SVPM's

COLLEGE OF ENGINEERING, Malegaon(Bk)

DEPARTMENT OF COMPUTER ENGINEERING

210257: Microprocessor Laboratory

LABORATORY MANUAL

ACADEMIC YEAR 2023-24

NAME: PROF. SARALA ASHOK DABHADE

INDEX

Note:Suggested list for Assignment experiment Any 10.

Sr. No.	Date	Experiment Performed	Page No	Sign	Remark
1		Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers.			
2		Write an X86/64 ALP to accept a string and to display its length.			
3		Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.			
4		Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,*, /) using suitable macros. Define procedure for each operation.			

5		Write an X86/64 ALP to count number of positive and negative numbers from the array.			
6		Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5- digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for: (a) HEX to BCD b) BCD to HEX (c) EXIT. Display proper strings to prompt the user while accepting the input and displaying the result. (Wherever necessary, use 64-bit registers).			
7		Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction.			
8		Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.			
9		Write X86/64 ALP to perform overlapped block transfer with string specific instructions Block containing data can be defined in the data segment.			
10		Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (use of 64-bit registers is expected).			
11		Write X86 Assembly Language Program (ALP) to implement following OS commands i) COPY, ii) TYPE Using file operations. User is supposed to provide command line arguments			
12		Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program_1 execution and write FAR PROCEDURES in Program_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.			
13		Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.			

14		Write an X86/64 ALP password program that operates as follows: a. Do not display what is actually typed instead display asterisk (“*”). If the password is correct display, “access is granted” else display “Access not Granted”			
		Study Assignment: Motherboards are complex. Break them down, component by component, and Understand how they work. Choosing a motherboard is a hugely important part of building a PC. Study- Block diagram, Processor Socket, Expansion Slots, SATA, RAM, Form Factor, BIOS, Internal Connectors, External Ports, Peripherals and Data Transfer, Display, Audio, Networking, Overclocking, and Cooling. 4. https://www.intel.in/content/www/in/en/support/articles/000006014/boards-and-kits/desktop-boards.html			

EXPERIMENT NO. 01

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

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof.

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 01

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

Introduction to Assembly Language Programming:

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

Advantages of Assembly Language

- An understanding of assembly language provides knowledge of:
- Interface of programs with OS, processor and BIOS;
- Representation of data in memory and other external devices;
- How processor accesses and executes instruction;
- How instructions accesses and process data;
- How a program access external devices.

Other advantages of using assembly language are:
- It requires less memory and execution time;
- It allows hardware-specific complex jobs in an easier way;
- It is suitable for time-critical jobs;

ALP Step By Step:

Installing NASM:

If you select "Development Tools" while installed Linux, you may NASM installed along with the Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps:
- Open a Linux terminal.
- Type ***whereis nasm*** and press ENTER.
- If it is already installed then a line like, *nasm: /usr/bin/nasm* appears. Otherwise, you will see just*nasm:*, then you need to install NASM.

To install NASM take the following steps:

Open Terminal and run below commands:
sudo apt-get update
sudo apt-get install nasm

Assembly Basic Syntax:
An assembly program can be divided into three sections:
- The **data** section
- The **bss** section
- The **text** section

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

The .data Section
The .data section contains data definitions of initialized data items. Initialized data is data that has a value before the program begins running. These values are part of the executable file. They are loaded
into memory when the executable file is loaded into memory for execution. You don't have to load them with their values, and no machine cycles are used in their creation beyond what it takes to load the
program as a whole into memory. The important thing to remember about the .data section is that the more initialized data items you define, the larger the executable file will be, and the longer it will take to load it from disk into memory when you run it.

The .bss Section
Not all data items need to have values before the program begins running. When you're reading data
from a disk file, for example, you need to have a place for the data to go after it comes in from disk.

Data buffers like that are defined in the .bss section of your program. You set aside some number of bytes for a buffer and give the buffer a name, but you don't say what values are to be present in the buffer. There's a crucial difference between data items defined in the .data section and data items defined in the .bss section: data items in the .data section add to the size of your executable file. Data items in the .bss section do not.

The .text Section

The actual machine instructions that make up your program go into the .text section. Ordinarily, no data
items are defined in .text. The .text section contains symbols called *labels* that identify locations in the
program code for jumps and calls, but beyond your instruction mnemonics, that's about it.
All global labels must be declared in the .text section, or the labels cannot be "seen" outside your program by the Linux linker or the Linux loader. Let's look at the labels issue a little more closely.

Labels
A label is a sort of bookmark, describing a place in the program code and giving it a name that's easier
to remember than a naked memory address. Labels are used to indicate the places where jump instructions should jump to, and they give names to callable assembly language procedures.
Here are the most important things to know about labels:
- *Labels must begin with a letter, or else with an underscore, period, or question mark.* These last three have special meanings to the assembler, so don't use them until you know how NASM interprets them.
- *Labels must be followed by a colon when they are defined.* This is basically what tells NASM that the identifier being defined is a label. NASM will punt if no colon is there and will not flag an error, but the colon nails it, and prevents a mistyped instruction mnemonic from being mistaken for a label. Use the colon!
- *Labels are case sensitive.* So yikes:, Yikes:, and YIKES: are three completely different labels.

Assembly Language Statements

Assembly language programs consist of three types of statements:
- Executable instructions or instructions
- Assembler directives or pseudo-ops
- Macros

Syntax of Assembly Language Statements

[label] mnemonic [operands] [;comment]

LIST OF INTERRRUPTS USED: NA

LIST OF ASSEMBLER DIRECTIVES USED: EQU,DB

LIST OF MACROS USED: NA

LIST OF PROCEDURES USED: NA

ALGORITHM:

INPUT: ARRAY

OUTPUT: ARRAY

STEP 1: Start.

STEP 2: Initialize the data segment.

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

STEP 4: Initialize counter to 05 and rbx as 00

STEP 5: Store element in array.

STEP 6: Move rdx by 17.

STEP 7: Add 17 to rbx.

STEP 8: Decrement Counter.

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

STEP 9: Display msg2.

STEP 10: Initialize counter to 05 and rbx as 00

STEP 11: Display element of array.

STEP 12: Move rdx by 17.

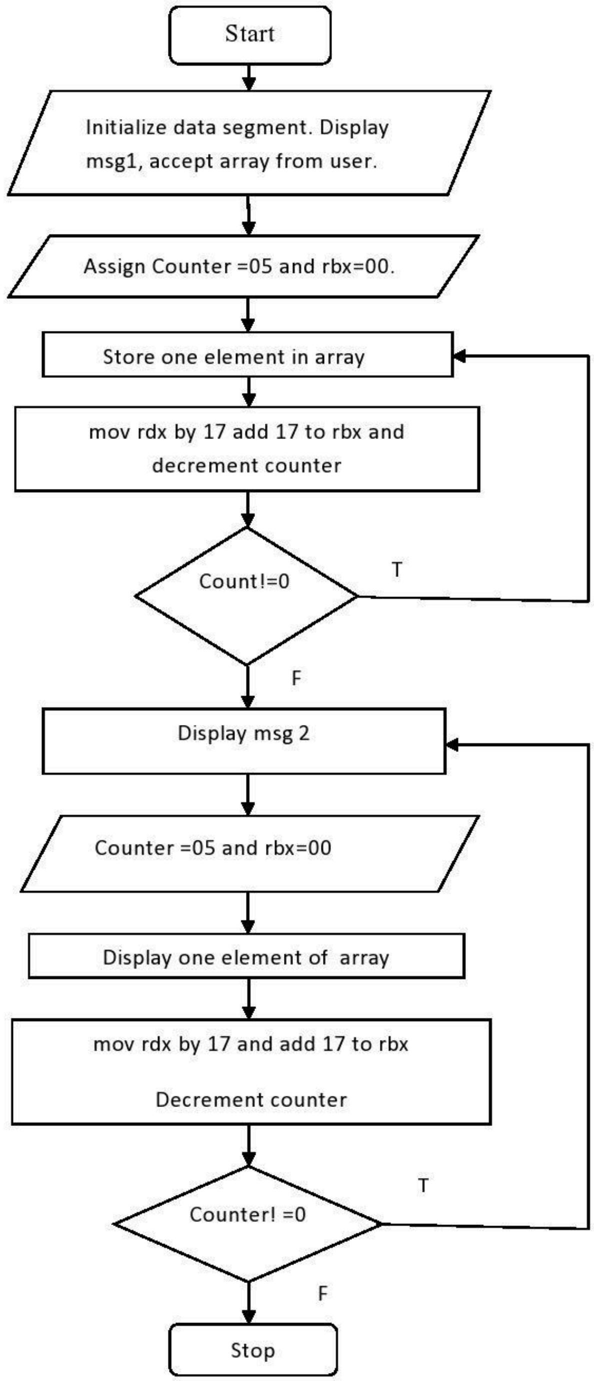
STEP 13: Add 17 to rbx.

STEP 14: Decrement Counter.

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

STEP 16: Stop

FLOWCHART:


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
;:$ cd ~/Desktop
;:~/Desktop$ nasm -f elf64 ass1.asm
;:~/Desktop$ ld -o ass1 ass1.o
;$ ./ass1

;Enter 5 64 bit numbers12
;23
;34
;45
;56

;Entered 5 64 bit numbers12
;23
;34
;45
;56
```

CONCLUSION:

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

.

EXPERIMENT NO. 02

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

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY:

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 02

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

MACRO:

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

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

The Syntax for macro definition −

%macro macro_name number_of_params
<macro body>
%endmacro

Where, *number_of_params* specifies the number parameters, *macro_name* specifies the name of the macro.

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

PROCEDURE:

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

Syntax

Following is the syntax to define a procedure −

```
proc_name:
  procedure body
  ...
  ret
```

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

CALL proc_name

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

LIST OF INTERRRUPTS USED: NA

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

LIST OF MACROS USED: DISPMSG

LIST OF PROCEDURES USED: DISPLAY

ALGORITHM:

INPUT: String

OUTPUT: Length of String in hex

STEP 1: Start.

STEP 2: Initialize data section.

STEP 3: Display msg1 on monitor

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

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

STEP 6. Take counter as 16 int cnt variable

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

STEP 8: Rotate left rbx register by 4 bit.

STEP 9: Move bl into al.

STEP 10: And al with 0fh

STEP 11: Compare al with 09h

STEP 12: If greater add 37h into al

STEP 13: else add 30h into al

STEP 14: Move al into memory location pointed by rdi

STEP 14: Increment rdi

STEP 15: Loop the statement till counter value becomes zero

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

STEP 17: Return from procedure

STEP 18: Stop

FLOWCHART:

PROGRAM:

```
section .data
        msg1 db 10,13,"Enter a string:"
        len1 equ $-msg1

section .bss
        str1 resb 200           ;string declaration
        result resb 16

section .text

global _start
        _start:

;display
        mov Rax,1
        mov Rdi,1
        mov Rsi,msg1
        mov Rdx,len1
        syscall

;store string

        mov rax,0
        mov rdi,0
        mov rsi,str1
        mov rdx,200
        syscall

call display

;exit system call
```

```
        mov Rax ,60
        mov Rdi,0
        syscall


%macro dispmsg 2
        mov Rax,1
        mov Rdi,1
        mov rsi,%1
        mov rdx,%2
        syscall
%endmacro



display:
        mov rbx,rax                 ; store no in rbx
        mov rdi,result              ;point rdi to result variable
        mov cx,16                   ;load count of rotation in cl

        up1:
                rol rbx,04          ;rotate no of left by four bits
                mov al,bl              ; move lower byte in al
                and al,0fh          ;get only LSB
                cmp al,09h            ;compare with 39h
                jg add_37            ;if greater than 39h skip add 37
                add al,30h
                jmp skip            ;else add 30
        add_37:
                add al,37h
        skip:
                mov [rdi],al          ;store ascii code in result variable
                inc rdi             ; point to next byte
                dec cx              ; decrement counter
                jnz up1             ; if not zero jump to repeat
                dispmsg result,16       ;call to macro
ret
```

OUTPUT:

CONCLUSION:

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

EXPERIMENT NO. 03

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

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY:

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 03

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

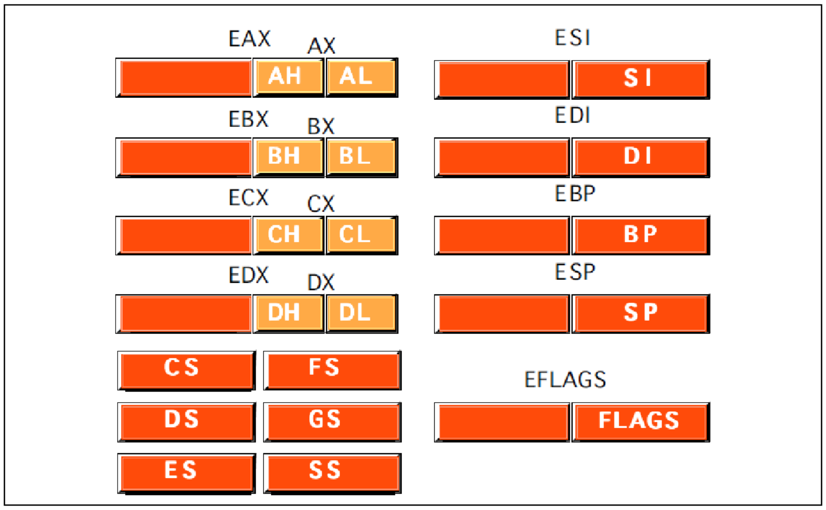
Datatype in 80386:

Datatypes of 80386:

The 80386 supports the following data types they are

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

Registers in 80386:



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

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

Size of operands in an Intel assembler instruction

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

Addressing modes in 80386:

The purpose of using addressing modes is as follows:

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

1. Register addressing mode:	MOV EAX, EDX
2. Immediate Addressing modes:	MOV ECX, 20305060H
3. Direct Addressing mode:	MOV AX, [1897 H]
4. Register Indirect Addressing mode	MOV EBX, [ECX]
5. Based Mode	MOV ESI, [EAX+23H]
6. Index Mode	SUB COUNT [EDI], EAX
7. Scaled Index Mode	MOV [ESI*8], ECX
8. Based Indexed Mode	MOV ESI, [ECX][EBX]
9. Based Index Mode with displacement	EA=EBX+EBP+1245678H
10. Based Scaled Index Mode with displacement	MOV [EBX*8] [ECX+5678H], ECX
11. String Addressing modes:	
12. Implied Addressing modes:	

LIST OF INTERRRUPTS USED:

LIST OF ASSEMBLER DIRECTIVES USED:

LIST OF MACROS USED:

LIST OF PROCEDURES USED:

ALGORITHM:

FLOWCHART:

