System Programming Lab

NASM Tutorial

Submitted By:

Uditi Arora

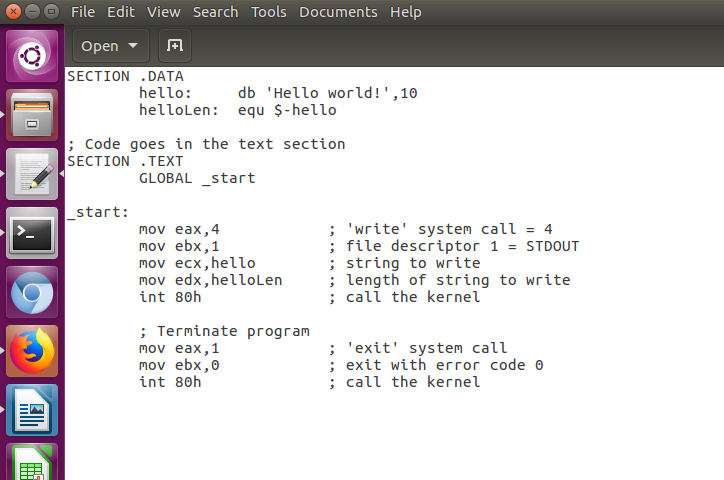
2016UCP1415

Section B

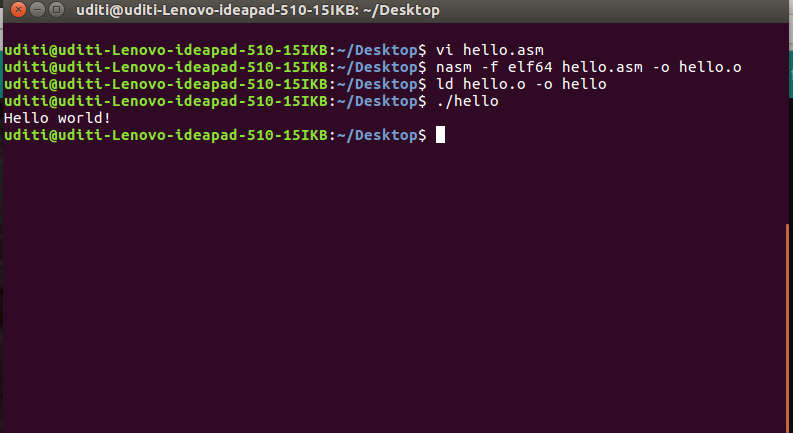
NASM, or The Netwide Assembler, is an x86 compiler that allows us to turn Assembly code in to machine code object files. Once we have an object file, we can link it and create the final executable. This example is meant for Unix systems or Windows with MinGW toolchain installed. On Debian systems, it can be installed with the nasm package.

Variables are defined the data section and code goes in the text section. We define the hello world string and the length of it. Then, the number that represents the system call for printing is moved into the eax register. That is where the computer will look for the system call number when the system interrupt int 80h is called.

The following program is to print Hello World!



On executing code:



An assembly program can be divided into three sections −

* The **data** section,
* The **bss** section, and
* The **text** section.

## The *data* Section

The **data** section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names, or buffer size, etc., in this section.

