# 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:**

**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-**

**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.

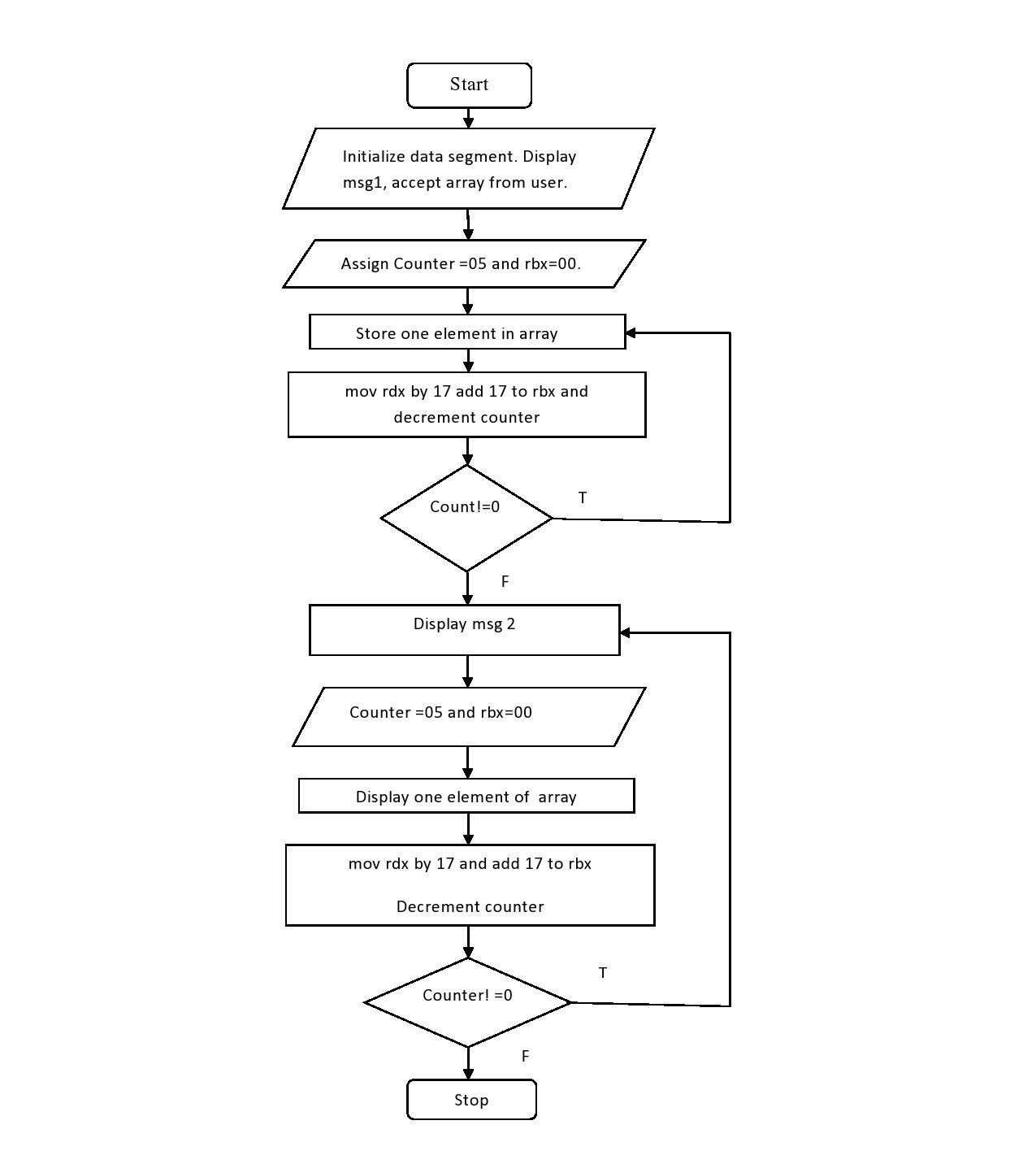
STEP 13: Add 17 to rbx.

STEP 14: Decrement Counter.

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

STEP 16: Stop

**FLOWCHART:**



**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

**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:**

**PROGRAM:**

**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.

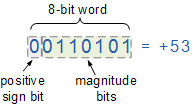
But how do we represent signed binary numbers if all we have is a bunch of one’s and zero’s. We know that binary digits, or bits only have two values, either a “1” or a “0” and conveniently for us, a sign also has only two values, being a “**+**” or a “**–**“.

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.

