**INDEX**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr.No.** | **Date** | **Experiment Performed** | **Page**  **No** | **Sign** | **Remark** |
| **1** |  | 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. |  |  |  |
| **2** |  | Write an X86/64 ALP to accept a string and to display its length. |  |  |  |
| **3** |  | Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers. |  |  |  |
| **4** |  | Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic  operations (+,-,\*, /) using suitable macros. Define procedure for each operation. |  |  |  |
| **5** |  | Write an X86/64 ALP to count number of positive and negative numbers from the array. |  |  |  |
| **6** |  | Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5- digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for: (a) HEX to BCD b) BCD to HEX (c) EXIT.  Display proper strings to prompt the user while accepting the input and displaying the  result. (Wherever necessary, use 64-bit registers). |  |  |  |
| **7** |  | Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction. |  |  |  |
| **8** |  | Write X86/64 ALP to perform non-overlapped block transfer without string specific  instructions. Block containing data can be defined in the data segment. |  |  |  |
| **9** |  | Write X86/64 ALP to perform overlapped block transfer with string specific instructions  Block containing data can be defined in the data segment. |  |  |  |
| **10** |  | Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use  successive addition and add and shift method. (use of 64-bit registers is expected). |  |  |  |
| **11** |  | Write X86 Assembly Language Program (ALP) to implement following OS commands  i) COPY, ii) TYPE  Using file operations. User is supposed to provide command line arguments |  |  |  |
| **12** |  | Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program\_1 execution and write FAR PROCEDURES in Program\_2 for the rest of  the processing. Use of PUBLIC and EXTERN directives is mandatory. |  |  |  |
| **13** |  | Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code. |  |  |  |
| **14** |  | Write an X86/64 ALP password program that operates as follows: a. Do not display what is actually typed instead display asterisk (“\*”).  If the password is correct display, “access is granted” else display “Access not Granted” |  |  |  |
|  |  | Study Assignment:  Motherboards are complex. Break them down, component by component, and Understand how they work. Choosing a motherboard is a hugely important part of building a PC. Study- Block diagram, Processor Socket, Expansion Slots, SATA, RAM, Form Factor, BIOS, Internal Connectors, External Ports, Peripherals and Data Transfer, Display, Audio, Networking, Overclocking, and Cooling. 4.  [https://www.intel.in/content/www/in/en/support/articles/000006014/boards-and-](https://www.intel.in/content/www/in/en/support/articles/000006014/boards-and-kits/desktop-boards.html) [kits/desktop-boards.html](https://www.intel.in/content/www/in/en/support/articles/000006014/boards-and-kits/desktop-boards.html) |  |  |  |

**EXPERIMENT NO. 01**

**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.

**EXP NO: 01**

**AIM: :** 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.

**OBJECTIVES:**

* To understand assembly language programming instruction set
* To understand different assembler directives with example
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**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 software 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.

Other advantages of using assembly language are:

 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:**

Open Terminal and run below commands:

sudo apt-get update

sudo apt-get install nasm

**Assembly Basic Syntax:**

An assembly program can be divided into three sections:

 The **data** section

 The **bss** section

 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.

**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.

**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. Let’s look at the labels issue a little more closely.

**Labels**

A label is a sort of bookmark, describing a place in the program code and giving it a name that’s easier

to remember than a naked memory address. Labels are used to indicate the places where jump instructions should jump to, and they give names to callable assembly language procedures.

Here are the most important things to know about labels:

 *Labels must begin with a letter, or else with an underscore, period, or question mark.* These last

three have special meanings to the assembler, so don’t use them until you know how NASM

interprets them.

 *Labels must be followed by a colon when they are defined.* This is basically what tells NASM

that the identifier being defined is a label. NASM will punt if no colon is there and will not flag

an error, but the colon nails it, and prevents a mistyped instruction mnemonic from being

mistaken for a label. Use the colon!

 *Labels are case sensitive.* So yikes:, Yikes:, and YIKES: are three completely different labels.

**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]

**LIST OF INTERRRUPTS USED:** NA

**LIST OF ASSEMBLER DIRECTIVES USED:** EQU,DB

**LIST OF MACROS USED:** NA

**LIST OF PROCEDURES USED:** NA

**ALGORITHM:**

INPUT: ARRAY

OUTPUT: ARRAY

STEP 1: Start.

STEP 2: Initialize the data segment.

STEP 3: Display msg1 “Accept array from user. “

STEP 4: Initialize counter to 05 and rbx as 00

STEP 5: Store element in array.

STEP 6: Move rdx by 17.

STEP 7: Add 17 to rbx.

STEP 8: Decrement Counter.

STEP 9: Jump to step 5 until counter value is not zero.

STEP 9: Display msg2.

STEP 10: Initialize counter to 05 and rbx as 00

STEP 11: Display element of array.

STEP 12: Move rdx by 17.

