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

Laboratory work 4:

Introduction in NASM

Elaborat:
st. gr. FAF-221 Berco Andrei

Verificat:

asist. Univ. Voitcovschi Vladislav

Chişinău - 2024

Purpose of the work:

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.
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.
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.

1. Fundamentals of Assembly Language

Assembly Language serves as a fundamental layer of programming, allowing direct interaction with a computer's hardware. Unlike high-level programming languages, Assembly Language provides a close correspondence between the instructions written by the programmer and the operations executed by the CPU. This direct control over hardware makes it powerful for tasks requiring optimization, real-time processing, or low-level system programming.

- Registers:

Registers are small, high-speed storage locations within the CPU used for temporary data manipulation. They play a crucial role in Assembly Language programming by facilitating arithmetic, logical, and data transfer operations. For example, in x86 architecture:

```
MOV AX, 10     ; Move the value 10 into the AX register

ADD BX, AX     ; Add the value of AX to the BX register
```

Here, the MOV instruction moves the value 10 into the AX register, and the ADD instruction adds the value of AX to the BX register.

- Memory:

Memory in Assembly Language refers to the primary storage of a computer system where data and instructions are stored during program execution. Accessing memory involves specifying memory addresses and using load and store operations to read from or write to those addresses. For instance:

```
MOV [memory_address], AX      ; Move the value of AX register to a memory
location

MOV CX, [memory_address]      ; Move the value from a memory location to CX
register
```

In these examples, memory_address represents a specific memory location, and MOV instructions are used to transfer data between registers and memory.

- Instructions:

Assembly Language instructions are low-level commands that direct the CPU to perform specific operations. These instructions are represented by mnemonics, which are human-readable abbreviations for machine operations. Examples include:

```
MOV AX, BX     ; Move data from BX register to AX register
ADD AX, 10     ; Add the value 10 to AX register
JMP label      ; Jump to a specified label in the code
```

Each instruction performs a distinct operation, such as data movement, arithmetic, or control flow manipulation.

- Syntax:

Assembly Language syntax varies depending on the specific architecture and assembler being used. However, it generally consists of mnemonic instructions followed by operands and optional comments. For example:

```
; Calculate the sum of two numbers and store the result in AX register
MOV AX, 5      ; Move the value 5 into AX register
ADD AX, 10     ; Add the value 10 to AX register
```

In this example, MOV and ADD are instructions, AX and 10 are operands, and the semicolon ; denotes a comment.

- Data Types:

Assembly Language supports various data types, including integers, characters, strings, and floating-point numbers. Data types are represented using different instruction sets and formats depending on the architecture. For instance:

```
DB  'A'         ; Define a single character
DW  1234        ; Define a 2-byte integer
DD  12345678    ; Define a 4-byte integer
```

These directives allocate memory and specify the data type for storage.

- Control Flow:

Control flow in Assembly Language governs the sequence of instructions executed by the CPU based on conditions and branching instructions. Examples include:

```
CMP AX, BX     ; Compare the values in AX and BX registers
JE  label      ; Jump to a label if the previous comparison was equal
```

These instructions enable decision-making and looping constructs in Assembly programs, allowing for flexible program control.

2. Install NASM

- Firstly, I installed NASM from thje official site. Next, I put it in PATH variable, so I coul acces it from anywhere. Using *nasm -v,* I can check its version.

```
C:\Users\Victor\Desktop\AC>nasm -v
NASM version 2.16.01 compiled on Dec 21 2022
```

3. Simple programs + Debbuging

First program:

```
section .data
    hello db 'Hello, World!', 10  ; String to be printed, with newline character (10)

section .text
    global _start   ; Entry point for the program

_start:
    ; System call to write to standard output (file descriptor 1)
    mov rax, 1      ; syscall number for sys_write
    mov rdi, 1      ; file descriptor 1 (stdout)
    mov rsi, hello  ; pointer to the string
    mov rdx, 13     ; length of the string
    syscall         ; invoke the system call

    ; System call to exit the program
    mov rax, 60     ; syscall number for sys_exit
    xor rdi, rdi    ; exit code 0
    syscall         ; invoke the system call
```

Figure 1. Program to print something to console

Console Output and Debugging with GDB:

