OUTPUT_HANDLE EQU -11

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

global Start

section .data
    Message db "I don't like to write code in .asm", 0
    MessageLength EQU $-Message

```

```

section .bss

    StandardHandle resd 1
    Written resd 1

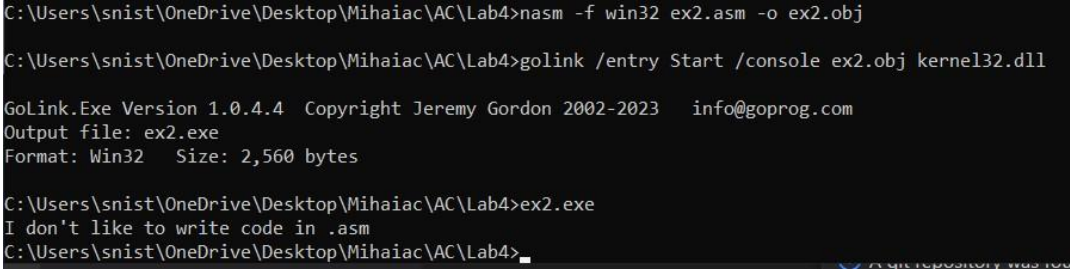
section .text
Start:

    ; Get the standard output handle
    push STD_OUTPUT_HANDLE
    call _GetStdHandle@4
    mov dword [StandardHandle], EAX

    ; Write the message to standard output
    push NULL
    push Written
    push MessageLength
    push Message
    push dword [StandardHandle]
    call _WriteFile@20

    ; Exit the program
    push NULL
    call _ExitProcess@4

```



```

C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>nasm -f win32 ex2.asm -o ex2.obj
C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>golink /entry Start /console ex2.obj kernel32.dll

Golink.Exe Version 1.0.4.4 Copyright Jeremy Gordon 2002-2023 info@goprog.com
Output file: ex2.exe
Format: Win32 Size: 2,560 bytes

C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>ex2.exe
I don't like to write code in .asm
C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>

```

*Output for the addition program*

### Problem 3

This code calculates the sum of an array of integers, converts the result to a string, and writes a message and the result string to standard output. It defines two constants using the EQU directive and declares three external functions. The Start label is defined as the entry point of the program and retrieves the standard output handle before calculating the sum of the array. The DecimalToString function converts an integer value to a string of decimal digits. The program then writes a message and the result string to standard output before exiting.

```
NULL EQU 0
STD_OUTPUT_HANDLE EQU -11

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

global Start

section .data
    Message db "The sum of the array is:", 0
    MessageLength EQU $-Message
    array dd 1, 7, 3, 4, 5
    array_size equ ($ - array) / 4

section .bss
    StandardHandle resd 1
    Written resd 1
    result dd 0
    result_str resb 11

section .text
Start:
    ; Get the standard output handle
```

```

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

; Calculate the sum of the array
mov ecx, array_size
xor eax, eax
mov esi, array
add_loop:
    add eax, [esi]
    add esi, 4
    loop add_loop
mov [result], eax

; Convert the result to a string
mov EAX, [result]
call DecimalToString

; Write the message and the result to standard output
push NULL
push Written
push MessageLength
push Message
push dword [StandardHandle]
call _WriteFile@20
push NULL
push Written
push 11
push result_str
push dword [StandardHandle]
call _WriteFile@20

; Exit the program
push NULL

```

```
call _ExitProcess@4
```

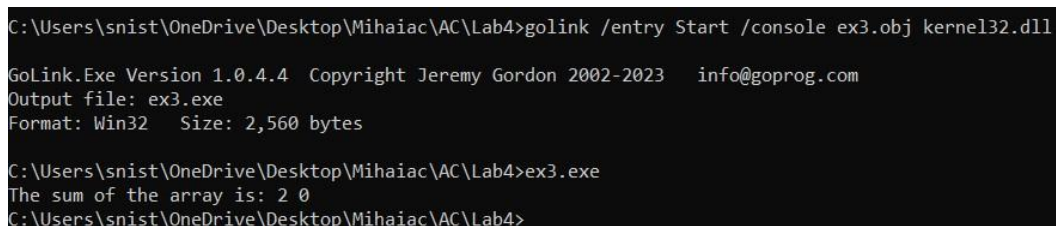
DecimalToString:

```
push EBP
mov EBP, ESP
sub ESP, 8
mov dword [EBP-4], 0
mov dword [EBP-8], 10
```

.Loop:

```
xor EDX, EDX
div dword [EBP-8]
add DL, '0'
mov byte [result_str+eax], DL
inc dword [EBP-4]
test EAX, EAX
jnz .Loop
```

```
mov EAX, dword [EBP-4]
add ESP, 8
pop EBP
ret
```



```
C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>golink /entry Start /console ex3.obj kernel32.dll
GoLink.Exe Version 1.0.4.4 Copyright Jeremy Gordon 2002-2023 info@goprog.com
Output file: ex3.exe
Format: Win32 Size: 2,560 bytes
C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>ex3.exe
The sum of the array is: 2 0
C:\Users\snist\OneDrive\Desktop\Mihaiac\AC\Lab4>
```

*Output for the addition program*

#### Task 4

Debugging code written in NASM can be a difficult task, but there are ways to make it simpler. The GNU Debugger (GDB) is a commonly used tool for debugging, allowing you to analyze the program by stepping through instructions, checking variable and memory values, and setting breakpoints. Another popular debugger is the OllyDbg, which is a graphical debugger that also lets you step through code and analyze variables and memory. Writing clear, organized code with descriptive variable names and comments can

also make debugging easier. Testing the code in smaller sections and printing values or messages to the console can be helpful for tracking the program's progress. To effectively debug in NASM, you need patience, attention to detail, and a thorough understanding of the code.



## Conclusions

In conclusion, this lab covers the basics of assembly language programming and debugging in NASM. Assembly Language is a low-level programming language that is used to write programs for computers, microcontrollers, and other devices. It is a powerful language for writing optimized and efficient code, but also requires a deep understanding of the underlying hardware architecture. NASM is a popular assembler used to write code in assembly language. The lab provides information on how to install NASM, Sublime Text, and Golink on a Windows system and how to configure Sublime Text for assembly language programming. The lab also includes three code examples: one that writes a message to standard output, another that calculates the sum of an array of integers and writes the result to standard output, and a third that adds two numbers and writes the result to standard output. The code examples demonstrate the use of constants, external functions, and loops in assembly language programming. Debugging in NASM can be a difficult task, but the lab provides tips and tools for making it easier, such as using the GNU Debugger (GDB) or the OllyDbg, writing clear and organized code with descriptive variable names and comments, testing code in smaller sections, and printing values or messages to the console. Overall, the lab provides a good introduction to assembly language programming and debugging in NASM.