NBN Sinhgad Technical Institute Campus

NBN Sinhgad School of Engineering, Ambegaon, Pune

DEPARTMENT OF COMPUTER ENGINEERING

210257: Microprocessor Laboratory

LABORATORY MANUAL

ACADEMIC YEAR 2020-21

NAME: Prof. Nandini Babbar

DEPARTMENT OF COMPUTER ENGINEERING

INDEX

Sr. No.	Date	Experiment Performed	Page No	Sign	Remark
1		Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers.			
2		Write an X86/64 ALP to count number of positive and negative numbers from the array.			
3		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).			
4		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.			
5		Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.			
6		Write X86/64 ALP to perform overlapped block transfer with string specific instructions Block containing data can be defined in the data segment.			
7		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).			
8		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			
9		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.			

10		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.			
		Mini Project: Design Calculator			

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. Nandini Babbar

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.

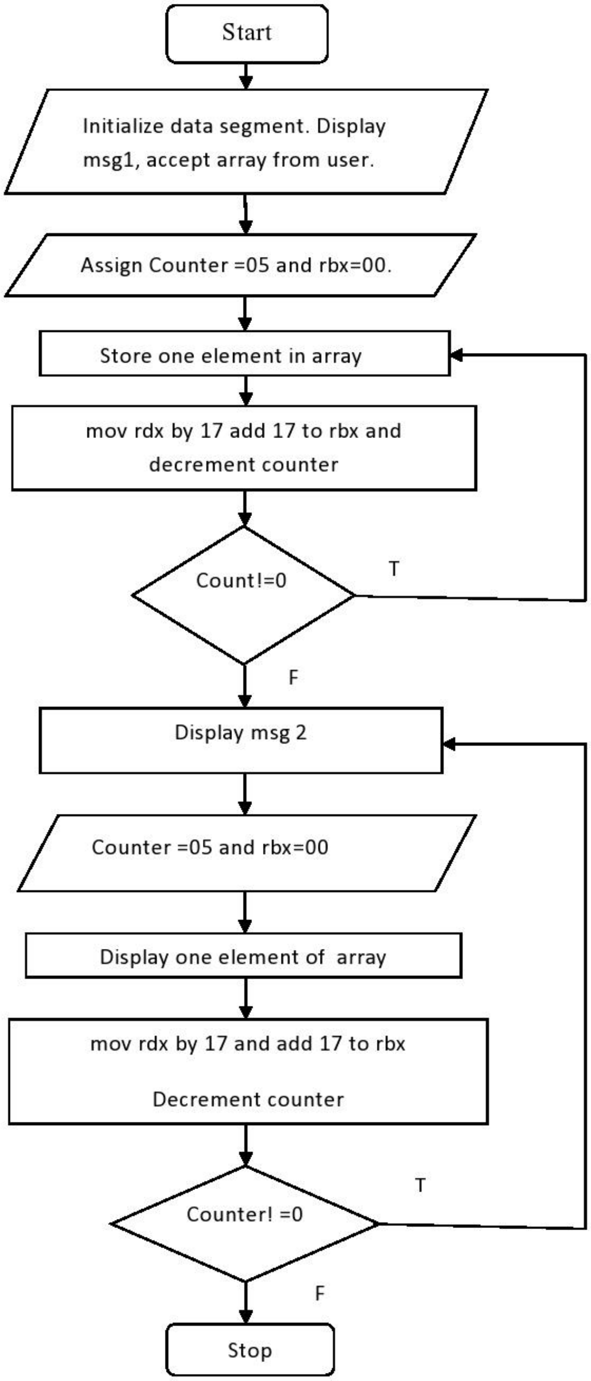
STEP 13: Add 17 to rbx.

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

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

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

CONCLUSION:

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

EXPERIMENT NO. 02

NAME: Write an X86/64 ALP to 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. Nandini Babbar

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 02

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.

