

**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.4 : Assembly Language  
*Computer Architecture*

Executed by: Pereteatcu Arina st. gr. FAF-223

Verified by:  
univ. assist. Voitcovschi Vladislav

Chişinău - 2024

## Objectives

- 1 Familiarize with the basics of Assembly Language
2. Install NASM
3. Write simple programs
4. Debug your programs
5. Confirm the completion by text or screenshot, and write 3 programs in NASM with the theme of your choice and comment each line of code it executes.

## Theory Overview

The microprocessor in every personal computer is responsible for handling its control, logic, and arithmetic functions.

Every processor family has a unique set of instructions for managing different tasks like entering data via the keyboard, showing data on the screen, and carrying out different tasks. 'Machine language instructions' is the term used to describe these instructions.

Only machine language instructions, which are just sequences of 1s and 0s, are understood by processors. Machine language is too complicated and cryptic to be used in software development, nevertheless. Thus, low-level ***assembly language*** is meant to describe different instructions in symbolic code in a more comprehensible format for a particular family of processors.

An assembly program can be divided into:

**-data** section - Declarations of initialized data or constants go in the data section. There are no runtime changes to this data

**-bss** section -Used for declaring variables

**-text** section-The actual code is stored in the text area. The declaration global \_start, which instructs the kernel where program execution begins, must come first in this section.

Assembly language comment begins with a semicolon (;)

The system memory is divided into groups of independent segments by a segmented memory model, with pointers to the segments contained in the segment registers. A certain kind of data is intended to be contained in each section. The data elements are stored in a separate segment, the program stack is kept in a third segment, and instruction codes are kept in one segment.

Considering the conversation above, we can designate different memory portions as –

**Data segment:** The .data section and the .bss file describe it. The memory space where the program's data items are kept is declared in the .data section. Once the data items are declared, this part cannot be expanded and stays the same throughout the program.

Another portion of static memory, the .bss section holds buffers for data that will be declared later in the program. There are zeros in this buffer's memory.

**Section of code:** The .text section is used to represent it. This designates a region of memory where the instruction codes are kept. This space is likewise fixed.

**Stack:** The data values given to the program's functions and procedures are included in this segment.

The CPU has certain registers—internal memory storage locations—to speed up processor processes. Data items are stored in the **registers** so they can be processed without accessing the memory. The CPU chip has a finite amount of registers built in.

## Implementation:

### Installing NASM

```
C:\Users\arina>nasm --version
NASM version 2.14.03rc2 compiled on Dec 30 2018

C:\Users\arina>gcc --version
gcc (MinGW.org GCC Build-2) 9.2.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

## Simple programs:

### First Program:

```
global  _main                ; declare main() method
extern  _printf              ; link to external library

segment .data
message: db  'Hello world', 0xA, 0 ; text message
        ; 0xA (10) is hex for (NL), carriage return
        ; 0 terminates the line

        ; code is put in the .text section
section .text
_main:                                ; the entry point! void main()
    push    message                  ; save message to the stack
    call    _printf                  ; display the first value on the stack
    add     esp, 4                   ; clear the stack
    ret                                ; return
```

This assembly code is a simple program that prints the message "Hello world" to the console.

## Global and Extern Declarations:

*global \_main:* Declares the `_main` label as the entry point for the program.

*extern \_printf:* Declares that `_printf` is an external symbol that will be linked from a library. In this case, it's expected to be a function provided by the C standard library.

## Data Segment:

*.data segment:* This is where static data is declared.

*message:* Declares a null-terminated string "Hello world" followed by a newline character (0xA) and a null terminator (0). This message will be displayed by the `_printf` function.

## Text Segment:

*.text segment:* This is where executable code is placed.

*\_main label:* This is the entry point of the program.

*push message:* Pushes the address of the message string onto the stack. The stack grows downward, so the address of the first character of the string is pushed last.

*call \_printf:* Calls the `_printf` function, passing it the address of the message string.

*add esp, 4:* Clears the stack by adjusting the stack pointer (esp) back by 4 bytes. This removes the address of the message string from the stack.

*ret:* Returns from the `_main` function.

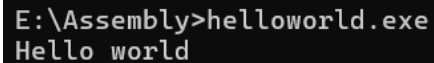
## **Debugging**