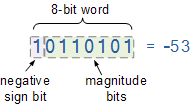
For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is “0”, this means the number is positive in value. If the sign bit is “1”, then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number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

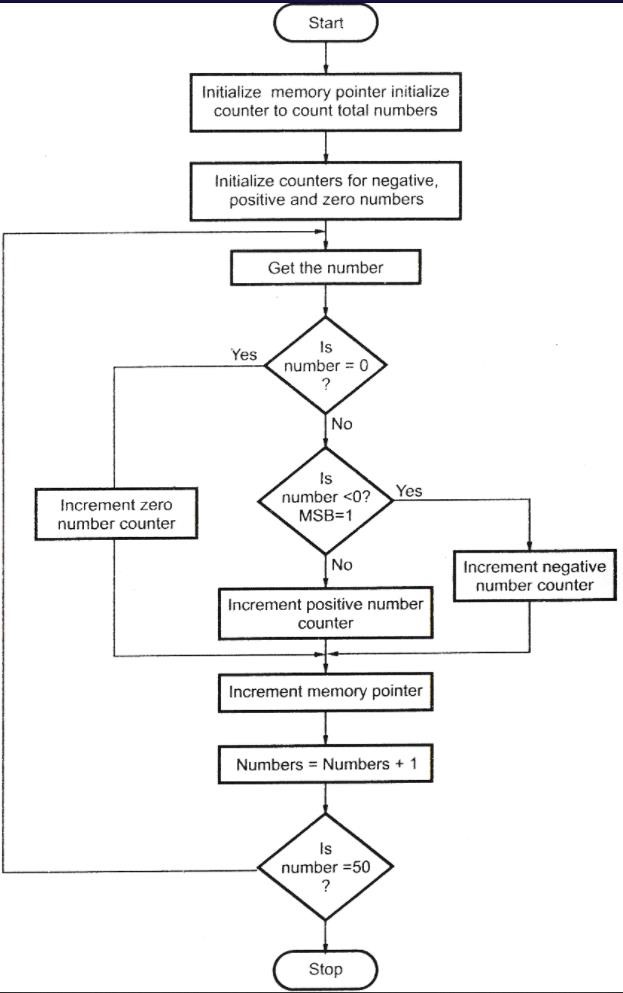
STEP 8: Point index register to the next element

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:**



**PROGRAM:**

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:**

**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:**

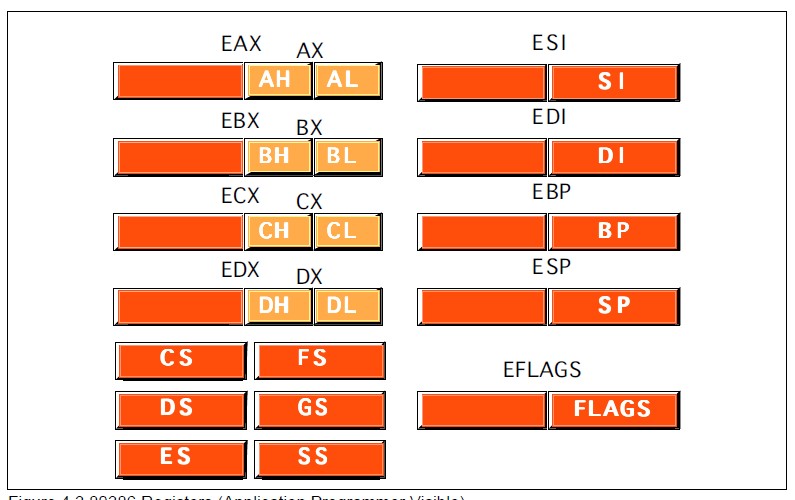
* Operating System: 64-bit Open source Linux or its derivative.
* Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
* Text Editor: geditor

**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:**

**OUTPUT:**

**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:**

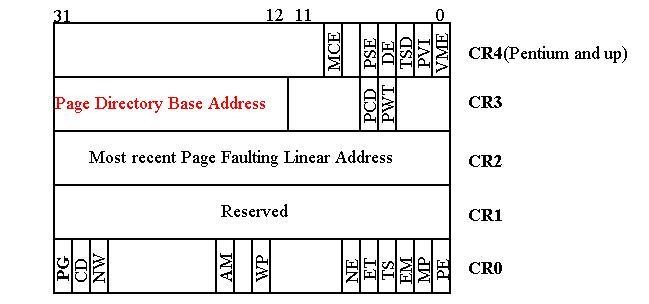
In computing, protected mode, also called protected virtual address mode is an operational mode of x86-compatible central processing units (CPUs). It allows system software to use features such as virtual memory, paging and safe multi-tasking designed to increase an operating system's control over application 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 :**