## The *bss* Section

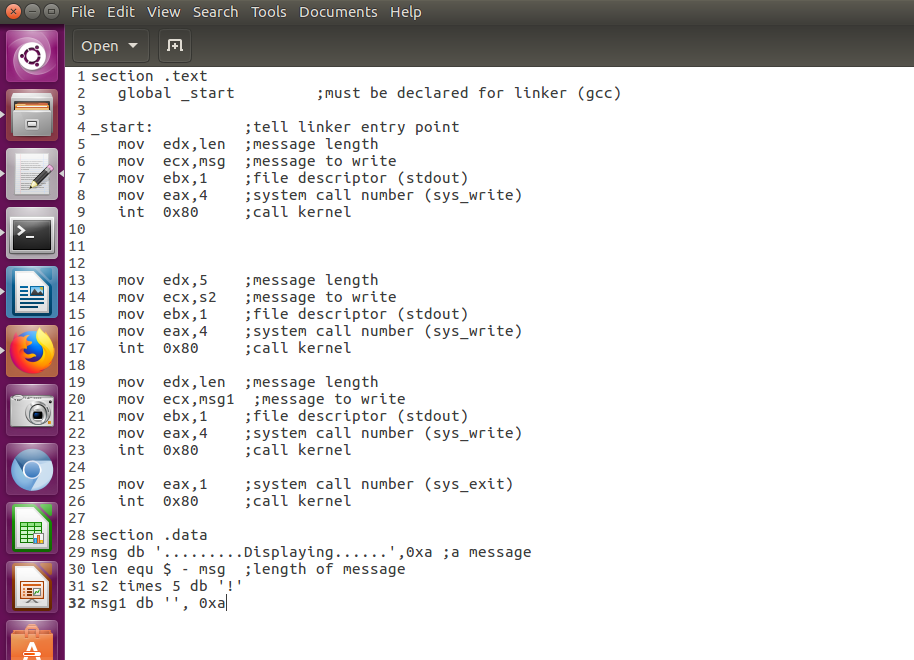
The **bss** section is used for declaring variables.

## The *text* section

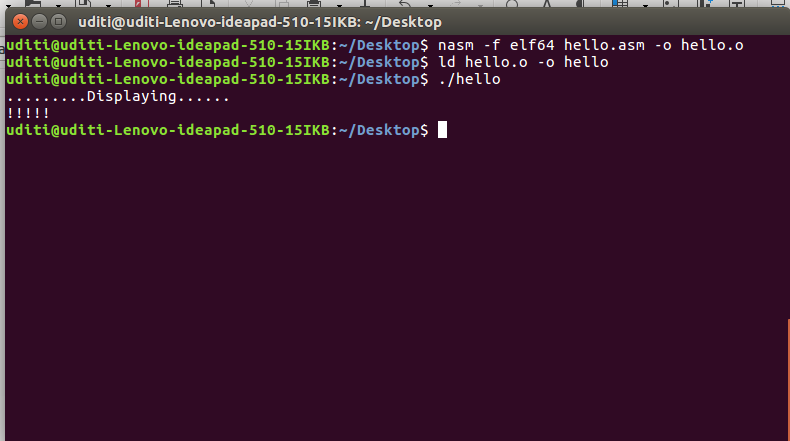
The **text** section is used for keeping the actual code. This section must begin with the declaration **global \_start**, which tells the kernel where the program execution begins.

USING REGISTERS

Let us try to print and store a string in register consisting of 5 ‘!’

