1 Abstract

Assembly is a low-level programming language that acts as an intermediary between a high-level programming language and processor-specific machine code. Assembly language provides a human-readable representation of a computer's Instruction Set Architecture, allowing programmers to write efficient code that directly interacts with computer hardware like CPU, main memory, and helps programmers optimize for specific hardware, resulting in faster and resource-efficient code.

In this lab, we will be working on understanding x86 architecture of the processor, different hardware, functionality of these hardware, and how we use assembly on these hardware to utilize them effectively for generating machine-specific object code.

Contents

2 Architecture of X86 Processor

2.1 Registers

2.1.1 General Purpose Registers

Data Registers There are four data registers in x86 processor: EAX, EBX, ECX, and EDX. They are 32-bit in size, and their significant 16 bits are named AX, BX, CX, and DX respectively.

EAX: Primary accumulator used for I/O and most arithmetic instructions.

EBX: Base register used in indexed addressing.

ECX: Count register used to store loop count in iterative operations.

EDX: Data register used in I/O operations, mainly used with EAX for multiplication and division operations involving large numbers.

Pointer Registers There are three categories of pointer registers in x86 processor: EIP, ESP, and EBP for 32-bit, and the corresponding 16-bit portions are IP, SP, and BP.

EIP: Instruction Pointer used to store the offset address of the next instruction to be executed.

ESP: Stack Pointer provides the offset value within the program stack, pointing to the top of the stack.

EBP: Base pointer stores the base address of the memory.

Index Registers There are two Index registers: Source Index (ESI) and Destination Index (EDI).

ESI: Used as Source Index for string operations.

EDI: Used as Destination Index for string operations.

2.1.2 Segment Registers

There are six segment registers: CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES (Extra Segment), FS, and GS (extra segments after E Segment).

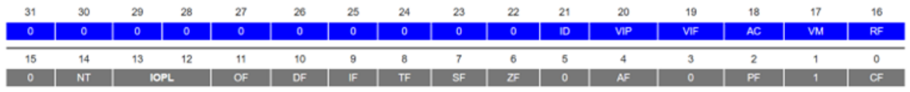
ECS: Code Segment contains all instructions to be executed, and ECS stores the starting address of the code segment.

EDS: Data Segment contains data, constants, and work areas. EDS stores the starting address of the data segment.

ESS: Stack Segment contains data and return addresses of procedures or subroutines. ESS register stores the starting address of the stack.

2.1.3 Flag Registers

The EFLAGS is a 32-bit register used as a collection of bits representing Boolean values to store the results of operations and the state of the processor.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Figure 1: EFLAGS Register

2.2 Control Unit

The control unit is responsible for fetching the instruction from the Instruction Queue (Main Memory) using the stack pointer or Program Counter, which refers to the top of the stack containing the next instruction to be executed. It also controls the flow of data between registers, ALU, and main memory.

2.3 Bus

The data bus transfers instructions and data between the CPU and memory. The I/O bus transfers data between the CPU and the devices in the computer system that have input/output commands.

2.4 ALU

The Arithmetic Logic Unit (ALU) operates on data stored in general-purpose registers. ALU operations take one or two inputs from registers and store the result back into the register. It supports Arithmetic, Logical, Relational, and Shift operations.

3 Software Used

Here we are installing Debian-based Linux OS in a virtualized manner. So, to accomplish that, we need two software: Oracle Virtual Box and Ubuntu or Kali Linux. Inside these guest OS, we need to install NASM shell to execute our assembly code by generating object code.

3.1 Oracle Virtual Box VM

1. Download Oracle Virtual Box from `Downloads -> Oracle VM VirtualBox`.
2. Follow the steps proceeding as the installation guide displays.

3.2 Ubuntu or Kali Linux

1. Download the Ubuntu ISO file from `Ubuntu Virtual Machine Images for VirtualBox and VMware (osboxes.org)` or directly pre-installed VM from `Get Kali | Kali Linux`.
2. Dynamically allocate HDD or SDD memory for the guest OS to work on and install its packages in.
3. For Kali Linux, after downloading the pre-installed VM, extract it to see two files of .vdi and .vbox format.
4. Click on Add and provide the installation guide the above two mentioned files when it requests.
5. Use the default username – osboxes.org and password osboxes.org if Ubuntu ISO is used.
6. Use default username kali and password kali if proceeded with Kali Linux.