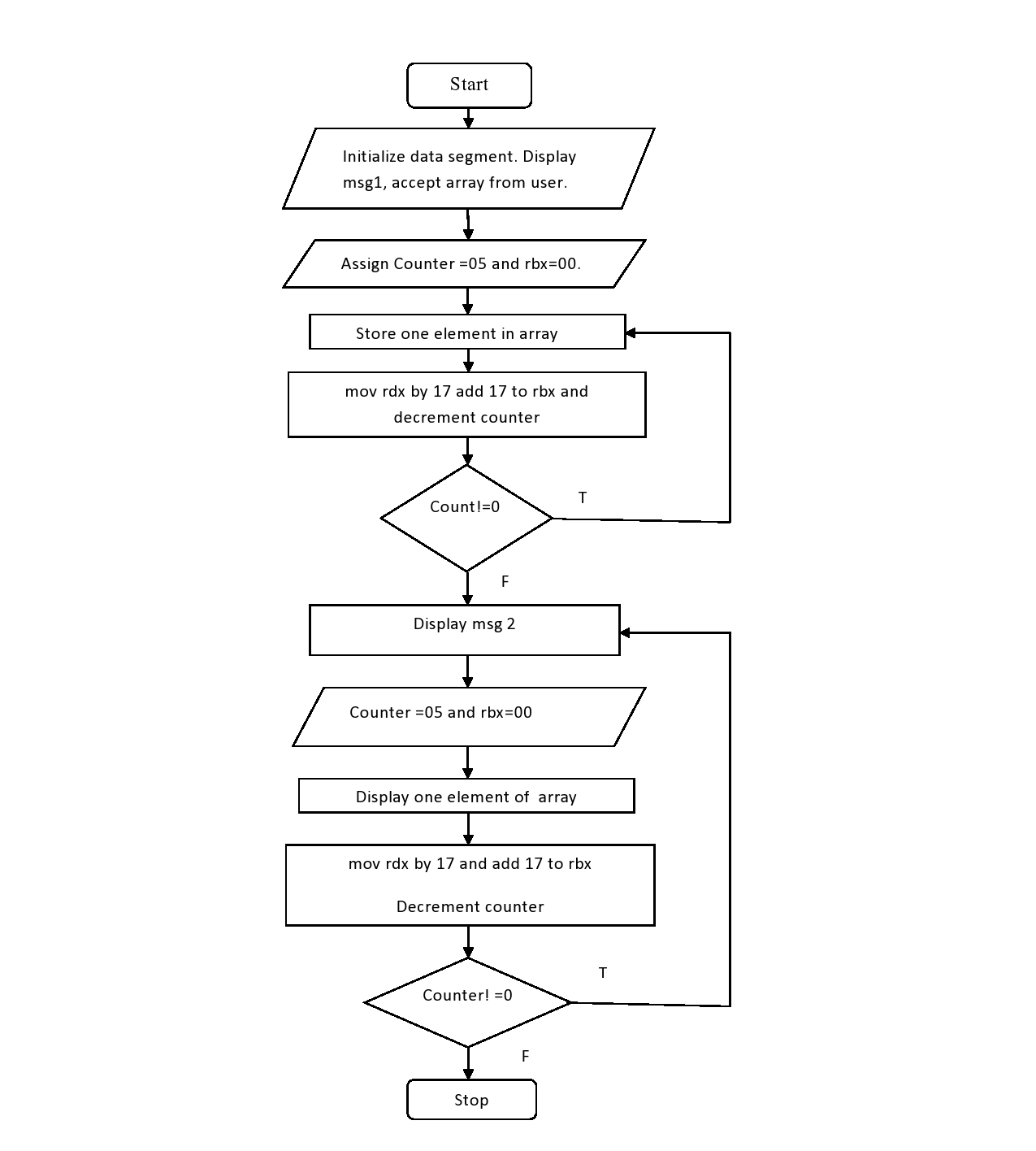
STEP 13: Add 17 to rbx.

STEP 14: Decrement Counter.

STEP 15: Jump to step 11 until counter value is not zero.

STEP 16: Stop

**FLOWCHART:**

****

**PROGRAM:**

section .data

msg1 db 10,13,"Enter 5 64 bit numbers"

len1 equ $-msg1

msg2 db 10,13,"Entered 5 64 bit numbers"

len2 equ $-msg2

section .bss

array resd 200

counter resb 1

section .text

global \_start

\_start:

;display

mov Rax,1

mov Rdi,1

mov Rsi,msg1

mov Rdx,len1

syscall

;accept

mov byte[counter],05

mov rbx,00

loop1:

mov rax,0 ; 0 for read

mov rdi,0 ; 0 for keyboard

mov rsi, array ;move pointer to start of array

add rsi,rbx

mov rdx,17

syscall

add rbx,17 ;to move counter

dec byte[counter]

JNZ loop1

;display

mov Rax,1

mov Rdi,1

mov Rsi,msg2

mov Rdx,len2

syscall

;display

mov byte[counter],05

mov rbx,00

loop2:

mov rax,1 ;1 for write

mov rdi, 1 ;1 for monitor

mov rsi, array

add rsi,rbx

mov rdx,17 ;16 bit +1 for enter

syscall

add rbx,17

dec byte[counter]

JNZ loop2

;exit system call

mov rax ,60

mov rdi,0

syscall

;output

;vacoea@vacoea-Pegatron:~$ cd ~/Desktop

;vacoea@vacoea-Pegatron:~/Desktop$ nasm -f elf64 ass1.asm

;vacoea@vacoea-Pegatron:~/Desktop$ ld -o ass1 ass1.o

;vacoea@vacoea-Pegatron:~/Desktop$ ./ass1

;Enter 5 64 bit numbers12

;23

;34

;45

;56

;Entered 5 64 bit numbers12

;23

;34

;45

;56

**CONCLUSION:**

In this practical session we learnt how to write assembly language program and Accept and display array in assembly language.

**EXPERIMENT NO. 02**

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

**EXP NO: 02**

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

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**MACRO:**

Writing a macro is another way of ensuring modular programming in assembly language.