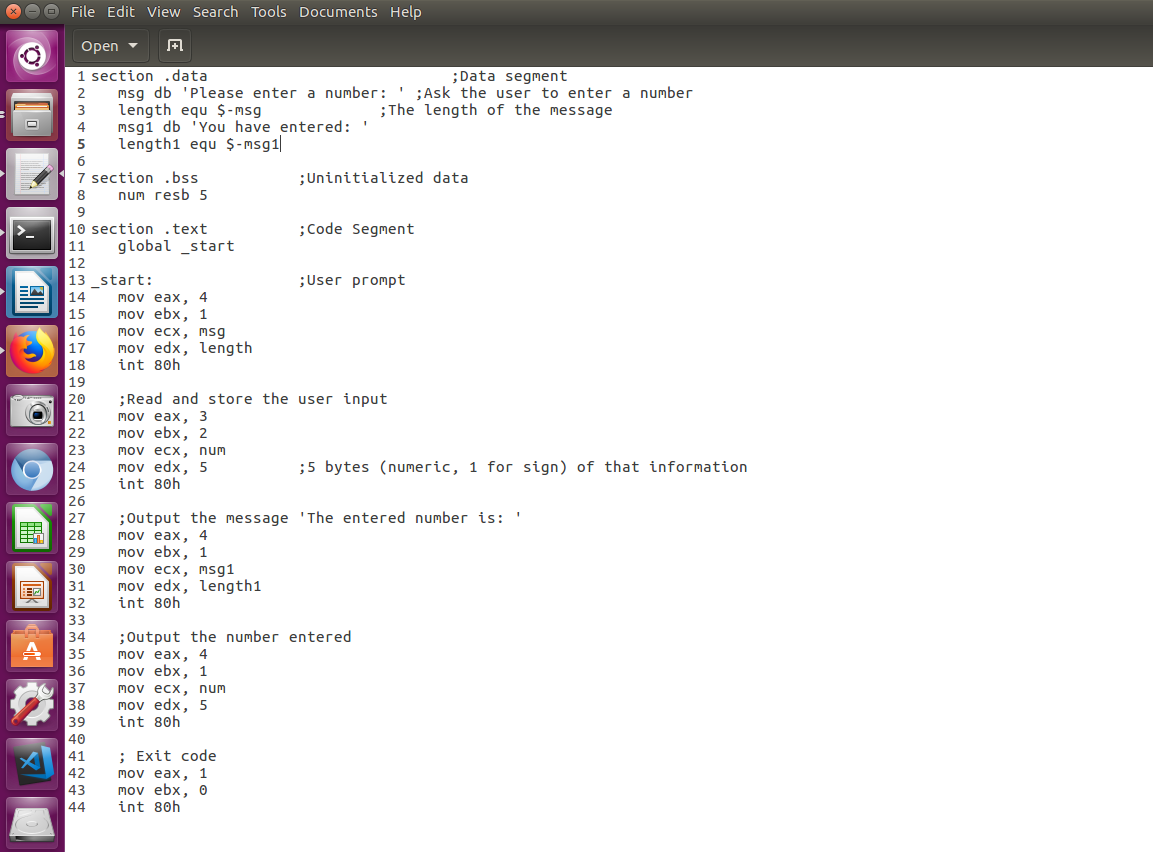


On running the code

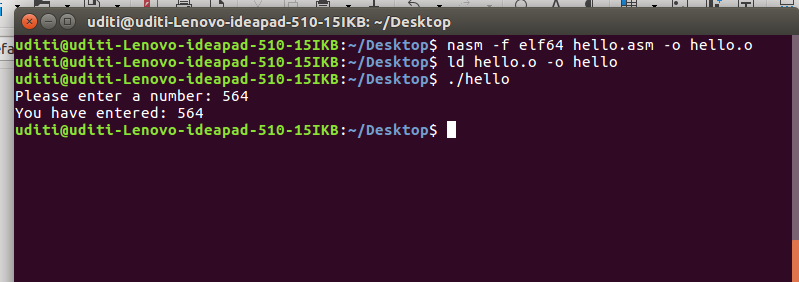


READING A NUMBER FROM USER

The code for reading and displaying a number is as follows



The output is



### Addressing Modes

Operands may be specified in one of three basic forms: immediate, register, and memory.

An immediate operand is just a number (or a label, which the assembler converts to the corresponding address). An immediate operand is used to specify a constant for one of the arithmetic or logical operations, or to give the jump address for a branching instruction. Most assemblers, including NASM, allow simple arithmetic expressions when computing immediate operands. For example, all of the following are equivalent:

MOV AL, 13

MOV AL, 0xD

MOV AL, 0Ah + 3 ;Note leading 0 to distinguish from register AH

MOV AL, George \* 2 - 1

assuming that the label George is associated with the address 7.

A register operand is one of the eight general- and special-purpose 16-bit registers listed above, or one of the eight general-purpose 8-bit registers (AL, AH, ...), or one of the four segment registers. The contents of the register are used and/or modified by the operation. In the example above, the destination operand of the MOV instruction is the low byte of the accumulator, AL; the effect of the instruction is to store the binary number 00001101 into the bottom eight bits of AX (leaving the other bits unchanged).