3.3 NASM Shell

NASM Shell is pre-installed in Kali Linux pre-built VM; however, it must be manually installed in Ubuntu Xenial from osboxes.org.

1. After logging into Ubuntu Xenial, open the link provided to download the `https://www.nasm.us/` tool:
2. Select the latest stable version.
3. Download the gzip file, which is in the tar.x.xxx.x.gz file format.
4. Once downloaded, extract the zip file to a convenient directory.
5. Change the directory into nasm (cd to nasm) location and execute configure file by `./configure` command.
6. Make the binaries using `make` to build the nasm and ndisasm binaries.
7. Install the final binaries using `make install` to install nasm and ndisasm in `/usr/local/bin` and to install the manual in man page.

4 Programs

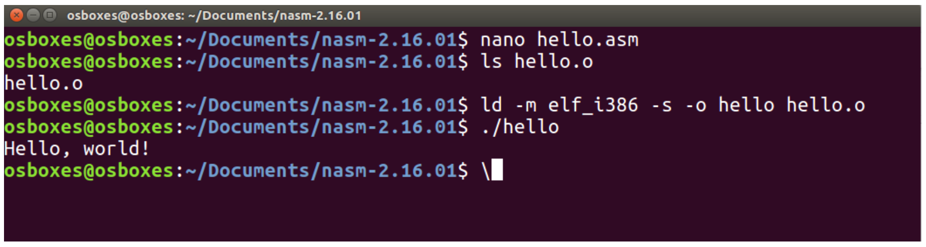
4.1 Program 1 – Hello World

In this program, we will display the message “Hello World!” using assembly language. The code is stored in section .text, and section .data is for initialized data or constants. `MOV eax, 4` is used to store the direct value 4 into EAX (accumulator), and `int` is an interrupt call used to call the kernel and interrupt the execution.

4.1.1 CODE

```
section .data
    hello db 'Hello, world!',0xa
    len equ $-hello

section .text
    global _start

_start:
    ; write to stdout
    mov eax, 4
    mov ebx, 1
    mov ecx, hello
    mov edx, len
    int 0x80

    ; exit
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

4.1.2 Result



```
osboxes@osboxes: ~/Documents/nasm-2.16.01
osboxes@osboxes:~/Documents/nasm-2.16.01$ nano hello.asm
osboxes@osboxes:~/Documents/nasm-2.16.01$ ls hello.o
hello.o
osboxes@osboxes:~/Documents/nasm-2.16.01$ ld -m elf_i386 -s -o hello hello.o
osboxes@osboxes:~/Documents/nasm-2.16.01$ ./hello
Hello, world!
osboxes@osboxes:~/Documents/nasm-2.16.01$ \
```

Figure 2: Hello World Output

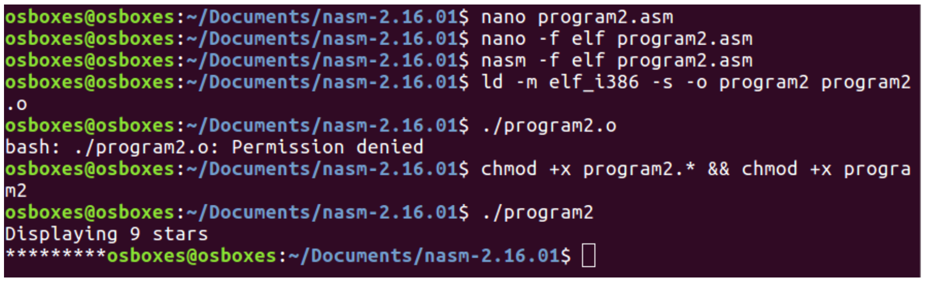
4.2 Program 2

In this program, we are printing a message "Displaying 9 stars" and 9 stars in a new line. Here we are printing output by using sys_write system call by loading eax with 4.

CODE

```
section .data
    stars db '*********', 0xa
    len equ $-stars

section .text
    global _start

_start:
    ; write to stdout
    mov eax, 4
    mov ebx, 1
    mov ecx, stars
    mov edx, len
    int 0x80