* A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.
* In NASM, macros are defined with **%macro** and **%endmacro** directives.
* The macro begins with the %macro directive and ends with the %endmacro directive.

The Syntax for macro definition −

%macro macro\_name number\_of\_params

<macro body>

%endmacro

Where, *number\_of\_params* specifies the number parameters, *macro\_name* specifies the name of the macro.

The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

**PROCEDURE:**

Procedures or subroutines are very important in assembly language, as the assembly language programs tend to be large in size. Procedures are identified by a name. Following this name, the body of the procedure is described which performs a well-defined job. End of the procedure is indicated by a return statement.

Syntax

Following is the syntax to define a procedure −

proc\_name:

procedure body

...

ret

The procedure is called from another function by using the CALL instruction. The CALL instruction should have the name of the called procedure as an argument as shown below −

CALL proc\_name

The called procedure returns the control to the calling procedure by using the RET instruction.

**LIST OF INTERRRUPTS USED:** NA

**LIST OF ASSEMBLER DIRECTIVES USED**: EQU, PROC, GLOBAL, DB,

**LIST OF MACROS USED:** DISPMSG

**LIST OF PROCEDURES USED:** DISPLAY

**ALGORITHM:**

INPUT: String

OUTPUT: Length of String in hex

STEP 1: Start.

STEP 2: Initialize data section.

STEP 3: Display msg1 on monitor

STEP 4: accept string from user and store it in Rsi Register (Its length gets stored in Rax register by default).

STEP 5: Display the result using “display” procedure. Load length of string in data register.

STEP 6. Take counter as 16 int cnt variable

STEP 7: move address of “result” variable into rdi.

STEP 8: Rotate left rbx register by 4 bit.

STEP 9: Move bl into al.

STEP 10: And al with 0fh

STEP 11: Compare al with 09h

STEP 12: If greater add 37h into al

STEP 13: else add 30h into al

STEP 14: Move al into memory location pointed by rdi

STEP 14: Increment rdi

STEP 15: Loop the statement till counter value becomes zero

STEP 16: Call macro dispmsg and pass result variable and length to it. It will print length of string.

STEP 17: Return from procedure

STEP 18: Stop

**FLOWCHART:**

**PROGRAM:**

section .data

msg1 db 10,13,"Enter a string:"

len1 equ $-msg1

section .bss

str1 resb 200 ;string declaration

result resb 16

section .text

global \_start

\_start:

;display

mov Rax,1

mov Rdi,1

mov Rsi,msg1

mov Rdx,len1

syscall

;store string

mov rax,0

mov rdi,0

mov rsi,str1

mov rdx,200

syscall

call display

;exit system call

mov Rax ,60

mov Rdi,0

syscall

%macro dispmsg 2

mov Rax,1

mov Rdi,1

mov rsi,%1

mov rdx,%2

syscall

%endmacro

display:

mov rbx,rax ; store no in rbx

mov rdi,result ;point rdi to result variable

mov cx,16 ;load count of rotation in cl

up1:

rol rbx,04 ;rotate no of left by four bits

mov al,bl ; move lower byte in al

and al,0fh ;get only LSB

cmp al,09h ;compare with 39h

jg add\_37 ;if greater than 39h skip add 37

add al,30h

jmp skip ;else add 30

add\_37:

add al,37h

skip:

mov [rdi],al ;store ascii code in result variable

inc rdi ; point to next byte

dec cx ; decrement counter

jnz up1 ; if not zero jump to repeat

dispmsg result,16 ;call to macro

ret

**OUTPUT:**

**CONCLUSION:**

In this practical session, we learnt how to display any number on monitor. (Convesion of hex to ascii number in ALP program).

**EXPERIMENT NO. 03**

**NAME:** Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers

**EXP NO: 03**

**AIM:** Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**Datatype in 80386:**

### Datatypes of 80386:

The 80386 supports the following  data types they are

* Bit
* Bit Field: A group of at the most 32 bits (4bytes)
* Bit String: A string of contiguous bits of maximum 4Gbytes in length.
* Signed Byte: Signed byte data
* Unsigned Byte: Unsigned byte data.
* Integer word: Signed 16-bit data.
* Long Integer: 32-bit signed data represented in 2's complement form.
* Unsigned Integer Word: Unsigned 16-bit data
* Unsigned Long Integer: Unsigned 32-bit data
* Signed Quad Word: A signed 64-bit data or four word data.
* Unsigned Quad Word: An unsigned 64-bit data.
* Offset: 16/32-bit displacement that points a memory location using any of the addressing modes.
* Pointer: This consists of a pair of 16-bit selector and 16/32-bit offset.
* Character: An ASCII equivalent to any of the alphanumeric or control characters.
* Strings: These are the sequences of bytes, words or double words. A string may contain minimum one byte and maximum 4 Gigabytes.
* BCD: Decimal digits from 0-9 represented by unpacked bytes.
* Packed BCD: This represents two packed BCD digits using a byte, i.e. from 00 to 99.