```
section .data
     array db 11h, 55h, 33h, 22h,44h
     msg1 db 10,13,"Largest no in an array is:"
     len1 equ $-msg1

section .bss
     cnt resb 1
     result resb 16

          %macro dispmsg 2
          mov Rax,1
          mov Rdi,1
          mov rsi,%1
          mov rdx,%2
          syscall
     %endmacro

section .text
     global _start
     _start:
          mov byte[cnt],5
          mov rsi,array
```

```
          mov al,0
      LP: cmp al,[rsi]
          jg skip
          mov al ,[rsi]
          skip: inc rsi
          dec byte[cnt]
          jnz LP


      ;display al

      call display

      ;display message
          mov Rax,1
          mov Rdi,1
          mov Rsi,msg1
          mov Rdx,len1
          syscall


   dispmsg result,16        ;call to macro

      ;exit system call
          mov Rax ,60
          mov Rdi,0
          syscall



      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 skip1                 ;else add 30
          add_37:
               add al,37h
          skip1:
               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

     ret
```

```
;Output[sarala@localhost ~]$ su
Password:
[root@localhost sarala]# nasm -f elf64 ass3.asm
nasm: fatal: unable to open input file `ass3.asm'
[root@localhost sarala]# nasm -f elf64 ass3.asm
[root@localhost sarala]# ld -o ass3 ass3.o
[root@localhost sarala]# ./ass3

Largest no in an array is:0000000000000055[root@localhost sarala]#
```

EXPERIMENT NO. 04

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

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof. Dabhade S. A

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 04

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

LIST OF INTERRRUPTS USED: 80h

LIST OF ASSEMBLER DIRECTIVES USED: equ, db

LIST OF MACROS USED: scall

LIST OF PROCEDURES USED: add_proc, sub_proc, mul_proc, div_proc, disp64num

ALGORITHM:

FLOWCHART:

PROGRAM:

```
section .data
        menumsg db 10,'****** Menu ******',
        db 10,'1: Addition'
        db 10,'2: Subtraction'
        db 10,'3: Multiplication'
        db 10,'4: Division'
        db 10,10,'Enter your choice:: '
```

```
        menumsg_len: equ $-menumsg

        addmsg db 10,'Welcome to additon',10
        addmsg_len equ $-addmsg

        submsg db 10,'Welcome to subtraction',10
        submsg_len equ $-submsg

        mulmsg db 10,'Welcome to Multiplication',10
        mulmsg_len equ $-mulmsg

        divmsg db 10,'Welcome to Division',10
        divmsg_len equ $-divmsg

        wrchmsg db 10,10,'You Entered a Wrong Choice....!',10
        wrchmsg_len equ $-wrchmsg

        no1 dq 08h
        no2 dq 02h

        nummsg db 10
        result dq 0

        resmsg db 10,'Result is:'
        resmsg_len equ $-resmsg

        qmsg db 10,'Quotient::'
        qmsg_len equ $-qmsg

        rmsg db 10,'Remainder::'
        rmsg_len equ $-rmsg

        nwmsg db 10
        resh dq 0
        resl dq 0

section .bss
        choice resb 2
        dispbuff resb 16

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

section .text
global _start
        _start:
up:
```

```
            scall 1,1,menumsg,menumsg_len
            scall 0,0,choice,2

case1:cmp byte[choice],'1'
            jne case2
            call add_proc
            jmp up


case2:
            cmp byte[choice],'2'
            jne case3
            call sub_proc
            jmp up

case3:
            cmp byte[choice],'3'
            jne case4
            call mul_proc
            jmp up
case4:
            cmp byte[choice],'4'
            jne caseinv
            call div_proc
            jmp up
caseinv:
            scall 1,1, wrchmsg,wrchmsg_len

exit:
            mov eax,01
            mov ebx,0
            int 80h

add_proc:
     mov rax,[no1]
            adc rax,[no2]
            mov [result],rax
            scall 1,1,resmsg,resmsg_len
            mov rbx,[result]
            call disp64num
            scall 1,1,nummsg,1
            ret

sub_proc:

     mov rax,[no1]
            subb rax,[no2]
            mov [result],rax
            scall 1,1,resmsg,resmsg_len
            mov rbx,[result]
            call disp64num
            scall 1,1,nummsg,1
```

```
	ret

mul_proc:
	scall 1,1,mulmsg,mulmsg_len
	mov rax,[no1]
	mov rbx,[no2]
	mul rbx
	mov [resh],rdx
	mov [resl],rax
	scall 1,1, resmsg,resmsg_len
	mov rbx,[resh]
	call disp64num
	mov rbx,[resl]
	call disp64num
	scall 1,1,nwmsg,1
	ret
div_proc:
	scall 1,1,divmsg,divmsg_len
	mov rax,[no1]
	mov rdx,0
	mov rbx,[no2]
	div rbx
	mov [resh],rdx       ;Remainder
	mov [resl],rax       ;Quotient
	scall 1,1, rmsg,rmsg_len
	mov rbx,[resh]
	call disp64num
	scall 1,1, qmsg,qmsg_len
	mov rbx,[resl]
	call disp64num
	scall 1,1, nwmsg,1
	ret
disp64num:
	mov ecx,16
	mov edi,dispbuff
	dup1:
		rol rbx,4
		mov al,bl
		and al,0fh
		cmp al,09
		jbe dskip
		add al,07h
		dskip: add al,30h
		mov [edi],al
		inc edi
		loop dup1
	scall 1,1,dispbuff,16
	ret
```

CONCLUSION:

EXPERIMENT NO. 05

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

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof. Dabhade S. A.

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 05

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

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

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

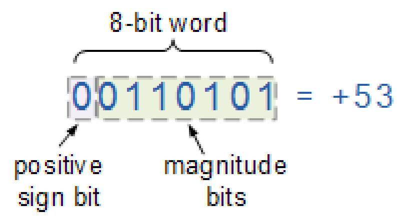
But how do we represent signed binary numbers if all we have is a bunch of one's and zero's. We know that binary digits, or bits only have two values, either a "1" or a "0" and conveniently for us, a sign also has only two values, being a "+" or a "–".

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

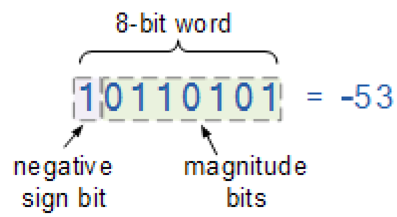
For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is "0", this means the number is positive in value. If the sign bit is "1", then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number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

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

Positive Signed Binary Numbers



Negative Signed Binary Numbers



LIST OF INTERRRUPTS USED: 80h

LIST OF ASSEMBLER DIRECTIVES USED: equ, db

LIST OF MACROS USED: print

LIST OF PROCEDURES USED: disp8num

ALGORITHM:

STEP 1: Initialize index register with the offset of array of signed numbers
STEP 2: Initialize ECX with array element count
STEP 3: Initialize positive number count and negative number count to zero
STEP 4: Perform MSB test of array element
STEP 5: If set jump to step 7
STEP 6: Else Increment positive number count and jump to step 8
STEP 7: Increment negative number count and continue
STEP 8: Point index register to the next element
STEP 9: Decrement the array element count from ECX, if not zero jump to step 4, else continue
STEP 10: Display Positive number message and then display positive number count
STEP 11: Display Negative number message and then display negative number count
STEP 12: EXIT

FLOWCHART:

PROGRAM:

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

section .data

welmsg db 10,'Welcome to count positive and negative numbers in an array',10
welmsg_len equ $-welmsg

pmsg db 10,'Count of +ve numbers::'
pmsg_len equ $-pmsg

nmsg db 10,'Count of -ve numbers::'
nmsg_len equ $-nmsg

nwline db 10

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

arrcnt equ 8

pcnt db 0
ncnt db 0

section .bss
        dispbuff resb 2

%macro print 2                  ;defining print function
        mov eax, 4              ; this 4 commands signifies the print sequence
        mov ebx, 1
        mov ecx, %1             ; first parameter
        mov edx, %2             ;second parameter
        int 80h                 ;interrupt command
%endmacro

section .text                   ;code segment
        global _start       ;must be declared for linker
        _start:         ;tells linker the entry point ;i.e start of code
        print welmsg,welmsg_len   ;print title
        mov esi,array
        mov ecx,arrcnt       ;store array count in extended counter reg


        up1:                    ;label
                bt word[esi],15
                ;bit test the array number (15th byte) pointed by esi.
                ;It sets the carray flag as the bit tested
                jnc pnxt     ;jump if no carry to label pskip

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

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

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

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

;ecx=0 loop automatically assums ecx has the counter

print pmsg,pmsg_len      ;prints pmsg
mov bl,[pcnt]    ;move the positive no count  to lower 8 bit of B reg
call disp8num           ;call disp8num subroutine
print nmsg,nmsg_len         ;prints nmsg
mov bl,[ncnt]    ;move the negative no count to lower 8 bits of b reg
call disp8num        ;call disp8num subroutine


print nwline,1        ;New line char

exit:
        mov eax,01
        mov ebx,0
        int 80h

disp8num:
        mov ecx,2        ;move 2 in ecx ;Number digits to display
        mov edi,dispbuff               ;Temp buffer

        dup1:      ;this command sequence which converts hex to bcd
        rol bl,4               ;Rotate number from bl to get MS digit to LS digit
        mov al,bl         ;Move  bl i.e. rotated number to AL
        and al,0fh          ;Mask upper digit (logical AND the contents ;of lower8 bits of accumulator with 0fh )

        cmp al,09          ;Compare al with 9

jbe dskip      ;If number below or equal to 9 go to add only 30h
        ;add al,07h ;Else first add 07h to accumulator

dskip:
add al,30h        ;Add 30h to accumulator
        mov [edi],al           ;Store ASCII code in temp buff (move contents       ;of accumulator to the location pointed by edi)
        inc edi
                        ;Increment destination index i.e. pointer to      ;next location in temp buff
loop dup1         ;repeat till ecx becomes zero

print dispbuff,2         ;display the value from temp buff
ret                     ;return to calling program
```

OUTPUT:

;[root@comppl2022 ~]# nasm -f elf64 Exp5.asm
;[root@comppl2022 ~]# ld -o Exp6 Exp5.o
;[root@comppl2022 ~]# ./Exp5
;Welcome to count +ve and -ve numbers in an array
;Count of +ve numbers::05
;Count of -ve numbers::03
;[root@comppl2022 ~]#

CONCLUSION:

EXPERIMENT NO. 06

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

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY:

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 06

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

OBJECTIVES:

- To understand assembly language programming instruction set.
- To understand different assembler directives with example.

- To apply instruction set for implementing X86/64 bit assembly language programs

ENVIRONMENT:

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

THEORY:

Hexadecimal Number System:

The “Hexadecimal” or simply “Hex” numbering system uses the **Base of 16** system and are a popular choice for representing long binary values because their format is quite compact and much easier to understand compared to the long binary strings of 1’s and 0’s.
Being a Base-16 system, the hexadecimal numbering system therefore uses 16 (sixteen) different digits with a combination of numbers from 0 to 9 and A to F.
Hexadecimal Numbers is a more complex system than using just binary or decimal and is mainly used when dealing with computers and memory address locations.

Binary Coded Decimal(BCD) Number System:

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

0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001
Numbers larger than 9, having two or more digits in the decimal system, are expressed digit by digit. For example, the BCD rendition of the base-10 number 1895 is
0001 1000 1001 0101
The binary equivalents of 1, 8, 9, and 5, always in a four-digit format, go from left to right.
The BCD representation of a number is not the same, in general, as its simple binary representation. In binary form, for example, the decimal quantity 1895 appears as
11101100111

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

HEX to BCD
Divide FFFF by 10 this FFFF is as decimal 65535 so
Division
65535 / 10 Quotient = 6553 Reminder = 5
6553 / 10 Quotient = 655 Reminder = 3
655 / 10 Quotient = 65 Reminder = 5
65 / 10 Quotient = 6 Reminder = 5
6 / 10 Quotient = 0 Reminder = 6
and we are pushing Reminder on stack and then printing it in reverse order.

BCD to HEX

1 LOOP : DL = 06 ; RAX = RAX * RBX = 0 ; RAX = RAX + RDX = 06
2 LOOP : DL = 05 ; 60 = 06 * 10 ; 65 = 60 + 5
3 LOOP : DL = 05 ; 650 = 60 * 10 ; 655 = 650 + 5
4 LOOP : DL = 03 ; 6550 = 655 * 10 ; 6553 = 6550 + 3
5 LOOP : DL = 06 ; 65530 = 6553 * 10 ; 65535 = 65530 + 5
Hence final result is in RAX = 65535 which is 1111 1111 1111 1111 and when we print this it is represented as FFFF.

LIST OF INTERRRUPTS USED:

LIST OF ASSEMBLER DIRECTIVES USED:

LIST OF MACROS USED:

LIST OF PROCEDURES USED:

ALGORITHM:

STEP 1: Start
STEP 2: Initialize data section.
STEP 3: Using Macro display the Menu for HEX to BCD, BCD to HEX and exit. Accept the choice
from user.
STEP 4: If choice = 1, call procedure for HEX to BCD conversion.
STEP 5: If choice = 2, call procedure for BCD to HEX conversion.
STEP 6: If choice = 3, terminate the program.

Algorithm for procedure for HEX to BCD conversion:

STEP 7: Accept 4-digit hex number from user.
STEP 8: Make count in RCX register 0.
STEP 9: Move accepted hex number in BX to AX.
STEP 10: Move base of Decimal number that is 10 in BX.
STEP 11: Move zero in DX.
STEP 12: Divide accepted hex number by 10. Remainder will return in DX.
STEP 13: Push remainder in DX on to stack.
STEP 14: Increment RCX counter.
STEP 15: Check whether AX contents are zero.
STEP 16: If it is not zero then go to step 5.
STEP 17: If AX contents are zero then pop remainders in stack in RDX.
STEP 18: Add 30 to get the BCD number.
STEP 19: Increment RDI for next digit and go to step 11.

Algorithm for procedure for BCD to HEX:

STEP 1: Accept 5-digit BCD number from user.
STEP 2: Take count RCX equal to 05.
STEP 3: Move 0A that is 10 in EBX.

STEP 4: Move zero in RDX register.
STEP 5: Multiply EBX with contents in EAX.
STEP 6: Move contents at RSI that is number accepted from user to DL.
STEP 7: Subtract 30 from DL.
STEP 8: Add contents of RDX to RAX and result will be in RAX.
STEP 9: Increment RSI for next digit and go to step 4 and repeat till RCX becomes zero.
STEP 10: Move result in EAX to EBX and call display procedure.