    ; exit
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

4.2.1 Result



```
osboxes@osboxes:~/Documents/nasm-2.16.01$ nano program2.asm
osboxes@osboxes:~/Documents/nasm-2.16.01$ nano -f elf program2.asm
osboxes@osboxes:~/Documents/nasm-2.16.01$ nasm -f elf program2.asm
osboxes@osboxes:~/Documents/nasm-2.16.01$ ld -m elf_i386 -s -o program2 program2
.o
osboxes@osboxes:~/Documents/nasm-2.16.01$ ./program2.o
bash: ./program2.o: Permission denied
osboxes@osboxes:~/Documents/nasm-2.16.01$ chmod +x program2.* && chmod +x progra
m2
osboxes@osboxes:~/Documents/nasm-2.16.01$ ./program2
Displaying 9 stars
*********osboxes@osboxes:~/Documents/nasm-2.16.01$
```

Figure 3: Displaying 9 Stars

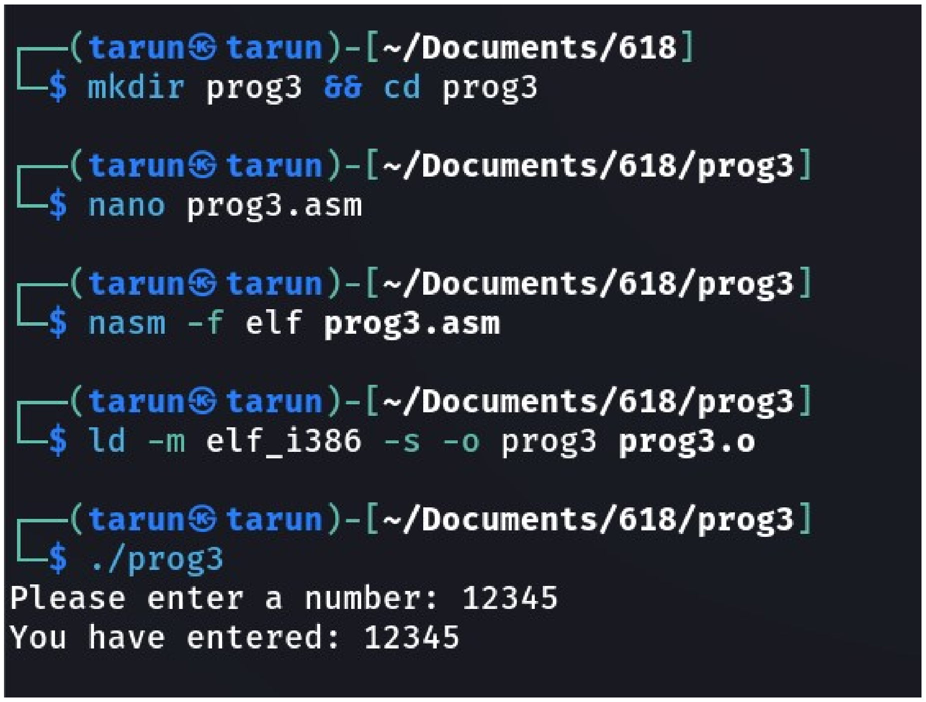
4.3 Program 3 – Display User Input Number

In the example code below, we are moving the value of the userMsg instruction to the edx data register using the `mov` data transfer instruction. The `int` system call instruction is used to call the kernel to an interrupt.

CODE

```
section .data
    userMsg db 'Enter a number: ', 0
    userLen equ $-userMsg
    userIn resb 5

section .text
    global _start

_start:
    ; write user prompt
    mov eax, 4
    mov ebx, 1
    mov ecx, userMsg
    mov edx, userLen
    int 0x80

    ; read user input
    mov eax, 3
    mov ebx, 0
    mov ecx, userIn
    mov edx, 5
    int 0x80

    ; exit
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

4.3.1 Result



