MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work n.4

Computer Architectures

Verificat:

Voitcovschi Vladislav *asist.univ*

Chișinău – 2024

1. Purpose of the task work:

Write 3 programs in NASM with the theme of your choice and comment each line of code it executes.

2. Introduction:

In the realm of computer science and software engineering, understanding the fundamental concepts of assembly language programming is paramount. Assembly language serves as a bridge between high-level programming languages and the machine code executed by the central processing unit (CPU). In this laboratory work report, we delve into the basics of assembly language and its implementation using NASM (Netwide Assembler).

Assembly language, often referred to simply as assembly, provides a low-level representation of computer programs. Unlike high-level languages such as Python or C, assembly language directly corresponds to the machine instructions understood by the CPU. It offers programmers precise control over hardware resources and facilitates optimization for performance-critical applications.

NASM, a widely used assembler in the realm of x86 and x86-64 architectures, plays a pivotal role in translating assembly code into machine code. It provides a robust set of features and directives that aid in the development of efficient and portable assembly programs. In this report, we will explore the fundamental concepts of assembly language programming, elucidate the role of NASM in assembling code, and demonstrate practical examples to reinforce comprehension.

By the end of this laboratory work, readers will have gained a solid foundation in assembly programming and proficiency in utilizing NASM as a tool for code development.

3. Implementation:

Hello world program:

```
section .data
    hello db 'Hello, World!', 0 ; Null-terminated string

section .text
    global _start

_start:
    ; Write the string to stdout
    mov eax, 4          ; System call number for sys_write
    mov ebx, 1          ; File descriptor 1 (stdout)
    mov ecx, hello      ; Pointer to the string
    mov edx, 13         ; Length of the string
    int 0x80            ; Call kernel

    ; Exit the program
    mov eax, 1          ; System call number for sys_exit
    xor ebx, ebx        ; Exit code 0
    int 0x80            ; Call kernel
```

In this code snippet, we first declare a section called .data, which is typically used to define initialized data. Here, we define a variable named hello as a null-terminated string containing the text "Hello, World!". The db directive is used to declare bytes, and we terminate the string with a null character (ASCII value 0).

Moving on to the .text section, this is where the actual program instructions reside. We start by declaring the entry point of the program using the _start label. mov eax, 4: We load the system call number for sys_write into the eax register. The sys_write system call is responsible for writing data to a file descriptor, in this case, stdout (standard output). mov ebx, 1: We set the file descriptor to 1, indicating that we want to write to stdout. mov ecx, hello: We load the memory address of the hello string into the ecx register. This register will point to the start of the string we want to write. mov edx, 13: We specify the length of the string to be written. In this case, the string "Hello, World!" consists of 12 characters plus the null terminator, so the length is 13. int 0x80: This instruction triggers a software interrupt, which causes the CPU to switch to kernel mode. The kernel then executes the system call indicated by the value in the eax register (in this case, sys_write), with the provided parameters (ebx, ecx, edx). After writing the string to stdout, we prepare to exit the program. mov eax, 1: We load the system call number for sys_exit into the eax register. The sys_exit system call terminates the program. xor ebx, ebx: We set the exit status code to 0. In POSIX systems, a status code of 0 indicates successful termination. int 0x80: Another software interrupt is triggered to execute the sys_exit system call with the provided exit status code (ebx).

Output:

```
Hello, World!
```

CPU Time: 0.00 sec(s) | Memory: 256 kilobyte(s) | Compiled and & executed in 0.513 sec(s)

Sum of two numbers program:

```
1  section .text          ; Code section
2  global _start         ; Global label for program entry point
3
4  _start:               ; Entry point label
5
6      mov     eax, [x]   ; Move value stored at memory address x into register eax
7      sub     eax, '0'   ; Convert ASCII character to integer (subtract ASCII '0')
8      mov     ebx, [y]   ; Move value stored at memory address y into register ebx
9      sub     ebx, '0'   ; Convert ASCII character to integer (subtract ASCII '0')
10     add     eax, ebx   ; Add contents of ebx to eax
11     add     eax, '0'   ; Convert result back to ASCII character
12
13     mov     [sum], eax  ; Move the result to the memory location labeled 'sum'
14
15     mov     ecx, msg   ; Move the address of message string to ecx
16     mov     edx, len   ; Move the length of message string to edx
17     mov     ebx, 1     ; File descriptor for stdout
18     mov     eax, 4     ; Syscall number for sys_write
19     int     0x80       ; Call kernel
20
21     mov     ecx, sum   ; Move the address of 'sum' to ecx
22     mov     edx, 1     ; Length of data to be printed (1 byte)
23     mov     ebx, 1     ; File descriptor for stdout
24     mov     eax, 4     ; Syscall number for sys_write
25     int     0x80       ; Call kernel
26
27     mov     eax, 1     ; Syscall number for sys_exit
28     int     0x80       ; Call kernel
29
30 section .data         ; Data section
31     x db '5'          ; Define variable 'x' with ASCII character '5'
32     y db '3'          ; Define variable 'y' with ASCII character '3'
33     msg db  "sum of x and y is "  ; Define message string
34     len equ $ - msg  ; Calculate length of message string
35
36 segment .bss          ; Uninitialized data section
37     sum resb 1        ; Define variable 'sum' with 1 byte of space
```