FLOWCHART:

PROGRAM

```
section .data
        msg1 db 10,10,'###### Menu for Code Conversion ######'
        db 10,'1: Hex to BCD'
        db 10,'2: BCD to Hex'
        db 10,'3: Exit'
        db 10,10,'Enter Choice:'
        msg1length  equ $-msg1

        msg2 db 10,10,'Enter 4 digit hex number::'
        msg2length equ $-msg2

        msg3 db 10,10,'BCD Equivalent:'
        msg3length  equ $-msg3

        msg4 db  10,10,'Enter 5 digit BCD number::'
        msg4length  equ $-msg4

        msg5 db 10,10,'Wrong Choice Entered....Please try again!!!',10,10
        msg5length  equ $-msg5

        msg6 db 10,10,'Hex Equivalent::'
        msg6length equ $-msg6
        cnt db  0

section .bss
        arr resb   06       ;common buffer for choice, hex and bcd input
        dispbuff resb   08
        ans resb   01


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

```
%macro accept 2
        mov rax,0
        mov rdi,0
        mov rsi,%1
        mov rdx,%2
        syscall
%endmacro


section .text
        global _start
_start:

menu:

        disp msg1,msg1length
        accept arr,2  ;      choice either 1,2,3 + enter

        cmp byte [arr],'1'
        jne l1
        call hex2bcd_proc

        jmp menu

l1:     cmp byte [arr],'2'
        jne l2
        call bcd2hex_proc
        jmp menu

l2:     cmp byte [arr],'3'
        je exit
        disp msg5,msg5length
        jmp menu

exit:
        mov rax,60
        mov rbx,0
        syscall


hex2bcd_proc:
        disp msg2,msg2length
        accept arr,5          ; 4 digits + enter
        call conversion
        mov rcx,0
        mov ax,bx
        mov bx,10             ;Base of Decimal No. system
l33:    mov dx,0
        div bx                ; Divide the no by 10
        push rdx              ; Push the remainder on stack
        inc rcx
inc byte[cnt]
```

```
        cmp ax,0
        jne l33
disp msg3,msg3length
l44:    pop rdx             ; pop the last pushed remainder from stack
        add dl,30h           ; convert it to ascii
        mov [ans],dl
disp ans,1
        dec byte[cnt]
jnz l44
        ret

bcd2hex_proc:
        disp msg4,msg4length
        accept arr,6      ; 5 digits + 1 for enter

        disp msg6,msg6length

        mov rsi,arr
        mov rcx,05
        mov rax,0
        mov ebx,0ah

l55:    mov rdx,0
        mul ebx       ; ebx * eax = edx:eax
        mov dl,[rsi]
        sub dl,30h
        add rax,rdx
        inc rsi
        dec rcx
jnz l55
        mov ebx,eax    ; store the result in ebx
        call disp32_num
        ret

conversion:
        mov bx,0
        mov ecx,04
        mov esi,arr
up1:
        rol bx,04
        mov al,[esi]
        cmp al,39h
        jbe l22
        sub al,07h
l22:    sub al,30h
        add bl,al
        inc esi
        loop up1
        ret
```

; the below procedure is to display 32 bit result in ebx why 32 bit & not 16 ;bit; because 5 digit bcd no ranges between 00000 to 99999 & for ;65535 ans ;is FFFF
; i.e if u enter the no between 00000-65535 u are getting the answer between
;0000-FFFF, but u enter i/p as 99999 urans is greater than 16 bit which is ;not; fitted in 16 bit register so 32 bit register is taken frresult

```
disp32_num:
        mov rdi,dispbuff
        mov rcx,08              ; since no is 32 bit,no of digits 8

l77:
        rol ebx,4
        mov dl,bl
        and dl,0fh
        add dl,30h
        cmp dl,39h
        jbe l66
        add dl,07h

l66:
        mov [rdi],dl
        inc rdi
        dec rcx
jnz  l77
        disp dispbuff+3,5  ;Dispays only lower 5 digits as upper three are '0'

        ret

;OUTPUT  OF PROGRAM

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


;###### Menu for Code Conversion ######
;1: Hex to BCD
;2: BCD to Hex
;3: Exit

;Enter Choice:1


;Enter 4 digit hex number::FFFF


;BCD Equivalent::65535

;###### Menu for Code Conversion ######
;1: Hex to BCD
```

```
;2: BCD to Hex
;3: Exit

;Enter Choice:1


;Enter 4 digit hex number::00FF


;BCD Equivalent::255

;###### Menu for Code Conversion ######
;1: Hex to BCD
;2: BCD to Hex
;3: Exit

;Enter Choice:1


;Enter 4 digit hex number::000F


;BCD Equivalent::15

;###### Menu for Code Conversion ######
;1: Hex to BCD
;2: BCD to Hex
;3: Exit

;Enter Choice:2


;Enter 5 digit BCD number::65535


;Hex Equivalent::0FFFF

;###### Menu for Code Conversion ######
;1: Hex to BCD
;2: BCD to Hex
;3: Exit

;Enter Choice:2


;Enter 5 digit BCD number::00255


;Hex Equivalent::000FF

;###### Menu for Code Conversion ######
;1: Hex to BCD
```

;2: BCD to Hex
;3: Exit
First Program:BCD To HEX

```
;*******************PROGRAM NO:5***********************
;                    ASSIGNMENT NO 5(A)
;AIM: A)WRITE 8086 ALP TO CONVERT FIVE DIGIT BCD NUMBER
; TO ITS EQUIVALENT HEX NUMBER
;*******************************************************

section .data                                 ;data section
     msg1 db 10,"Enter the five digit BCD number=>"
    msg1len equ $-msg1
    msg2 db 10,"The hexadecimal number is=>"
    msg2len equ $-msg2

section .bss
  numascii  resb 05                          ;initialise byte ascii number
  dispbuff resb  05                          ;initialise byte for display
buffer

%macro disp 2                                ;defination of macro message
  mov rax,1
 mov rdi,1
 mov rsi,%1
 mov rdx,%2
syscall
%endmacro

%macro accept 2                               ;defination of macro message
  mov rax,0
  mov rdi,0
  mov rsi,%1
  mov rdx,%2
syscall
%endmacro


section .text                                  ;code section
global _start
_start:


BTH:
      disp msg1,msg1len                        ;display the count message
      accept numascii,05                        ;accept input value
      call packnum                              ;call packnum
       mov edx,ebx
       mov ebx,00
      call bcd_hex
       call display
       disp msg2,msg2len                     ;display msg2
       disp dispbuff,5
```

```
exit:                                   ;termination of program
     mov rax,60
     mov rdi,0
     syscall

packnum:
      mov rbx,00                        ;display the numbers
       mov rcx,05                       ;clear the bl register
      mov rsi,numascii                 ;load input value in esi
up2:
      rol rbx,04                  ;rol four digit so that msb comes to
lsb
      mov al,[rsi]                    ;move the esi element in al
register
      sub al,30h                    ;if letter sub 37H else only sub 30H
      add bl,al
      inc rsi
      loop up2
ret

packnum1:

      mov bl,0
      mov rcx,04
      mov rsi,numascii

      up1:  rol ebx,04
            mov al,[rsi]
            cmp al,39h
            jbe skip1
            sub al,07h

      skip1:     sub al,30h
            add bl,al
            inc rsi
            loop up1
      ret

bcd_hex:
      cmp edx,10000h
      jb xx
      add ebx,10000
      sub edx,10000h
      jmp bcd_hex

xx:   cmp edx,1000h
      jb next1
      add ebx,1000
      sub edx,1000h
      jmp xx

next1:     cmp edx,100h
      jb next2
```

```
	add ebx,100
	sub edx,100h
	jmp next1

next2:	cmp edx,10h
	jb next3
	add ebx,10
	sub edx,10h
	jmp next2

next3:	add ebx,edx
ret

display:
	mov rsi,dispbuff
	mov rcx,05
	rol ebx,12

up3:	rol ebx,04			;rol four digit so that msb comes to
lsb
	mov al,bl
	and al,0fh			;get only lsb
	cmp al,09
	jbe skip
	add al,07h			;if letter add 37H else only add 30H
skip:	add al,30h
	mov [rsi],al
	inc esi				;increment count of edi
	dec rcx
	jnz up3
ret

;***************output***************

;[student@cocomp73 ~]$ nasm -f elf64 bcdtohex.asm
;[student@cocomp73 ~]$ ld -o bcdtohex bcdtohex.o
;[student@cocomp73 ~]$ ./bcdtohex

;enter the five digit bcd number=>65535

;The hexadecimal number is=>0FFFF[student@cocomp73 ~]$
;[student@cocomp73 ~]$
```

Second Program:HEX TO BCD

```
;******************PROGRAM NO:5**********************
;                  ASSIGNMENT NO 5(B)
;AIM: A)WRITE 8086 ALP TO CONVERT FIVE DIGIT HEX NUMBER
; TO ITS EQUIVALENT BCD NUMBER
;*******************************************************

section .data                                ;data section
     msg1 db 10,"Enter the five digit HEX number=>"
    msg1len equ $-msg1
    msg2 db 10,"The BCD number is=>"
    msg2len equ $-msg2

section .bss
  numascii  resb 05                         ;initialise byte ascii number
  dispbuff resb  05                         ;initialise byte for display
buffer

%macro display 2                               ;defination of macro
message
  mov rax,1
 mov rdi,1
 mov rsi,%1
 mov rdx,%2
syscall
%endmacro

%macro accept 2                              ;defination of macro message
  mov rax,0
  mov rdi,0
  mov rsi,%1
  mov rdx,%2
syscall
%endmacro


section .text                                 ;code section
global _start
_start:



      display msg1,msg1len                       ;display the count
message
      accept numascii,05                       ;accept input value
      call packnum                             ;call packnum

      call hex


 exit: mov rax,60
```

```
        mov rdi,0
        syscall

packnum:
        mov rbx,00                              ;display the numbers
        mov rcx,05                             ;clear the bl register
        mov rsi,numascii                       ;load input value in esi
up2:
      rol ebx,04                            ;rol four digit so that msb
comes to lsb
      mov al,[rsi]                          ;move the esi element in al
register
         cmp al,39h
         jbe skip1
         sub al,07
skip1:
      sub al,30h                              ;if letter sub 37H else only
sub 30H
      add bl,al
      inc rsi
      loop up2
ret




disp:
      mov rsi,dispbuff
      mov rcx,05
      rol ebx,12

up3:  rol ebx,04                    ;rol four digit so that msb comes to
lsb
      mov al,bl
      and al,0fh                    ;get only lsb
      cmp al,09
      jbe skip
      add al,07h                    ;if letter add 37H else only add 30H
skip: add al,30h
      mov [rsi],al
      inc esi                       ;increment count of edi
      loop up3

ret



hex:
        mov edx,ebx
        mov ebx,00h
        call hex_bcd
        call disp
```

```
        display msg2,msg2len
         display dispbuff,5

hex_bcd:
      cmp edx,10000
      jb step2
      add ebx,10000h
      sub edx,10000
      jmp hex_bcd
step2:     cmp edx,1000
      jb step3
      add ebx,1000h
      sub edx,1000
      jmp hex_bcd

step3:     cmp edx,100
      jb step4
      add ebx,100h
      sub edx,100
      jmp hex_bcd

step4:     cmp edx,10
      jb step5
      add ebx,10h
      sub edx,10
      jmp hex_bcd

step5:     add ebx,edx
ret

;[student@cocomp73 ~]$ nasm -f elf64 bcdtohex.asm
;[student@cocomp73 ~]$ ld -o bcdtohex bcdtohex.o
;[student@cocomp73 ~]$ ./bcdtohex