```
┌──(tarun㉿tarun)-[~/Documents/618]
└─$ mkdir prog3 && cd prog3

┌──(tarun㉿tarun)-[~/Documents/618/prog3]
└─$ nano prog3.asm

┌──(tarun㉿tarun)-[~/Documents/618/prog3]
└─$ nasm -f elf prog3.asm

┌──(tarun㉿tarun)-[~/Documents/618/prog3]
└─$ ld -m elf_i386 -s -o prog3 prog3.o

┌──(tarun㉿tarun)-[~/Documents/618/prog3]
└─$ ./prog3
Please enter a number: 12345
You have entered: 12345
```

Figure 4: Return the user entered number

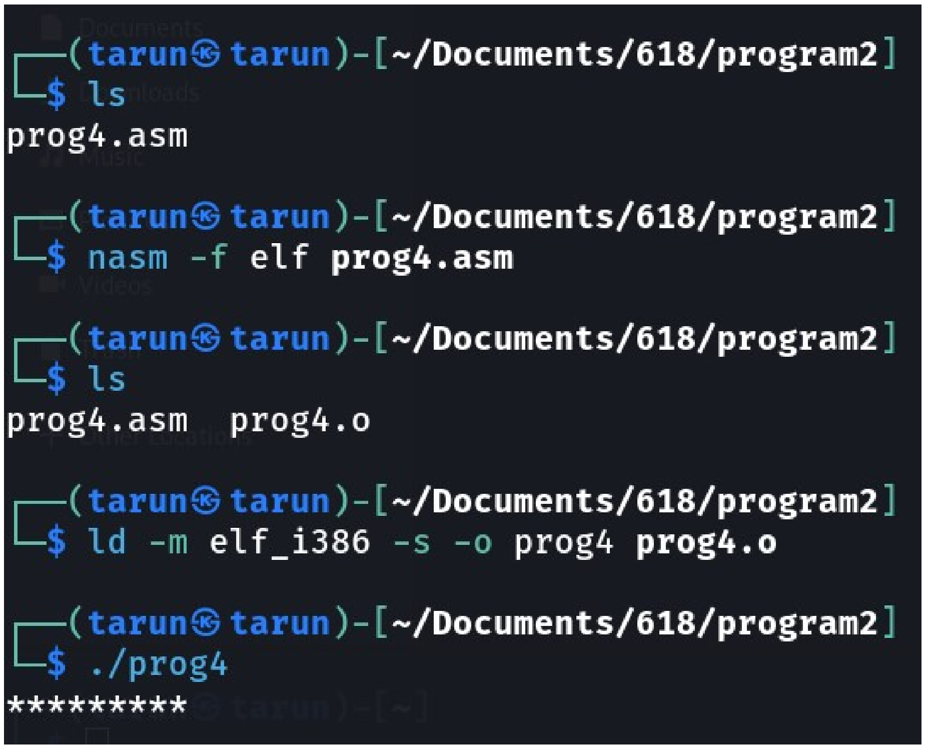
4.4 Program 4

In this program, we will display a sequence of “*********”. So here in the below code, we are implementing double byte “db” in the data segment. (`mov ebx,1`) is the file descriptor for stdout which will print output using sys_write system call by using the file descriptor of 4 (`mov eax,4`) and called the kernel with interrupt function defined as 80h, which is equivalent to 0x80.

CODE

```
section .data
    stars db '*********', 0xa
    len equ $-stars

section .text
    global _start

_start:
    ; write to stdout
    mov eax, 4
    mov ebx, 1
    mov ecx, stars
    mov edx, len
    int 0x80

    ; exit
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

4.4.1 Result



```
┌──(tarun㉿tarun)-[~/Documents/618/program2]
└─$ ls
prog4.asm

┌──(tarun㉿tarun)-[~/Documents/618/program2]
└─$ nasm -f elf prog4.asm

┌──(tarun㉿tarun)-[~/Documents/618/program2]
└─$ ls
prog4.asm  prog4.o

┌──(tarun㉿tarun)-[~/Documents/618/program2]
└─$ ld -m elf_i386 -s -o prog4 prog4.o

┌──(tarun㉿tarun)-[~/Documents/618/program2]
└─$ ./prog4
*********
```