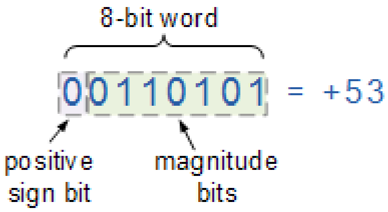
But how do we represent signed binary numbers if all we have is a bunch of one’s and zero’s. We know that binary digits, or bits only have two values, either a “1” or a “0” and conveniently for us, a sign also has only two values, being a “+” or a “–“.

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.

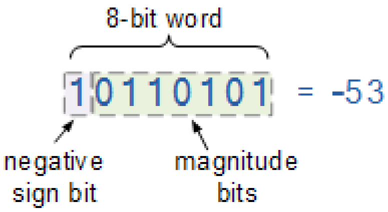
For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is “0”, this means the number is positive in value. If the sign bit is “1”, then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number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

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. 03

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: Prof. Nandini Babbar

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 03

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
```

CONCLUSION:

EXPERIMENT NO. 04

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. Nandini Babbar

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 04

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

LIST OF INTERRRUPTS USED:

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.

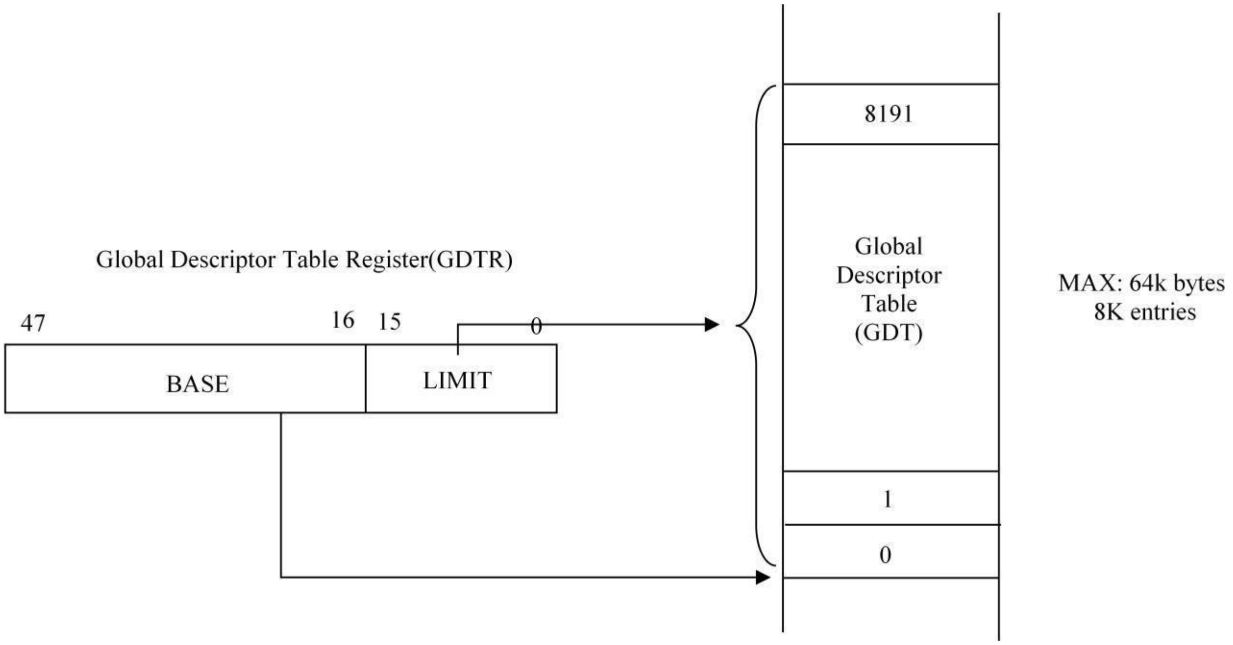
SIDT D:- Store the interrupt descriptor table register. D specifies both the memory location that gets the first of the six bytes to be stored from the IDTR.

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 control of GDT

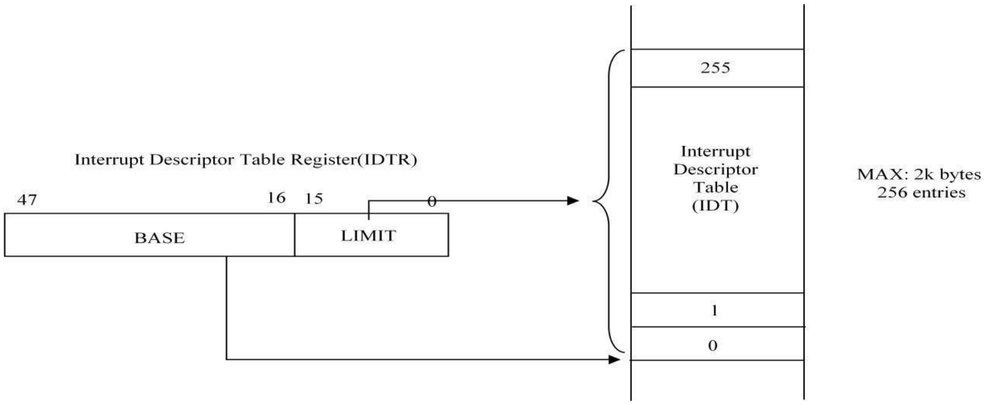
8) Store control of LDT
9) Store control of IDT
10) Store contains of TR
11) Call display processor to display control of GDT
12) Call display processor to display control of LDT
13) Call display processor to display control of IDT
14) Call display processor to display control of TR
15) Call display processor to display control of MSW
16) Point to esi buffer 17)Load no. of digit to display