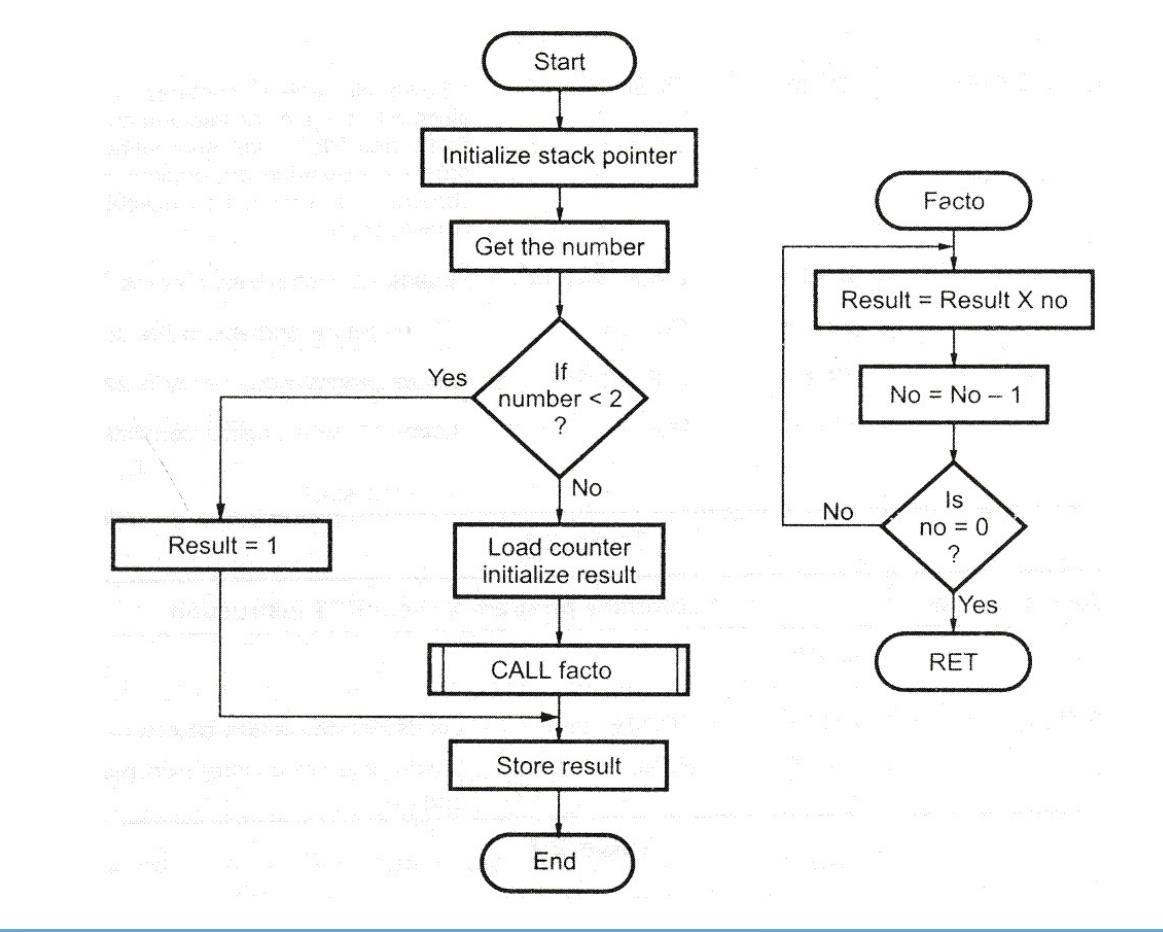
**Local Descriptor Table Register**

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:**



**PROGRAM:**

**OUTPUT:**

**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.

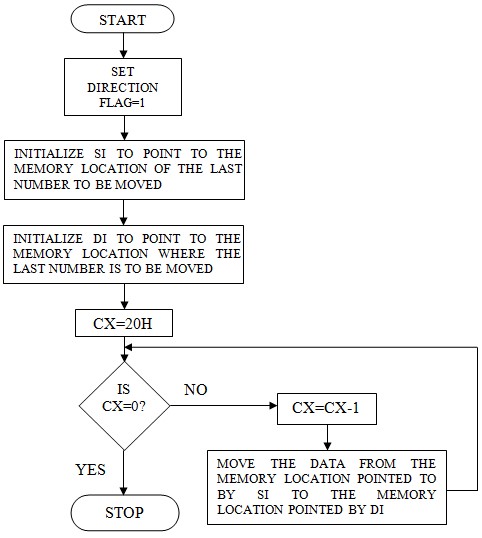
**Instructions needed:**

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 stringinstruction 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:

**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.

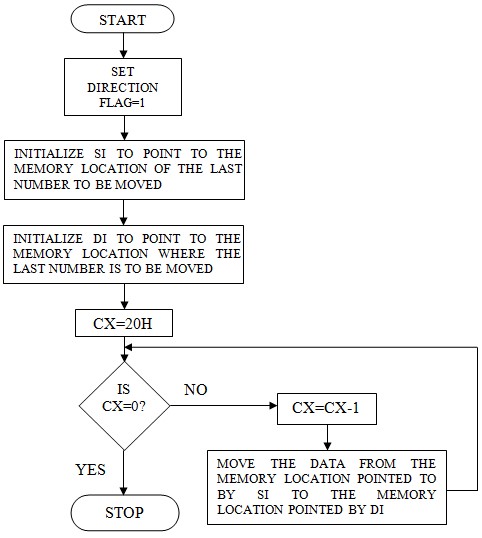
**Instructions needed:**

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 stringinstruction 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:**

**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 2n or 2n ±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 othernumber 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 leftalong with carry. If carry is present add the two numbers.

**Step 3:** Initialize the count to 4 as we are scanning for 4 digits. Decrementcounter 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.

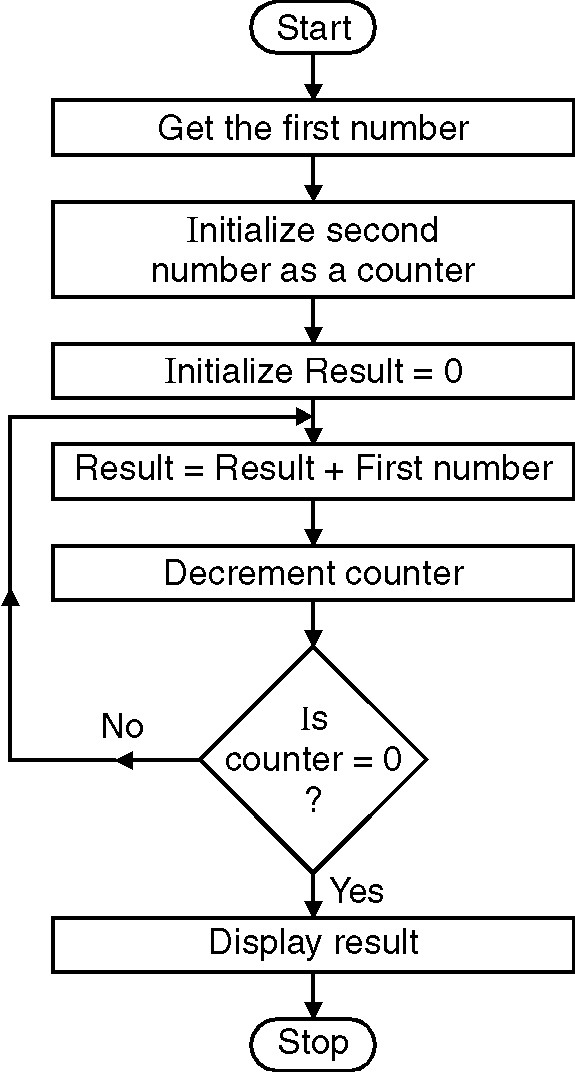
\ add AX, BX

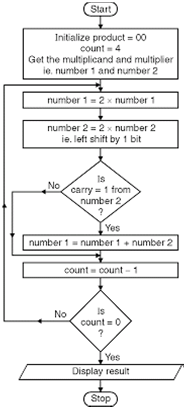
|  |  |
| --- | --- |
| 0110 | i.e.11 H |
| + 0000 | i.e.10 H |
| 0110 H | 0110H |