Figure 5: Displaying 9 stars

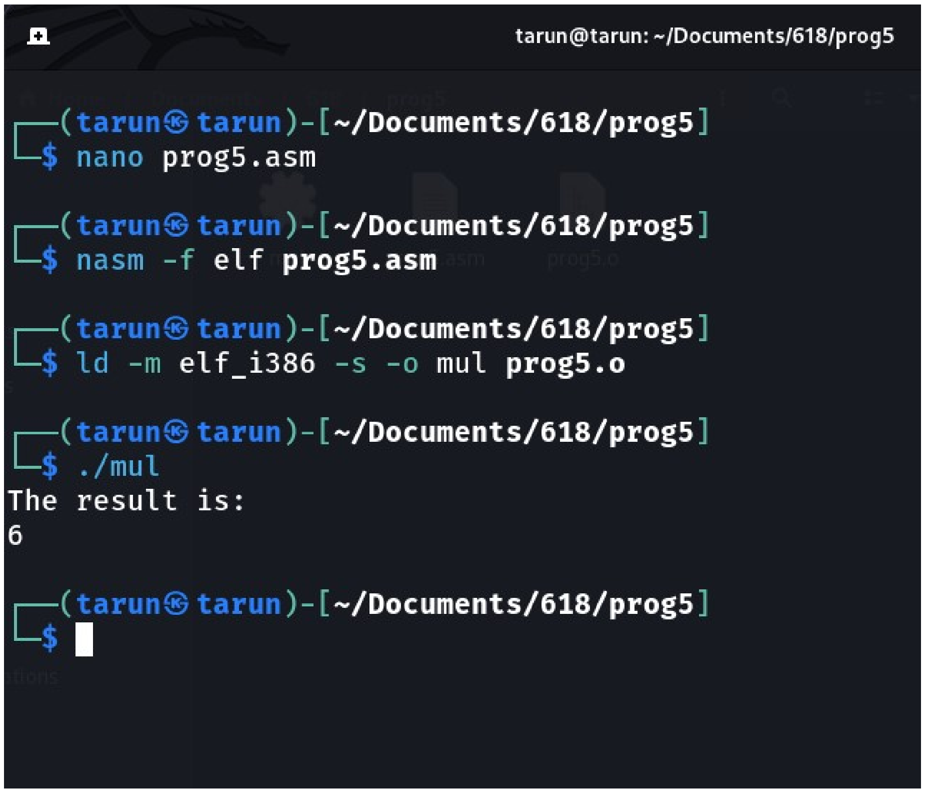
4.5 Program 5

This program will display the output of printing a message "The sum is:" and sum by accessing the value located at the address in the variable res. In the below code, `section.text` is the actual code section, `section.data` is for initialized data or constants, `section.bss` is a static variable created, but a temporary value is assigned to it and that value will be changed during runtime.

CODE

```
section .data
    msg db 'The sum is: ', 0
