AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

Department Of Computer Engineering

Microprocessor Lab Experiments

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AISSMS IOIT

SE COMPTER ENGINEERING

SUBMITTED BY

Ankit Patil
ERP No.- 50
Teams No.-32

2020 -2021

EXPERIMENT NO. 01

AIM: Write an X86/64 ALP to accept a number and display it using macros.

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.

MACROS:

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.

For example, a very common need for programs is to write a string of characters in the screen. For displaying a string of characters, you need the following sequence of instructions –

```
mov     edx,len                 ;message length
mov     ecx,msg                 ;message to write
mov     ebx,1     ;file descriptor (stdout)
mov     eax,4     ;system call number (sys_write)
int     0x80      ;call kernel
```

In the above example of displaying a character string, the registers EAX, EBX, ECX and EDX have been used by the INT 80H function call. So, each time you need to display on screen, you need to save these registers on the stack, invoke INT 80H and then restore the original value of the registers from the stack. So, it could be useful to write two macros for saving and restoring data.

We have observed that, some instructions like IMUL, IDIV, INT, etc., need some of the information to be stored in some particular registers and even return values in some specific register(s). If the program was already using those registers for keeping important data, then the existing data from these registers should be saved in the stack and restored after the instruction is executed.

AlGORITHM:

Step 1: Start

Step 2: Show the message, “Enter a number”, using the display macro.

Step 3: Accept the number from the user using the input macro and store it in a variable.

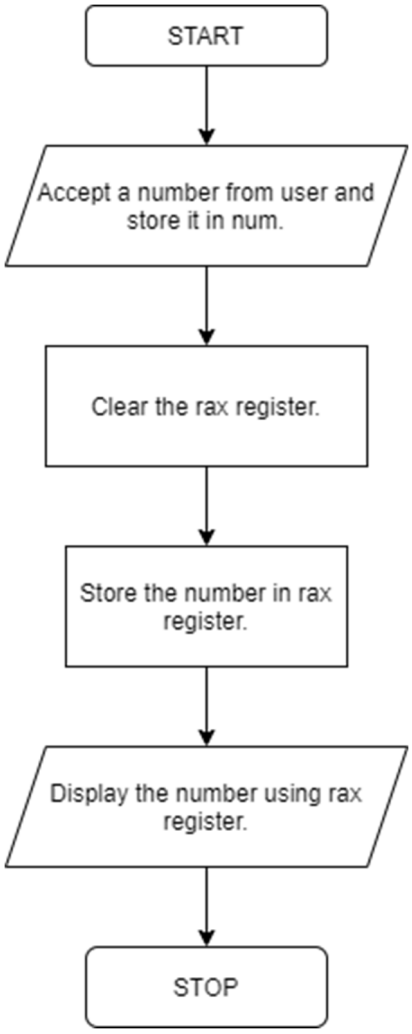
Step 4: Clear the rax register.

Step 5: Copy the number into the rax register.

Step 6: Display the number present in rax register using macro.

Step 7: Stop

FLOWCHART:



PROGRAM:

```
%macro dis_inp 4
mov rax,%1
mov rdi,%2
mov rsi,%3
mov rdx,%4
syscall
%endmacro

section .data

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

len1 equ $-msg1
```

```
msg2 db 10,13,"Entered number is: "

len2 equ $-msg2

section .bss

num resd 2

section .text

global _start

_start:

;display

dis_inp 01h,01h,msg1,len1

;input

dis_inp 00h,00h,num,2

xor rax,rax
mov rax,num

;display

dis_inp 01h,01h,msg2,msg2
dis_inp 01,01,rax,2

;exit system call

mov rax ,60
```

```
mov rdi,0


syscall
```

OUTPUT-

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd
Exp1AcceptNumber
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp1AcceptNumber$ nasm -f elf64 acceptnumber.asm -o acceptnumber.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp1AcceptNumber$ ld -o acceptnumber acceptnumber.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp1AcceptNumber$ ./acceptnumber

Enter a number: 29

Entered number is: 29
```

CONCLUSION: In this practical session we learnt how to insert and display number using macros.

EXPERIMENT NO. 02

AIM: Write an X86/64 ALP to accept five 64bit 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 X86/64bit assembly language programs

ENVIRONMENT:

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

THEORY:

Hexadecimal Number System

Hexadecimal Number System is one the type of Number Representation techniques, in which there value of base is 16. That means there are only 16 symbols or possible digit values, there are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Introduction to Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). The base value is index 0 and the difference between the two indexes is the offset.

MACROS:

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.

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.

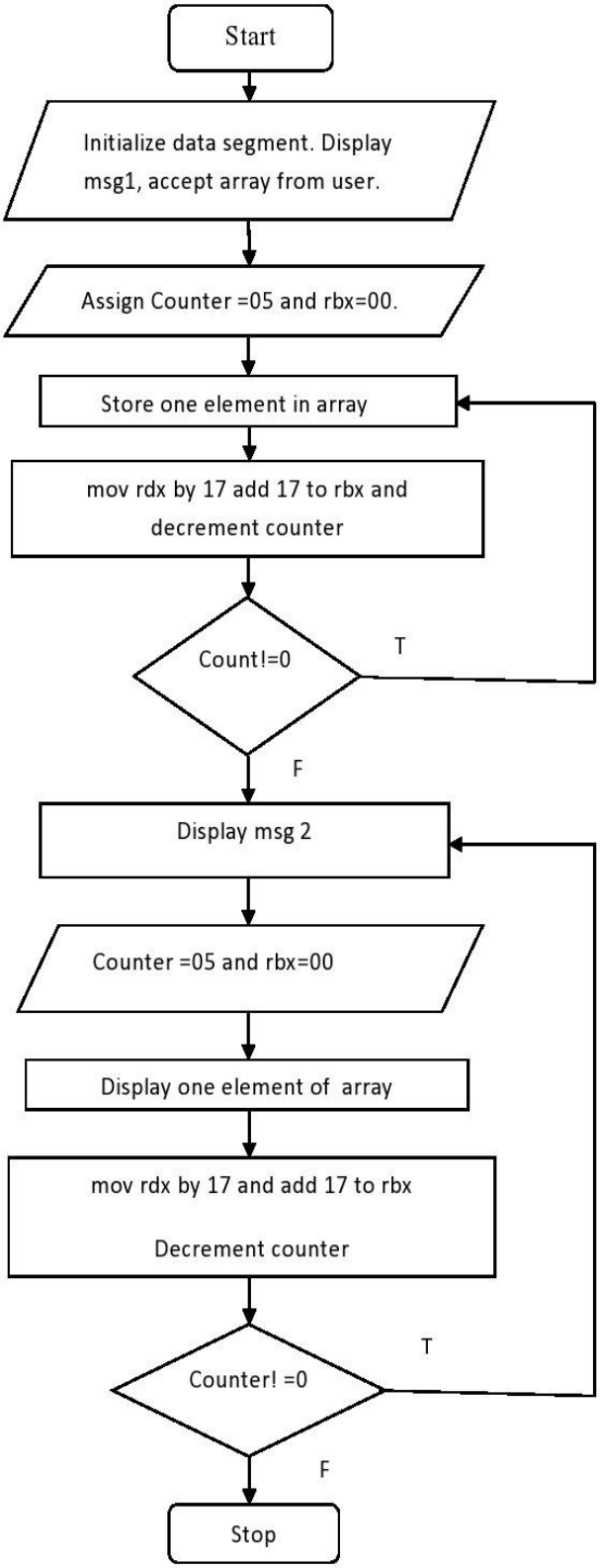
STEP 13: Add 17 to rbx.

STEP 14: Decrement Counter.

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

STEP 16: Stop

FLOWCHART:


Start
Initialize data segment. Display msg1, accept array from user.
Assign Counter =05 and rbx=00.
Store one element in array
mov rdx by 17 add 17 to rbx and decrement counter
Count!=0
T
F
Display msg 2
Counter =05 and rbx=00
Display one element of array
mov rdx by 17 and add 17 to rbx
Decrement counter
Counter! =0
T
F
Stop

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:

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd
Exp2_5HexNumbers
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp2_5HexNumbers$ nasm -f elf64 5hexnumbers.asm -o 5hexnumbers.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp2_5HexNumbers$ ld -o 5hexnumbers 5hexnumbers.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp2_5HexNumbers$ ./5hexnumbers

Enter 5 64 bit numbers
29
05
2
00
1

Entered 5 64 bit numbers
29
05
2
00
1
```

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

EXPERIMENT NO. 03

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:

String:

A string is a data type used in programming, such as an integer and floating point unit, but is used to represent text rather than numbers. ... For example, the word "hamburger" and the phrase "I ate 3 hamburgers" are both strings. Even "12345" could be considered a string, if specified correctly.

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.