;enter the HEX number=>OFFFF

;The  BCD number is=>65535[student@cocomp73 ~]$
;[student@cocomp73 ~]$
```

CONCLUSION:

EXPERIMENT NO. 07

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

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof. Dabhade S. A.

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 07

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

OBJECTIVES:

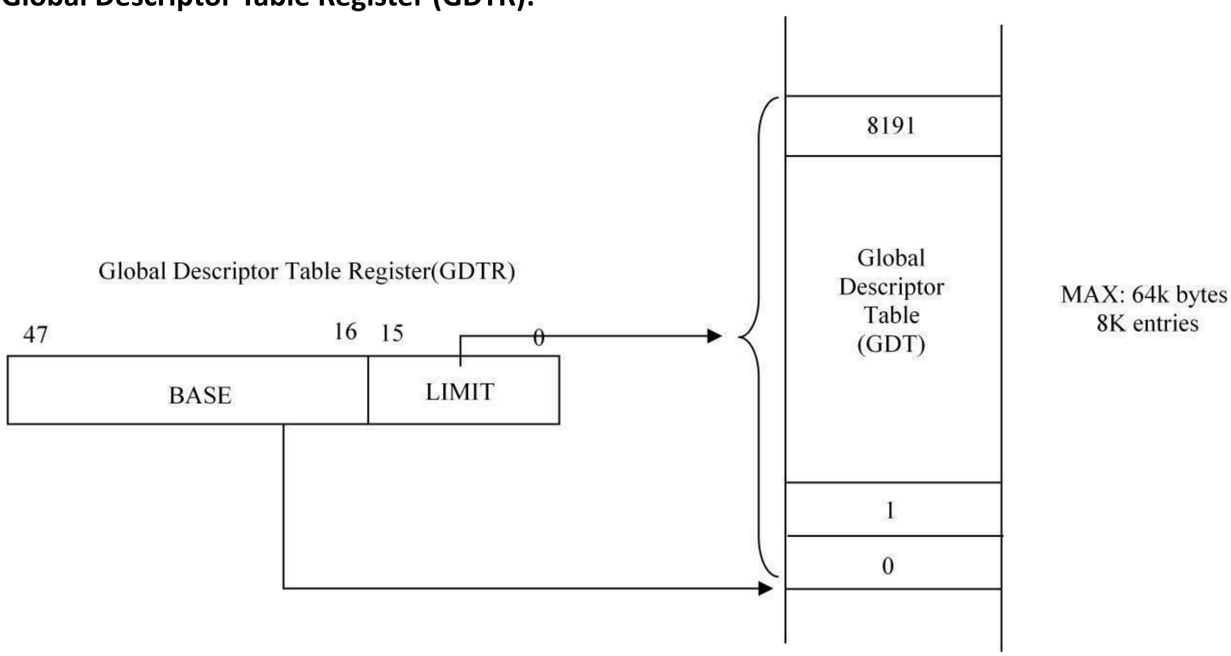
- To understand assembly language programming instruction set.
- To understand different assembler directives with example.
- To apply instruction set for implementing X86/64 bit assembly language programs

ENVIRONMENT:

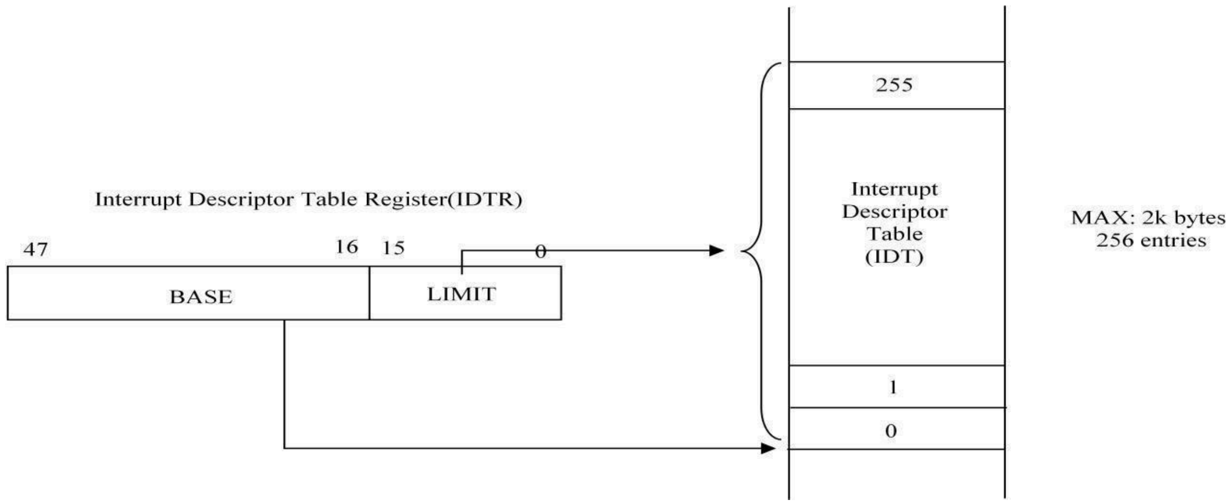
- Operating System: 64-bit Open source Linux or its derivative.
- Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
- Text Editor: geditor

THEORY: Theory:-

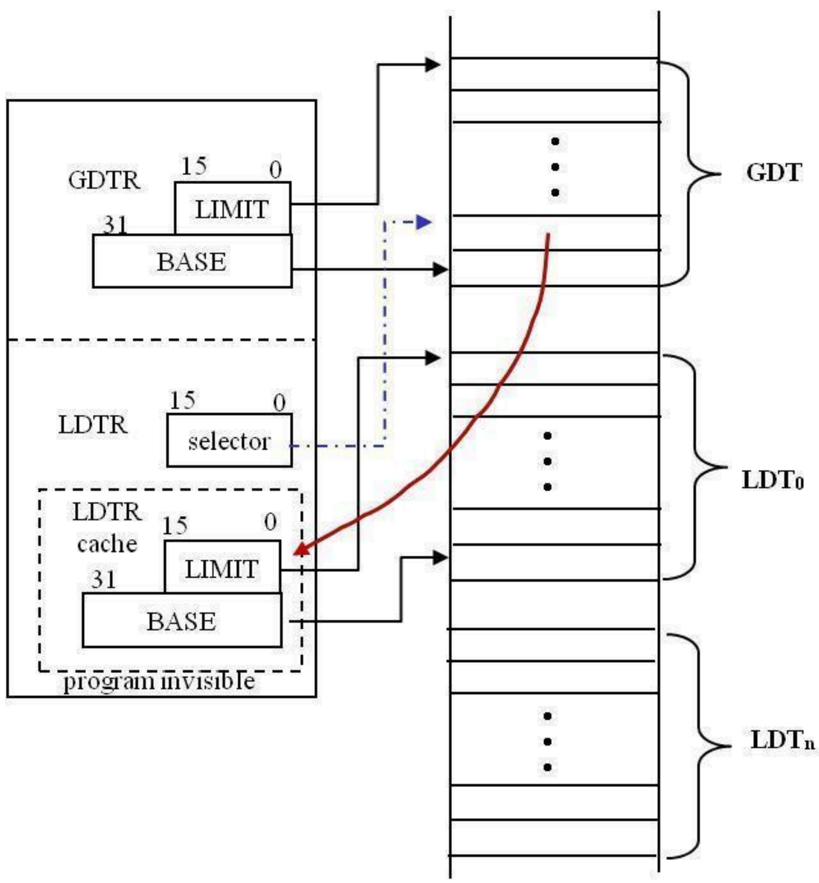
Global Descriptor Table Register (GDTR):



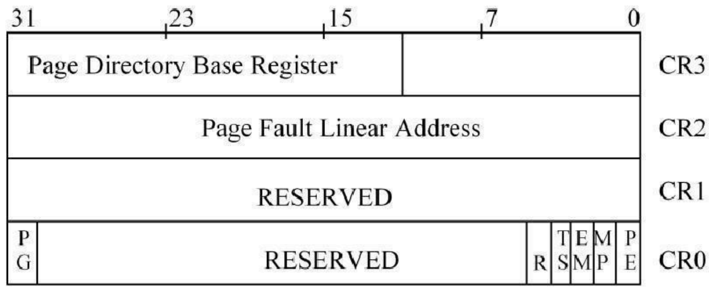
Interrupt Descriptor Table Register (IDTR):



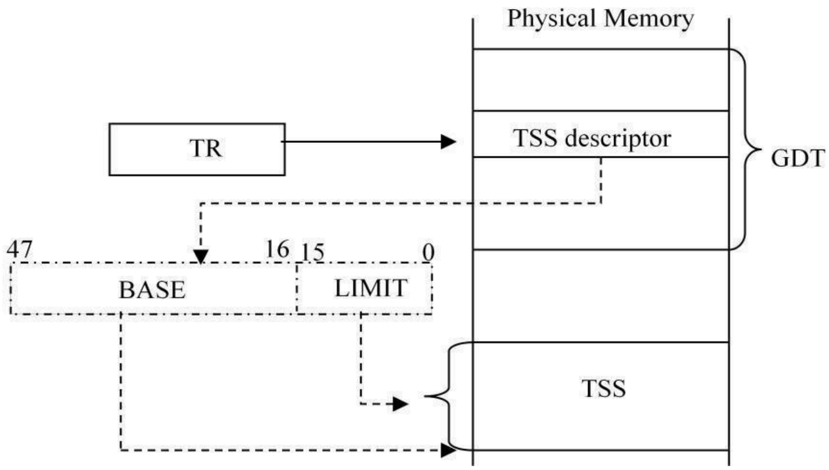
Local Descriptor Table Register (LDTR):



Control Registers:



- MSW : CR0
  - the lower 5 bits of CR0 are system-control flags
  - PE: protected-mode enable bit
    - At reset, PE is cleared.(real mode)
    - Set PE to 1 to enter protected mode
    - Once in protected mode, 386 cannot be switched back to real mode under SW control
  - MP: math present
  - EM: emulate
  - R: extension type
  - TS: task switched



Instruction Description Mode

LGDT S:- Load the global descriptor table register. S specifies both the memory location that contains the first byte of the 6 bytes to be loaded into the GDTR.

SGDT D:- Store the global descriptor table register. D specifies both the memory location that gets the first of the six bytes to be stored from the GDTR.

LIDT S: - Load the interrupt descriptor table register. S specifies both the memory location that contains the first byte of the 6 bytes to be loaded into the IDTR.

SIDT D:- Store the interrupt descriptor table register. D specifies both the memory location that gets the first of the six bytes to be stored from the IDTR.

Algorithm:

1) Start
2) Variable declaration in data section with initialization
3) Variable bss. section without initialization
4) Macro definition for display msg on screen
5) Read CRo
6) If PE beat =1
7) Store contains of GDT
8) Store contains of LDT
9) Store contains of IDT
10) Store contains of TR
11) Call display processor to display control of GDT
12) Call display processor to display contain of LDT
13) Call display processor to display contain of IDT
14) Call display processor to display control of TR
15) Call display processor to display control of MSW

16)Point to esi buffer 17)Load no. of digit to display

18) Rotate no. left by 4 bit
19) Move lower byte in DL
20) Mask upper digit of byte in DL
21) Add 30h to calculate ASCCI code
22) If DL < 39 , no add 7, yes Skip adding 07 more

23) Store ASCCI code in buffer

24) Point to next byte
25) 25)Display the no. from buffer
26) END

Program:
```
;This program first check the mode of processor(Real or
Protected),
;then reads GDTR, LDTR, IDTR, TR & MSW and displays the same.

section .data
rmodemsg db 10,"Processor is in Real Mode"
rmsg_len:equ $-rmodemsg

pmodemsg db 10,"Processor is in Protected Mode"
pmsg_len:equ $-pmodemsg

gdtmsg db 10,"GDT Contents are::"
gmsg_len:equ $-gdtmsg

ldtmsg db 10,"LDT Contents are::"
lmsg_len:equ $-ldtmsg

idtmsg db 10,"IDT Contents are::"
imsg_len:equ $-idtmsg

trmsg db 10,"Task Register Contents are::"
tmsg_len: equ $-trmsg

mswmsg db 10,"Machine Status Word::"
mmsg_len:equ $-mswmsg

colmsg db ":"

nwline db 10

section .bss
gdt resd 1
resw 1
ldt resw 1
idt resd 1
resw 1
tr  resw 1

cr0_data resd 1

dnum_buff resb 04

%macro disp 2
```

```
mov eax,04
mov ebx,01
mov ecx,%1
mov edx,%2
int 80h
%endmacro

section .text
global _start
_start:
smsw eax        ;Reading CR0

mov [cr0_data],eax

bt eax,0        ;Checking PE bit(LSB), if 1=Protected Mode, else Real
Mode
jc prmode
disp rmodemsg,rmsg_len
jmp nxt1

prmode:    disp pmodemsg,pmsg_len

nxt1:    sgdt [gdt]
sldt [ldt]
sidt [idt]
str [tr]

disp gdtmsg,gmsg_len

mov bx,[gdt+4]
call disp_num

mov bx,[gdt+2]
call disp_num

disp colmsg,1

mov bx,[gdt]
call disp_num

disp ldtmsg,lmsg_len
mov bx,[ldt]
call disp_num

disp idtmsg,imsg_len

mov bx,[idt+4]
call disp_num

mov bx,[idt+2]
call disp_num
```

```
disp colmsg,1

mov bx,[idt]
call disp_num

disp trmsg,tmsg_len

mov bx,[tr]
call disp_num

disp mswmsg,mmsg_len

mov bx,[cr0_data+2]
call disp_num

mov bx,[cr0_data]
call disp_num

disp nwline,1
exit:    mov eax,01
mov ebx,00
int 80h

disp_num:
mov esi,dnum_buff    ;point esi to buffer

mov ecx,04        ;load number of digits to display

up1:
rol bx,4        ;rotate number left by four bits
mov dl,bl        ;move lower byte in dl
and dl,0fh        ;mask upper digit of byte in dl
add dl,30h        ;add 30h to calculate ASCII code
cmp dl,39h        ;compare with 39h
jbe skip1        ;if less than 39h skip adding 07 more
add dl,07h        ;else add 07
skip1:
mov [esi],dl        ;store ASCII code in buffer
inc esi            ;point to next byte
loop up1        ;decrement the count of digits to display
;if not zero jump to repeat

disp dnum_buff,4    ;display the number from buffer
ret
```

Output:

```
[root@localhost sarala]# nasm -f elf64 Ass7.asm
[root@localhost sarala]# ld -o Ass7 Ass7.o
[root@localhost sarala]# ./Ass7
```