17) Rotate no. left by 4 bit
18) Move lower byte in DL
19) Mask upper digit of byte in DL
20) Add 30h to calculate ASCCI code
21) If DL < 39 , no add 7, yes Skip adding 07 more
22) Store ASCCI code in buffer
23) Point to next byte
24) Display the no. from buffer
25) END

FLOWCHART:

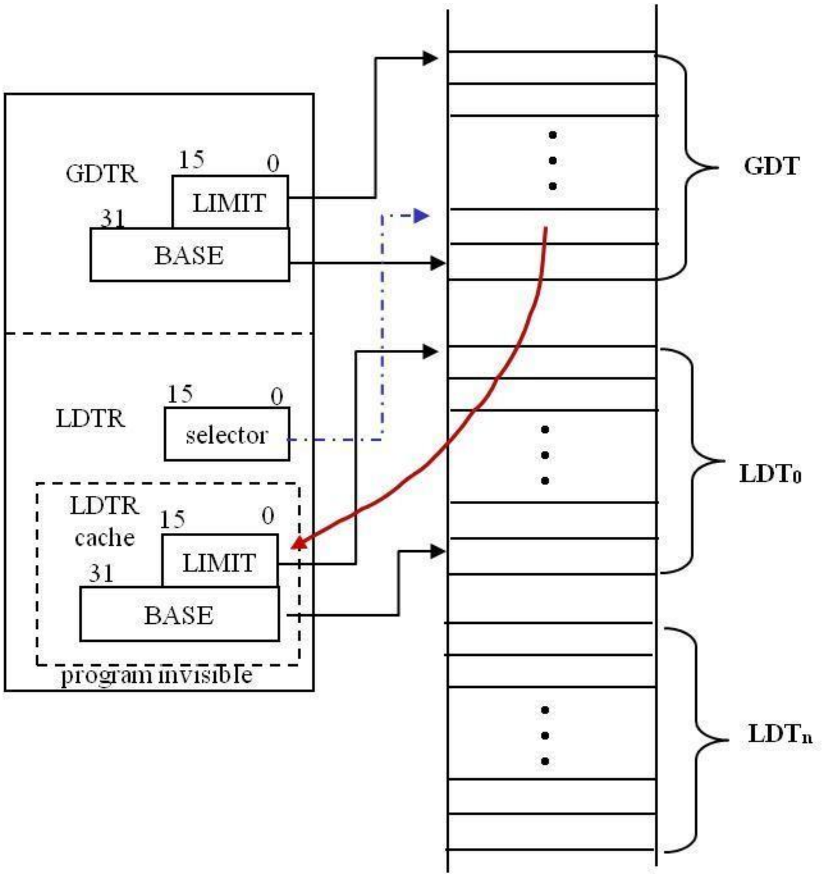

Global Descriptor Table Register(GDTR)
47
16
15
0
BASE
LIMIT
8191
Global Descriptor Table (GDT)
1
0
MAX: 64k bytes
8K entries

Global Descriptor Table Register (GDTR):

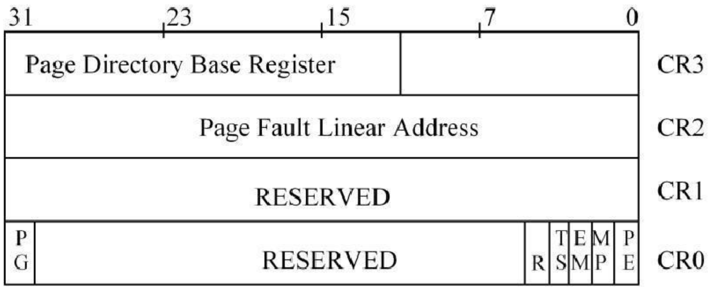
Interrupt Descriptor Table Register (IDTR):


Interrupt Descriptor Table Register(IDTR)
47
16
15
0
BASE
LIMIT
255
Interrupt Descriptor Table (IDT)
1
0
MAX: 2k bytes
256 entries

Local Descriptor Table Register (LDTR):

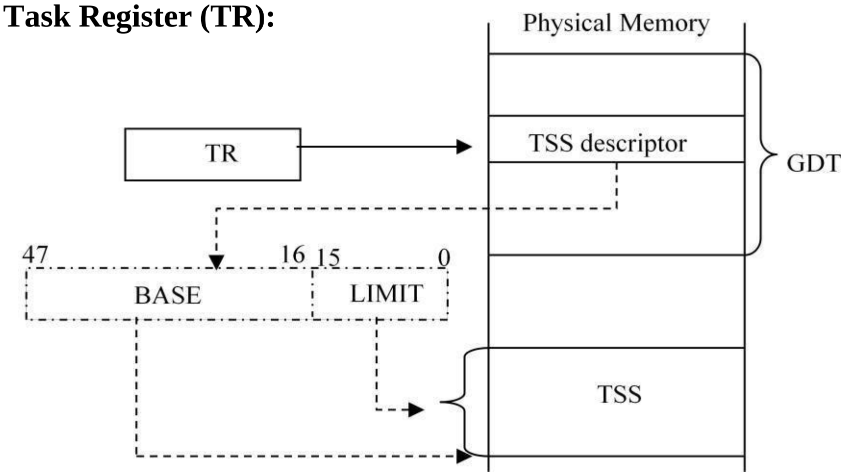


Control Registers:



- 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

Task Register (TR):



PROGRAM:

```
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    ;base register (upper part)
            resw 1   ;limit (lower part)
        ldt resw 1
        idt resd 1     ;base register (upper part)
            resw 1    ;limit  (lower part)

        tr  resw 1           ;16 bit

        cr0_data resd 1 ;32 bit

        dnum_buff resb 04  ;lowest TR,LDTR,Limit (2 times call)

%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,1;checking PE bit,if 1=protectrd mode else realmode
            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]              ;higher part base address
            call disp_num

            mov bx,[gdt+2]              ;lower part limit
            call disp_num

            disp colmsg,1

            mov bx,[gdt]                ;lower part
            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]     ;32 bit higher part
        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 no. of digits to display
up1:
        rol bx,4                ;rotate no. left by four bits
        mov dl,bl               ;mov lower byte in dl
        add dl,0fh
        add dl,30h              ;add 30 h to calculate ASCII code
        cmp dl,39h      ;compare with 39h
        jbe skip1
        add dl,07h              ;else add 07
skip1:
        mov [esi],dl
        inc esi
        loop up1

        disp dnum_buff, 4  ;display the no.from buffer'
        ret



;RESULT:
;**********************OUTPUT***********************
;[a@localhost ~]$ nasm -f elf32 sub.asm -o sub.o
;[a@localhost ~]$ ld -m elf_i386 -s -o sub sub.o