A memory operand gives the address of a location in main memory to use in the operation. The NASM syntax for this is very simple: put the address in square brackets. The address can be given as an arithmetic expression involving constants and labels (the displacement), plus an optional base or index register. Here are some examples:

MOV DX, [1234h]

ADD DX, [BX + 8]

MOV [BP + SI], DL

INC BYTE [0x100 + CS:DI]

A few of comments are needed on these examples. In the third example, the address is given by the sum of the contents of BP and SI; you can imagine that there is a default displacement of zero here, so the address is 0 + BP + SI. In the first example, the destination is 16 bits wide, so a 16-bit quantity will be fetched from two adjacent memory locations: DL will be loaded with the byte from 1234h, and DH will be loaded from 1235h. The same will happen in the second example: DL will have the contents of address BX + 8 added to it, and DH will have the contents of address BX + 9 added (plus any carry from the low byte). In the third example, the source is only 8 bits wide, so only the byte at address BP + SI will be changed. Finally, in the fourth example, the INC operation by itself is ambiguous about whether it is incrementing a single byte or a full 16-bit word; the keyword BYTE in front of the operand determines that only the byte at the address 100h + DI will be affected (the alternative would be to use the keyword WORD, to add 1 to the combination of bytes at 100h + DI and 101h + DI).

All of these addresses are really offsets into a particular segment. In the fourth example, the code segment is explicitly called for by the segment override CS:. The default segment is the data segment, except when the base register BP is involved, in which case the stack segment is used (as in the third example).

As one more example of a memory operand, consider the following sequence of instructions:

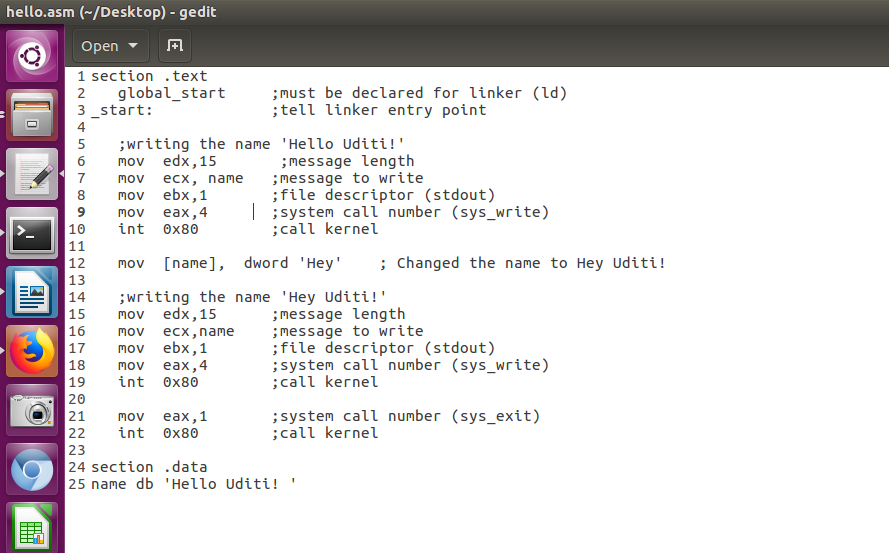
MOV BX, 100h

MOV SI, 20h

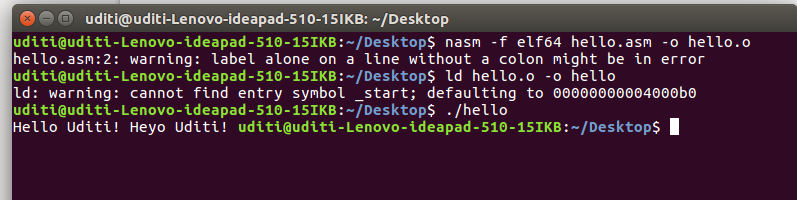
MOV AL, [BX + SI + 3]

The effective address of the third move instruction is computed by adding the contents of the BX and SI registers, plus the constant 3; therefore, the byte that is moved into AL comes from address 0123h (interpreted as an offset within the data segment).