```
Processor is in Protected Mode
GDT Contents are::37C0A000:007F
LDT Contents are::0000
IDT Contents are::FF577000:0FFF
Task Register Contents are::0040
Machine Status Word::8005FFFF
[root@localhost sarala]#
```

EXPERIMENT NO. 08

NAME: Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof. Dabhade S. A.

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 08

AIM: Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.

OBJECTIVES:

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

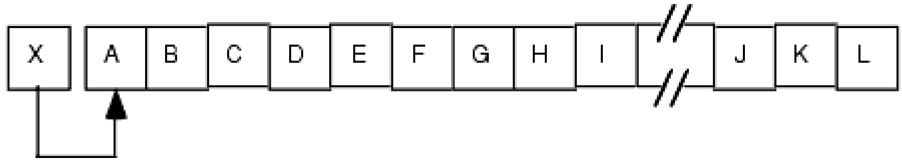
ENVIRONMENT:

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

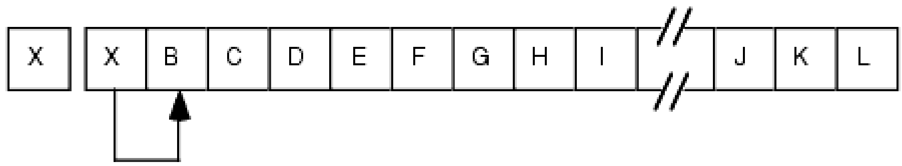
THEORY: All members of the 80x86family support five different string instructions: movs, cmps, scas, lods, and stos. They are the string primitives since you can build most other string operations from these five instructions. How you use these five instructions is the topic of the next several sections This sequence of instructions treats CharArray1 and CharArray2 as a pair of 384 byte strings. However, the last 383 bytes in the CharArray1 array overlap the first 383 bytes in theCharArray2 array. Let's trace the operation of this code byte by byte. When the CPU executes the MOVSB instruction, it copies the byte at ESI (CharArray1) to the byte pointed at by EDI (CharArray2). Then it increments ESI and EDI, decrements ECX by one, and repeats this process. Now the ESI register points at CharArray1+1 (which is the address of CharArray2) and the EDI register points at CharArray2+1. The MOVSB instruction copies the byte pointed at by ESI to the byte pointed at by EDI. However, this is the byte originally copied from location CharArray1. So the MOVSB instruction copies the value originally in location CharArray1 to both locations CharArray2 and CharArray2+1. Again, the CPU increments ESI and EDI, decrements ECX, and repeats this operation. Now the movsb instruction copies the byte from location CharArray1+2 (CharArray2+1) to location CharArray2+2. But once again, this is the value that originally appeared in location CharArray1. Each repetition of the loop copies the next element in CharArray1 [0] to the next available location in the C charArray2 array. Pictorially, it looks something like that shown in figure.

The end result is that the MOVSB instruction replicates X throughout the string. The MOVSB instruction copies the source operand into the memory location which will

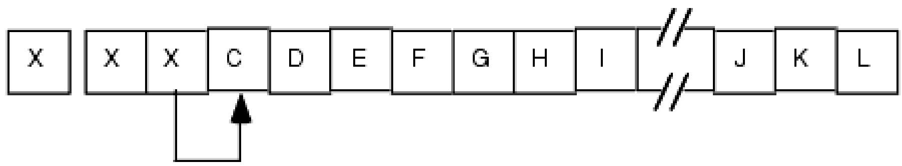
1st move operation:



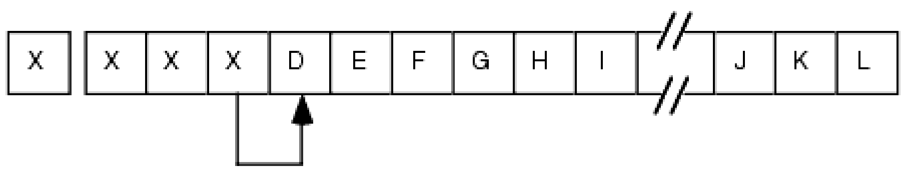
2nd move operation:



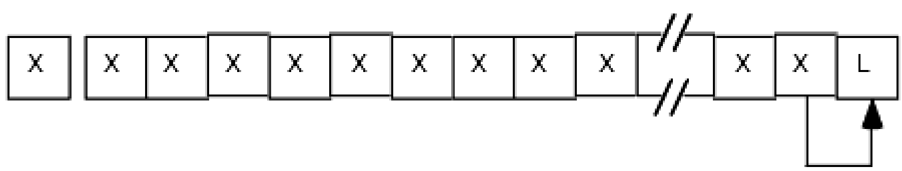
3rd move operation:



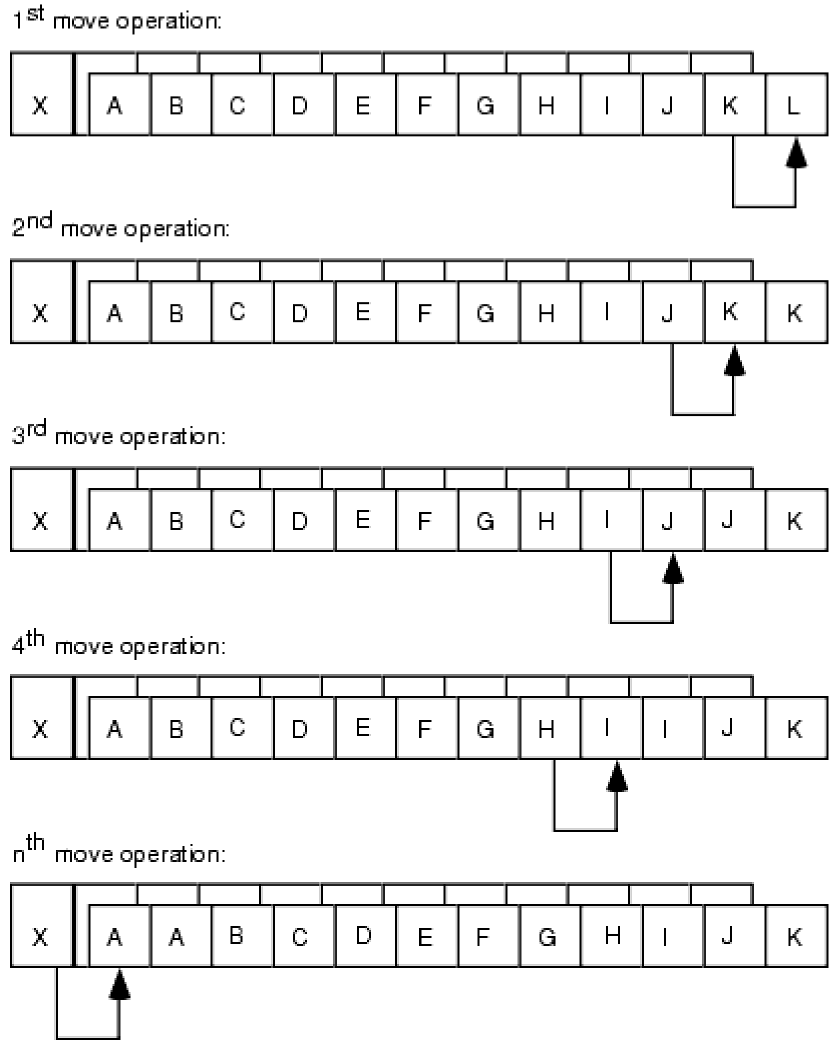
4th move operation:



nth move operation:



Become the source operand for the very next move operation, which causes the replication. If you really want to move one array into another when they overlap, you should move each element of the source string to the destination string.



Setting the direction flag and pointing ESI and EDI at the end of the strings will allow you to (correctly) move one string to another when the two strings overlap and the source string begins at a lower address than the destination string. If the two strings overlap and the source string begins at a higher address than the destination string, then clear the direction flag and point ESI and EDI at the beginning of the two strings.

If the two strings do not overlap, then you can use either technique to move the strings around in memory. Generally, operating with the direction flag clear is the easiest, so that makes the most sense in this case.

You shouldn't use the MOVSx instruction to fill an array with a single byte, word, or double word value. Another string instruction, STOS, is much better for this purpose. However, for arrays whose elements are larger than four bytes, you can use the MOVS instruction to initialize the entire array to the content of the first element.

The MOVS instruction is generally more efficient when copying double words than it is copying bytes or words. In fact, it typically takes the same amount of time to copy a byte using MOVSB as it does to copy a double word using MOVSD[3]. Therefore, if you are moving a large number of bytes from one array to another, the copy operation will be faster if you can use the MOVSD instruction rather than the MOVSB instruction. Of course, if the number of bytes you wish to move is an even multiple of four, this is a trivial change; just divide the number of bytes to copy by four, load this value into ECX, and then use the MOVSB instruction. If the number of bytes is not evenly divisible by four, then you can use the MOVSD instruction to copy all but the last one, two, or three bytes of the array (that is, the remainder after you divide the byte count by four). For example, if you want to efficiently move 4099 bytes, you can do so with the following instruction sequence:

Algorithm For Overlapping Blocks Transfer:-

(TYPE A : Latter half of source overlapped)

1. Physical initialization of data segment.
2. Initialization of source memory pointer to last element in source array.
3. Initialization of destination memory pointer to last element in destination array.
4. Initialize counter to no. of elements in source array.
5. Copy element in a source array pointed by source memory pointer to a location in a destination array pointed by destination memory pointer.
6. Decrement destination memory pointer, decrement source memory pointer and decrement counter by 1.
7. If (counter ≠0), goto step 5.
8. Terminate program and exit to DOS.

Algorithm For Overlapping Block Transfer:-

(TYPE B: Prior half of source overlapped)

1. Physical initialization of data segment.
2. Initialization of memory pointer to first element of source.
3. Initialization of memory pointer to first element of destination.
4. Initialization of counter to no. of elements in source array.
5. Copy element in a source array pointed by source memory pointer to a location in a destination array pointed by destination memory pointer.
6. Increment destination memory pointer, Increment source memory pointer and decrement counter.
7. If (counter ≠0), goto step 5.
8. Terminate program and exit to DOS.

```
;*-*-Non-overlap Block transfer-*-*
section .data
	menumsg db 10,10,'***Nonoverlap block transfer***',10
		db 10,'1.Block transfer without string '
		db 10,'2.Block transfer with string '
		db 10,'3.exit   '
	menumsg_len equ $-menumsg
	wrmsg db 10,10,'Wrong choice entered',10,10
	wrmsg_len equ $-wrmsg
	bfrmsg db 10,'**Block contents before transfer: '
	bfrmsg_len equ $-bfrmsg
	afrmsg db 10,'**Block contents after transfer:'
	afrmsg_len equ $-afrmsg
	srcmsg db 10,'*_*Source block contents   '
	srcmsg_len equ $-srcmsg
	dstmsg db 10,'*_*Destination block contents   '
	dstmsg_len equ $-dstmsg
	srcblk db 01h,02h,03h,04h,05h
	dstblk times 5 db 0			;destination block is
defined 5 times
	cnt equ 05
	spacechar db 20h
	lfmsg db 10,10

section .bss
	optionbuff resb 02
	dispbuff resb 02

%macro dispmsg 2
```

```
     mov eax,04
     mov ebx,01
     mov ecx,%1
     mov edx,%2
     int 80h
%endmacro

%macro accept 2
     mov eax,03
     mov ebx,00
     mov ecx,%1
     mov edx,%2
     int 80h
%endmacro

section .text
global _start
_start:
     dispmsg bfrmsg,bfrmsg_len
     call show
     menu:
          dispmsg menumsg,menumsg_len
          accept optionbuff,02
          cmp byte [optionbuff],'1'
          jne case2
          call wos             ;wos=With Out String
          jmp exit1

     case2:
          cmp byte [optionbuff],'2'
          jne case3
          call ws              ;ws=with string
          jmp exit1

     case3:
          cmp byte [optionbuff],'3'
          je exit
          dispmsg wrmsg,wrmsg_len
          jmp menu

     exit1:
          dispmsg afrmsg,afrmsg_len
          call show
          dispmsg lfmsg,2
     exit:
          mov eax,01
          mov ebx,00
          int 80h

     dispblk:
          mov rcx,cnt
```

```
rdisp:
     push rcx
     mov bl,[esi]
     call disp8
     inc esi
     dispmsg spacechar,1
     pop rcx
     loop rdisp
ret

wos:
     mov esi,srcblk
     mov edi,dstblk
     mov ecx,cnt
     x:
          mov al,[esi]
          mov [edi],al
          inc esi
          inc edi
          loop x
          ret

ws:
     mov esi,srcblk
     mov edi,dstblk
     mov ecx,cnt
     cld                  ;clear direction flag
     rep movsb

show:
     dispmsg srcmsg,srcmsg_len
     mov esi,srcblk
     call dispblk
     dispmsg dstmsg,dstmsg_len
     mov esi,dstblk
     call dispblk
     ret

disp8:
     mov ecx,02
     mov edi,dispbuff
     dub1:
          rol bl,4
          mov al,bl
          and al,0fh
          cmp al,09h
          jbe x1
          add al,07
     x1:
          add al,30h
          mov [edi],al
          inc edi
```

```
                loop dub1
                dispmsg dispbuff,3
           ret


;****OUTPUT****

[sarala@localhost ~]$ su
Password:
[root@localhost sarala]# nasm -f elf64 ass8.asm
[root@localhost sarala]# ld -o ass8 ass8.o
[root@localhost sarala]# ./ass8

**Block contents before transfer:
*_*Source block contents    01 02 03 04 05
*_*Destination block contents    00 00 00 00 00

***Nonoverlap block transfer***

1.Block transfer without string
2.Block transfer with string
3.exit   1

**Block contents after transfer:
*_*Source block contents    01 02 03 04 05
*_*Destination block contents    01 02 03 04 05

[root@localhost sarala]#
```

Conclusion:

Questions:

1) What are the interrupts used in above program. Explain in detail.
2) What is the assembler directives used in above program?
3) Explain the flag register of 8086 and 80386 microprocessor?

Assignment

LIST OF INTERRRUPTS USED:

LIST OF ASSEMBLER DIRECTIVES USED:

LIST OF MACROS USED:

LIST OF PROCEDURES USED:

ALGORITHM:

FLOWCHART:

EXPERIMENT NO. 09

NAME: Write X86/64 ALP to perform overlapped block transfer with string specific instructions Block containing data can be defined in the data segment.

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof. Dabhade S. A

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 09

AIM: Write X86/64 ALP to perform overlapped block transfer with string specific instructions Block containing data can be defined in the data segment.

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

LIST OF INTERRRUPTS USED:

LIST OF ASSEMBLER DIRECTIVES USED:

LIST OF MACROS USED:

LIST OF PROCEDURES USED:

ALGORITHM:

FLOWCHART:

Program:

```
;TITLE: Overlap Block transfer

section .data
     menumsg db 10,10,'***Overlap block transfer***',10
          db 10,'1.Block transfer without string '
          db 10,'2.Block transfer with string '
          db 10,'3.exit   '
     menumsg_len equ $-menumsg
     wrmsg db 10,10,'Wrong choice entered',10,10
     wrmsg_len equ $-wrmsg
```

```
	bfrmsg db 10,'**Block contents before transfer: '
	bfrmsg_len equ $-bfrmsg
	afrmsg db 10,'**Block contents after transfer:'
	afrmsg_len equ $-afrmsg
	srcmsg db 10,'*_*Source block contents   '
	srcmsg_len equ $-srcmsg
	dstmsg db 10,'*_*Destination block contents   '
	dstmsg_len equ $-dstmsg
	srcblk db 01h,02h,03h,04h,05h
	dstblk times 3 db 0
	cnt equ 05
	spacechar db 20h
	lfmsg db 10,10

section .bss
	optionbuff resb 02
	dispbuff resb 02

%macro dispmsg 2
	mov eax,04
	mov ebx,01
	mov ecx,%1
	mov edx,%2
	int 80h
%endmacro

%macro accept 2
	mov eax,03
	mov ebx,00
	mov ecx,%1
	mov edx,%2
	int 80h
%endmacro

section .text
global _start
_start:
	dispmsg bfrmsg,bfrmsg_len
	call show
	menu:
		dispmsg menumsg,menumsg_len
		accept optionbuff,02
		cmp byte [optionbuff],'1'
		jne case2
		call wos              ;wos=With Out String
		jmp exit1

	case2:
		cmp byte [optionbuff],'2'
		jne case3
		call ws               ;ws=with string
		jmp exit1
```

```
case3:
     cmp byte [optionbuff],'3'
     je exit
     dispmsg wrmsg,wrmsg_len
     jmp menu

exit1:
     dispmsg afrmsg,afrmsg_len
     call show
     dispmsg lfmsg,2
exit:
     mov eax,01
     mov ebx,00
     int 80h

dispblk:
     mov rcx,cnt

rdisp:
     push rcx
     mov bl,[esi]
     call disp8
     inc esi
     dispmsg spacechar,1
     pop rcx
     loop rdisp
ret

wos:
     mov esi,srcblk + 04h
     mov edi,dstblk + 02h
     mov ecx,cnt
     x:
          mov al,[esi]
          mov [edi],al
          dec esi
          dec edi
          loop x
          ret

ws:
     mov esi,srcblk + 04h
     mov edi,dstblk + 02h
     mov ecx,cnt
     std             ;set direction flag,si and di inc
     rep movsb

show:
     dispmsg srcmsg,srcmsg_len
     mov esi,srcblk
     call dispblk
```

```
          dispmsg dstmsg,dstmsg_len
          mov esi,dstblk-02h
          call dispblk
          ret

     disp8:
          mov ecx,02
          mov edi,dispbuff
          dub1:
               rol bl,4
               mov al,bl
               and al,0fh
               cmp al,09h
               jbe x1
               add al,07
          x1:
               add al,30h
               mov [edi],al
               inc edi
               loop dub1
               dispmsg dispbuff,3
          ret