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

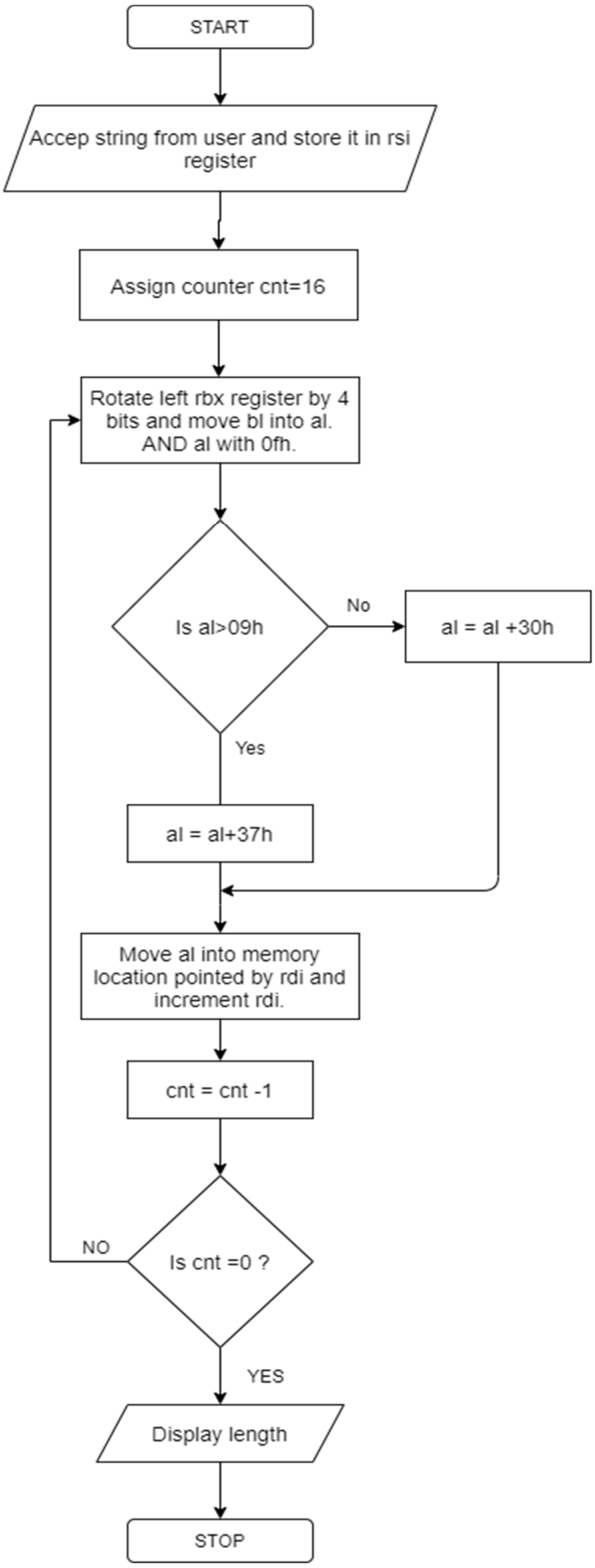
STEP 15: Loop the statement till counter value becomes zero

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:


START
Accep string from user and store it in rsi register
Assign counter cnt=16
Rotate left rbx register by 4 bits and move bl into al. AND al with 0fh.
Is al>09h
No
al = al +30h
Yes
al = al+37h
Move al into memory location pointed by rdi and increment rdi.
cnt = cnt -1
Is cnt =0 ?
NO
YES
Display length
STOP

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 dl
                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:

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd Exp3String
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp3String$
nasm -f elf64 string.asm -o string.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp3String$ ld
-o string string.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp3String$
./string

Enter a string:My self Kaustubh Kabra
0000000000000016

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp3String$
nasm -f elf64 string.asm -o string.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp3String$
./string

Enter a string:Kaustubh Kabra
000000000000000E

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp3String$
nasm -f elf64 string.asm -o string.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp3String$
./string

Enter a string:12345678
0000000000000008(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp3String$ ./string
```

Enter a string:1
0000000000000001

CONCLUSION: In this practical session we learnt how to accept string and display its length.

EXPERIMENT NO. 04

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.

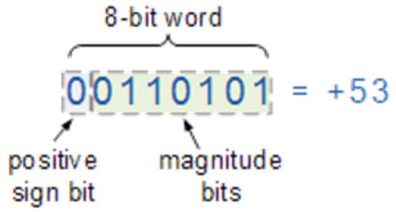
But how do we represent signed binary numbers if all we have is a bunch of one's and zero's. We know that binary digits, or bits only have two values, either a "1" or a "0" and conveniently for us, a sign also has only two values, being a "+" or a "–".

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.

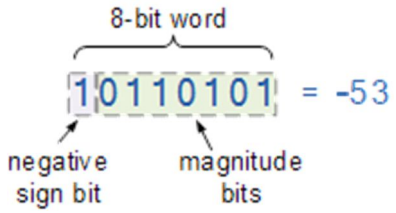
For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is "0", this means the number is positive in value. If the sign bit is "1", then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number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

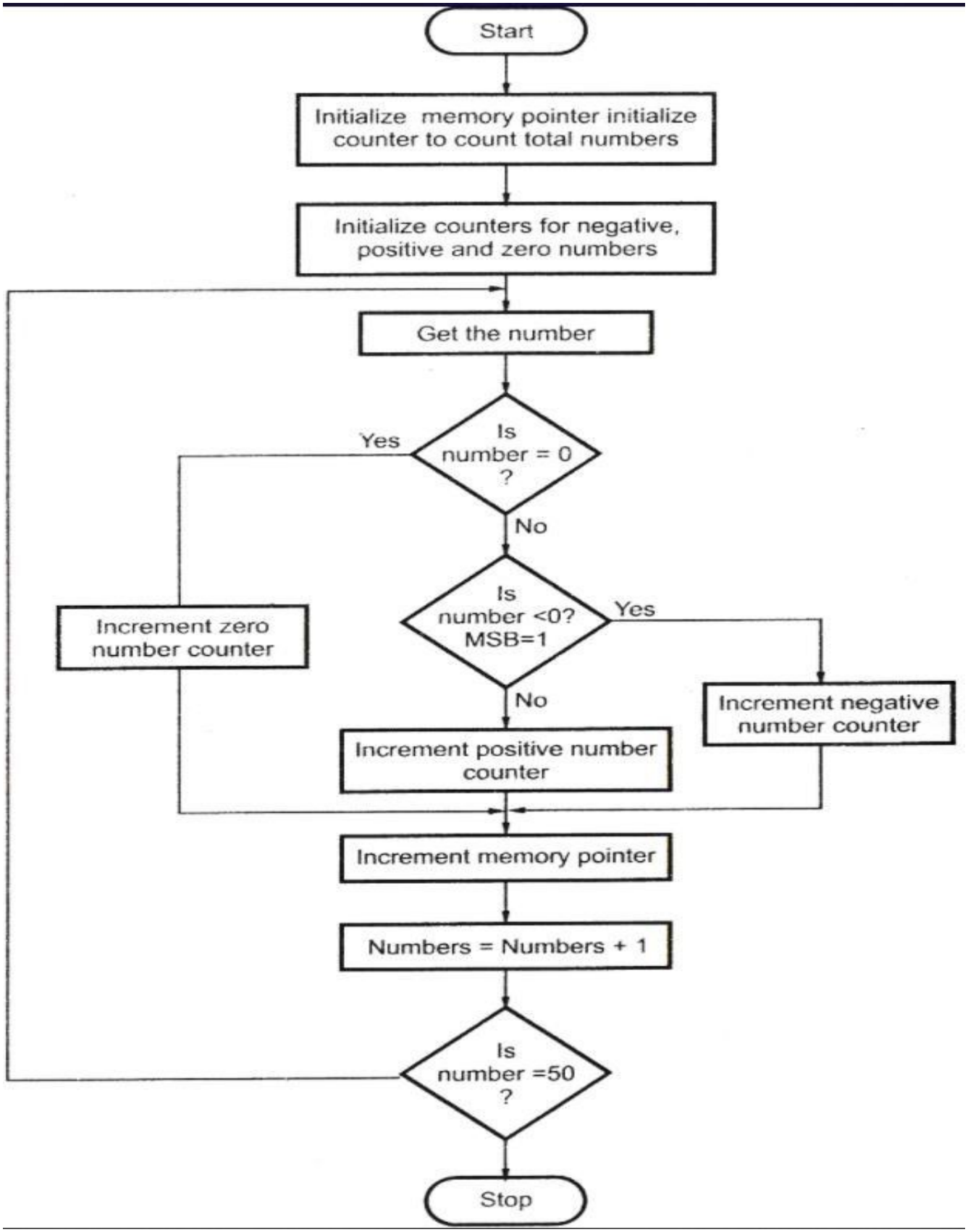
STEP 8: Point index register to the next element

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:


Start
Initialize memory pointer initialize counter to count total numbers
Initialize counters for negative, positive and zero numbers
Get the number
Is number = 0 ?
Yes
No
Is number <0? MSB=1
Yes
No
Increment zero number counter
Increment negative number counter
Increment positive number counter
Increment memory pointer
Numbers = Numbers + 1
Is number =50 ?
Stop

PROGRAM:

```
section .data
 ncnt db 0
 pcnt db 0
 array: dw -80H,4CH,-3FH
 len equ 3
 msg1: db 'positive numbers are:',0xa
 len1: equ $-msg1
 msg2: db 'negative numbers are:',0xa
 len2: equ $-msg2

section .bss
 buff resb 02

section .text

 global _start
 _start:

  mov rsi,array
  mov rcx,03

  A1:
  bt word[rsi],15
  jnc A
  inc byte[ncnt]
  jmp skip
```

```
A:
inc byte[pcnt]

skip:
inc rsi
inc rsi
loop A1

mov rax,1
mov rdi,1
mov rsi,msg1
mov rdx,len1
syscall

mov bl,[pcnt]
mov rdi,buff
mov rcx,02

call display

mov rax,1
mov rdi,1
mov rsi,msg2
mov rdx,len2
syscall

mov bl,[ncnt]
mov rdi,buff
```

```
mov rcx,02

call display

mov rax,60
mov rdi,0
syscall

display:
rol bl,4
mov al,bl
and al,0FH
cmp al,09
jbe B
        add al,07h
B:
add al,30H
mov[rdi],al
inc rdi
loop display

mov rax,1
mov rdi,1
mov rsi,buff
mov rdx,02
syscall
ret
```

OUTPUT:

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd
Exp4PositiveNegative
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp4PositiveNegative$ nasm -f elf64 positive_negative.asm -o positive_negative.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp4PositiveNegative$ ld -o positive_negative positive_negative.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp4PositiveNegative$ ./positive_negative
positive numbers are:
01
negative numbers are:
02
```

CONCLUSION: In this practical session we learnt to count number of positive and negative numbers from the array.