**Registers in 80386:**

****

* General Purpose Register: EAX, EBX, ECX, EDX
* Pointer register: ESP, EBP
* Index register: ESI, EDI
* Segment Register: CS, FS, DS, GS, ES, SS
* Eflags register: EFLAGS
* System Address/Memory management Registers : GDTR, LDTR, IDTR
* Control Register: Cr0, Cr1, Cr2, Cr3
* Debug Register : DR0, DR,1 DR2, DR3, DR4, DR5, DR6, DR7
* Test Register: TR0, TR,1 TR2, TR3, TR4, TR5, TR6, TR7

|  |  |  |
| --- | --- | --- |
| EAX | AX | AH,AL |
| EBX | BX | BH,BL |
| ECX | CX | CH,CL |
| EDX | DX | DH,DL |
| EBP | BP |  |
| EDI | DI |  |
| ESI | SI |  |
| ESP |  |  |

Size of operands in an Intel assembler instruction

* Specifying the size of an operand in Intel
* The size of the operand (byte, word, double word) is conveyed by the operand itself
  + - * + EAX means: a 32 bit operand
        + AX means: a 16 bit operand
    - AL means: a 8 bit operand The size of the source operand and the destination operand must be equal

**Addressing modes in 80386:**

The purpose of using addressing modes is as follows:

1. To give the programming versatility to the user.
2. To reduce the number of bits in addressing field of instruction.

1. Register addressing mode: MOV EAX, EDX

2. Immediate Addressing modes: MOV ECX, 20305060H

3. Direct Addressing mode: MOV AX, [1897 H]

4. Register Indirect Addressing mode MOV EBX, [ECX]

5. Based Mode MOV ESI, [EAX+23H]

6. Index Mode SUB COUNT [EDI], EAX

7. Scaled Index Mode MOV [ESI\*8], ECX

8. Based Indexed Mode MOV ESI, [ECX][EBX]

9. Based Index Mode with displacement EA=EBX+EBP+1245678H

10. Based Scaled Index Mode with displacement MOV [EBX\*8] [ECX+5678H], ECX

11. String Addressing modes:

12. Implied Addressing modes:

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**FLOWCHART:**

**EXPERIMENT NO. 04**

**NAME:** Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,\*, /) using suitable macros. Define procedure for each operation.

**NAME OF LABORATORY: MICROPROCESSOR LAB**

**EXP NO: 04**

**AIM:** Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,\*, /) using suitable macros. Define procedure for each operation.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**LIST OF INTERRRUPTS USED:** 80h

**LIST OF ASSEMBLER DIRECTIVES USED:** equ, db

**LIST OF MACROS USED:** scall

**LIST OF PROCEDURES USED:** add\_proc, sub\_proc, mul\_proc, div\_proc, disp64num

**ALGORITHM:**

**FLOWCHART:**

**PROGRAM:**

section .data

menumsg db 10,'\*\*\*\*\*\* Menu \*\*\*\*\*\*',

db 10,'1: Addition'

db 10,'2: Subtraction'

db 10,'3: Multiplication'

db 10,'4: Division'

db 10,10,'Enter your choice:: '

menumsg\_len: equ $-menumsg

addmsg db 10,'Welcome to additon',10

addmsg\_len equ $-addmsg

submsg db 10,'Welcome to subtraction',10

submsg\_len equ $-submsg

mulmsg db 10,'Welcome to Multiplication',10

mulmsg\_len equ $-mulmsg

divmsg db 10,'Welcome to Division',10

divmsg\_len equ $-divmsg

wrchmsg db 10,10,'You Entered a Wrong Choice....!',10

wrchmsg\_len equ $-wrchmsg

no1 dq 08h

no2 dq 02h

nummsg db 10

result dq 0

resmsg db 10,'Result is:'

resmsg\_len equ $-resmsg

qmsg db 10,'Quotient::'

qmsg\_len equ $-qmsg

rmsg db 10,'Remainder::'

rmsg\_len equ $-rmsg

nwmsg db 10

resh dq 0

resl dq 0

section .bss

choice resb 2

dispbuff resb 16

%macro scall 4

mov rax,%1

mov rdi,%2

mov rsi,%3

mov rdx,%4

syscall

%endmacro

section .text

global \_start

\_start:

up:

scall 1,1,menumsg,menumsg\_len

scall 0,0,choice,2

case1:cmp byte[choice],'1'

jne case2

call add\_proc

jmp up

case2:

cmp byte[choice],'2'

jne case3

call sub\_proc

jmp up

case3:

cmp byte[choice],'3'

jne case4

call mul\_proc

jmp up

case4:

cmp byte[choice],'4'

jne caseinv

call div\_proc

jmp up

caseinv:

scall 1,1, wrchmsg,wrchmsg\_len

exit:

mov eax,01

mov ebx,0

int 80h

add\_proc:

mov rax,[no1]

adc rax,[no2]

mov [result],rax

scall 1,1,resmsg,resmsg\_len

mov rbx,[result]

call disp64num

scall 1,1,nummsg,1

ret

sub\_proc:

mov rax,[no1]

subb rax,[no2]

mov [result],rax

scall 1,1,resmsg,resmsg\_len

mov rbx,[result]

call disp64num

scall 1,1,nummsg,1

ret

mul\_proc:

scall 1,1,mulmsg,mulmsg\_len

mov rax,[no1]

mov rbx,[no2]

mul rbx

mov [resh],rdx

mov [resl],rax

scall 1,1, resmsg,resmsg\_len

mov rbx,[resh]

call disp64num

mov rbx,[resl]

call disp64num

scall 1,1,nwmsg,1

ret

div\_proc:

scall 1,1,divmsg,divmsg\_len

mov rax,[no1]

mov rdx,0

mov rbx,[no2]

div rbx

mov [resh],rdx ;Remainder

mov [resl],rax ;Quotient

scall 1,1, rmsg,rmsg\_len

mov rbx,[resh]

call disp64num

scall 1,1, qmsg,qmsg\_len

mov rbx,[resl]

call disp64num

scall 1,1, nwmsg,1

ret

disp64num:

mov ecx,16

mov edi,dispbuff

dup1:

rol rbx,4

mov al,bl

and al,0fh

cmp al,09

jbe dskip

add al,07h

dskip: add al,30h

mov [edi],al

inc edi

loop dup1

scall 1,1,dispbuff,16

ret

**CONCLUSION:**

**EXPERIMENT NO. 05**

**NAME:** Write an X86/64 ALP to count number of positive and negative numbers from the array.

**EXP NO: 05**

**AIM:** Write an X86/64 ALP to count number of positive and negative numbers from the array.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