;*****OUTPUT*****
;[root@comppl208 ~]# cd nasm-2.10.07
;[root@comppl208 nasm-2.10.07]# gedit overlap26.asm
;[root@comppl208 nasm-2.10.07]# nasm -f elf64 overlap26.asm
;[root@comppl208 nasm-2.10.07]# ld -o overlap26 overlap26.o
;[root@comppl208 nasm-2.10.07]# ./overlap26

;**Block contents before transfer:
;*_*Source block contents   01 02 03 04 05
;*_*Destination block contents   04 05 00 00 00

;***Overlap block transfer***

;1.Block transfer without string
;2.Block transfer with string
;3.exit   1

;**Block contents after transfer:
;*_*Source block contents   01 02 03 01 02
;*_*Destination block contents   01 02 03 04 05

;[root@comppl208 nasm-2.10.07]# ./overlap26

;**Block contents before transfer:
;*_*Source block contents   01 02 03 04 05
;*_*Destination block contents   04 05 00 00 00

;***Overlap block transfer***
```

```
;1.Block transfer without string
;2.Block transfer with string
;3.exit   2

;*_*Source block contents   01 02 03 01 02
;*_*Destination block contents   01 02 03 04 05
;**Block contents after transfer:
;*_*Source block contents   01 02 03 01 02
;*_*Destination block contents   01 02 03 04 05

;[root@comppl208 nasm-2.10.07]# ./overlap26

;**Block contents before transfer:
;*_*Source block contents   01 02 03 04 05
;*_*Destination block contents   04 05 00 00 00

;***Overlap block transfer***

;1.Block transfer without string
;2.Block transfer with string
;3.exit   3
;[root@comppl208 nasm-2.10.07]#
```

EXPERIMENT NO. 10

NAME: Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (use of 64-bit registers is expected).

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof. Dabhade S. A.

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 10

AIM: Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (use of 64-bit registers is expected).

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY: Write an 8086 assembly language program to perform the multiplication of two 8 bit Hex numbers by

1) Successive Addition

2) Shift & Add Method

Objectives:

1. To study basics of assembly language programming i.e. Software commands to use them, format of alp, assembler directives.
2. To study step in assembly language programming.
3. To study DOS-DEBUG to execute program and to check the results.
4. To study basic 8086 instructions & an interrupt instruction

Assember directives used:-

1. segment
2. ends
3. macro & endm
4. proc & endp

ALGORITHMS FOR PROCEDURE MAIN:-

1. Start
2. Physical initialization of data segment

3.Display the following menu for user :-

**** MULTIPLICATION ****

1. Accept the numbers
2. Successive Addition
3. Shift & add method
4. Exit

Enter your choice::

4. Accept the choice from user.
5. If (choice =1), then call procedure „ACCEPT".
6. If (choice =2), then call procedure „SUCC".
7. If (choice =3), then call procedure „SHIFT".
8. STOP / Exit to DOS.

ALGORITHMS FOR PROCEDURE 'SUCC':-

1. Start
2. Copy the multiplicand in count register & copy the multiplier in base register.
3. Add the content of base register with itself.
4. Decrement the contents in count register.
5. If (choice !=0), then goto step (3)
6. Display the result in base register.
7. STOP / Exit to DOS.

ALGORITHMS FOR PROCEDURE 'SHIFT':-

1. Start
2. Get the LSB of multiplier.
3. Do the multiplication of LSB of multiplier with multiplicand by Successive Addition Method.
4. Store the result in accumulator.
5. Get the MSB of multiplier.
6. Do the multiplication of MSB of multiplier with multiplicand by successive addition method.
7. Store the result in base register. Shift the contents of base register towards left by 4 bits.
8. Add the contents of accumulator & base register.
9. Display the result.
10. RETURN.

Program:
```
section .data
  msg1 db 10,'enter the first number :-'
  msg1len equ $-msg1
  msg2 db 10,'enter the second  number :-'
  msg2len equ $-msg2
  resmsg db 10,'addition of no:-'
  resmsglen equ $-resmsg

section .bss
   numascii resb 03
     count resb 01
     num1 resb 01
     num2 resb 01
     result resb 02
     dispbuff resb 08
```

```
%macro dispmsg 2
   mov rax,1
   mov rdi,1
   mov rsi,%1
   mov rdx,%2
   syscall
%endmacro

%macro accept 2
   mov rax,0
   mov rdi,0
   mov rsi,%1
   mov rdx,%2
   syscall
%endmacro

section .text
 global_start:

_start:
  call input
  dispmsg resmsg,resmsglen
  call succ_add
  mov bx,[result]
  call  disp_proc

exit:  mov rax,60
       mov rdi,00
       syscall
input:
     dispmsg msg1,msg1len
     accept numascii,03
     call packnum
     mov [num1],bl
     dispmsg msg2,msg2len
     accept numascii,03
     call packnum
     mov [num2],bl
     ret

packnum:
        mov bl,0
        mov rcx,02
        mov rsi,numascii

up:
    rol bx,04
    mov al,[rsi]
    cmp al,39h
    jbe skip
    sub al,07h
```

```
skip:
      sub al,30h
      add bl,al
      inc rsi
      loop up
      ret

succ_add:
mov word[result],0

up1:
     mov bh,00
     mov bl,[num1]
     add word[result],bx
     dec  byte[num2]
     cmp byte[num2],00
     jne up1
     ret

disp_proc:
      mov rcx,4
      mov rdi,dispbuff

up2:
     rol bx,04
     mov al,bl
     and al,0fH
     cmp al,09h
     jbe skip2
     add al,07h

skip2:
      add al,30h
      mov [rdi],al
      inc rdi
      loop up2
    dispmsg dispbuff,4
  ret



;[sarala@localhost ~]$ nasm -f elf64 ass10.asm
[sarala@localhost ~]$ ld -o ass10 ass10.o
ld: warning: cannot find entry symbol _start; defaulting to
00000000004000b0
[sarala@localhost ~]$ ./ass10

enter the first number :-04

enter the second  number :-05
```

```
addition of no:-0014[sarala@localhost ~]$
[sarala@localhost ~]$
```

Conclusion:

Questions:

1. Explain Logic for the program
2. Explain Successive addition process
3. **3.** Explain Shift & rotate method.

EXPERIMENT NO. 11

NAME: Write X86 Assembly Language Program (ALP) to implement following OS commands i) COPY, ii) TYPE Using file operations. User is supposed to provide command line arguments

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof. Dabhade S. A.

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 11

AIM: Write X86 Assembly Language Program (ALP) to implement following OS commands i) COPY, ii) TYPE Using file operations. User is supposed to provide command line arguments

OBJECTIVES:

- To understand assembly language programming instruction set.
- To understand different assembler directives with example.

- To apply instruction set for implementing X86/64 bit assembly language programs

ENVIRONMENT:

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

THEORY: Aim:- Write X86menu driven Assembly Language Program (ALP) to implement OS (DOS) commands TYPE, COPY and DELETE using file operations. User is supposed to provide command line arguments in all cases.

Theory:-

- OPEN File

mov rax, 2 ; 'open' syscall
mov rdi, fname1 ; file name mov
rsi, 0 ;
mov rdx, 0777 ; permissions set
Syscall
mov [fd_in], rax

- OPEN File/Create file mov rax, 2

; 'open' syscall mov rdi, fname1 ;
file name
mov rsi, 0102o ; read and write mode, create if not
mov rdx, 0666o ; permissions set
Syscall

mov [fd_in], rax

- READ File

mov rax, 0 ; „Read' syscall mov
rdi, [fd_in] ; file Pointer mov
rsi, Buffer ; Buffer for read
mov rdx, length ; len of data want to read
Syscall

- WRITE File

```
mov rax, 01          ; „Write' syscall mov
rdi, [fd_in]         ; file Pointer mov rsi,
Buffer               ; Buffer for write
mov rdx, length      ; len of data want to read
Syscall
```

- DELETE File mov rax,87

mov rdi,Fname
syscall

- CLOSE File mov
  rax,3
  mov rdi,[fd_in]
  syscall

TYPE Command:-

- Open file in read mode using open interrupt.
- Read contents of file using read interrupt.
- Display contents of file using write interrupt.
- Close file using close interrupt
-

COPY Command

- Open file in read mode using open interrupt.
- Read contents of file using read interrupt.
- Create another file using read interrupt change only attributes.
- Open another file using open interrupt.
- Write contents of buffer into opened file.
- Close both files using close interrupt.

DELETE Command

1. DELETE file using delete interrupt

Algorithm

1. Accept Filenames from Command line.
2. Display MENU:-
    1. TYPE
    2. COPY
    3. DEL

3.Procedure for TYPE command

4.Procedure for COPE command

5. Procedure for DELETE command
6. EXIT

Program 1:DOS1.asm

```
%macro cmn 4                ;input/output
      mov rax,%1
      mov rdi,%2
      mov rsi,%3
      mov rdx,%4
      syscall
%endmacro
%macro exit 0
      mov rax,60
      mov rdi,0
      syscall
%endmacro

%macro fopen 1
      mov   rax,2      ;open
      mov   rdi,%1     ;filename
      mov   rsi,2      ;mode RW
      mov   rdx,0777o  ;File permissions
      syscall
%endmacro

%macro fread 3
      mov   rax,0      ;read
      mov   rdi,%1     ;filehandle
      mov   rsi,%2     ;buf
      mov   rdx,%3     ;buf_len
      syscall
%endmacro

%macro fwrite 3
```

```
     mov  rax,1      ;write/print
     mov  rdi,%1     ;filehandle
     mov  rsi,%2     ;buf
     mov  rdx,%3     ;buf_len
     syscall
%endmacro

%macro fclose 1
     mov  rax,3      ;close
     mov  rdi,%1     ;file handle
     syscall
%endmacro

section .data
     menu db 'MENU : ',0Ah
          db "1. TYPE",0Ah
          db "2. COPY",0Ah
          db "3. DELETE",0Ah
          db "4. Exit",0Ah
          db "Enter your choice : "
     menulen equ $-menu
     msg db "Command : "
     msglen equ $-msg
     cpysc db "File copied successfully !!",0Ah
     cpysclen equ $-cpysc
     delsc db 'File deleted successfully !!',0Ah
     delsclen equ $-delsc
     err db "Error ...",0Ah
     errlen equ $-err
     cpywr db 'Command does not exist',0Ah
     cpywrlen equ $-cpywr
     err_par db 'Insufficient parameter',0Ah
     err_parlen equ $-err_par


section .bss
     choice resb 2
     buffer resb 50
     name1 resb 15
     name2 resb 15
     cmdlen resb 1
     filehandle1 resq 1
     filehandle2 resq 1

     abuf_len        resq 1          ; actual buffer length
     dispnum resb 2

     buf resb   4096
     buf_len equ $-buf          ; buffer initial length

section .text
global _start
```

```
_start:

again:     cmn 1,1,menu,menulen
     cmn 0,0,choice,2

     mov al,byte[choice]
     cmp al,31h
     jbe op1            ;CF=1 and ZF=1
     cmp al,32h
     jbe op2
     cmp al,33h
     jbe op3

        exit
        ret

op1:
     call tproc
     jmp again

op2:
     call cpproc
     jmp again


op3:
     call delproc
     jmp again



;type command procedure
tproc:
     cmn 1,1,msg,msglen
     cmn 0,0,buffer,50
     mov byte[cmdlen],al
     dec byte[cmdlen]

     mov rsi,buffer
     mov al,[rsi]                ;search for correct type command
     cmp al,'t'
     jne skipt              ; ZF=0
     inc rsi
     dec byte[cmdlen]
     jz skipt
     mov al,[rsi]
     cmp al,'y'
     jne skipt
     inc rsi
     dec byte[cmdlen]
     jz skipt
     mov al,[rsi]
```

```
	cmp al,'p'
	jne skipt
	inc rsi
	dec byte[cmdlen]
	jz skipt
	mov al,[rsi]
	cmp al,'e'
	jne skipt
	inc rsi
	dec byte[cmdlen]
	jnz correctt
	cmn 1,1,err_par,err_parlen
	call exit

skipt:	cmn 1,1,cpywr,cpywrlen
	exit
correctt:
	mov rdi,name1		;finding file name
	call find_name

	fopen name1		; on succes returns handle
	cmp rax,-1H		; on failure returns -1
	jle error
	mov [filehandle1],rax

	xor rax,rax
	fread [filehandle1],buf, buf_len
	mov [abuf_len],rax
	dec byte[abuf_len]

	cmn 1,1,buf,abuf_len	;printing file content on screen

ret


;copy command procedure
cpproc:
	cmn 1,1,msg,msglen
	cmn 0,0,buffer,50		;accept command
	mov byte[cmdlen],al
	dec byte[cmdlen]

	mov rsi,buffer
	mov al,[rsi]		;search for copy
	cmp al,'c'
	jne skip
	inc rsi
	dec byte[cmdlen]
	jz skip
	mov al,[rsi]
	cmp al,'o'
	jne skip
```