    msgLen equ $-msg

section .bss
    res resd 1

section .text
    global _start

_start:
    ; calculate the sum
    mov eax, 5
    add eax, 7
    mov [res], eax

    ; write message
    mov eax, 4
    mov ebx, 1
    mov ecx, msg
    mov edx, msgLen
    int 0x80

    ; write the sum
    mov eax, 4
    mov ebx, 1
    mov ecx, [res]
    mov edx, 4
    int 0x80

    ; exit
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

4.5.1 Result



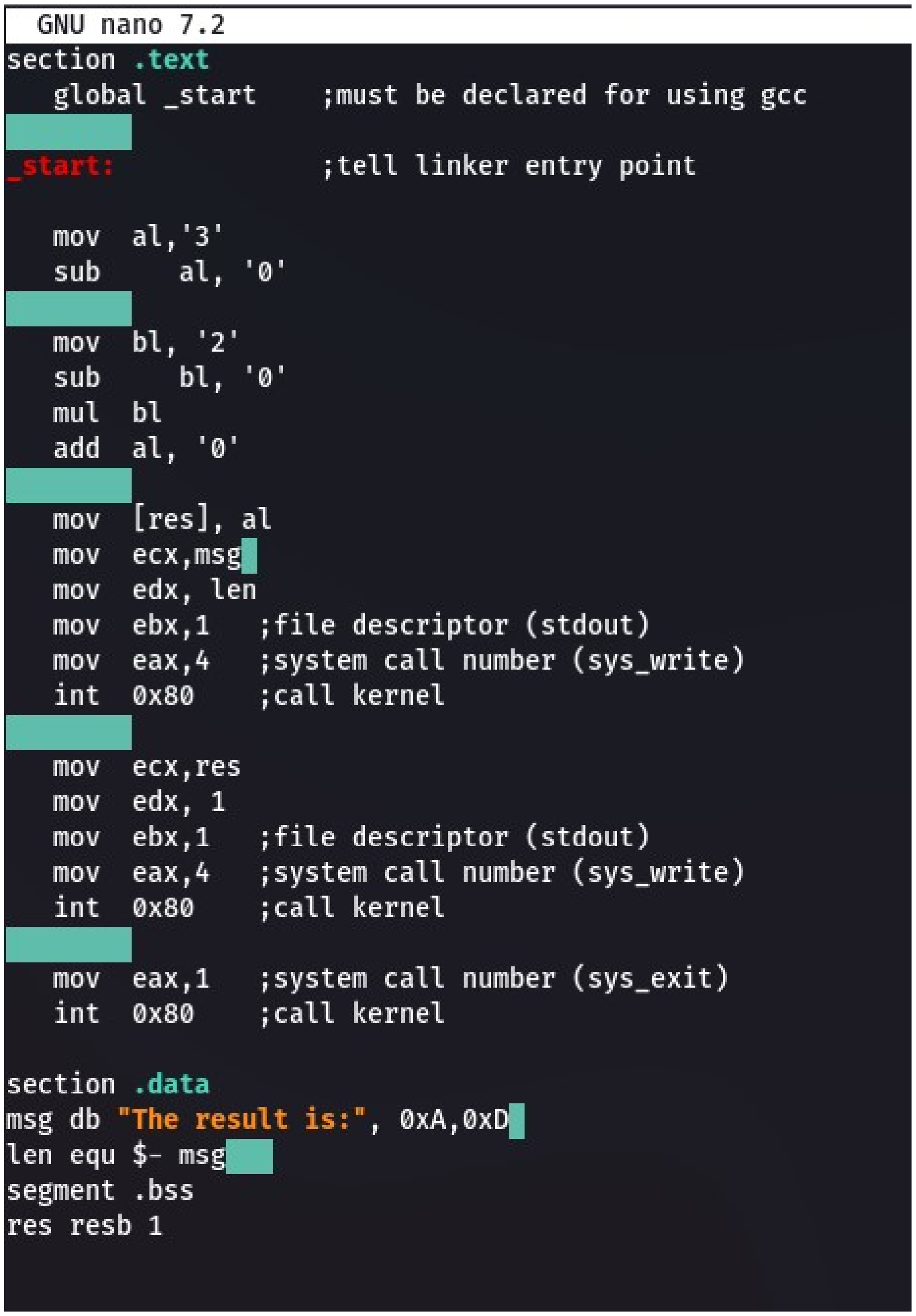
```
tarun@tarun: ~/Documents/618/prog5