This code is more complicated compared to the hello world program so I will make an easy breakdown line by line:

1. Declare the start of the code section.

2. Declare the entry point of the program.

3. Start of the program.

4-8. Read the values stored at memory addresses 'x' and 'y', convert them from ASCII characters to integers, perform addition, and convert the result back to an ASCII character. Save the result in memory.

9-13. Prepare to write a message to the screen: set up the message address and its length, set file descriptor for standard output, and set up the syscall for writing.

14-18. Write the message to the screen using the syscall.

19-23. Prepare to write the sum to the screen: set up the memory address of the sum, length of data to be printed, file descriptor for standard output, and syscall for writing.

24-28. Write the sum to the screen using the syscall.

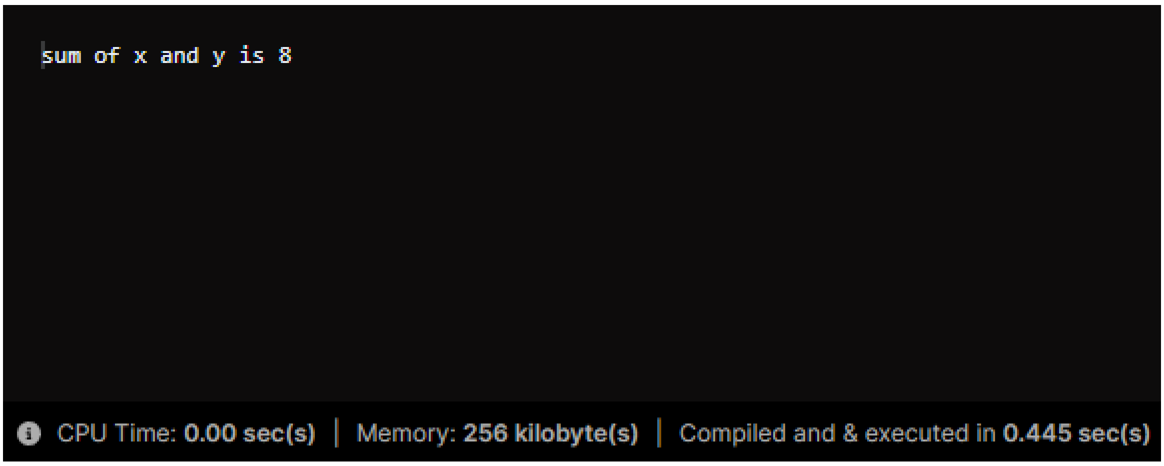
29-32. Prepare to exit the program: set up the syscall for program exit and exit code.

33-34. Exit the program using the syscall.

35-39. Declare the start of the data section and define variables 'x', 'y', and 'msg' with their respective initial values.

40-42. Declare the start of the uninitialized data section and define the variable 'sum' with 1 byte of space reserved for it.

Output:



Subtraction of two numbers program:

```
1  section .text        ; Code section
2  global _start       ; Global label for program entry point
3
4  _start:             ; Entry point label
5
6      mov     eax, [x]    ; Move value stored at memory address x into register eax
7      sub     eax, '0'    ; Convert ASCII character to integer (subtract ASCII '0')
8      mov     ebx, [y]    ; Move value stored at memory address y into register ebx
9      sub     ebx, '0'    ; Convert ASCII character to integer (subtract ASCII '0')
10     sub     eax, ebx    ; Subtract contents of ebx from eax (perform subtraction)
11     add     eax, '0'    ; Convert result back to ASCII character
12
13     mov     [difference], eax  ; Store the result of subtraction in memory location 'difference'
14
15     mov     ecx, msg_diff   ; Move the address of message string for difference to ecx
16     mov     edx, len_diff   ; Move the length of message string for difference to edx
17     mov     ebx, 1          ; File descriptor for stdout
18     mov     eax, 4          ; Syscall number for sys_write
19     int     0x80            ; Call kernel to write the message to stdout
20
21     mov     ecx, difference   ; Move the address of 'difference' to ecx
22     mov     edx, 1            ; Length of data to be printed (1 byte)
23     mov     ebx, 1            ; File descriptor for stdout
24     mov     eax, 4            ; Syscall number for sys_write
25     int     0x80              ; Call kernel to write the result to stdout
26
27     mov     eax, 1            ; Syscall number for sys_exit
28     int     0x80              ; Call kernel to exit the program
29
30 section .data        ; Data section
31     x db '5'         ; Define variable 'x' with ASCII character '5'
32     y db '3'         ; Define variable 'y' with ASCII character '3'
33     msg_diff db  "difference of x and y is "  ; Define message string for difference
34     len_diff equ $ - msg_diff  ; Calculate length of message string for difference
35
36 section .bss         ; Uninitialized data section
37     difference resb 1      ; Define variable 'difference' with 1 byte of space
```