```
        inc rsi
        dec byte[cmdlen]
        jz skip
        mov al,[rsi]
        cmp al,'p'
        jne skip
        inc rsi
        dec byte[cmdlen]
        jz skip
        mov al,[rsi]
        cmp al,'y'
        jne skip           ZF=1
        inc rsi
        dec byte[cmdlen]
        jnz correct
        cmn 1,1,err_par,err_parlen
        exit

skip:   cmn 1,1,cpywr,cpywrlen
        exit
correct:
        mov rdi,name1              ;finding first file name
        call find_name

        mov rdi,name2              ;finding second file name
        call find_name

skip3:      fopen name1                 ; on succes returns handle
        cmp rax,-1H                ; on failure returns -1
        jle error
        mov [filehandle1],rax

        fopen name2                ; on succes returns handle
        cmp rax,-1H                ; on failure returns -1
        jle error
        mov [filehandle2],rax

        xor rax,rax
        fread [filehandle1],buf, buf_len
        mov [abuf_len],rax
        dec byte[abuf_len]

        fwrite [filehandle2],buf, [abuf_len]       ;write to file

        fclose     [filehandle1]
        fclose     [filehandle2]
        cmn 1,1,cpysc,cpysclen

        jmp again
error:
        cmn 1,1,err,errlen
        exit
```

```
ret



;delete command procedure
delproc:

      cmn 1,1,msg,msglen
      cmn 0,0,buffer,50          ;accept command
      mov byte[cmdlen],al
      dec byte[cmdlen]

      mov rsi,buffer
      mov al,[rsi]               ;search for copy
      cmp al,'d'
      jne skipr
      inc rsi
      dec byte[cmdlen]
      jz skipr
      mov al,[rsi]
      cmp al,'e'
      jne skipr
      inc rsi
      dec byte[cmdlen]
      jz skipr
      mov al,[rsi]
      cmp al,'l'
      jne skipr
      inc rsi
      dec byte[cmdlen]
      jnz correctr
      cmn 1,1,err_par,err_parlen
      exit

skipr:    cmn 1,1,cpywr,cpywrlen
      exit

correctr:
      mov rdi,name1              ;finding first file name
      call find_name

      mov rax,87           ;unlink system call
      mov rdi,name1
      syscall

      cmp rax,-1H                ; on failure returns -1
      jle errord
      cmn 1,1,delsc,delsclen
      jmp again

errord:
      cmn 1,1,err,errlen
```

```
	exit

ret


find_name:			;finding file name from command
	inc rsi
	dec byte[cmdlen]
cont1:	mov al,[rsi]
	mov [rdi],al
	inc rdi
	inc rsi
	mov al,[rsi]
	cmp al,20h		;searching for space
	je skip2
	cmp al,0Ah		;searching for enter key
	je skip2
	dec byte[cmdlen]
	jnz cont1
	cmn 1,1,err,errlen
	exit

skip2:
ret
```

```
;Program2: Actual logic of command
%macro cmn 4			;input/output
	mov rax,%1
	mov rdi,%2
	mov rsi,%3
	mov rdx,%4
	syscall
%endmacro
%macro exit 0
	mov rax,60
	mov rdi,0
	syscall
%endmacro

%macro fopen 1
	mov  rax,2	;open
	mov  rdi,%1	;filename
	mov  rsi,2	;mode RW
	mov  rdx,0777o ;File permissions
	syscall
%endmacro

%macro fread 3
```

```
     mov  rax,0      ;read
     mov  rdi,%1     ;filehandle
     mov  rsi,%2     ;buf
     mov  rdx,%3     ;buf_len
     syscall
%endmacro

%macro fwrite 3
     mov  rax,1      ;write/print
     mov  rdi,%1     ;filehandle
     mov  rsi,%2     ;buf
     mov  rdx,%3     ;buf_len
     syscall
%endmacro

%macro fclose 1
     mov  rax,3      ;close
     mov  rdi,%1     ;file handle
     syscall
%endmacro

section .data
     menu db 'MENU : ',0Ah
          db "1. TYPE",0Ah
          db "2. COPY",0Ah
          db "3. DELETE",0Ah
          db "4. Exit",0Ah
          db "Enter your choice : "
     menulen equ $-menu
     msg db "Command : "
     msglen equ $-msg
     cpysc db "File copied successfully !!",0Ah
     cpysclen equ $-cpysc
     delsc db 'File deleted successfully !!',0Ah
     delsclen equ $-delsc
     err db "Error ...",0Ah
     errlen equ $-err
     cpywr db 'Command does not exist',0Ah
     cpywrlen equ $-cpywr
     err_par db 'Insufficient parameter',0Ah
     err_parlen equ $-err_par


section .bss
     choice resb 2
     buffer resb 50
     name1 resb 15
     name2 resb 15
     cmdlen resb 1
     filehandle1 resq 1
     filehandle2 resq 1
```

```
	abuf_len		resq 1		; actual buffer length
	dispnum resb 2

	buf resb  4096
	buf_len equ $-buf		; buffer initial length

section .text
global _start
_start:

again:	cmn 1,1,menu,menulen
	cmn 0,0,choice,2

	mov al,byte[choice]
	cmp al,31h
	jbe op1
	cmp al,32h
	jbe op2
	cmp al,33h
	jbe op3

	  exit
	  ret

op1:
	call tproc
	jmp again

op2:
	call cpproc
	jmp again


op3:
	call delproc
	jmp again



;type command procedure
tproc:
	cmn 1,1,msg,msglen
	cmn 0,0,buffer,50
	mov byte[cmdlen],al
	dec byte[cmdlen]

	mov rsi,buffer
	mov al,[rsi]			;search for correct type command
	cmp al,'t'
	jne skipt		ZF=0
	inc rsi
	dec byte[cmdlen]
```

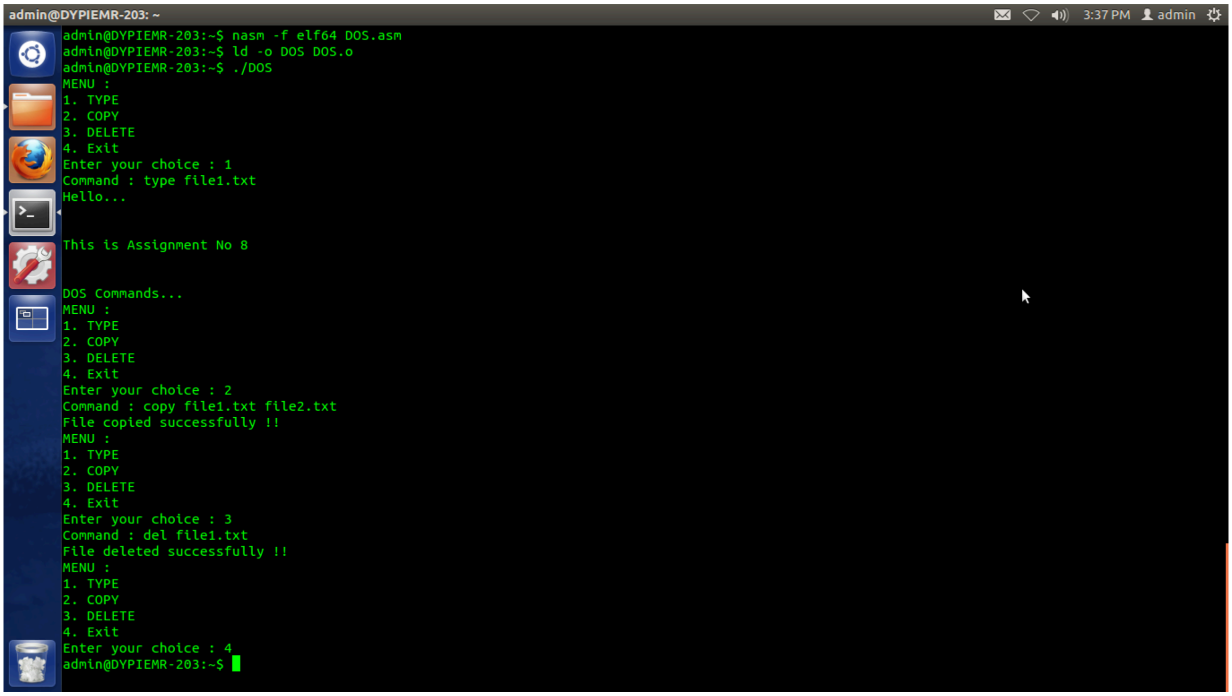
```
	jz skipt
	mov al,[rsi]
	cmp al,'y'
	jne skipt
	inc rsi
	dec byte[cmdlen]
	jz skipt
	mov al,[rsi]
	cmp al,'p'
	jne skipt
	inc rsi
	dec byte[cmdlen]
	jz skipt
	mov al,[rsi]
	cmp al,'e'
	jne skipt
	inc rsi
	dec byte[cmdlen]
	jnz correctt
	cmn 1,1,err_par,err_parlen
	call exit

skipt:	cmn 1,1,cpywr,cpywrlen
	exit
correctt:
	mov rdi,name1			;finding file name
	call find_name

	fopen name1			; on succes returns handle
	cmp rax,-1H			; on failure returns -1
	jle error
	mov [filehandle1],rax

	xor rax,rax
	fread [filehandle1],buf, buf_len
	mov [abuf_len],rax
	dec byte[abuf_len]

	cmn 1,1,buf,abuf_len		;printing file content on screen

ret


;copy command procedure
cpproc:
	cmn 1,1,msg,msglen
	cmn 0,0,buffer,50		;accept command
	mov byte[cmdlen],al
	dec byte[cmdlen]

	mov rsi,buffer
	mov al,[rsi]			;search for copy
```

```
      cmp al,'c'
      jne skip
      inc rsi
      dec byte[cmdlen]
      jz skip
      mov al,[rsi]
      cmp al,'o'
      jne skip
      inc rsi
      dec byte[cmdlen]
      jz skip
      mov al,[rsi]
      cmp al,'p'
      jne skip
      inc rsi
      dec byte[cmdlen]
      jz skip
      mov al,[rsi]
      cmp al,'y'
      jne skip           ZF=1
      inc rsi
      dec byte[cmdlen]
      jnz correct
      cmn 1,1,err_par,err_parlen
      exit

skip: cmn 1,1,cpywr,cpywrlen
      exit
correct:
      mov rdi,name1               ;finding first file name
      call find_name

      mov rdi,name2               ;finding second file name
      call find_name

skip3:     fopen name1                 ; on succes returns handle
      cmp rax,-1H                 ; on failure returns -1
      jle error
      mov [filehandle1],rax

      fopen name2                 ; on succes returns handle
      cmp rax,-1H                 ; on failure returns -1
      jle error
      mov [filehandle2],rax

      xor rax,rax
      fread [filehandle1],buf, buf_len
      mov [abuf_len],rax
      dec byte[abuf_len]

      fwrite [filehandle2],buf, [abuf_len]       ;write to file
```

```
        fclose      [filehandle1]
        fclose      [filehandle2]
        cmn 1,1,cpysc,cpysclen

        jmp again
error:
        cmn 1,1,err,errlen
        exit
ret



;delete command procedure
delproc:

        cmn 1,1,msg,msglen
        cmn 0,0,buffer,50           ;accept command
        mov byte[cmdlen],al
        dec byte[cmdlen]

        mov rsi,buffer
        mov al,[rsi]                ;search for copy
        cmp al,'d'
        jne skipr
        inc rsi
        dec byte[cmdlen]
        jz skipr
        mov al,[rsi]
        cmp al,'e'
        jne skipr
        inc rsi
        dec byte[cmdlen]
        jz skipr
        mov al,[rsi]
        cmp al,'l'
        jne skipr
        inc rsi
        dec byte[cmdlen]
        jnz correctr
        cmn 1,1,err_par,err_parlen
        exit

skipr:      cmn 1,1,cpywr,cpywrlen
        exit

correctr:
        mov rdi,name1               ;finding first file name
        call find_name

        mov rax,87            ;unlink system call
        mov rdi,name1
        syscall
```

```
	cmp rax,-1H			; on failure returns -1
	jle errord
	cmn 1,1,delsc,delsclen
	jmp again

errord:
	cmn 1,1,err,errlen
	exit

ret


find_name:			;finding file name from command
	inc rsi
	dec byte[cmdlen]
cont1:	mov al,[rsi]
	mov [rdi],al
	inc rdi
	inc rsi
	mov al,[rsi]
	cmp al,20h		;searching for space
	je skip2
	cmp al,0Ah		;searching for enter key
	je skip2
	dec byte[cmdlen]
	jnz cont1
	cmn 1,1,err,errlen
	exit