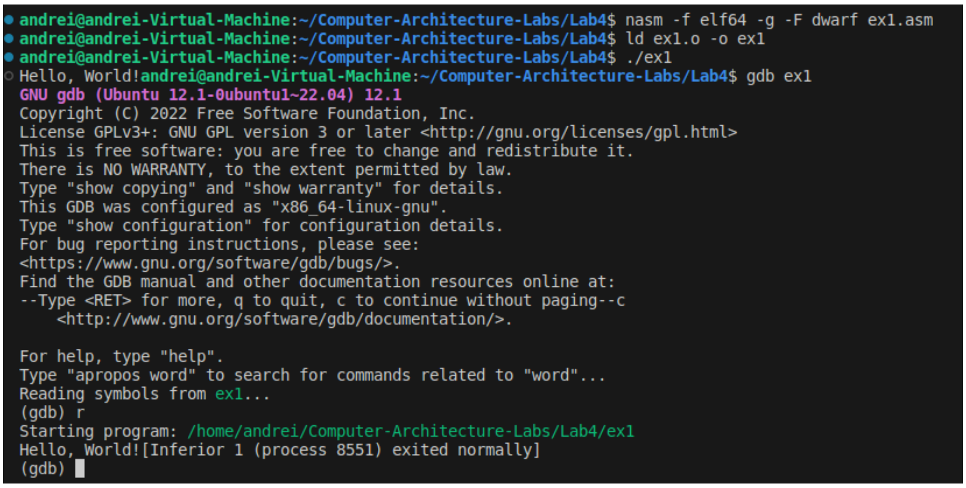

Figure 2. Debug info

Console instructions:

nasm –f elf64 -g –F dwarf -o ex1.o ex1.asm

ld ex1.o -o ex1

./ex1

gdb ./ex1

b _start

Code Explanation:

This assembly code is a simple program that prints the string "Hello, World!" followed by a newline character to the standard output. Here's a brief explanation:

Data Section:

hello: Stores the string "Hello, World!" followed by a newline character (10).

Text Section:

The _start label marks the beginning of the program.

It uses the sys_write system call to print the string stored in the hello variable to standard output (stdout).

The mov instructions set up the registers rax, rdi, rsi, and rdx with the appropriate values for the sys_write system call.

rax is set to 1, indicating the syscall number for sys_write.

rdi is set to 1, indicating stdout (file descriptor 1).

rsi is set to the memory address where the string is stored.

rdx is set to 13, indicating the length of the string.

The syscall instruction is then used to invoke the system call and print the string to the standard output.

Next, it uses another system call, sys_exit, to terminate the program with an exit code of 0. The exit code is set to 0 by xor rdi, rdi, indicating successful termination.

Second program:

Take the user input (name) and print “Hello ” [name]

```
section .data
    prompt db 'Enter your name: ', 0    ; Prompt message
    greeting db 'Hello, ', 0             ; Greeting message prefix
    newline db 10, 0                     ; Newline character for printing

section .bss
    name resb 32                         ; Buffer to store user's name (up
to 31 characters)

section .text
    global _start

_start:
    ; Display prompt message
    mov rax, 1                ; syscall number for sys_write
    mov rdi, 1                ; file descriptor 1 (stdout)
    mov rsi, prompt           ; pointer to the prompt message
    mov rdx, 17               ; length of the prompt message
    syscall                   ; invoke the system call

    ; Read user input
    mov rax, 0                ; syscall number for sys_read
    mov rdi, 0                ; file descriptor 0 (stdin)
    mov rsi, name             ; pointer to the buffer
    mov rdx, 32               ; maximum number of bytes to read
```

```
syscall                     ; invoke the system call

; Print greeting
mov rax, 1                  ; syscall number for sys_write
mov rdi, 1                  ; file descriptor 1 (stdout)
mov rsi, greeting           ; pointer to the greeting prefix
mov rdx, 7                  ; length of the greeting prefix
syscall                     ; invoke the system call

; Print the user's name
mov rax, 1                  ; syscall number for sys_write
mov rdi, 1                  ; file descriptor 1 (stdout)
mov rsi, name               ; pointer to the user's name
syscall                     ; invoke the system call

; Print newline character
mov rax, 1                  ; syscall number for sys_write
mov rdi, 1                  ; file descriptor 1 (stdout)
mov rsi, newline            ; pointer to the newline character
mov rdx, 1                  ; length of the newline character
syscall                     ; invoke the system call

; Exit the program
mov rax, 60                 ; syscall number for sys_exit
xor rdi, rdi                ; exit code 0
syscall                     ; invoke the system call
```