Mathematical numbers are generally made up of a sign and a value (magnitude) in which the sign indicates whether the number is positive, ( + ) or negative, ( – ) with the value indicating the size of the number, for example 23, +156 or -274. Presenting numbers is this fashion is called “sign-magnitude” representation since the left most digit can be used to indicate the sign and the remaining digits the magnitude or value of the number.

Sign-magnitude notation is the simplest and one of the most common methods of representing positive and negative numbers either side of zero, (0). Thus negative numbers are obtained simply by changing the sign of the corresponding positive number as each positive or unsigned number will have a signed opposite, for example, +2 and -2, +10 and -10, etc.

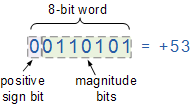
But how do we represent signed binary numbers if all we have is a bunch of one’s and zero’s. We know that binary digits, or bits only have two values, either a “1” or a “0” and conveniently for us, a sign also has only two values, being a “**+**” or a “**–**“.

Then we can use a single bit to identify the sign of a signed binary number as being positive or negative in value. So to represent a positive binary number (+n) and a negative (-n) binary number, we can use them with the addition of a sign.

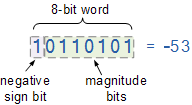
For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is “0”, this means the number is positive in value. If the sign bit is “1”, then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number in the usual unsigned binary number format way.

Then we can see that the Sign-and-Magnitude (SM) notation stores positive and negative values by dividing the “n” total bits into two parts: 1 bit for the sign and n–1 bits for the value which is a pure binary number. For example, the decimal number 53 can be expressed as an 8-bit signed binary number as follows.

### Positive Signed Binary Numbers



### Negative Signed Binary Numbers



**LIST OF INTERRRUPTS USED:** 80h

**LIST OF ASSEMBLER DIRECTIVES USED:** equ, db

**LIST OF MACROS USED:** print

**LIST OF PROCEDURES USED:** disp8num

**ALGORITHM:**

STEP 1: Initialize index register with the offset of array of signed numbers

STEP 2: Initialize ECX with array element count

STEP 3: Initialize positive number count and negative number count to zero

STEP 4: Perform MSB test of array element

STEP 5: If set jump to step 7

STEP 6: Else Increment positive number count and jump to step 8

STEP 7: Increment negative number count and continue

STEP 8: Point index register to the next element

STEP 9: Decrement the array element count from ECX, if not zero jump to step 4, else continue

STEP 10: Display Positive number message and then display positive number count

STEP 11: Display Negative number message and then display negative number count

STEP 12: EXIT

**FLOWCHART:**

**PROGRAM:**

;Write an ALP to count no. of positive and negative numbers from the array.

section .data

welmsg db 10,'Welcome to count positive and negative numbers in an array',10

welmsg\_len equ $-welmsg

pmsg db 10,'Count of +ve numbers::'

pmsg\_len equ $-pmsg

nmsg db 10,'Count of -ve numbers::'

nmsg\_len equ $-nmsg

nwline db 10

array dw 8505h,90ffh,87h,88h,8a9fh,0adh,02h,8507h

arrcnt equ 8

pcnt db 0

ncnt db 0

section .bss

dispbuff resb 2

%macro print 2 ;defining print function

mov eax, 4 ; this 4 commands signifies the print sequence

mov ebx, 1

mov ecx, %1 ; first parameter

mov edx, %2 ;second parameter

int 80h ;interrupt command

%endmacro

section .text ;code segment

global \_start ;must be declared for linker

\_start: ;tells linker the entry point ;i.e start of code

print welmsg,welmsg\_len ;print title

mov esi,array

mov ecx,arrcnt ;store array count in extended counter reg

up1: ;label

bt word[esi],15

;bit test the array number (15th byte) pointed by esi.

;It sets the carray flag as the bit tested

jnc pnxt ;jump if no carry to label pskip

inc byte[ncnt] ;if the 15th bit is 1 it signifies it is a ;negative no and so we ;use this command to increment ncnt counter.

jmp pskip ;unconditional jump to label skip

pnxt: inc byte[pcnt] ;label pnxt if there no carry then it is ;positive no

;and so pcnt is incremented

pskip: inc esi ;increment the source index but this ;instruction only increments it by 8 bit but the no’s in array ;are 16 bit word and hence it needs to be incremented twice.