┌──(tarun㉿tarun)-[~/Documents/618/prog5]
└─$ nano prog5.asm

┌──(tarun㉿tarun)-[~/Documents/618/prog5]
└─$ nasm -f elf prog5.asm

┌──(tarun㉿tarun)-[~/Documents/618/prog5]
└─$ ld -m elf_i386 -s -o mul prog5.o

┌──(tarun㉿tarun)-[~/Documents/618/prog5]
└─$ ./mul
The result is:
6

┌──(tarun㉿tarun)-[~/Documents/618/prog5]
└─$ 
```

Figure 6: MUL operation in x86

4.6 Program 6

`MUL` is used to perform multiplication in assembly. There are two instructions for multiplying binary data. The `MUL` (Multiply) instruction handles unsigned data and the `IMUL` (Integer Multiply) handles signed data. Both instructions affect the Carry and Overflow flag.

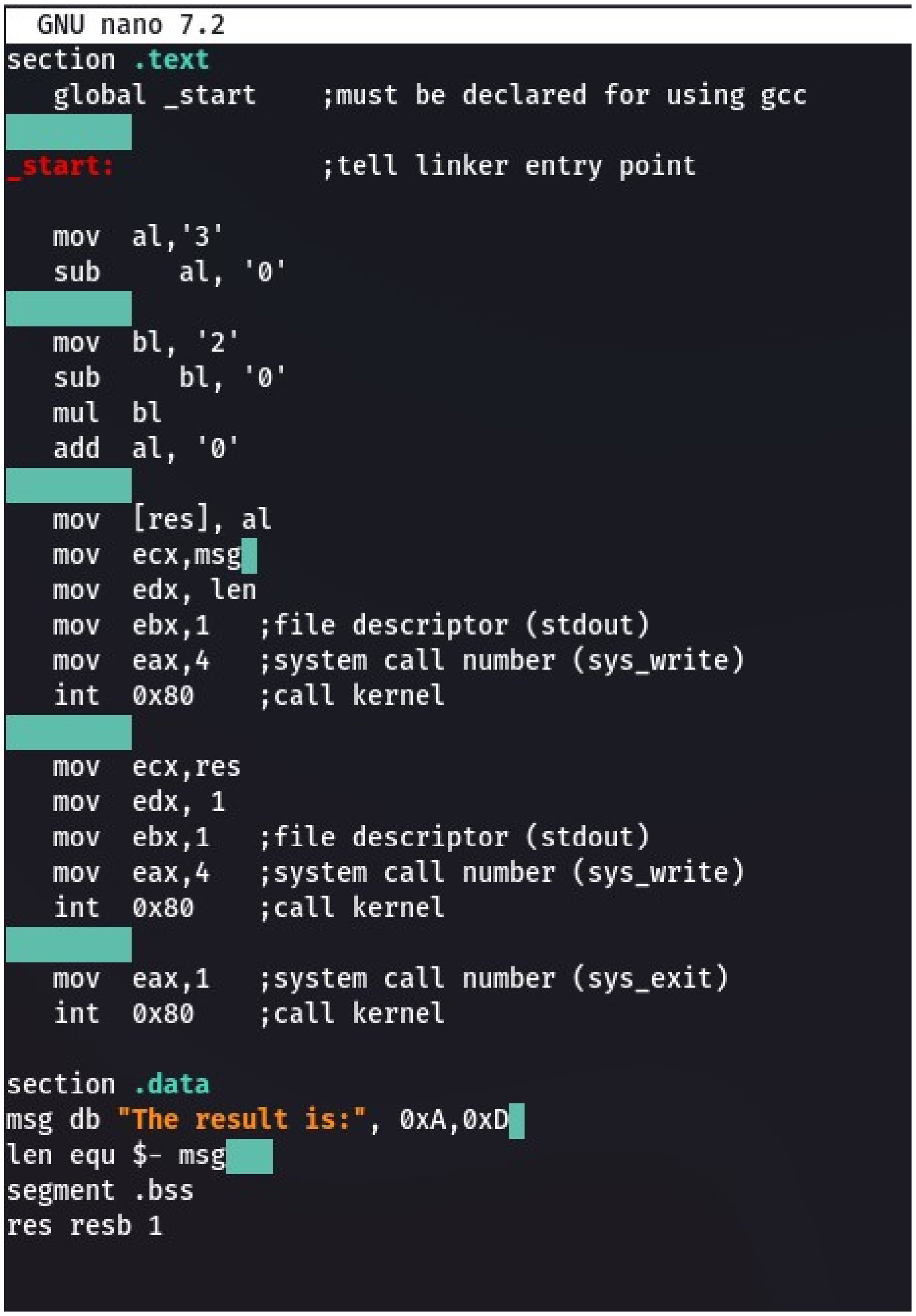
CODE



```
  GNU nano 7.2
section .text
   global _start    ;must be declared for using gcc

_start:             ;tell linker entry point

   mov  al,'3'
   sub     al, '0'

   mov  bl, '2'
   sub     bl, '0'
   mul  bl
   add  al, '0'

   mov  [res], al
   mov  ecx,msg
   mov  edx, len
   mov  ebx,1   ;file descriptor (stdout)
   mov  eax,4   ;system call number (sys_write)
   int  0x80    ;call kernel

   mov  ecx,res
   mov  edx, 1
   mov  ebx,1   ;file descriptor (stdout)
   mov  eax,4   ;system call number (sys_write)
   int  0x80    ;call kernel

   mov  eax,1   ;system call number (sys_exit)
   int  0x80    ;call kernel

section .data
msg db "The result is:", 0xA,0xD
len equ $- msg
segment .bss
res resb 1
```

Figure 7: MUL and IMUL in x86

4.6.1 Result



```
 GNU nano 7.2
section .text
   global _start    ;must be declared for using gcc

_start:             ;tell linker entry point

   mov  al,'3'
   sub     al, '0'

   mov  bl, '2'
   sub     bl, '0'
   mul  bl
   add  al, '0'

   mov  [res], al
   mov  ecx,msg
   mov  edx, len
   mov  ebx,1   ;file descriptor (stdout)
   mov  eax,4   ;system call number (sys_write)
   int  0x80    ;call kernel

   mov  ecx,res
   mov  edx, 1
   mov  ebx,1   ;file descriptor (stdout)
   mov  eax,4   ;system call number (sys_write)
   int  0x80    ;call kernel

   mov  eax,1   ;system call number (sys_exit)
   int  0x80    ;call kernel

section .data
msg db "The result is:", 0xA,0xD
len equ $- msg
segment .bss
res resb 1
```

Figure 8: Program to perform MUL operation in x86 Assembly