A sample code which changes the string contained in the data section

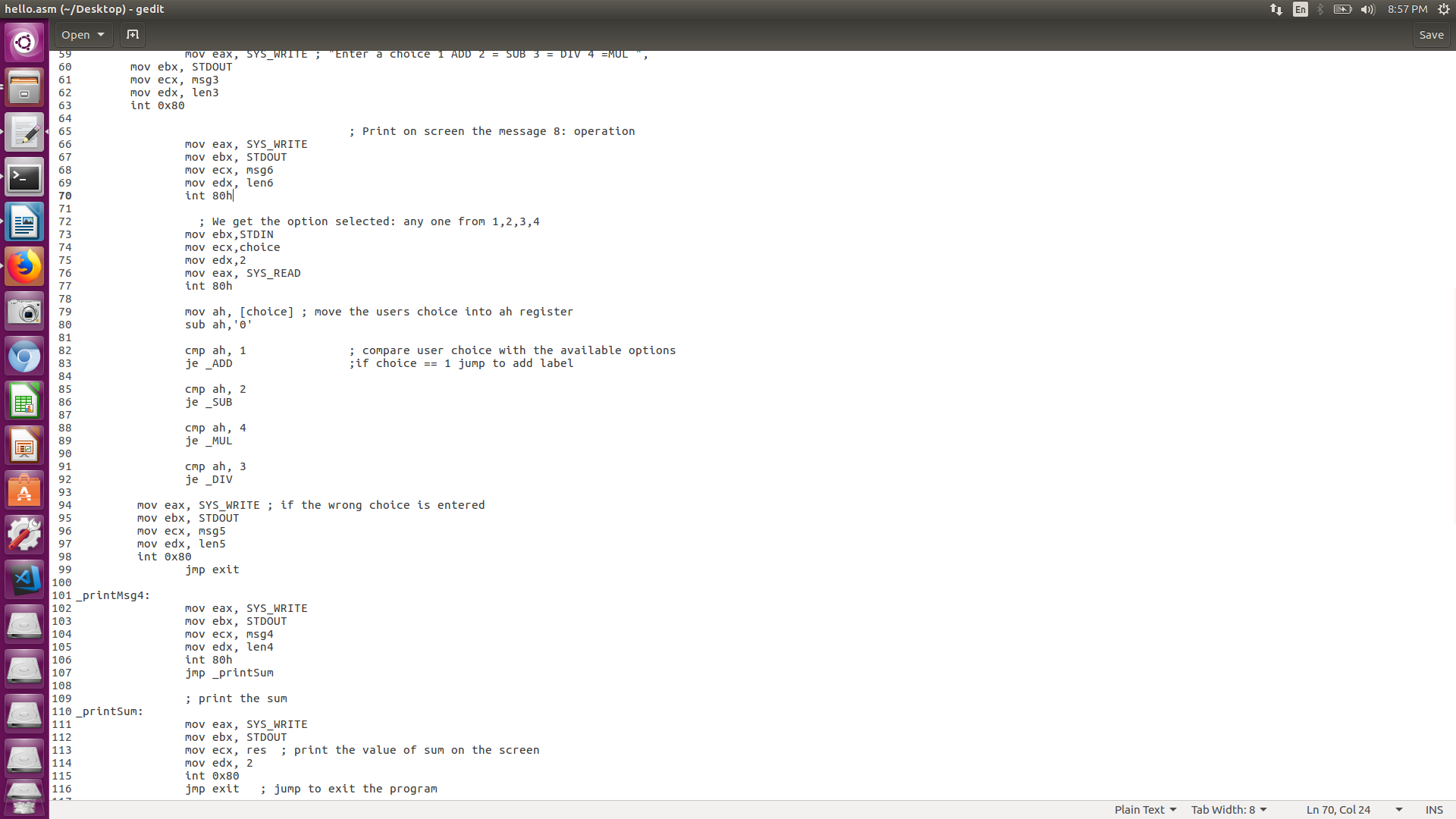
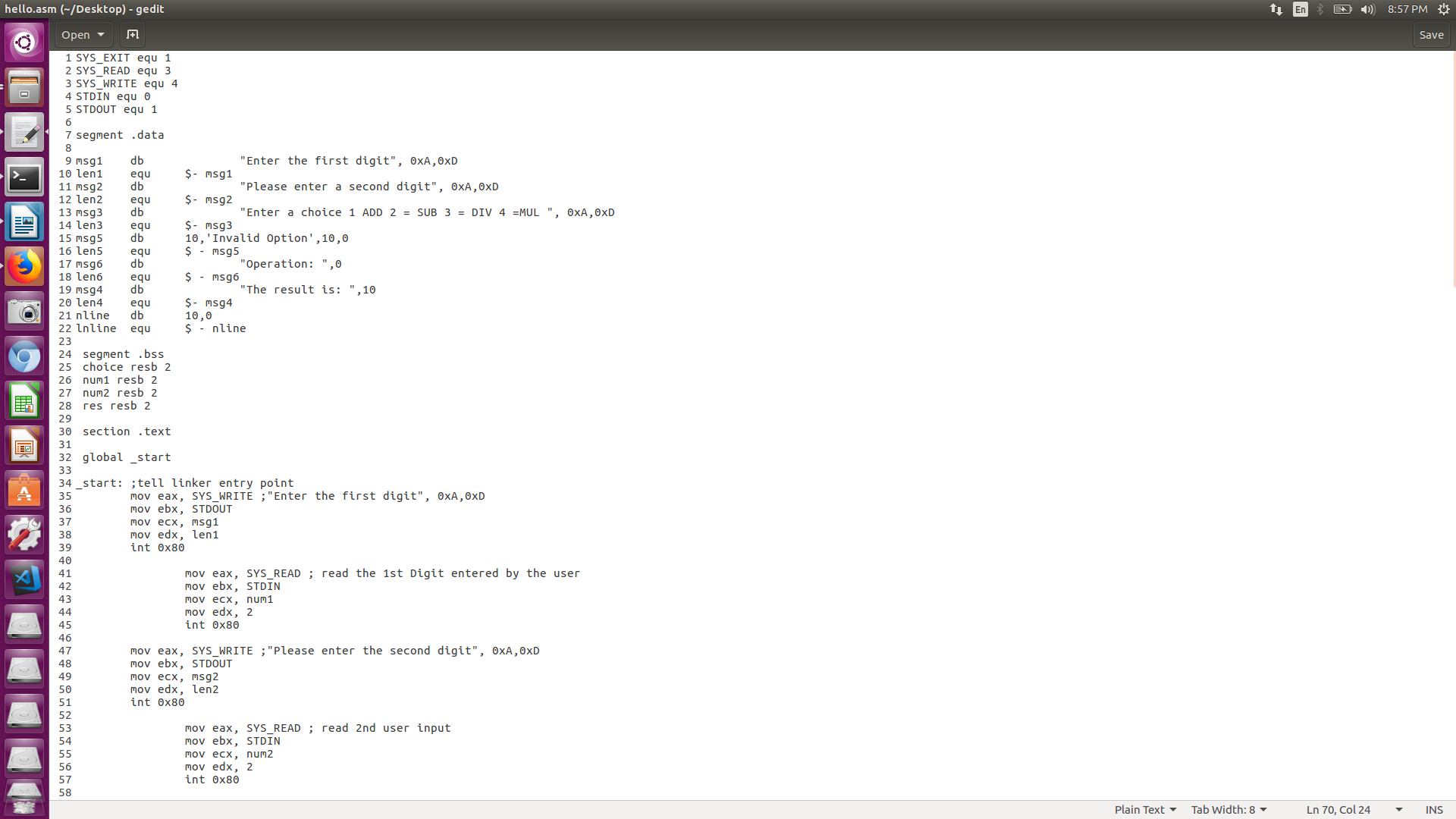