skip2:
ret
```

Output:


admin@DYPIEMR-203: ~
3:37 PM admin
admin@DYPIEMR-203:~$ nasm -f elf64 DOS.asm
admin@DYPIEMR-203:~$ ld -o DOS DOS.o
admin@DYPIEMR-203:~$ ./DOS
MENU :
1. TYPE
2. COPY
3. DELETE
4. Exit
Enter your choice : 1
Command : type file1.txt
Hello...
This is Assignment No 8
DOS Commands...
MENU :
1. TYPE
2. COPY
3. DELETE
4. Exit
Enter your choice : 2
Command : copy file1.txt file2.txt
File copied successfully !!
MENU :
1. TYPE
2. COPY
3. DELETE
4. Exit
Enter your choice : 3
Command : del file1.txt
File deleted successfully !!
MENU :
1. TYPE
2. COPY
3. DELETE
4. Exit
Enter your choice : 4
admin@DYPIEMR-203:~$

Conclusion
:

FAQ:-

1. Why we use TYPE command?
2. Why we use COPY command?
3. Why we use DEL command?

EXPERIMENT NO. 12

NAME: Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program_1 execution and write FAR PROCEDURES in Program_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof. Dabhade S. A.

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 12

AIM: Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program_1 execution and write FAR PROCEDURES in Program_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

Title:- Near and FAR Procedure

Aim:- Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program_1 execution and write FAR PROCEDURES in Program_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.

Prerequisites:

1. Basic knowledge about String operations
2. Knowledge about procedure in ALP

Objectives:

1. To study basics of assembly language programming i.e. Software development alp, assembler directives.
2. To study step in assembly language programming.
3. To study dos-debug to execute program and to check the results.
4. To study basic 8086 instructions & interrupt instruction

Theory:

Far CALL and RET Operation

When executing a far call, the processor performs these actions

1. Pushes current value of the CS register on the stack.

2. Pushes the current value of the EIP register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the EIP register.
5. Begins execution of the called procedure.

When executing a far return, the processor does the following:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. Pops the top-of-stack value (the segment selector for the code segment being returned to) into the CS register.
3. (If the RET instruction has an optional n argument.) Increments pointer by the number of bytes specified with the n operand to release parameters from the stack.
4. Resumes execution of the calling procedure.

ASSEMBER DIRECTIVES USED:-

1. MACRO & ENDM
2. PROC & ENDP
3. EXTRN
4. PUBLIC

LIST OF PROCESDURES USED:-

1. ACCEPT PROC
2. CONCAT PROC
3. COMPARE PROC
4. SUBSTR PROC
5. NO_WORD PROC
6. NO_CHAR PROC

LIST OF MACROS USED:-

MACRO IS USED TO DISPLAY A STRING:-

DISP MACRO MESSAGE

MOV AH, 09H

LEA DX, MESSAGE

INT 21H

ENDM

ALGORITHMS:-

ALGORITHMS FOR PROCEDURE MAIN:-

1. Start
2. Physical initialization of data segment
3. Display the following menu using macro :-

**** STRING OPERATIONS ****

1. Accept the string
2. Concatenation
3. Check for substring
4. Compare the strings
5. Number of words
6. Number of characters
7. Number of digits
8. Number of Capital characters
9. Exit

Select your option ::

4. Accept the choice from user.
5. If (choice =1), then call FAR procedure „ACCEPT".
6. If (choice =2), then call FAR procedure „CONCAT".
7. If (choice =3), then call FAR procedure „SUBSTR".
8. If (choice =4), then call FAR procedure „COMPARE".
9. If (choice =5), then call FAR procedure „NO_WORDS".
10. If (choice =6), then call FAR procedure „NO_CHAR".
11. If (choice =7), then call FAR procedure „NO_DIGIT".

12. If (choice =8), then call FAR procedure „NO_CAP“.
13. If (choice =9), then Exit to DOS, terminating the program.

14. If (choice!=9), repeat the steps (3), (4) & (5).
15. Stop.

ALGORITHMS FOR PROCEDURE 'CONCAT':-

1. Start
2. Initialization of pointer1 to first string & pointer2 to second string.
3. Initialize the count1 & count2 to length of first & second string respectively.
4. Display the character pointed by pointer1.
5. Increment the pointer1 & decrement the count1.
6. If count1 is not zero, goto step (4).
7. Display the character pointed by pointer2.
8. Increment the pointer2 & decrement the count2.
9. If count2 is not zero, gotostep(7).
10. Stop.

ALGORITHMS FOR PROCEDURE 'SUBSTR':-

1. Start
2. Initialize the source pointer to main string & destination pointer to substring.
3. Initialize the count1 & count2 to length of main string & substring respectively.
4. Compare the characters pointed by source & destination pointer.
5. If they are equal, goto step (6), else goto ( ).
6. Increment the source pointer & destination pointer. Decrement the count1 & count2.
7. If (count2!=0), then goto step (4) else goto step (9).
8. Increment the source pointer & reinitialize the destination pointer. Decrement the count1 & reinitialize the count2 &goto step (4).
9. Increment the count for number of occurrences.
10. If (count1!=0), then goto step (10) else goto step (12).
11. Reinitialize the destination pointer & count2 &goto step (4).
12. If count for no. of occurrences of substring is „zero", then print "NOT SUBSTRING", else print "SUBSTRING" & print the number of occurrences of string.
13. Stop.

ALGORITHMS FOR PROCEDURE COMPARE:-

1. Start
2. Initialize the source pointer the source pointer to string1 & destination pointer to string2. Initialize the count1 & count2 to the length of string1 & string2 respectively.
3. If length of string1 & string2 are not same, goto step (9).
4. Compare the characters pointed by source & destination pointer.
5. If they are equal, goto step (6), else goto(9).
6. Increment the source pointer & destination pointer.
   Decrement count1.

7. If (count1!=0), goto step (4), else goto (8).
8. Print “STRING ARE EQUAL……!” &goto (10).

9. Print “STRING IS NOT EQUAL……!”
10. STOP.

ALGORITHMS FOR PROCEDURE ‘NO_WORD’:-

1. Start
2. Initialize the source pointer to the given string & the count to length of string.
3. Compare the character pointed by source pointer to “ “ (space).
4. If equal, increment the count for no. of words & increment source pointer.
   Else increment the source pointer, goto step (3) till count!=0.
5. Print the no. of words.
6. STOP.

ALGORITHMS FOR PROCEDURE ‘NO_CHAR’:-

1. Start
2. Initialize the source pointer to the given string & the count to length of string.
3. Compare the character with „30H‟. If below, goto step (8).
   If greater, goto step (4).
4. Compare the character with „39H‟. If below or equal, „increment the count for no. of digits‟. If greater, goto step (5).
5. Compare the character with „5AH‟. If below or equal, „increment the count for no. of capital letters‟ & also increment the count for no. of characters. If greater, goto step (6).
6. Compare the character with „60H‟, If below or equal, goto step (8).
   If greater, goto step (7).
7. Compare the character with „7AH‟. If below or equal, „increment the count for no. of characters‟.
8. Decrement the count and increment the source pointer &goto step (9).
9. If count is not zero, goto step (3). Else goto step (10).
10. Display the result i.e. no. of words, no. of characters, no. of capital letters.
11. STOP.

Progam1:
```
;Assignment no. :12
;Assignment Name :X86/64 Assembly language program (ALP) to find
;     a) Number of Blank spaces
;     b) Number of lines
;     c) Occurrence of a particular character.
;Accept the data from the text file. The text file has to be accessed
during Program_1 execution.
;Write FAR PROCEDURES in Program_2 for the rest of the processing.
;Use of PUBLIC/GLOBAL and EXTERN directives is mandatory.
;-------------------------------------------------------------------
---
```

```
extern     far_proc          ; [ FAR PROCRDURE
                             ;   USING EXTERN DIRECTIVE ]

global     filehandle, char, buf, abuf_len

%include   "macro.asm"

;-----------------------------------------------------------------------
---
section .data
     nline      db   10
     nline_len  equ  $-nline

     ano        db   10,10,10,10,"ML assignment 05 :- String Operation
using Far Procedure"
                db
10,"---------------------------------------------------",10
     ano_len    equ  $-ano

     filemsg    db   10,"Enter filename for string operation    : "
     filemsg_len     equ  $-filemsg

     charmsg    db   10,"Enter character to search   : "
     charmsg_len     equ  $-charmsg

     errmsg     db   10,"ERROR in opening File...",10
     errmsg_len equ  $-errmsg

     exitmsg    db   10,10,"Exit from program...",10,10
     exitmsg_len     equ  $-exitmsg

;-----------------------------------------------------------------------
------
section .bss
     buf             resb 4096
     buf_len         equ  $-buf      ; buffer initial length

     filename        resb 50
     char            resb 2

     filehandle      resq 1
     abuf_len        resq 1          ; actual buffer length

;-----------------------------------------------------------------------
-----
section .text
     global _start

_start:
          print ano,ano_len          ;assignment no.
```

```
          print filemsg,filemsg_len
          read  filename,50
          dec   rax
          mov   byte[filename + rax],0          ; blank char/null char

          print charmsg,charmsg_len
          read  char,2

          fopen filename              ; on succes returns handle
          cmp   rax,-1H               ; on failure returns -1
          jle   Error
          mov   [filehandle],rax

          fread [filehandle],buf, buf_len
          mov   [abuf_len],rax

          call  far_proc
          jmp   Exit

Error:    print errmsg, errmsg_len

Exit:     print exitmsg,exitmsg_len
          exit
;----------------------------------------------------------------------
-----------
Program 2:
;----------------------------------------------------------------------
section .data
     nline      db   10,10
     nline_len: equ  $-nline

     smsg       db   10,"No. of spaces are : "
     smsg_len:  equ  $-smsg

     nmsg       db   10,"No. of lines are  : "
     nmsg_len:  equ  $-nmsg

     cmsg       db   10,"No. of character occurances are   : "
     cmsg_len:  equ  $-cmsg

;----------------------------------------------------------------------
section .bss

     scount     resq 1
     ncount     resq 1
     ccount     resq 1

     char_ans   resb 16

;----------------------------------------------------------------------
global    far_proc
```

```
extern     filehandle, char, buf, abuf_len

%include   "macro.asm"
;---------------------------------------------------------------------
section .text
     global     _main
_main:

far_proc:                  ;FAR Procedure

          xor  rax,rax
          xor  rbx,rbx
          xor  rcx,rcx
          xor  rsi,rsi

          mov  bl,[char]
          mov  rsi,buf
          mov  rcx,[abuf_len]

again:    mov  al,[rsi]

case_s:   cmp  al,20h          ;space : 32 (20H)
          jne  case_n
          inc  qword[scount]
          jmp  next

case_n:   cmp  al,0Ah          ;newline : 10(0AH)
          jne  case_c
          inc  qword[ncount]
          jmp  next

case_c:   cmp  al,bl           ;character
          jne  next
          inc  qword[ccount]

next:     inc  rsi
          dec  rcx             ;
          jnz  again           ;loop again
               print smsg,smsg_len
          mov  rax,[scount]
          call display
            print nmsg,nmsg_len
          mov  rax,[ncount]
          call display

          print cmsg,cmsg_len
          mov  rax,[ccount]
          call display

     fclose     [filehandle]
     ret
```

```
;---------------------------------------------------------------------
display:
     mov   rsi,char_ans+3  ; load last byte address of char_ans in rsi
     mov   rcx,4           ; number of digits

cnt: mov   rdx,0           ; make rdx=0 (as in div instruction
rdx:rax/rbx)
     mov   rbx,10          ; divisor=10 for decimal and 16 for hex
     div   rbx
;    cmp   dl, 09h         ; check for remainder in RDX
;    jbe   add30
;    add   dl, 07h
;add30:
     add   dl,30h          ; calculate ASCII code
     mov   [rsi],dl        ; store it in buffer
     dec   rsi             ; point to one byte back

     dec   rcx             ; decrement count
     jnz   cnt             ; if not zero repeat

     print char_ans,4           ; display result on screen
ret
;-------------------------------------------------------------------

Add Macro file
Input Text file

;nasm –f elf64 file1.asm
;nasm –f elf64 file2.asm
;ld file1.o file2.0  -file
;./file
```

Conclusion:

FAQ:

Q. 1 Explain which instructions are used for string operations in ALP

Q.2 Explain logic for the program.

Q.3 Explain What is the Procedure in ALP

Q.4 Explain FAR procedure.

Q.5 Which interrupts are used in the program.

Assi

EXPERIMENT NO. 13

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

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY:

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 13

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

1.Accept Number from User

2.Call Factorial Procedure

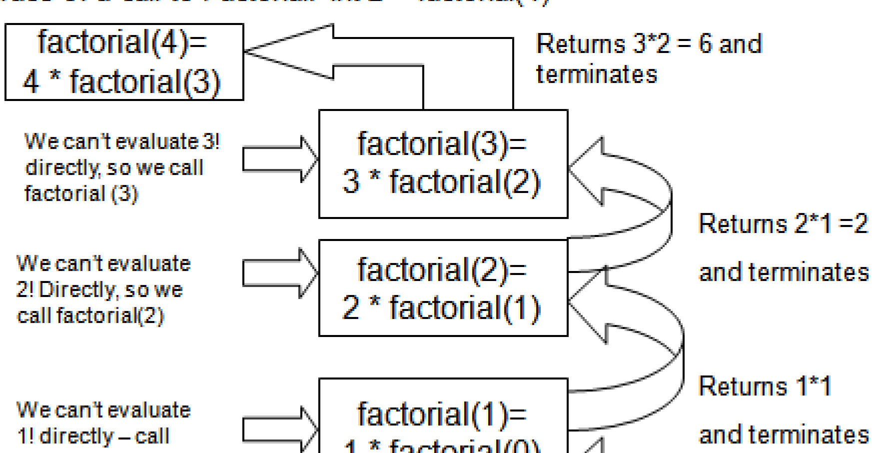
3. Define Recursive Factorial Procedure 4.Disply Result.

Conclusion:

FAQ:-

```
; Microprocessor Lab
; Assignment no. : 13
;Problem Statement: 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

%macro cmn 4                ;common macro for input/output
     mov rax,%1
     mov rdi,%2
```

Trace of a call to Factorial: int z = factorial(4)



```
	mov rsi,%3
	mov rdx,%4
	syscall
%endmacro

section .data
	num db 00h
	msg db "Factorial is : "
	msglen equ $-msg
	msg1 db "*****Program to find Factorial of a number***** ",0Ah
	     db "Enter the number : ",
	msg1len equ $-msg1
	msgth db 0Ah,"Thank you for using program ",0Ah
	msgthlen equ $-msgth
	zerofact db " 00000001 "
	zerofactlen equ $-zerofact

section .bss
	dispnum resb 16
	result resb 4
	temp resb 3		;buffer for input


section .text
global _start
_start:

	cmn 1,1,msg1,msg1len
	cmn 0,0,temp,3			;accept number from user
	call convert			;convert number from ascii to hex
	mov [num],dl

	cmn 1,1,msg,msglen

	xor rdx,rdx				;rdx=0
	xor rax,rax				;rax=0
	mov al,[num]			;store number in accumulator
	cmp al,01h
	jbe endfact				;cf=1 and zf=1
	xor rbx,rbx
	mov bl,01h
	call factr		;call factorial procedure
	call display

	call exit
endfact:
	cmn 1,1,zerofact,zerofactlen
	call exit

	factr:				;recursive procedure

			cmp rax,01h
```

```
          je retcon1    ;if ZF=1
          push rax
          dec rax

          call factr

     retcon:
          pop rbx
          mul ebx
          jmp endpr

     retcon1:               ;if rax=1 return
          pop rbx
          jmp retcon
     endpr:

ret

display:               ; procedure to convert hex to ascii

          mov rsi,dispnum+15
          xor rcx,rcx
          mov cl,16

     cont:
          xor rdx,rdx
          xor rbx,rbx
          mov bl,10h
          div ebx
          cmp dl,09h
          jbe skip
          add dl,07h
     skip:
          add dl,30h
          mov [rsi],dl
          dec rsi
          loop cont

          cmn 1,1,dispnum,16

ret

convert:               ;procedure to convert ascii to hex
          mov rsi,temp
          mov cl,02h
          xor rax,rax
          xor rdx,rdx
     contc:
          rol dl,04h
          mov al,[rsi]
          cmp al,39h
          jbe skipc
```

```
            sub al,07h
        skipc:
            sub al,30h
            add dl,al
            inc rsi
            dec cl
            jnz contc

    ret

    exit:               ;exit system call
            cmn 1,1,msgth,msgthlen
            mov rax,60
            mov rdi,0
            syscall

    ret
```

```
;Output

[root@localhost sarala]# nasm -f elf64 ass13.asm
[root@localhost sarala]# ld -o ass13 ass13.o
[root@localhost sarala]# ./ass13
*****Program to find Factorial of a number*****
Enter the number : 04
Factorial is : 0000000000000018
Thank you for using program
[root@localhost sarala]#
```

1.What are the applications of factorial?

1. Why to use factorial?

EXPERIMENT NO. 14

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

NAME OF LABORATORY: MICROPROCESSOR LAB

DATE OF EXPERIMENT:

DATE OF SUBMISSION:

NAME OF STUDENT:

ROLL NO:

SEMESTER: FOURTH SEMESTER

YEAR: SECOND YEAR

NAME OF FACULTY: Prof.

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 14

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

LIST OF INTERRRUPTS USED:

LIST OF ASSEMBLER DIRECTIVES USED:

LIST OF MACROS USED:

LIST OF PROCEDURES USED:

ALGORITHM:

FLOWCHART: