|  |
| --- |
| **Subject: MICROPROCESSOR LAB (MPL)** |
| **Name: Priyanka suresh Salunke** |
| **Class: SE COMP a Roll No.: F19111151** |
| **Semester: Sem-IV Year: 2020-2021** |
| **Date of Performance: Date of Submission:** |
| **Examined:** |

**Assignment No-01**

**Title:-** Accept and display Numbers

**Assignment Name: -** Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers.

**Objective-**

* To understand the assembly language program
* To understand 64 bit interrupt.

**Outcome-**

* Students will be able to write code for how to count positive and negative number from array
* Students will be able to understand different assembly language instruction.

### [Prerequisite](http://dictionary.reference.com/browse/prerequisite) -

System call of Unix for Assembly language Program.

**Hardware Requirement-**

Desktop PC

**Software Requirement-**

Ubuntu 14.04,

Assembler: NASM version 2.10.07

Linker: ld

**Introduction:-**

**Theory:**

**Introduction to Assembly Language Programming:**

Each personal computer has a microprocessor that manages the computer's arithmetical, logical and control activities. Each family of processors has its own set of instructions for handling various operations like getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instruction. Processor understands only machine language instructions which are strings of 1s and 0s. However machine language is too obscure and complex for using in oftware development. So the low level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form. Assembly language is a low-level programming language for a computer, or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM etc.

**Advantages of Assembly Language**

* An understanding of assembly language provides knowledge of Interface of programs with OS, processor and BIOS.
* Representation of data in memory and other external devices.
* How processor accesses and executes instruction
* How instructions accesses and process data
* How a program access external devices.
* It requires less memory and execution time.
* It allows hardware-specific complex jobs in an easier way.
* It is suitable for time-critical jobs.

**ALP Step By Step:**

**Installing NASM:**

If you select "Development Tools" while installed Linux, you may NASM installed along with the Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps:

Open a Linux terminal.

Type ***whereis nasm*** and press ENTER.

If it is already installed then a line like, *nasm: /usr/bin/nasm* appears. Otherwise, you will see just *nasm:*, then you need to install NASM.

**To install NASM take the following steps:**

Check the netwide assembler (NASM) website for the latest version.

1. Download the Linux source archive nasm-X.XX. ta .gz, where X.XX is the NASM version number in the archive.
2. Unpack the archive into a directory, which creates a subdirectory nasm-X. XX.
3. cd to *nasm-X. XX* and type ***./configure*** . This shell script will find the best C compiler to use and set up Makefiles accordingly.
4. Type ***make*** to build the nasm and ndisasm binaries.
5. Type ***make install*** to install nasm and ndisasm in /usr/local/bin and to install the man pages.

**Assembly Basic Syntax:**

An assembly program can be divided into three sections:

1. **The data section**
2. **The bss section**
3. **The text section**

The order in which these sections fall in your program really isn’t important, but by convention the .data section comes first, followed by the .bss section, and then the .text section.

**The .data Section**

The .data section contains data definitions of initialized data items. Initialized data is data that has a value before the program begins running. These values are part of the executable file. They are loaded into memory when the executable file is loaded into memory for execution. You don’t have to load them with their values, and no machine cycles are used in their creation beyond what it takes to load the program as a whole into memory.

The important thing to remember about the .data section is that the more initialized data items you define, the larger the executable file will be, and the longer it will take to load it from disk into memory when you run it.

Section .data

**The .bss Section**

Not all data items need to have values before the program begins running. When you’re reading data from a disk file, for example, you need to have a place for the data to go after it comes in from disk. Data buffers like that are defined in the .bss section of your program. You set aside some number of bytes for a buffer and give the buffer a name, but you don’t say what values are to be present in the buffer.

There’s a crucial difference between data items defined in the .data section and data items defined in the .bss section: data items in the .data section add to the size of your executable file. Data items in the .bss section do not.

Section .bss

**The .text Section**

The actual machine instructions that make up your program go into the .text section. Ordinarily, no data items are defined in .text. The .text section contains symbols called *labels* that identify locations in the program code for jumps and calls, but beyond your instruction mnemonics, that’s about it.

All global labels must be declared in the .text section, or the labels cannot be ‘‘seen’’ outside your program by the Linux linker or the Linux loader

Section .text

**Assembly Language Statements**

Assembly language programs consist of three types of statements:

* Executable instructions or instructions
* Assembler directives or pseudo-ops
* Macros

Syntax of Assembly Language Statements

|  |
| --- |
| [label] mnemonic [operands] [;comment] |

**64 bit Interrupt**

**Write system Call**

mov rax,1

mov rdi,1

mov rsi,%1

mov rdx,%2

syscall

**Read system call**

mov rax,0

mov rdi,0

mov rsi,%1

mov rdx,%2

syscall

**Compiling and Linking an Assembly Program in NASM**

1. Type the above code using a text editor and save it as assignment1.asm.
2. Make sure that you are in the same directory as where you saved assignment1**.asm**.
3. To assemble the program, type **nasm -f elf64 assignment1.asm**
4. If there is any error, you will be prompted about that at this stage. Otherwise an object file of your program named **assignment1.o** will be created.
5. To link the object file and create an executable file named assignment1, type **ld -o assignment assignment1.o**
6. Execute the program by typing **./assignment1**

**Algorithm:**

1. Start

2. Initialise section .data

3. Define variable for array,pcount,ncount

4. Count Positive and negative number using BT command.

5. Display counts

6. Terminate program using system call

6. Stop

**Conclusion:-** Hence we implemented an ALP to count positive and negative number from array and display count.

**Questions:-**

1. Explain Architecture of 80386?
2. Explain segmentation of 80386?
3. Explain and draw bit pattern for flag register of 80386DX with significance of each bit.

|  |
| --- |
| **Subject: MICROPROCESSOR LAB (MPL)**  **Name:** |
| **Class: Roll No.:** |
| **Semester: Sem-II Year: 2016-17** |
| **Date of Performance: Date of Submission:** |
| **Examined:** |

**Assignment No-02**

**Title:-** String length calculation

**Assignment Name:** - Write an X86/64 ALP to accept a string and to display its length.

**Objective-**

* To study various string instruction
* To understand how to define string in data segment.
* To calculate str length.

**Outcome-**

* Students will be able to write code to accept str and display string length.

### [Prerequisite](http://dictionary.reference.com/browse/prerequisite) -

System call of Unix for Assembly language Program.

**Hardware Requirement-**

Desktop PC

**Software Requirement-**

Ubuntu 14.04,

Assembler: NASM version 2.10.07

Linker: ld

**Introduction:-**

**Guidelines for the algorithm:**

1. Initialize Data section.
2. Declare string and other required variables.
3. Accept string from user.
4. Count of entered string including enter character is availabe with RAX register.
5. Display length as value available in RAX.

**Conclusion:-** Hence we implemented an ALP to calculate string length.

**Questions:-**

1. Explain string instruction of 80386?
2. Explain direction flag?