Console Output and Debugging with GDB:

```
andrei@andrei-Virtual-Machine:~/Computer-Architecture-Labs/Lab4$ nasm -f elf64 -g -F dw
arf ex2.asm
andrei@andrei-Virtual-Machine:~/Computer-Architecture-Labs/Lab4$ ld ex2.o -o ex2
andrei@andrei-Virtual-Machine:~/Computer-Architecture-Labs/Lab4$ ./ex2
Enter your name: Andrei
Hello, Andrei

andrei@andrei-Virtual-Machine:~/Computer-Architecture-Labs/Lab4$ gdb ex2
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
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 ex2...
(gdb) b _start
Breakpoint 1 at 0x401000
(gdb) r
Starting program: /home/andrei/Computer-Architecture-Labs/Lab4/ex2

Breakpoint 1, 0x0000000000401000 in _start ()
(gdb) c
Continuing.
Enter your name: Andrei
Hello, Andrei

[Inferior 1 (process 8969) exited normally]
(gdb) 
```

Figure 3. Debug info

Code Explanation:

This assembly code is a simple program that prompts the user to enter their name, reads the input, and then greets the user by printing "Hello, " followed by their name. Here's a brief explanation of the key parts:

Data Section:

prompt: Stores the message "Enter your name: " with a null terminator.

greeting: Stores the message prefix "Hello, " with a null terminator.

newline: Stores the newline character (ASCII value 10) followed by a null terminator.

BSS Section:

name: Reserves space to store the user's name, with a maximum length of 31 characters (plus null terminator).

Text Section:

The _start label marks the beginning of the program.

It displays the prompt message using the sys_write system call.

It reads the user's input using the sys_read system call.

It prints the greeting prefix "Hello, " using sys_write.

It prints the user's name stored in the name buffer using sys_write.

It prints a newline character to move to the next line.

Finally, it exits the program using the sys_exit system call with an exit code of 0.

Third program:

Calculate the sum of 2 digits taken from the input

```
; Define system call numbers

SYS_EXIT  equ 1

SYS_READ  equ 3

SYS_WRITE equ 4

STDIN     equ 0

STDOUT    equ 1


section .data

    ; Define message to prompt user to enter a digit

    msg db "Enter a digit ", 0xA,0xD   ; Newline and carriage return
characters

    len equ $- msg                      ; Calculate length of message


section .bss

    ; Reserve space for variables

    number1 resb 2                      ; Reserve space for the first
number

    number2 resb 2                      ; Reserve space for the second
number

    result resb 1                       ; Reserve space for the result


section .text

    ; Define another message to prompt user to enter second digit
```

```
    msg2 db "Enter second digit", 0xA,0xD
    len2 equ $- msg2

    ; Define message for displaying the sum
    msg3 db "The sum is: "
    len3 equ $- msg3

global _start

_start:
    ; Prompt user to enter the first digit
    mov eax, SYS_WRITE          ; System call to write
    mov ebx, STDOUT             ; File descriptor for standard output
    mov ecx, msg                ; Address of the message
    mov edx, len                ; Length of the message
    int 0x80                    ; Invoke the system call

    ; Read the first digit entered by the user
    mov eax, SYS_READ           ; System call to read
    mov ebx, STDIN              ; File descriptor for standard input
    mov ecx, number1            ; Address to store the input
    mov edx, 2                  ; Number of bytes to read
    int 0x80                    ; Invoke the system call

    ; Prompt user to enter the second digit
    mov eax, SYS_WRITE          ; System call to write
    mov ebx, STDOUT             ; File descriptor for standard output
    mov ecx, msg2               ; Address of the message
    mov edx, len2               ; Length of the message
    int 0x80                    ; Invoke the system call

    ; Read the second digit entered by the user
    mov eax, SYS_READ           ; System call to read
    mov ebx, STDIN              ; File descriptor for standard input
```

```
    mov ecx, number2            ; Address to store the input
    mov edx, 2                  ; Number of bytes to read
    int 0x80                    ; Invoke the system call

    ; Display message indicating sum calculation
    mov eax, SYS_WRITE          ; System call to write
    mov ebx, STDOUT             ; File descriptor for standard output
    mov ecx, msg3               ; Address of the message
    mov edx, len3               ; Length of the message
    int 0x80                    ; Invoke the system call

    ; Load number1 into eax and subtract '0' to convert from ASCII to
decimal
    mov eax, [number1]
    sub eax, '0'
    ; Do the same for number2
    mov ebx, [number2]
    sub ebx, '0'

    ; Add eax and ebx, storing the result in eax
    add eax, ebx
    ; Add '0' to eax to convert the digit from decimal to ASCII
    add eax, '0'

    ; Store the result in result
    mov [result], eax

    ; Print the result digit
    mov eax, SYS_WRITE          ; System call to write
    mov ebx, STDOUT             ; File descriptor for standard output
    mov ecx, result             ; Address of the result
    mov edx, 1                  ; Length of the result
    int 0x80                    ; Invoke the system call
```

```
exit:
    ; Exit the program
    mov eax, SYS_EXIT          ; System call to exit
    xor ebx, ebx               ; Exit code 0
    int 0x80                   ; Invoke the system call
```