EXPERIMENT NO. 05

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:

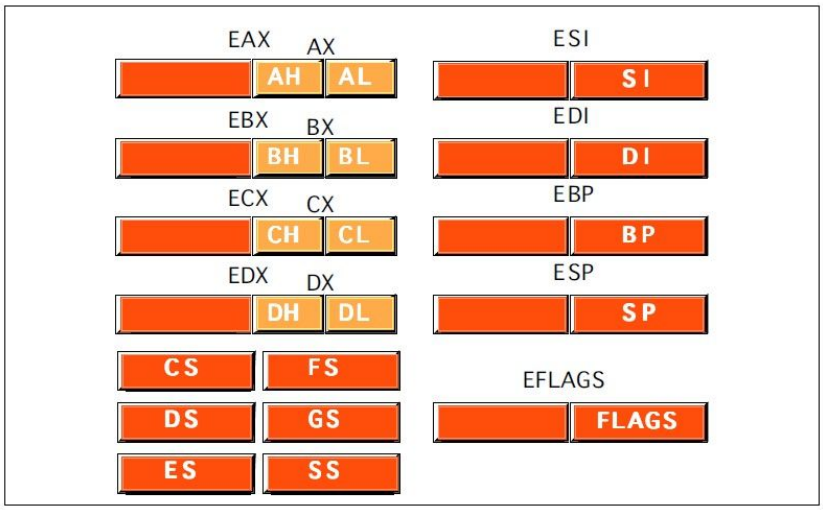
- Operating System: 64-bit Open source Linux or its derivative.
- Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
- Text Editor: geditor

THEORY:

Datatypes of 80386:

- 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		

ALGORITHM:

Step 1: Start

Step 2: Initialize Block Size and get address of first element.

Step 3: Load the data from the memory.

Step 4: Decrement Block Size and Increment address of first element.

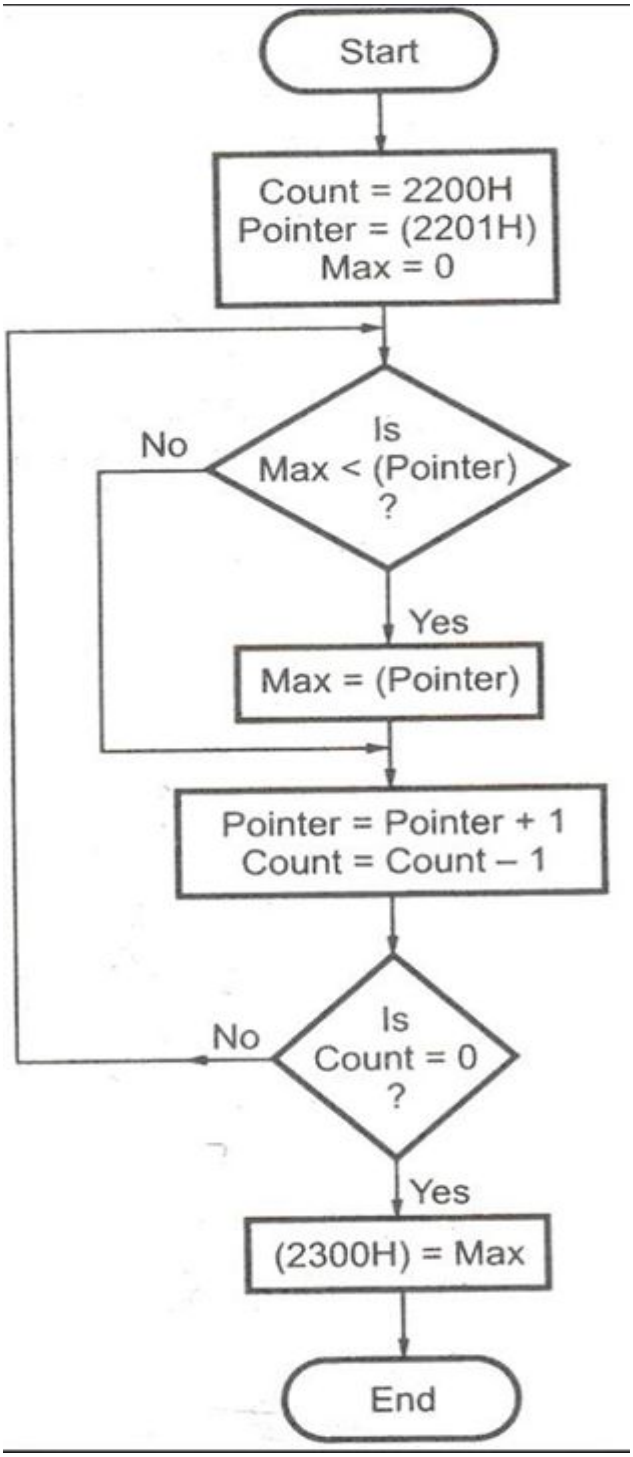
Step 5: Store first element in A.

Step 6: Compare A with other elements, if A is smaller then store that element in B otherwise compare with next element.

Step 7: The value of B is the answer.

Step 8: Stop.

FLOWCHART:



PROGRAM:

```
section .data
        welmsg db 10,"The maximum number in the array is: ",10
        welmsg_len equ $-welmsg
        array dQ
8abc123456781234h,90ff123456781234h,7700123456781234h,8800123456781234h,8a9fdd345678123
4h
        arrcnt equ 5

section .bss
        dispbuff resb 2
        buf resb 16
%macro print 2
        mov eax,4
        mov ebx,1
        mov ecx,%1
        mov edx,%2
        int 0x80
%endmacro

section .text
        global _start
_start:
        print welmsg,welmsg_len
        mov esi,array
        mov rax,[esi]
        mov ecx,arrcnt
```

```
up1:    add esi,8


        mov rbx,[esi]
        cmp rax,rbx
        jnc skip
        mov rax,rbx
        mov edi,buf
skip:   loop up1
        mov rbx,rax
        mov edi,buf
        mov ecx,16
dispup1:
        rol rbx,4
        mov dl,bl
        and dl,0fh
        add dl,30h
        cmp dl,39h
        jbe dispskip1
        add dl,07h
dispskip1:
        mov [edi],dl
        inc edi
        loop dispup1
        print buf,16
        ret
```

OUTPUT:

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd
Exp5Largest
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp5Largest$
nasm -f elf64 largest.asm -o largest.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp5Largest$
ld -o largest largest.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments/Exp5Largest$
./largest