```
Copyright (C) 2013 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 "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from E:\Assembly\helloworld.exe...done.
(gdb) b _main+4 ; Before displaying the prompt for the first integer
Function "_main+4" ; Before displaying the prompt for the first integer" not defined.
Make breakpoint pending on future shared library load? (y or [n]) b _main+14 ; After reading the first integer
Please answer y or [n].
Make breakpoint pending on future shared library load? (y or [n]) b _main+24 ; Before displaying the prompt for the sec
ond integer
Please answer y or [n].
Make breakpoint pending on future shared library load? (y or [n]) b _main+34 ; After reading the second integer
Please answer y or [n].
Make breakpoint pending on future shared library load? (y or [n]) b _main+44 ; Before calculating the product
Please answer y or [n].
Make breakpoint pending on future shared library load? (y or [n]) b _main+53 ; Before displaying the result
Please answer y or [n].
Make breakpoint pending on future shared library load? (y or [n]) b _main+60 ; Before exiting the program
Please answer y or [n].
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (_main+4 ; Before displaying the prompt for the first integer) pending.
(gdb) run
Starting program: E:\Assembly\helloworld.exe
[New Thread 26300.0x1af8]
[New Thread 26300.0x1444]
Hello world
[Inferior 1 (process 26300) exited with code 014]
```

Figure 1. Debug first program

## Output



```
E:\Assembly>helloworld.exe
Hello world
```

*Figure2. Output first program*

## Second Program:

```
section .data
    prompt1 db "Enter the first integer: ", 0
    prompt2 db "Enter the second integer: ", 0
    format db "%d", 0
    result db "The product is: %d", 0

section .bss
    num1 resd 1
    num2 resd 1
    product resd 1

section .text
    extern _printf, _scanf, _exit

global _main

_main:
    ; Display prompt for first integer
    push prompt1
    call _printf
    add esp, 4

    ; Read first integer
    lea eax, [num1]
    push eax
    push format
    call _scanf
    add esp, 8

    ; Display prompt for second integer
    push prompt2
    call _printf
    add esp, 4

    ; Read second integer
    lea eax, [num2]
```

```

push eax
push format
call _scanf
add esp, 8

; Calculate product
mov eax, [num1]
imul eax, [num2]
mov [product], eax

; Display the result
push dword [product]
push result
call _printf
add esp, 8

; Exit the program
push 0
call _exit

```

This program prompts the user to enter two integers, multiplies them, and then displays the product.

#### Data Section:

*.data section:* This section contains static data such as strings.

*prompt1 and prompt2:* Strings prompting the user to enter the first and second integers respectively.

*format:* A format string for scanf to read integers.

*result:* A string indicating the format of the output message displaying the product.

#### BSS Section:

*.bss section:* This section is typically used for statically-allocated variables that are uninitialized.

*num1 and num2:* Reserved space (4 bytes each) to store the two integers entered by the user.

*product:* Reserved space (4 bytes) to store the product of the two integers.

#### Text Section:

*.text section:* This section contains executable instructions.

#### External Declarations:

*extern \_printf, \_scanf, \_exit:* Declares external symbols \_printf, \_scanf, and \_exit, which are functions provided by an external library (likely the C standard library).

#### Global Declaration:

*global \_main*: Declares the *\_main* label as the entry point of the program.

#### Main Function ( *\_main* ):

Pushes the address of *prompt1* onto the stack and calls *\_printf* to display the prompt for the first integer.

Uses *scanf* to read the first integer entered by the user and stores it in *num1*.

Similar steps are repeated for the second integer (*prompt2* and *num2*).

#### Calculating the Product:

Uses the *imul* (integer multiply) instruction to multiply the values stored in *num1* and *num2*, and stores the result in the *eax* register.

The product is then moved from *eax* to the product variable.

#### Displaying the Result:

Pushes the address of *product* and the address of *result* onto the stack.

Calls *\_printf* to display the product using the format specified in *result*.

#### Exiting the Program:

Pushes 0 onto the stack and calls *\_exit* to terminate the program.

## Debugging

```
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (_main) pending.
(gdb) run
Starting program: E:\Assembly\product.exe
[New Thread 22972.0x5f64]
[New Thread 22972.0x63bc]
Enter the first integer: 2
Enter the second integer: 4
The product is: 8[Inferior 1 (process 22972) exited normally]
(gdb)
```

*Figure3. Output second program*

## Output

```
E:\Assembly>product.exe
Enter the first integer: 1
Enter the second integer: 2
The product is: 2
E:\Assembly>product.exe
Enter the first integer: 5
Enter the second integer: 4
The product is: 20
E:\Assembly>
```

*Figure4. Output second program*

### Third Program:

```
section .data
    prompt1 db "Enter the first integer: ", 0
    prompt2 db "Enter the second integer: ", 0
    format db "%d", 0
    result db "The difference is: %d", 0

section .bss
    num1 resd 1
    num2 resd 1
    difference resd 1

section .text
    extern _printf, _scanf, _exit

global _main

_main:
    ; Display prompt for first integer
    push prompt1
    call _printf
    add esp, 4

    ; Read first integer
    lea eax, [num1]
    push eax
    push format
    call _scanf
    add esp, 8

    ; Display prompt for second integer
    push prompt2
    call _printf
    add esp, 4

    ; Read second integer
    lea eax, [num2]
    push eax
    push format
    call _scanf
    add esp, 8

    ; Calculate difference
    mov eax, [num1]
```



```

sub eax, [num2]
mov [difference], eax

; Display the result
push dword [difference]
push result
call _printf
add esp, 8

; Exit the program
push 0
call _exit

```

This assembly code prompts the user to enter two integers, calculates their difference, and then displays the result.

### Data Section:

*.data section:* This section contains static data such as strings.

*prompt1 and prompt2:* Strings prompting the user to enter the first and second integers respectively.

*format:* A format string for scanf to read integers.

*result:* A string indicating the format of the output message displaying the difference.

### BSS Section:

*.bss section:* This section is typically used for statically-allocated variables that are uninitialized.

*num1 and num2:* Reserved space (4 bytes each) to store the two integers entered by the user.

*difference:* Reserved space (4 bytes) to store the difference between the two integers.

### Text Section:

*.text section:* This section contains executable instructions.

### External Declarations:

*extern \_printf, \_scanf, \_exit:* Declares external symbols \_printf, \_scanf, and \_exit, which are functions provided by an external library (likely the C standard library).

### Global Declaration:

*global \_main:* Declares the \_main label as the entry point of the program.

### Main Function ( \_main):

Pushes the address of prompt1 onto the stack and calls \_printf to display the prompt for the first integer.

Uses scanf to read the first integer entered by the user and stores it in num1.

Similar steps are repeated for the second integer (prompt2 and num2).

#### Calculating the Difference:

Subtracts the value stored in num2 from the value stored in num1 using the sub (subtract) instruction.

Stores the result in the difference variable.

#### Displaying the Result:

Pushes the address of difference and the address of result onto the stack.

Calls \_printf to display the difference using the format specified in result.

#### Exiting the Program:

Pushes 0 onto the stack and calls \_exit to terminate the program.

## Debugging

```
E:\Assembly>gdb difference.exe
GNU gdb (GDB) 7.6.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
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 "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from E:\Assembly\difference.exe...done.
(gdb) break _main
Function "_main" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (_main) pending.
(gdb) run
Starting program: E:\Assembly\difference.exe
[New Thread 26728.0x4744]
[New Thread 26728.0x49d8]
Enter the first integer: 4
Enter the second integer: 3
The difference is: 1[Inferior 1 (process 26728) exited normally]
```

*Figure5.Debug of third program*

## Output

```
E:\Assembly>difference.exe
Enter the first integer: 4
Enter the second integer: 1
The difference is: 3
E:\Assembly>difference.exe
Enter the first integer: 3
Enter the second integer: 4
The difference is: -1
E:\Assembly>|
```

*Figure6.Output third program*

**Conclusion:**

Our goal in this lab exercise was to become acquainted with the fundamentals of Assembly Language, which included setting up NASM, creating basic programs, and debugging them. We learned how microprocessors do arithmetic, logic, and control operations as well as how machine language instructions are necessary for these operations through this procedure.

We discovered that, in contrast to machine language, assembly language offers a more comprehensible framework for low-level programming. A typical assembly program has sections like the text, data, and bss sections, each of which has a specific function in arranging the program's variables, data, and code.

We partitioned memory during the laboratory work into segments, each one reserved for a certain kind of data (e.g., the text segment contained instruction codes and the data segment contained initialized data). We also looked at the idea of registers, which are CPU internal memory storage areas that are used to speed up processing operations.

We successfully developed and debugged assembly programs by finishing this lab work, which gave us a great understanding of the complexities of low-level programming. These fundamental skills will be a great starting point for future research into and comprehension of computer architecture and systems programming.