inc esi

loop up1 ;loop it ends as soon as the array end “count” or

;ecx=0 loop automatically assums ecx has the counter

print pmsg,pmsg\_len ;prints pmsg

mov bl,[pcnt] ;move the positive no count to lower 8 bit of B reg

call disp8num ;call disp8num subroutine

print nmsg,nmsg\_len ;prints nmsg

mov bl,[ncnt] ;move the negative no count to lower 8 bits of b reg

call disp8num ;call disp8num subroutine

print nwline,1 ;New line char

exit:

mov eax,01

mov ebx,0

int 80h

disp8num:

mov ecx,2 ;move 2 in ecx ;Number digits to display

mov edi,dispbuff ;Temp buffer

dup1: ;this command sequence which converts hex to bcd

rol bl,4 ;Rotate number from bl to get MS digit to LS digit

mov al,bl ;Move bl i.e. rotated number to AL

and al,0fh ;Mask upper digit (logical AND the contents ;of lower8 bits of accumulator with 0fh )

cmp al,09 ;Compare al with 9

jbe dskip ;If number below or equal to 9 go to add only 30h

;add al,07h ;Else first add 07h to accumulator

dskip:

add al,30h ;Add 30h to accumulator

mov [edi],al ;Store ASCII code in temp buff (move contents ;of accumulator to the location pointed by edi)

inc edi

;Increment destination index i.e. pointer to ;next location in temp buff

loop dup1 ;repeat till ecx becomes zero

print dispbuff,2 ;display the value from temp buff

ret ;return to calling program

**OUTPUT:**

;[root@comppl2022 ~]# nasm -f elf64 Exp5.asm

;[root@comppl2022 ~]# ld -o Exp6 Exp5.o

;[root@comppl2022 ~]# ./Exp5

;Welcome to count +ve and -ve numbers in an array

;Count of +ve numbers::05

;Count of -ve numbers::03

;[root@comppl2022 ~]#

**CONCLUSION:**

|  |
| --- |
|  |

**EXPERIMENT NO. 06**

**NAME:** Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5- digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for: (a) HEX to BCD b) BCD to HEX (c) EXIT. Display proper strings to prompt the user while accepting the input and displaying the

result. (Wherever necessary, use 64-bit registers).

**EXP NO: 06**

**AIM:** Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5- digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for: (a) HEX to BCD b) BCD to HEX (c) EXIT. Display proper strings to prompt the user while accepting the input and displaying the

result. (Wherever necessary, use 64-bit registers).

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

Hexadecimal Number System:

The “Hexadecimal” or simply “Hex” numbering system uses the **Base of 16** system and are a popular choice for representing long binary values because their format is quite compact and much easier to understand compared to the long binary strings of 1’s and 0’s.

Being a Base-16 system, the hexadecimal numbering system therefore uses 16 (sixteen) different digits with a combination of numbers from 0  to 9 and A to F.

**Hexadecimal Numbers** is a more complex system than using just binary or decimal and is mainly used when dealing with computers and memory address locations.

Binary Coded Decimal(BCD) Number System:

Binary coded decimal (BCD) is a system of writing numerals that assigns a four-digit [binary](https://whatis.techtarget.com/definition/binary) code to each digit 0 through 9 in a [decimal](https://whatis.techtarget.com/definition/decimal) (base-10) numeral. The four-[bit](https://whatis.techtarget.com/definition/bit-binary-digit) BCD code for any particular single base-10 digit is its representation in binary notation, as follows:

0 = 0000  
1 = 0001  
2 = 0010  
3 = 0011  
4 = 0100  
5 = 0101  
6 = 0110  
7 = 0111  
8 = 1000  
9 = 1001

Numbers larger than 9, having two or more digits in the decimal system, are expressed digit by digit. For example, the BCD rendition of the base-10 number 1895 is

0001 1000 1001 0101

The binary equivalents of 1, 8, 9, and 5, always in a four-digit format, go from left to right.

The BCD representation of a number is not the same, in general, as its simple binary representation. In binary form, for example, the decimal quantity 1895 appears as

11101100111

|  |  |  |  |
| --- | --- | --- | --- |
| Decimal Number | 4-bit Binary Number | Hexadecimal Number | BCD Number |
| 0 | 0000 | 0 | 0000 0000 |
| 1 | 0001 | 1 | 0000 0001 |
| 2 | 0010 | 2 | 0000 0010 |
| 3 | 0011 | 3 | 0000 0011 |
| 4 | 0100 | 4 | 0000 0100 |
| 5 | 0101 | 5 | 0000 0101 |
| 6 | 0110 | 6 | 0000 0110 |
| 7 | 0111 | 7 | 0000 0111 |
| 8 | 1000 | 8 | 0000 1000 |
| 9 | 1001 | 9 | 0000 1001 |
| 10 | 1010 | A | 0001 0000 |
| 11 | 1011 | B | 0001 0001 |
| 12 | 1100 | C | 0001 0010 |
| 13 | 1101 | D | 0001 0011 |
| 14 | 1110 | E | 0001 0100 |
| 15 | 1111 | F | 0001 0101 |
| 16 | 0001 0000 | 10 (1+0) | 0001 0110 |
| 17 | 0001 0001 | 11 (1+1) | 0001 0111 |

**HEX to BCD**

Divide FFFF by 10 this FFFF is as decimal 65535 so

Division

65535 / 10 Quotient = 6553 Reminder = 5

6553 / 10 Quotient = 655 Reminder = 3

655 / 10 Quotient = 65 Reminder = 5

65 / 10 Quotient = 6 Reminder = 5

6 / 10 Quotient = 0 Reminder = 6

and we are pushing Reminder on stack and then printing it in reverse order.