The maximum number in the array is:
90FF12345678123
```

CONCLUSION: In this practical session we learnt to find the largest of given Byte / Word / Dword / 64-Bit Numbers.

EXPERIMENT NO. 06

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:

Real Mode:

Real mode, also called real address mode, is an operating mode of all x86-compatible CPUs. Real mode is characterized by a 20-bit segmented memory address space (giving exactly 1 MiB of addressable memory) and unlimited direct software access to all addressable memory, I/O addresses and peripheral hardware. Real mode provides no support for memory protection, multitasking, or code privilege levels.

Protected Mode:

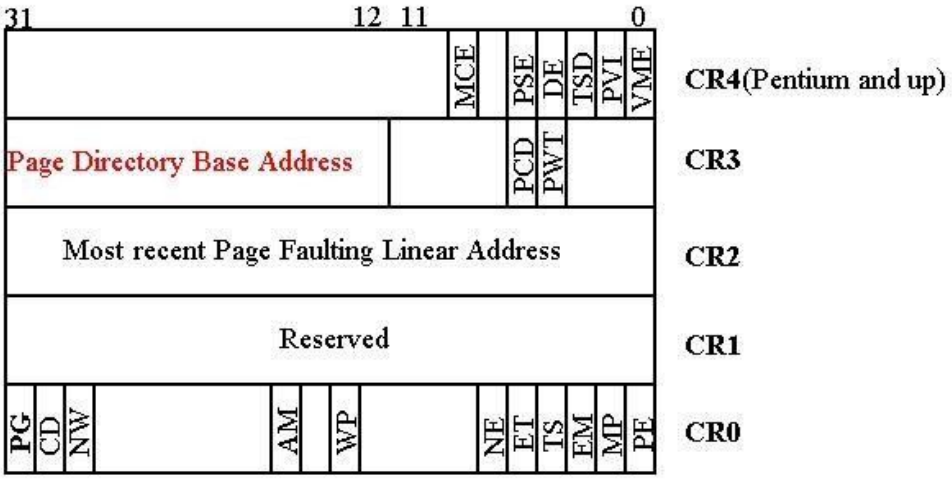
In computing, protected mode, also called protected virtual address mode is an operational mode of x86-compatible central processing units (CPUs). It allows system software to use features such as virtual memory, paging and safe multi-tasking designed to increase an operating system's control over application software.
When a processor that supports x86 protected mode is powered on, it begins executing instructions in real mode, in order to maintain backward compatibility with earlier x86 processors. Protected mode may only be entered after the system software sets up several descriptor tables and enables the Protection Enable (PE) bit in the control register 0 (CR0).

Interrupt Descriptor Table Register

This register holds the 32-bit base address and 16-bit segment limit for the interrupt descriptor table (IDT). When an interrupt occurs, the interrupt vector is used as an index to get a gate

descriptor from this table. The gate descriptor contains a far pointer used to start up the interrupt handler.

Control Register :



Local Descriptor Table Register

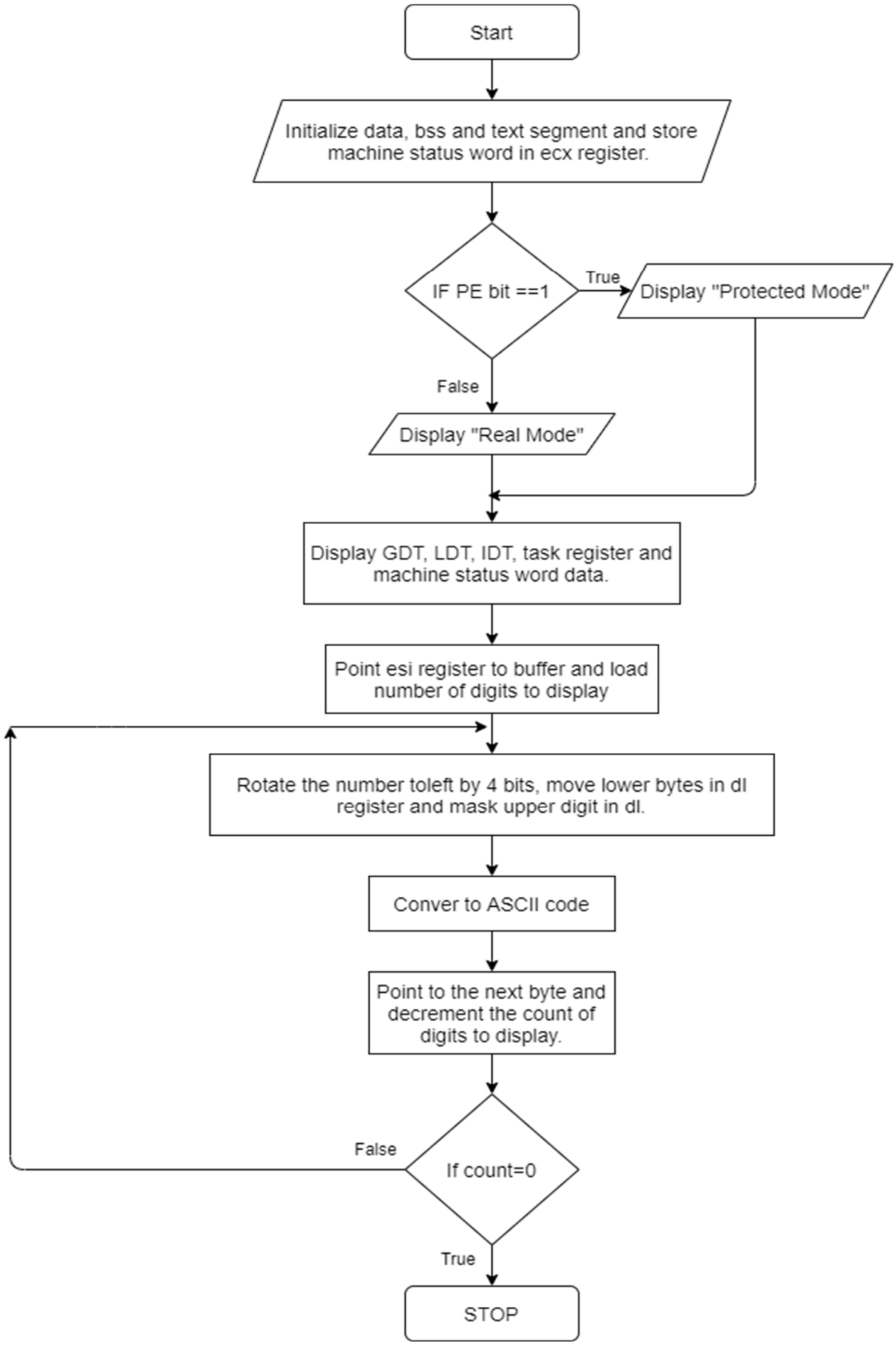
This register holds the 32-bit base address, 16-bit segment limit, and 16-bit segment selector for the local descriptor table (LDT). The segment which contains the LDT has a segment descriptor in the GDT. There is no segment descriptor for the GDT. When a reference is made to data in memory, a segment selector is used to find a segment descriptor in the GDT or LDT. A segment descriptor contains the base address for a segment

ALGORITHM:

1. Start
2. Initialize data segment
3. Initialize bss segment
4. Initialize text segment
5. Store the machine status word into eax
6. Check PE bit, if 1=>Protected mode, else Real mode
7. Display GDT data
8. Display LDT data
9. Display IDT data

10. Display task register data
11. Display machine status word data
12. Point ESI to buffer
13. Load number of digit to display
14. Rotate number left by 4-bit
15. Move lower byte in DL
16. Mark upper digit of byte in DL
17. Add 30h to calculate ASCII code
18. Compare with 39h, if less than 39h skip adding 07,else add 07
19. Store ASCII code in buffer
20. Point to next byte
21. Decrement the count of digit to display
22. If not zero, jump to step 14 and repeat, else stop
23. Stop

FLOWCHART:


Start
Initialize data, bss and text segment and store machine status word in ecx register.
IF PE bit ==1
True
Display "Protected Mode"
False
Display "Real Mode"
Display GDT, LDT, IDT, task register and machine status word data.
Point esi register to buffer and load number of digits to display
Rotate the number toleft by 4 bits, move lower bytes in dl register and mask upper digit in dl.
Conver to ASCII code
Point to the next byte and decrement the count of digits to display.
If count=0
False
True
STOP

PROGRAM:

```
%macro scall 4
mov rax,%1
mov rdi,%2
mov rsi,%3
mov rdx,%4
syscall
%endmacro

Section .data
title: db 0x0A,"----Assignment 6-----", 0x0A
title_len: equ $-title
regmsg: db 0x0A,"***** REGISTER CONTENTS *****"
regmsg_len: equ $-regmsg
gmsg: db 0x0A,"Contents of GDTR : "
gmsg_len: equ $-gmsg
lmsg: db 0x0A,"Contents of LDTR : "
lmsg_len: equ $-lmsg
imsg: db 0x0A,"Contents of IDTR : "
imsg_len: equ $-imsg
tmsg: db 0x0A,"Contents of TR : "
tmsg_len: equ $-tmsg
mmsg: db 0x0A,"Contents of MSW : "
mmsg_len: equ $-mmsg
realmsg: db "---- In Real mode. ----"
realmsg_len: equ $-realmsg
protmsg: db "---- In Protected Mode. ----"
```

```
protmsg_len: equ $-protmsg
cnt2:db 04H
newline: db 0x0A

Section .bss
g: resd 1
 resw 1
l: resw 1
idtr: resd 1
 resw 1
msw: resd 1

tr: resw 1
value :resb 4

Section .text
global _start
_start:
scall 1,1,title,title_len
smsw [msw]
mov eax,dword[msw]
bt eax,0
jc next
scall 1,1,realmsg,realmsg_len
jmp EXIT
next:
 scall 1,1,protmsg,protmsg_len
```

```
 scall 1,1, regmsg,regmsg_len
;printing register contents
scall 1,1,gmsg,gmsg_len
SGDT [g]
mov bx, word[g+4]
call HtoA
mov bx,word[g+2]
call HtoA
mov bx, word[g]
call HtoA

;--- LDTR CONTENTS----t find valid values for all labels after 1001 passes, giving up.

scall 1,1, lmsg,lmsg_len
SLDT [l]
mov bx,word[l]
call HtoA

;--- IDTR Contents -------
scall 1,1,imsg,imsg_len
SIDT [idtr]
mov bx, word[idtr+4]
call HtoA
mov bx,word[idtr+2]
call HtoA
mov bx, word[idtr]
```

```
call HtoA

;---- Task Register Contents -0-----
scall 1,1, tmsg,tmsg_len
mov bx,word[tr]
call HtoA

;------- Content of MSW ---------
scall 1,1,mmsg,mmsg_len
mov bx, word[msw+2]
call HtoA
mov bx, word[msw]
call HtoA
scall 1,1,newline,1
EXIT:
mov rax,60
mov rdi,0
syscall

;------HEX TO ASCII CONVERSION METHOD ----------------
HtoA: ;hex_no to be converted is in bx //result is stored in rdi/user defined variable
mov rdi,value
mov byte[cnt2],4H
aup:
rol bx,04
mov cl,bl
and cl,0FH
```

```
cmp cl,09H
jbe ANEXT
ADD cl,07H
ANEXT:
add cl, 30H
mov byte[rdi],cl
INC rdi
dec byte[cnt2]
JNZ aup
scall 1,1,value,4
ret
```

OUTPUT:

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd
Exp6Registers
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp6Registers$ nasm -f elf64 registers.asm -o registers.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp6Registers$ ld -o registers registers.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp6Registers$ ./registers

----Assignment 6-----
---- In Protected Mode. ----
***** REGISTER CONTENTS ***

Contents of GDTR : 0002D000007F

Contents of LDTR : 0000

Contents of IDTR : 000000000FFF

Contents of TR : 0000

Contents of MSW : FFFFFE00
```

CONCLUSION: In this practical session we learnt to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identified CPU type using CPUID instruction.