Line by line, this code works approximately like the sum of two numbers program but has some differences:

1. Declare the start of the code section.

2. Declare the global label '_start' as the entry point for the program.

4-8. Read the values stored at memory addresses 'x' and 'y', convert them from ASCII characters to integers, perform subtraction, and convert the result back to an ASCII character. Save the result in a variable called 'difference'.

9-13. Prepare to write a message about the difference to the screen: set up the message address and its length, set file descriptor for standard output, and set up the syscall for writing.

14-18. Write the message about the difference to the screen using the syscall.

19-23. Prepare to write the value of 'difference' to the screen: set up the memory address of 'difference', length of data to be printed, file descriptor for standard output, and syscall for writing.

24-28. Write the value of 'difference' to the screen using the syscall.

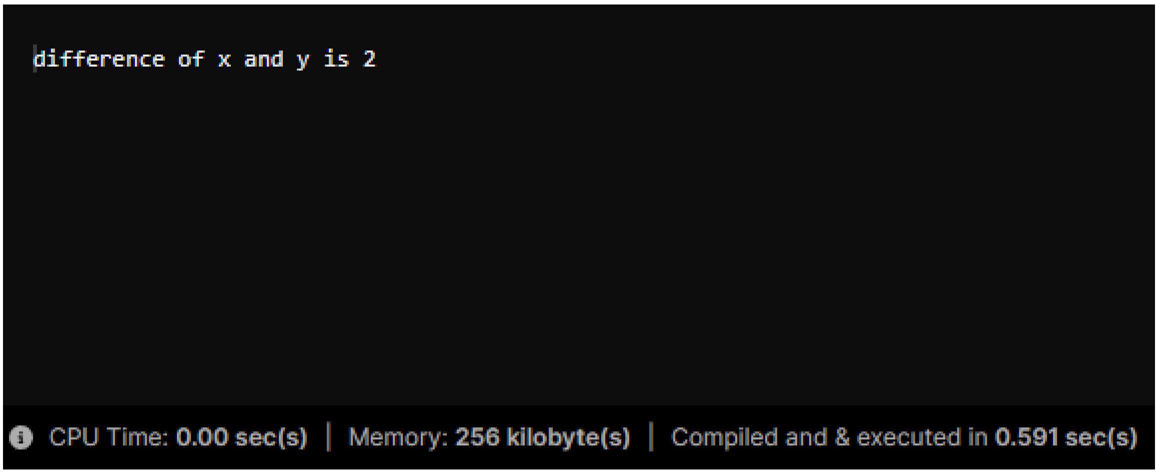
29-32. Prepare to exit the program: set up the syscall for program exit and exit code.

33-34. Exit the program using the syscall.

35-40. Declare the start of the data section and define variables 'x', 'y', 'msg_diff', and 'len_diff' with their respective initial values.

41-45. Declare the start of the uninitialized data section and define the variable 'difference' with 1 byte of space reserved for it.

Output:



4. Conclusion

In conclusion, this laboratory work provided a comprehensive introduction to assembly language programming using NASM (Netwide Assembler). Throughout the sessions, we gained hands-on experience by writing three fundamental programs: printing "Hello, World!", performing addition of two numbers, and conducting subtraction of two numbers.

The first program, which printed "Hello, World!", served as an initial exercise to familiarize ourselves with the basic syntax and structure of NASM assembly language. Through this program, we learned about defining data sections, accessing memory, and invoking system calls to interact with the operating system.

Moving forward, the second program delved into arithmetic operations by implementing addition of two numbers. We grasped the concepts of data manipulation, conversion between ASCII characters and integers, as well as utilizing system calls to display results to the user.

Furthermore, the third program extended our understanding by introducing subtraction. We applied similar principles from the addition program but incorporated subtraction operations instead. This allowed us to reinforce our comprehension of arithmetic operations and memory management in assembly programming.

Overall, this laboratory work provided a solid foundation in assembly language programming with NASM. Through practical implementation of various programs, we not only gained proficiency in writing assembly code but also enhanced our problem-solving skills and understanding of low-level computer architecture. These skills are invaluable for further exploration in the realm of system programming, embedded systems, and software optimization.