**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:**



****

**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:

**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

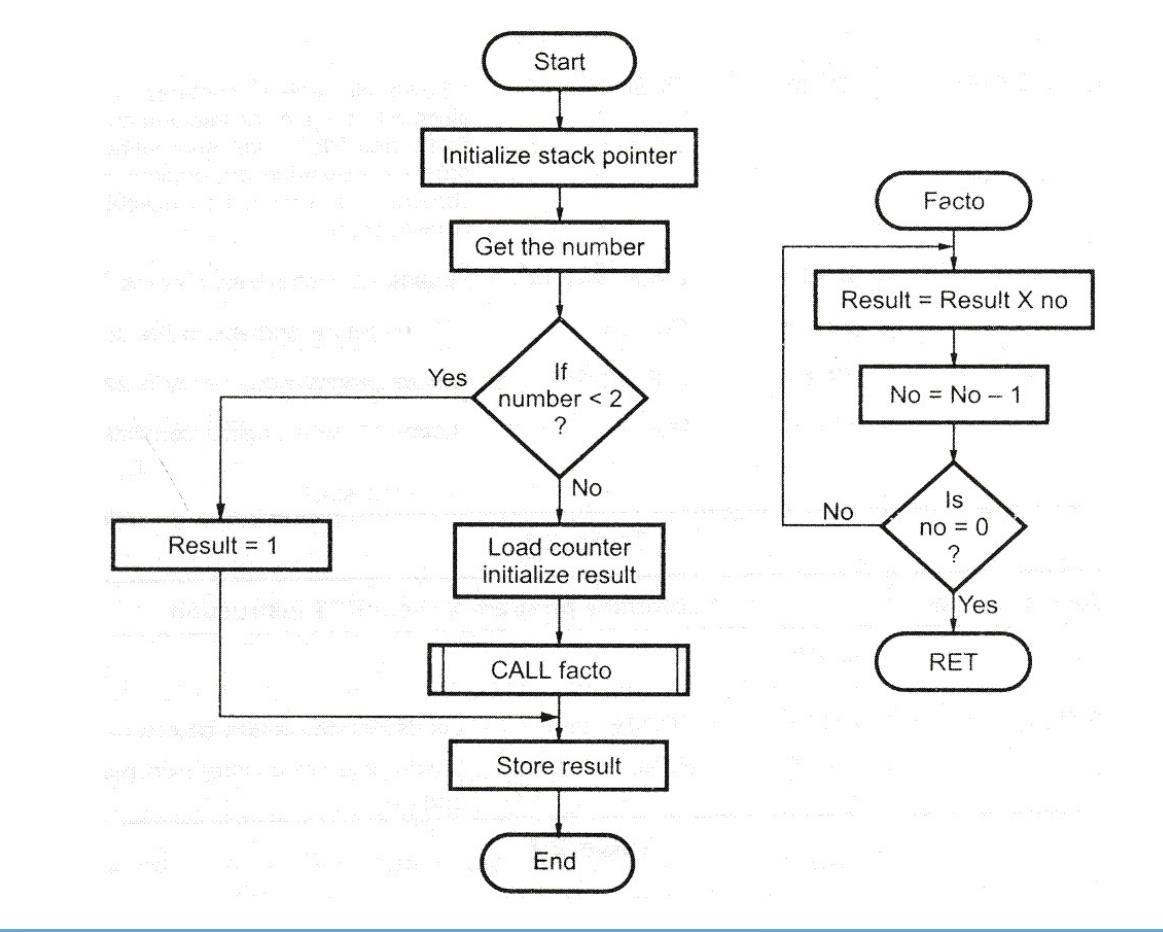
Step6: Repeat step until stack becomes empty

Step7: Convert number from Hex to ASCII

Step8: Print the number

Step9: Stop

**FLOWCHART:**



**PROGRAM:**

section .data msgFact: db 'Factorial is:',0xa msgFactSize: equ $-msgFact newLine: db 10 section .bss fact: resb 8

num: resb 2

section .txt global \_start \_start:

pop rbx ;Remove number of arguments

pop rbx ;Remove the program name

pop rbx ;Remove the actual number whoes factorial is to be calculated (Address of number)

mov [num],rbx

;print number accepted from command line

mov rax,1 mov rdi,1 mov rsi,[num] mov rdx,2

syscall

mov rsi,[num] mov rcx,02 xor rbx,rbx call aToH

mov rax,rbx

call factP

mov rcx,08 mov rdi,fact xor bx,bx mov ebx,eax

call hToA

mov rax,1 mov rdi,1 mov rsi,newLine mov rdx,1

syscall

mov rax,1 mov rdi,1 mov rsi,fact mov rdx,8

syscall

mov rax,1 mov rdi,1 mov rsi,newLine mov rdx,1 syscall

mov rax,60 mov rdi,0 syscall

factP: dec rbx cmp rbx,01 je comeOut cmp rbx,00 je comeOut mul rbx call factP comeOut:

ret aToH: up1: rol bx,04 mov al,[rsi] cmp al,39H jbe A2 sub al,07H A2: sub al,30H add bl,al inc rsi loop up1 ret

hToA:

d: rol ebx,4 mov ax,bx and ax,0fH

cmp ax,09H

jbe ii

add ax,07H

ii: add ax,30H mov [rdi],ax inc rdi

loop d

ret

OUTPUT:

**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
3. DDR4-RAM
4. RD-RAM
5. SD-RAM
6. 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-

1. 20+4 pin
2. SATA
3. Floppy Connectors
4. 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.