Console Output and Debugging with GDB:

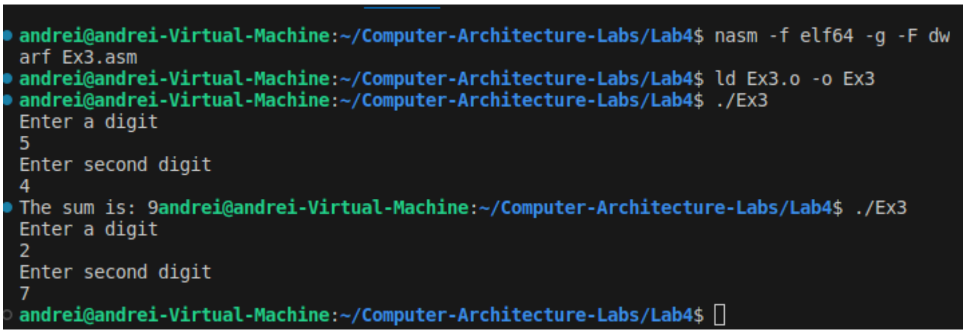

Figure 4. Console output

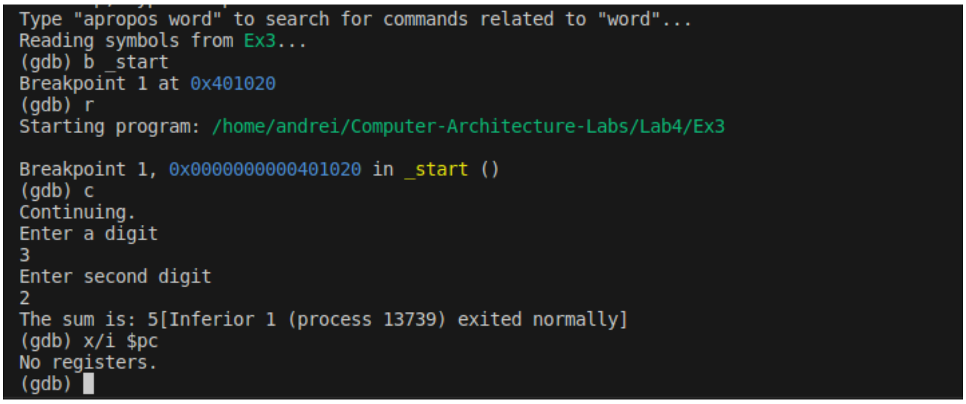

Figure 5. Debug info

Code Explanation:

This assembly code is a simple program that prompts the user to enter two digits, reads them from the standard input, calculates their sum, and then displays the result. Here's a brief explanation of the key parts:

Definitions: The code starts by defining constants for system call numbers and file descriptors.

Data Section: Messages are defined to prompt the user for input and to display the sum.

Text Section:

The _start label marks the beginning of the program.

The program prompts the user to enter the first digit and reads it from standard input.

Then, it prompts for the second digit and reads it.

Next, it displays a message indicating the sum calculation.

It converts ASCII digits to their numerical equivalents.

It adds the two numbers together.

Finally, it converts the result back to ASCII and prints it to standard output.

Exit: The program exits by making a system call to terminate the process.

Conclusions:

In conclusion, this report provides a comprehensive overview of the fundamentals of Assembly Language and the initial steps to get started with NASM (Netwide Assembler). Before delving into NASM, understanding the basic concepts of Assembly Language is crucial, including registers, memory, instructions, syntax, data types, and control flow.

Once equipped with this foundational knowledge, the installation of NASM is straightforward. By downloading and configuring NASM for their respective platforms, programmers can begin writing and assembling assembly code. The report emphasizes the significance of starting with simple programs to gain familiarity with NASM's syntax and functionality. These initial programs may involve printing messages, reading user input, or performing basic arithmetic operations.

Moreover, debugging is highlighted as an essential aspect of programming in NASM. Utilizing debugging tools such as GDB enables programmers to identify and rectify errors efficiently, enhancing their understanding of the code and its execution flow.

Ultimately, Assembly Language serves as a powerful tool for low-level system programming, offering direct control over hardware and enabling optimization and real-time processing. With this report as a guide, aspiring programmers can embark on their journey to master NASM and leverage the capabilities of Assembly Language for various applications.