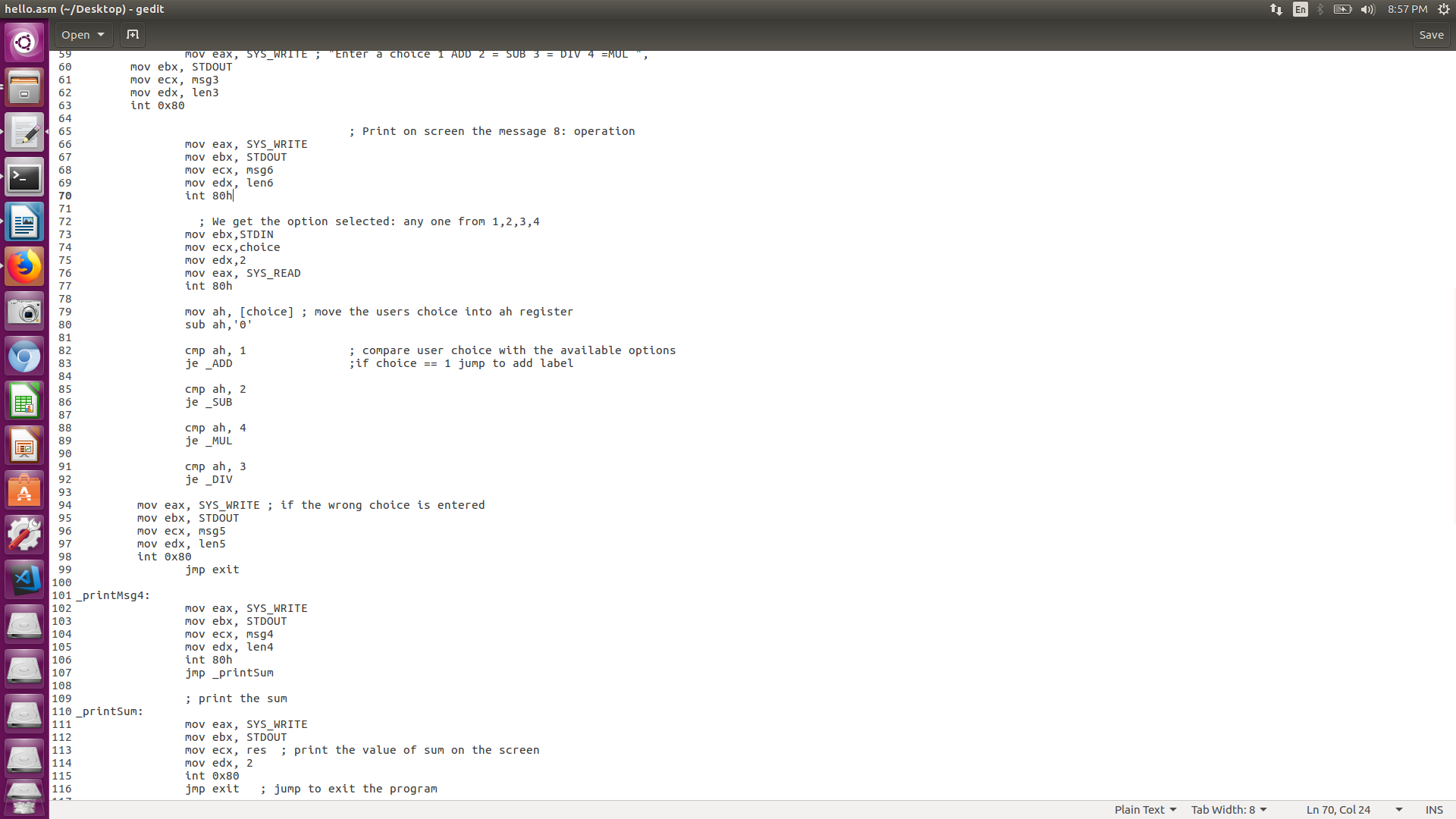
The output is



Arithemetic in NASM

Following is the code for operating on single digit numbers(The code uses switch case )





The results on executing program multiple times