**BCD to HEX**

1 LOOP : DL = 06 ; RAX = RAX \* RBX = 0 ; RAX = RAX + RDX = 06

2 LOOP : DL = 05 ; 60 = 06 \* 10 ; 65 = 60 + 5

3 LOOP : DL = 05 ; 650 = 60 \* 10 ; 655 = 650 + 5

4 LOOP : DL = 03 ; 6550 = 655 \* 10 ; 6553 = 6550 + 3

5 LOOP : DL = 06 ; 65530 = 6553 \* 10 ; 65535 = 65530 + 5

Hence final result is in RAX = 65535 which is 1111 1111 1111 1111 and when we print this it is

represented as FFFF.

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**STEP** 1: Start

**STEP** 2: Initialize data section.

**STEP 3:** Using Macro display the Menu for HEX to BCD, BCD to HEX and exit. Accept the choice

from user.

**STEP 4:** If choice = 1, call procedure for HEX to BCD conversion.

**STEP 5:** If choice = 2, call procedure for BCD to HEX conversion.

**STEP 6**: If choice = 3, terminate the program.

**Algorithm for procedure for HEX to BCD conversion:**

**STEP** 7: Accept 4-digit hex number from user.

**STEP** 8: Make count in RCX register 0.

**STEP** 9: Move accepted hex number in BX to AX.

**STEP 10:** Move base of Decimal number that is 10 in BX.

**STEP 11:** Move zero in DX.

**STEP** 12: Divide accepted hex number by 10. Remainder will return in DX.

**STEP** 13: Push remainder in DX on to stack.

**STEP** 14: Increment RCX counter.

**STEP** 15: Check whether AX contents are zero.

**STEP** 16: If it is not zero then go to step 5.

**STEP** 17: If AX contents are zero then pop remainders in stack in RDX.

**STEP** 18: Add 30 to get the BCD number.

**STEP** 19: Increment RDI for next digit and go to step 11.

**Algorithm for procedure for BCD to HEX:**

**STEP** 1: Accept 5-digit BCD number from user.

**STEP** 2: Take count RCX equal to 05.

**STEP** 3: Move 0A that is 10 in EBX.

**STEP** 4: Move zero in RDX register.

**STEP** 5: Multiply EBX with contents in EAX.

**STEP** 6: Move contents at RSI that is number accepted from user to DL.

**STEP** 7: Subtract 30 from DL.

**STEP** 8: Add contents of RDX to RAX and result will be in RAX.

**STEP** 9: Increment RSI for next digit and go to step 4 and repeat till RCX becomes zero.

**STEP** 10: Move result in EAX to EBX and call display procedure.

**FLOWCHART:**

**PROGRAM**

section .data

msg1 db 10,10,'###### Menu for Code Conversion ######'

db 10,'1: Hex to BCD'

db 10,'2: BCD to Hex'

db 10,'3: Exit'

db 10,10,'Enter Choice:'

msg1length equ $-msg1

msg2 db 10,10,'Enter 4 digit hex number::'

msg2length equ $-msg2

msg3 db 10,10,'BCD Equivalent:'

msg3length equ $-msg3

msg4 db 10,10,'Enter 5 digit BCD number::'

msg4length equ $-msg4

msg5 db 10,10,'Wrong Choice Entered....Please try again!!!',10,10

msg5length equ $-msg5

msg6 db 10,10,'Hex Equivalent::'

msg6length equ $-msg6

cnt db 0

section .bss

arr resb 06 ;common buffer for choice, hex and bcd input

dispbuff resb 08

ans resb 01

%macro disp 2

mov rax,01

mov rdi,01

mov rsi,%1

mov rdx,%2

syscall

%endmacro

%macro accept 2

mov rax,0

mov rdi,0

mov rsi,%1

mov rdx,%2

syscall

%endmacro

section .text

global \_start

\_start:

menu:

disp msg1,msg1length

accept arr,2 ; choice either 1,2,3 + enter

cmp byte [arr],'1'

jne l1

call hex2bcd\_proc

jmp menu

l1: cmp byte [arr],'2'

jne l2

call bcd2hex\_proc

jmp menu

l2: cmp byte [arr],'3'

je exit

disp msg5,msg5length

jmp menu

exit:

mov rax,60

mov rbx,0

syscall

hex2bcd\_proc:

disp msg2,msg2length

accept arr,5 ; 4 digits + enter

call conversion

mov rcx,0

mov ax,bx

mov bx,10 ;Base of Decimal No. system

l33: mov dx,0

div bx ; Divide the no by 10

push rdx ; Push the remainder on stack

inc rcx

inc byte[cnt]

cmp ax,0

jne l33

disp msg3,msg3length

l44: pop rdx ; pop the last pushed remainder from stack

add dl,30h ; convert it to ascii

mov [ans],dl

disp ans,1

dec byte[cnt]

jnz l44

ret

bcd2hex\_proc:

disp msg4,msg4length

accept arr,6 ; 5 digits + 1 for enter

disp msg6,msg6length

mov rsi,arr

mov rcx,05

mov rax,0

mov ebx,0ah

l55: mov rdx,0

mul ebx ; ebx \* eax = edx:eax

mov dl,[rsi]

sub dl,30h

add rax,rdx

inc rsi

dec rcx

jnz l55

mov ebx,eax ; store the result in ebx

call disp32\_num

ret

conversion:

mov bx,0

mov ecx,04

mov esi,arr

up1:

rol bx,04

mov al,[esi]

cmp al,39h

jbe l22

sub al,07h

l22: sub al,30h

add bl,al

inc esi

loop up1

ret

; the below procedure is to display 32 bit result in ebx why 32 bit & not 16 ;bit; because 5 digit bcd no ranges between 00000 to 99999 & for ;65535 ans ;is FFFF