EXPERIMENT NO. 07

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:

- Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
- Let the number of bytes N = 05.
- We will have to initialize this as count.
- We know that source address is in the ESI register and destination address is in the EDI register.
- For block transfer without string instruction, move contents at ESI to accumulator and from accumulator to memory location of EDI and increment ESI and EDI for next content transfer.
- For block transfer with string instruction, clear the direction flag. Move the data from source location to the destination location using string instruction.

Instructions needed:

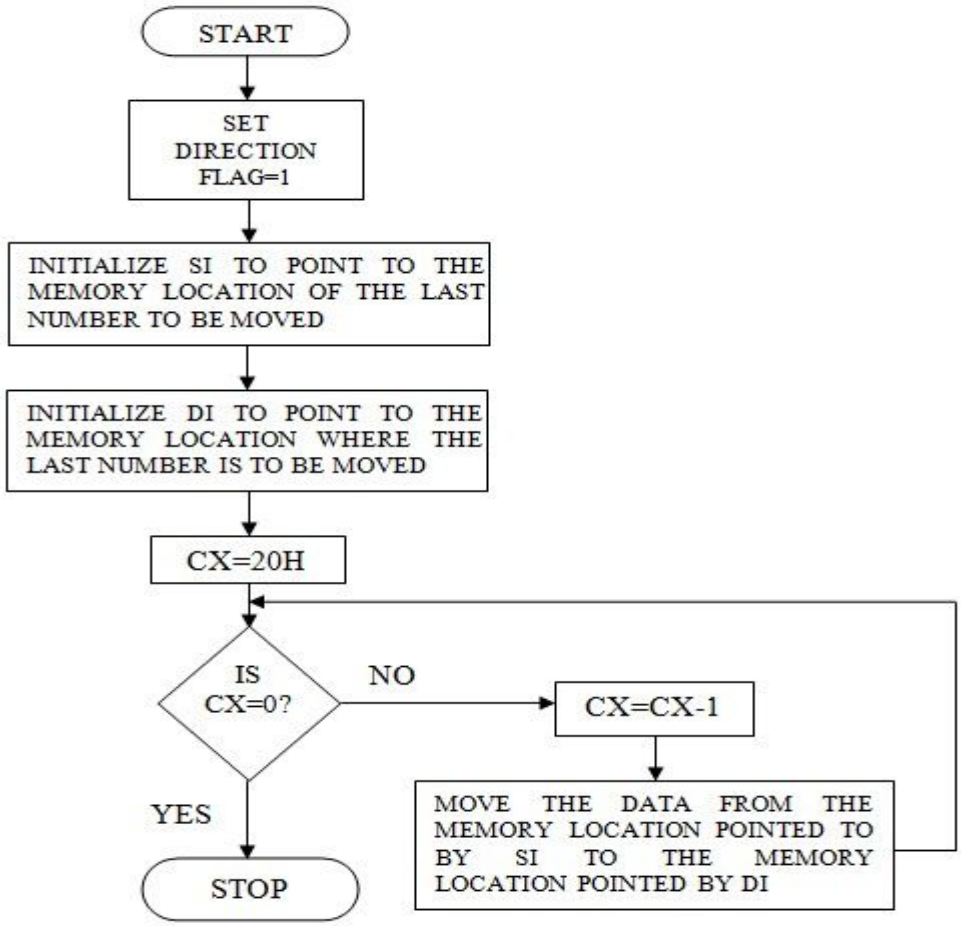
1. **MOVSB**-This is a string instruction and it moves string byte from source to destination.
2. **REP-** This is prefix that are applied to string operation. Each prefix cause the string instruction that follows to be repeated the number of times indicated in the count register.
3. **CLD-** Clear Direction flag. ESI and EDI will be incremented and DF = 0
4. **STD-** Set Direction flag. ESI and EDI will be incremented and DF = 1
5. **ROL**-Rotates bits of byte or word left.
6. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word.
7. **INC**-Increments specified byte/word by1.

8. **DEC**-Decrements specified byte/word by1.
9. **JNZ**-Jumps if not equal to Zero.
10. **JNC**-Jumps if no carry is generated.
11. **CMP**-Compares to specified bytes or words.
12. **JBE**-Jumps if below or equal.
13. **ADD**-Adds specified byte to byte or word to word.
14. **CALL**-Transfers the control from calling program to procedure.
15. **RET**-Return from where call is made.

ALGORITHM:

1. Initialize ESI and EDI with source and destination address.
2. Move count in ECX register.
3. Move contents at ESI to accumulator and from accumulator to memory location of EDI.
4. Increment ESI and EDI to transfer next content.
5. Repeat procedure till count becomes zero.

FLOWCHART:



PROGRAM:

```
section .data
msg1 db "the source block is:",0ah,0dh
len1: equ $-msg1
msg2 db "the destination block is:",0ah,0dh
len2: equ $-msg2
arr1 db "se computer",0ah
len: equ $-arr1

section .bss
arr2: resb len
%macro disp 2
mov rax,01
mov rdi,01
mov rsi,%1
mov rdx,%2
syscall
%endmacro
section .text
global _start
_start:
mov rsi,arr1
mov rdi,arr2
mov rcx,len

  xor al,al     ;without using movsb
  up:       ; copy the string character by character to the destination
```

```
  mov al,[rsi]
  mov [rdi],al
 inc rsi
 inc rdi
 dec rcx
 jnz up

;cld
;rep movsb                    ;comment need to be removed for movsb
      ; copy entire string at a time to destination
disp msg1,len1
disp arr1,len
disp msg2,len2
disp arr2,len

mov rax,3ch
mov rdi,00
syscall
```

OUTPUT:

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd
Exp7NonOverlapped
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp7NonOverlapped$ nasm -f elf64 nonoverlapped.asm -o nonoverlapped.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp7NonOverlapped$ ld -o nonoverlapped nonoverlapped.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp7NonOverlapped$ ./nonoverlapped
the source block is:
Kaustubh Kabra
the destination block is:
Kaustubh Kabra
```

CONCLUSION: In this practical session we learnt how to perform non-overlapped block transfer without string specific instructions.

EXPERIMENT NO. 08

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:

- Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
- Let the number of bytes N = 05.
- We will have to initialize this as count.
- Overlap the source block and destination block.
- We know that source address is in the ESI register and destination address is in the EDI register.
- For block transfer without string instruction, move contents at ESI to accumulator and from accumulator to memory location of EDI and decrement ESI and EDI for next content transfer.
- For block transfer with string instruction, set the direction flag. Move the data from source location to the destination location using string instruction.

Instructions needed:

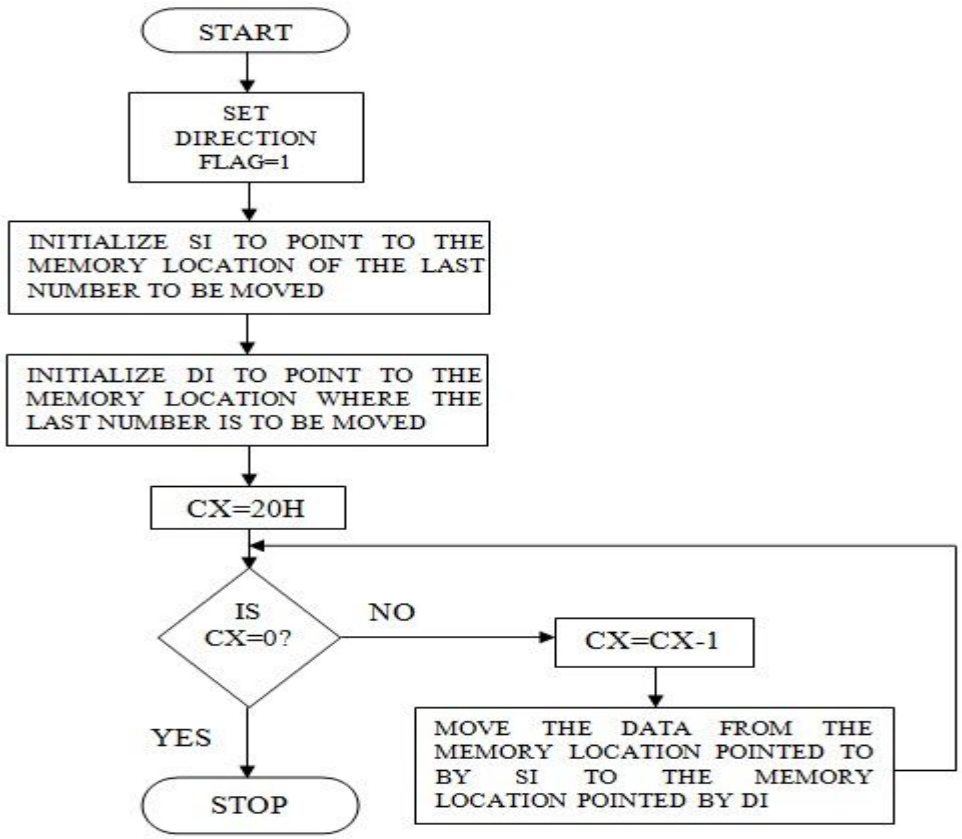
1. **MOVSB**-This is a string instruction and it moves string byte from source to destination.
2. **REP-** This is prefix that are applied to string operation. Each prefix cause the string instruction that follows to be repeated the number of times indicated in the count register.
3. **CLD-** Clear Direction flag. ESI and EDI will be incremented and DF = 0
4. **STD-** Set Direction flag. ESI and EDI will be decremented and DF = 1
5. **ROL**-Rotates bits of byte or word left.
6. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word.
7. **INC**-Increments specified byte/word by1.
8. **DEC**-Decrements specified byte/word by1.
9. **JNZ**-Jumps if not equal to Zero.

10. **JNC**-Jumps if no carry is generated.
11. **CMP**-Compares to specified bytes or words.
12. **JBE**-Jumps if below or equal.
13. **ADD**-Adds specified byte to byte or word to word.
14. **CALL**-Transfers the control from calling program to procedure.
15. **RET**-Return from where call is made.

ALGORITHM:

1. Initialize ESI and EDI with source and destination address.
2. Move count in ECX register.
3. Move source block's and destination block's last content address in ESI and EDI.
4. Move contents at ESI to accumulator and from accumulator to memory location of EDI.
5. Decrement ESI and EDI to transfer next content.
6. Repeat procedure till count becomes zero.

FLOWCHART:



PROGRAM:

```
section .data
msg db "enter an offset:",10
len: equ $-msg

arr1 db "se computer",0ah
len1: equ $-arr1

section .bss
n resb 2
len4 resb 2

%macro disp 2
mov rax,01
mov rdi,01
mov rsi,%1
mov rdx,%2
syscall
%endmacro

%macro inn 2
mov rax,00
mov rdi,00
mov rsi,%1
mov rdx,%2
syscall
%endmacro

section .text
global _start
_start:

disp msg,len
inn n,2

cmp byte[n],39h
jng skip
sub byte[n],07h
skip:
sub byte[n],30h

mov rsi,arr1+len1-1
mov rdi,rsi
mov rcx,len1
xor rax,rax
mov al,[n]
```

```
    add rdi,rax

    ;up:
    ;mov al,[rsi]
    ;mov [rdi],al
    ;dec rdi
    ;dec rsi
    ;dec cl
    ;jnz up
    ;mov al,[n]
    ;mov len4,al

    std
    rep movsb

    disp arr1,len1+len4
    mov rax,3ch
    mov rdi,00
    syscall
```

OUTPUT:

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd
Exp8Overlapped
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ nasm -f elf64 overlapped.asm -o overlapped.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ ld -o overlapped overlapped.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ ./overlapped
enter an offset:
7
KaustubKaustubh Kabra

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ ./overlapped
enter an offset:
6
KaustuKaustubh Kabra

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ ./overlapped
enter an offset:
5
KaustKaustubh Kabra

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ ./overlapped
```

```
enter an offset:
4
KausKaustubh Kabra

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ ./overlapped
enter an offset:
3
KauKaustubh Kabra

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ ./overlapped
enter an offset:
2
KaKaustubh Kabra

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ ./overlapped
enter an offset:
1
KKaustubh Kabra

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp8Overlapped$ ./overlapped
enter an offset:
0
Kaustubh Kabra
```

CONCLUSION: In this practical session we learnt how to perform non-overlapped block transfer with string specific instructions.

EXPERIMENT NO. 09

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:

A) Multiplication of two numbers using successive addition method:

Historically, computers used a "shift and add" algorithm for multiplying small integers. Both base 2 long multiplication and base 2 peasant multiplications reduce to this same algorithm. In base 2, multiplying by the single digit of the multiplier reduces to a simple series of logical AND operations. Each partial product is added to a running sum as soon as each partial product is computed. Most currently available microprocessors implement this or other similar algorithms (such as Booth encoding) for various integer and floating-point sizes in hardware multipliers or in microcode.

On currently available processors, a bit-wise shift instruction is faster than a multiply instruction and can be used to multiply (shift left) and divide (shift right) by powers of two. Multiplication by a constant and division by a constant can be implemented using a sequence of shifts and adds or subtracts. For example, there are several ways to multiply by 10 using only bit-shift and addition.

((x << 2) + x) << 1 # Here 10*x is computed as (x*2^2 + x)*2
(X << 3) + (x << 1) # Here 10*x is computed as x*2^3 + x*2 In some cases such sequences of shifts and adds or subtracts will outperform hardware multipliers and especially dividers. A division by a number of the form 2^n or $2^n \pm 1$ often can be converted to such a short sequence. These types of sequences have to always be used for computers that do not have a "multiply" instruction,[4] and can also be used by extension to floating point numbers if one replaces the shifts with computation of *2*x* as *x*+*x*, as these are logically equivalent.

Example:

Consider that a byte is present in the AL register and second byte is present in the BL register.

Step 1: We have to multiply the byte in AL with the byte in BL.

Step 2: We will multiply the numbers using successive addition method.

Step 3: In successive addition method, one number is accepted and other number is taken as a counter. The first number is added with itself, till the counter decrements to zero.

Step 4: Result is stored in DX register. Display the result, using display routine.

For example: AL = 12 H, BL = 10 H **Solution:**

Result = 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H+ 12H Result = 0120 H

Algorithm to Multiply Two 8 Bit Numbers Successive Addition Method:

1. Initialize the data segment.
2. Get the first number.
3. Get the second number as counter.
4. Initialize result = 0.
5. Result = Result + First number.
6. Decrement counter
7. If count ¹ 0, go to step V.
8. Display the result.
9. Stop.

B) Multiply Two 8 Bit Numbers using Add and Shift Method:

Program should take first and second numbers as input to the program. Now it should implement certain logic to multiply 8 bit Numbers using Add and Shift Method. Consider that one byte is present in the AL register and another byte is present in the BL register. We have to multiply the byte in AL with the byte in BL.

Steps for multiply the numbers using add and shift method:

Step 1: In this method, you add number with itself

Step 2: Rotate the other number each time and shift it by one bit to left along with carry. If carry is present add the two numbers.

Step 3: Initialize the count to 4 as we are scanning for 4 digits. Decrement counter each time the bits are added. The result is stored in AX. Display the result.

For example: AL = 11 H, BL = 10 H, Count = 4 **Solution:**

Step I : AX= 11

\+ **11**
22H

Rotate BL by one bit to left along with carry

BL=10 H 0 0001 0000

CY 10

After Rotate BL by one bit to left along with carry

BL= 0 0010 0000

CY 20

Step II : Now decrement counter count = 3.

Check for carry, carry is not there so add number with itself.

AX=22

\+ **22**
44H

Rotate BL to left,

BL= 0 0010 0000

CY 20

After Rotate BL by one bit to left along with carry

BL= 0 0100 0000

CY 40

Carry is not there.
Decrement count, count=2

Step III : Add number with itself

AX=44

\+ **44**

88H

Rotate BL to left,

BL= 0 0100 0000

CY 40

After Rotate BL by one bit to left along with carry

BL= 0 1000 0000

CY 80

Carry is not there.

Step IV : Decrement counter count = 1.

Add number with itself as carry is not there.

AX=88

+ **88**

110H

Rotate BL to left,

BL= 0 1000 0000

CY 80

After Rotate BL by one bit to left along with carry

BL= 1 0000 0000

CY 00

Carry is there.

Step V : Decrement counter = 0.

Carry is present.

\ add AX, BX

0110 i.e.11 H

\+ 0000 i.e.10 H

0110 H 0110H

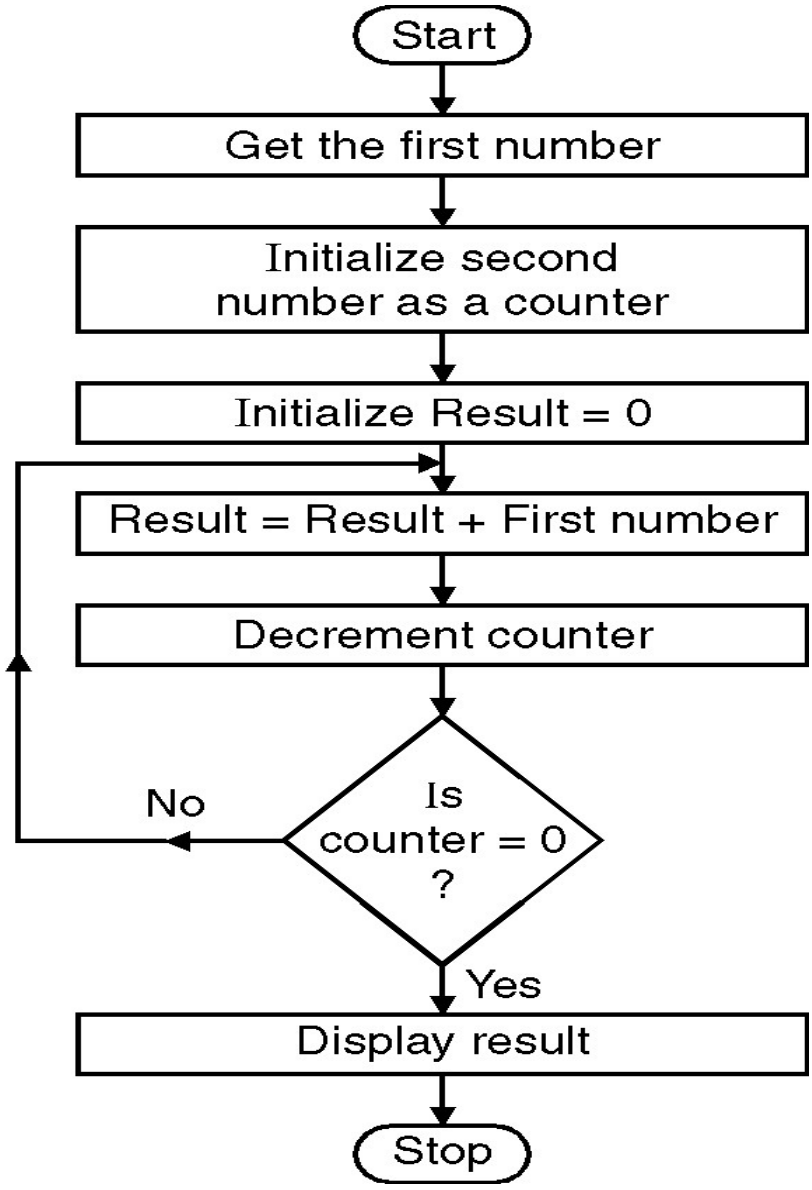
Algorithm to Multiply Two 8 Bit Numbers using Add and Shift Method:

1. Initialize the data segment.
2. Get the first number.
3. Get the second number.
4. Initialize count = 04.

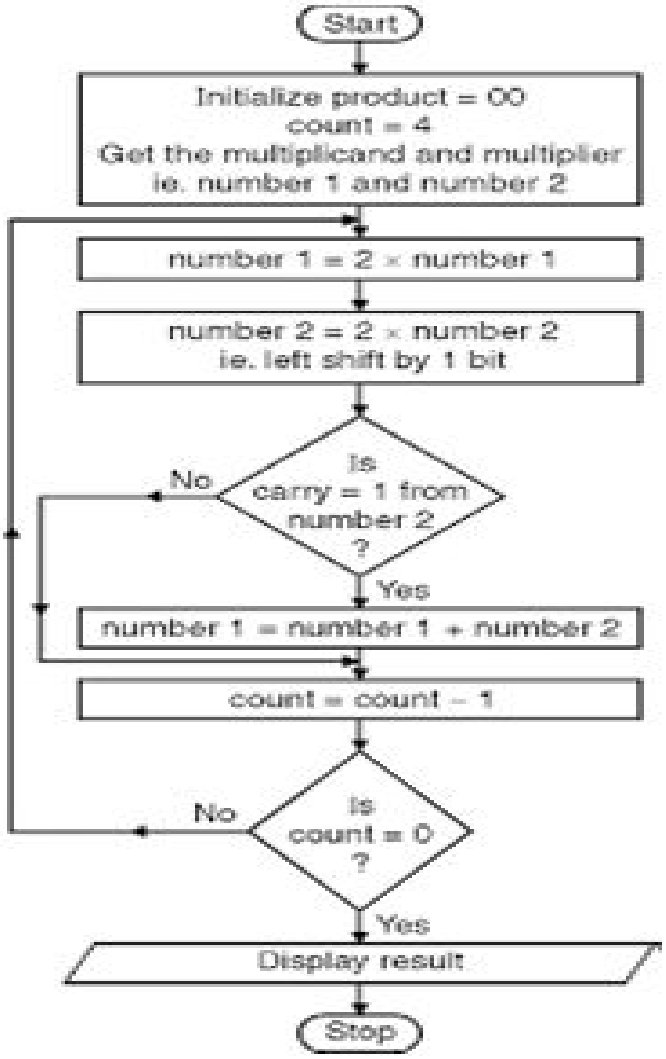
5. number 1 = number 1 ´ 2.
6. Shift multiplier to left along with carry.
7. Check for carry, if present go to step VIII else go to step IX.
8. number 1 = number1 + shifted number 2.
9. Decrement counter.
10. If not zero, go to step V.
11. Display the result.
12. Stop.

FLOWCHART:

Successive Addition



Add and Shift method



PROGRAM:

```
section .data

msg db 'Enter two digit Number::',0xa
msg_len equ $-msg
res db 10,'Multiplication of elements is::'
res_len equ $-res
choice db 'Enter your Choice:',0xa
      db'1.Successive Addition',0xa
      db '2.Add and Shift method',0xa
      db '3.Exit',0xa
choice_len equ $-choice

section .bss
num resb 03
```

```
num1 resb 01
result resb 04
cho resb 2


section .text

global _start
_start:

 xor rax,rax
 xor rbx,rbx
 xor rcx,rcx
 xor rdx,rdx
 mov byte[result],0
 mov byte[num],0
 mov byte[num1],0

       mov rax,1
 mov rdi,1
 mov rsi,choice
 mov rdx,choice_len
 syscall



       mov rax,0                ;; read choice
 mov rdi,0
 mov rsi,cho
 mov rdx,2
 syscall



 cmp byte[cho],31h          ;; comparing choice
 je a

 cmp byte[cho],32h
 je b

       jmp exit

 a:  call Succe_addition

 jmp _start

 b:  call Add_shift
```

```
 jmp _start

exit:
 mov rax,60
 mov rdi,0
 syscall

convert:                        ;; ASCII to Hex conversion
 xor rbx,rbx
 xor rcx,rcx
 xor rax,rax

 mov rcx,02
 mov rsi,num
 up1:
 rol bl,04

 mov al,[rsi]
 cmp al,39h
 jbe p1
 sub al,07h
 jmp p2
 p1: sub al,30h
 p2: add bl,al
 inc rsi
 loop up1
ret

display:                ;; Hex to ASCII conversion
 mov rcx,4
 mov rdi,result
 dup1:
 rol bx,4
 mov al,bl
 and al,0fh
 cmp al,09h
 jbe p3
 add al,07h
 jmp p4
 p3: add al,30h
 p4:mov [rdi],al
 inc rdi
 loop dup1

       mov rax,1
 mov rdi,1
```

```
mov rsi,result
mov rdx,4
syscall


ret


Succe_addition:

      mov rax,1
mov rdi,1
mov rsi,msg
mov rdx,msg_len
syscall



      mov rax,0
mov rdi,0
mov rsi,num
mov rdx,3
syscall



call convert
mov [num1],bl

      mov rax,1
mov rdi,1
mov rsi,msg
mov rdx,msg_len
syscall



      mov rax,0
mov rdi,0
mov rsi,num
mov rdx,3
syscall


call convert
xor rcx,rcx
xor rax,rax
```

```
mov rax,[num1]

repet:
add rcx,rax
dec bl
jnz repet

mov [result],rcx

      mov rax,1
mov rdi,1
mov rsi,res
mov rdx,res_len
syscall



mov rbx,[result]

call display
ret



Add_shift:

      mov rax,1
mov rdi,1
mov rsi,msg
mov rdx,msg_len
syscall



      mov rax,0
mov rdi,0
mov rsi,num
mov rdx,3
syscall



call convert
mov [num1],bl


      mov rax,1
```

```
mov rdi,1
mov rsi,msg
mov rdx,msg_len
syscall



      mov rax,0
mov rdi,0
mov rsi,num
mov rdx,3
syscall



call convert

mov [num],bl

xor rbx,rbx
xor rcx,rcx
xor rdx,rdx
xor rax,rax
mov dl,08
mov al,[num1]
mov bl,[num]

p11:
      shr bx,01
jnc p
add cx,ax
p:
      shl ax,01
dec dl
jnz p11

mov [result],rcx



      mov rax,1
mov rdi,1
mov rsi,res
mov rdx,res_len
syscall

;dispmsg res,res_len
```

```
 mov rbx,[result]
 call display

ret
```

OUTPUT:

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd
Exp9Multiplication
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp9Multiplication$ nasm -f elf64 multiplication_addandshift.asm -o
multiplication_addandshift.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp9Multiplication$ ld -o multiplication_addandshift multiplication_addandshift.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp9Multiplication$ ./multiplication_addandshift
Enter your Choice:
1.Successive Addition
2.Add and Shift method
3.Exit
1
Enter two digit Number::
10
Enter two digit Number::
14

Multiplication of elements is::0140
Enter your Choice:
1.Successive Addition
2.Add and Shift method
3.Exit
2
Enter two digit Number::
12
Enter two digit Number::
13

Multiplication of elements is::0156
Enter your Choice:
1.Successive Addition
2.Add and Shift method
3.Exit
3
```

CONCLUSION: In this practical session we learnt how to perform multiplication of two 8-bit hexadecimal numbers using successive addition and add and shift method.

EXPERIMENT NO. 10

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:

A recursive procedure is one that calls itself. There are two kind of recursion: direct and indirect. In direct recursion, the procedure calls itself and in indirect recursion, the first procedure calls a second procedure, which in turn calls the first procedure.

Recursion could be observed in numerous mathematical algorithms. For example, consider the case of calculating the factorial of a number. Factorial of a number is given by the equation −

Fact (n) = n * fact (n-1) for n > 0

For example: factorial of 5 is 1 x 2 x 3 x 4 x 5 = 5 x factorial of 4 and this can be a good example of showing a recursive procedure. Every recursive algorithm must have an ending condition, i.e., the recursive calling of the program should be stopped when a condition is fulfilled. In the case of factorial algorithm, the end condition is reached when n is 0.

Recursion occurs when a procedure calls itself. The following for example is a recursive procedure:

```
Recursive proc
callRecursive
ret
Recursive  endp
```

Of course the CPU will never execute the ret instruction at the end of this procedure. Upon entry into Recursive this procedure will immediately call itself again and control will never pass to the ret instruction. In this particular case run away recursion results in an infinite loop.

In many respects recursion is very similar to iteration (that is the repetitive execution of a loop).

The following code also produces an infinite loop:

```
Recursive proc

jmp Recursive

ret

Recursive endp
```

There is however one major difference between these two implementations. The former version of Recursive pushes a return address onto the stack with each invocation of the subroutine. This does not happen in the example immediately above (since the jmp instruction does not affect the stack).

Like a looping structure recursion requires a termination condition in order to stop infinite recursion. Recursive could be rewritten with a termination condition as follows:

```
Recursive proc

dec ax

jzQuitRecurse call

Recursive


QuitRecurse: ret


Recursiveendp
```

This modification to the routine causes Recursive to call itself the number of times appearing in the ax register. On each call Recursive decrements the ax register by one and calls itself again. Eventually Recursive decrements ax to zero and returns. Once this happens the CPU executes a string of ret instructions until control returns to the original call to Recursive.

So far however there hasn't been a real need for recursion. After all you could efficiently code this procedure as follows:

```
Recursive proc
RepeatAgain: dec ax
jnzRepeatAgain
ret
Recursive endp
```

Both examples would repeat the body of the procedure the number of times passed in the ax register. As it turns out there are only a few recursive algorithms that you cannot implement in an iterative fashion. However many recursively implemented algorithms are more efficient than their iterative counterparts and most of the time the recursive form of the algorithm is much easier to understand.

ALGORITHM:

Step1: Start

Step2: Accept the number from user

Step3: Convert that number into Hexadecimal (ascii to hex)

Step4: Compare accepted number with 1. If it is equal to 1 go to step 5 else push the number on stack and decrement the number and go to step 4

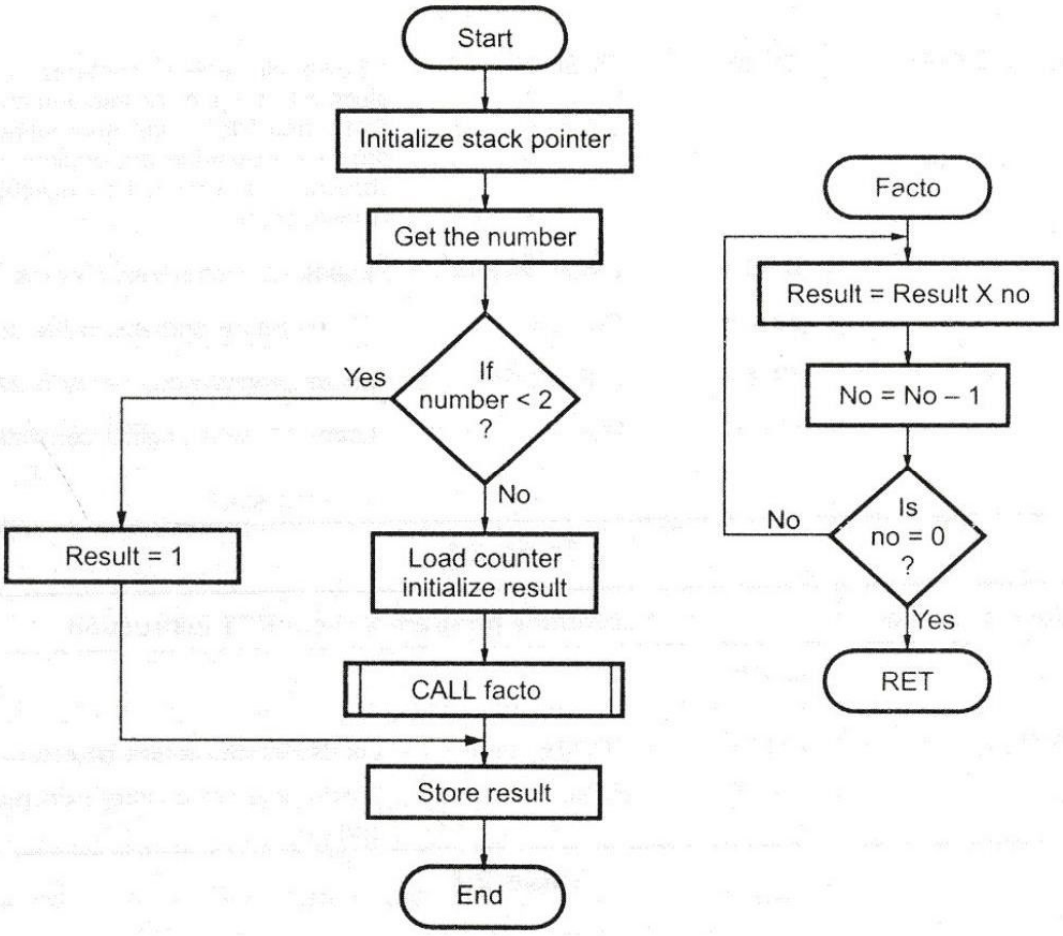
Step5: pop the content of the stack and multiply with number
Step6: Repeat step until stack becomes empty

Step7: Convert number from Hex to ASCII

Step8: Print the number

Step9: Stop

FLOWCHART:



PROGRAM:

```
%macro disp 2
mov rax,01h
mov rdi,01h
mov rsi ,%1
mov rdx,%2
syscall
%endmacro

%macro inn 2
mov rax,00h
mov rdi,00h
mov rsi,%1
mov rdx,%2
```

```
syscall
%endmacro

section .data
msg1 db "Enter the 8 bit number:",0ah,0dh
len1 equ $  -msg1

msg2 db "The factorial of given 8 bit number is:",0ah,0dh
len2 equ $  -msg2

msg3 db "The factorial for 0 or 1 is:",0ah,0dh
len3 equ $  -msg3

zeroonefact db "0001"
zeroonefactlen equ $-zeroonefact

section .bss
num resb 3
res resb 16

section .text
global _start
_start:

disp msg1, len1
inn num, 3
call accept

xor rax, rax
mov ax,bx
cmp ax,01h
jbe onezero

call factorial
call display
mov rax,60
mov rdi,0
syscall



onezero:
disp msg3,  len3
disp zeroonefact,  zeroonefactlen

;exit:
```

```
accept:
        mov rsi,num
        mov cl,04
        xor rbx,rbx
        mov ch,02
        up:
        cmp byte[rsi],39h
        jng sk
        sub byte[rsi],07h
        sk:
        sub byte[rsi],30h
        rol bl,cl
        add bl,[rsi]
        inc rsi
        dec ch
        jnz up
ret

factorial:
 xor rbx, rbx
 mov rbx,rax
 up1:sub rbx ,01
 mul rbx
 cmp rbx,01
 jne up1
ret

display:
 mov rsi,res
 mov ch,16
 mov cl,04

 again1:
 rol rax,cl
 mov bl,al
 and bl,0fh
 cmp bl,09h
 jng skip2
 add bl,07h
 skip2:
 add bl, 30h
 mov [rsi],bl
 inc rsi
 dec ch
 jnz again1
```

```
 disp msg2, len2
 disp res, 16
ret
```

OUTPUT:

```
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~$ cd Desktop
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop$ cd MPL\ Experiments/
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL Experiments$ cd
Exp10Factorial
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp10Factorial$ nasm -f elf64 factorial.asm -o factorial.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp10Factorial$ ld -o factorial factorial.o
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp10Factorial$ ./factorial
Enter the 8 bit number:
07h
The factorial of given 8 bit number is:
00000000000013B0

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp10Factorial$
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp10Factorial$ ./factorial
Enter the 8 bit number:
03h
The factorial of given 8 bit number is:
0000000000000006

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp10Factorial$
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp10Factorial$ ./factorial
Enter the 8 bit number:
02h
The factorial of given 8 bit number is:
0000000000000002

(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp10Factorial$
(base) ubuntu@ubuntu-TUF-Gaming-FA506IH-FA566IH:~/Desktop/MPL
Experiments/Exp10Factorial$ ./factorial
Enter the 8 bit number:
01h
The factorial for 0 or 1 is:
0001
```

CONCLUSION: In this practical session we learnt how to find the factorial of a given integer number on a command line by using recursion.

STUDY ASSIGNMENT

Motherboard -

A motherboard (sometimes manly known as the main board, system board) is the main printed circuit board (PCB) found in computer and other expanded systems. It heads many of the crucial electronic component of the system such as the Central Processing Unit (CPU) and memory and provides converter for other peripheral.

Main component of Motherboard are

- CPU Socket
- Memory Slots
- CMOS Battery
- ISA, PCI, and AGP slots
- Power Connector
- Chipset
- Graphical Devices • Back Panel

CPU Socket-

CPU socket or CPU slot is a mechanical component that provides mechanical and electrical connections between a microprocessor and a PCB. It allows CPU to be replaced without soldering. Common socket have retention chips that apply a constant force which must be overcome. Then a device is inserted.

Memory Slot-

A memory slot, memory socket or RAM slot is what allows computer memory (RAM) to be inserted into the computer or motherboard, there will usually be 2-4 memory slots.

Types of RAMs-

1. DDR-RAM
2. DDR2-RAM 3. DDR3-RAM
4. DDR4-RAM
5. RD-RAM
6. SD-RAM
7. 72Pin-SIMM

CMOS Battery-

Non volatile BIOS memory space to a small memory on PC motherboard that is used to store BIOS setting. It was traditionally called CMOS RAM because it use a volatile low power complementary metal oxide semiconductor.

ISA-

Industry Standard Architecture is a 8 bit 16 bit parallel bus system that allows up to 6 devices to be connected to PC.

AGP-

Accelerate Graphics Part is high speed point to point channel to attaching a video card to computers motherboard.

PCI-

Peripheral Component Inter-connected bus uses a local bus system. This system is independent of the processor bus speed.

Power Connectors-

i. 20+4 pin
ii. SATA
iii. Floppy Connectors
iv. PCIE Connectors

Chipset-

A chipset is a set of electronic component which is an integral circuit that manages the data flow between the processor, memory and peripherals. It is usually designed to work with a specific family of microprocessors.

Graphical Devices-

A video card (also called a video adapter) is an expansion card which generates a feed of output images to display, such as computer monitor.