; i.e if u enter the no between 00000-65535 u are getting the answer between

;0000-FFFF, but u enter i/p as 99999 urans is greater than 16 bit which is ;not; fitted in 16 bit register so 32 bit register is taken frresult

disp32\_num:

mov rdi,dispbuff

mov rcx,08 ; since no is 32 bit,no of digits 8

l77:

rol ebx,4

mov dl,bl

and dl,0fh

add dl,30h

cmp dl,39h

jbe l66

add dl,07h

l66:

mov [rdi],dl

inc rdi

dec rcx

jnz l77

disp dispbuff+3,5 ;Dispays only lower 5 digits as upper three are '0'

ret

;OUTPUT OF PROGRAM

;[admin@localhost ~]$ vi conv.nasm

;[admin@localhost ~]$ nasm -f elf64 conv.nasm -o conv.o

;[admin@localhost ~]$ ld -o conv conv.o

;[admin@localhost ~]$ ./conv

;###### Menu for Code Conversion ######

;1: Hex to BCD

;2: BCD to Hex

;3: Exit

;Enter Choice:1

;Enter 4 digit hex number::FFFF

;BCD Equivalent::65535

;###### Menu for Code Conversion ######

;1: Hex to BCD

;2: BCD to Hex

;3: Exit

;Enter Choice:1

;Enter 4 digit hex number::00FF

;BCD Equivalent::255

;###### Menu for Code Conversion ######

;1: Hex to BCD

;2: BCD to Hex

;3: Exit

;Enter Choice:1

;Enter 4 digit hex number::000F

;BCD Equivalent::15

;###### Menu for Code Conversion ######

;1: Hex to BCD

;2: BCD to Hex

;3: Exit

;Enter Choice:2

;Enter 5 digit BCD number::65535

;Hex Equivalent::0FFFF

;###### Menu for Code Conversion ######

;1: Hex to BCD

;2: BCD to Hex

;3: Exit

;Enter Choice:2

;Enter 5 digit BCD number::00255

;Hex Equivalent::000FF

;###### Menu for Code Conversion ######

;1: Hex to BCD

;2: BCD to Hex

;3: Exit

**CONCLUSION:**

**EXPERIMENT NO. 07**

**NAME:** Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction.

**EXP NO: 07**

**AIM:** Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**FLOWCHART:**

**EXPERIMENT NO. 08**

**NAME:** Write X86/64 ALP to perform non-overlapped block transfer without string specific

instructions. Block containing data can be defined in the data segment.

**EXP NO: 08**

**AIM:** Write X86/64 ALP to perform non-overlapped block transfer without string specific

instructions. Block containing data can be defined in the data segment.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**FLOWCHART:**

**EXPERIMENT NO. 09**

**NAME:** Write X86/64 ALP to perform overlapped block transfer with string specific instructions

Block containing data can be defined in the data segment.

**EXP NO: 09**

**AIM:** Write X86/64 ALP to perform overlapped block transfer with string specific instructions

Block containing data can be defined in the data segment.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**FLOWCHART:**

**EXPERIMENT NO. 10**

**NAME:** Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use

successive addition and add and shift method. (use of 64-bit registers is expected).

**EXP NO: 10**

**AIM:** Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use

successive addition and add and shift method. (use of 64-bit registers is expected).

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**FLOWCHART:**

**EXPERIMENT NO. 11**

**NAME:** Write X86 Assembly Language Program (ALP) to implement following OS commands

i) COPY, ii) TYPE Using file operations. User is supposed to provide command line arguments

**EXP NO: 11**

**AIM:** Write X86 Assembly Language Program (ALP) to implement following OS commands

i) COPY, ii) TYPE Using file operations. User is supposed to provide command line arguments

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**FLOWCHART:**

**EXPERIMENT NO. 12**

**NAME:** Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program\_1 execution and write FAR PROCEDURES in Program\_2 for the rest of

the processing. Use of PUBLIC and EXTERN directives is mandatory.

**EXP NO: 12**

**AIM:** Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program\_1 execution and write FAR PROCEDURES in Program\_2 for the rest of

the processing. Use of PUBLIC and EXTERN directives is mandatory.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**FLOWCHART:**

**EXPERIMENT NO. 13**

**NAME:** Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.

**EXP NO: 13**

**AIM:** Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**FLOWCHART:**

**EXPERIMENT NO. 14**

**NAME:** Write an X86/64 ALP password program that operates as follows: a. Do not display what is actually typed instead display asterisk (“\*”). If the password is correct display, “access is granted” else display “Access not Granted”

**EXP NO: 14**

**AIM:** Write an X86/64 ALP password program that operates as follows: a. Do not display what is actually typed instead display asterisk (“\*”). If the password is correct display, “access is granted” else display “Access not Granted”

**OBJECTIVES:**

* To understand assembly language programming instruction set.
* To understand different assembler directives with example.
* To apply instruction set for implementing X86/64 bit assembly language programs

**ENVIRONMENT:**

* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**THEORY:**

**LIST OF INTERRRUPTS USED:**

**LIST OF ASSEMBLER DIRECTIVES USED:**

**LIST OF MACROS USED:**

**LIST OF PROCEDURES USED:**

**ALGORITHM:**

**FLOWCHART:**