;[a@localhost ~]$ ./sub
;Processor is in protected mode
;GDT contents are1F384000007F
;LDT contents are0000
;IDT contents are81DF50000FFF
;[a@localhost ~]$
```

EXPERIMENT NO. 05

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. Nandini Babbar

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 05

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:

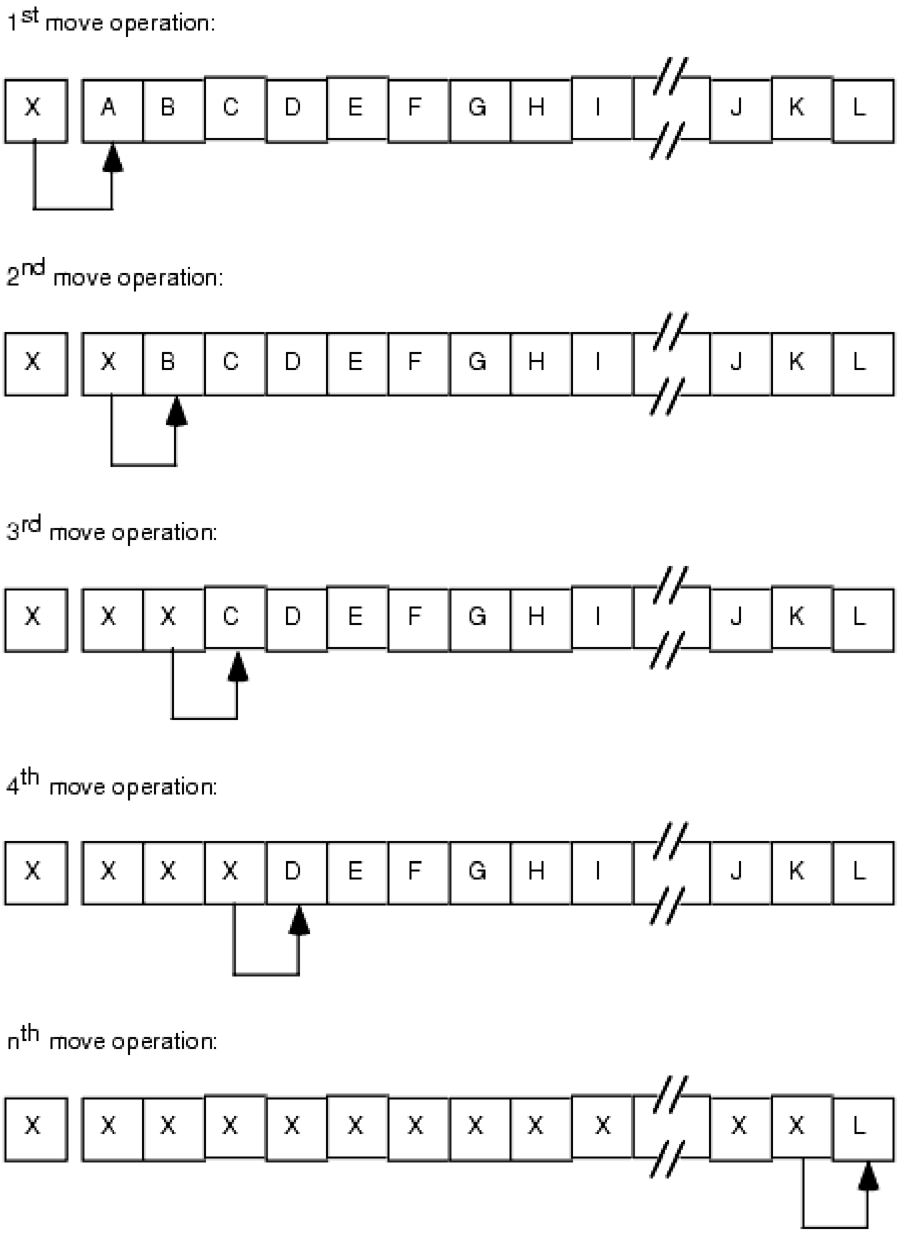
- Operating System: 64-bit Open source Linux or its derivative.
- Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
- Text Editor: geditor

THEORY:

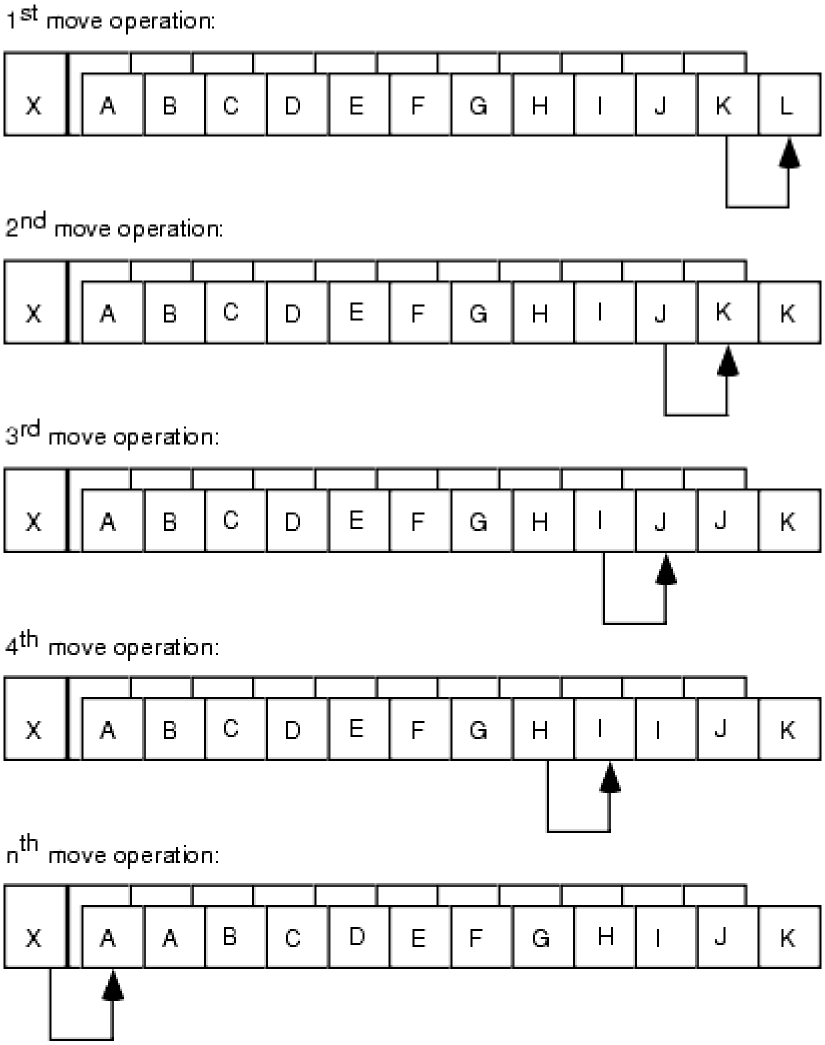
All members of the 80x86family support five different string instructions: movs, cmps, scas, lods, and stos. They are the string primitives since you can build most other string operations from these five instructions. How you use these five instructions is the topic of the next several sections This 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:
X A B C D E F G H I J K L
2nd move operation:
X X B C D E F G H I J K L
3rd move operation:
X X X C D E F G H I J K L
4th move operation:
X X X X D E F G H I J K L
nth move operation:
X X X X X X X X X X X X L

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:

(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.

PROGRAM:

```
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****
        ;[root@comppl208 nasm-2.10.07]# gedit nonoverlap26.asm
        ;[root@comppl208 nasm-2.10.07]# nasm -f elf64 nonoverlap26.asm
        ;[root@comppl208 nasm-2.10.07]# ld -o nonoverlap26 nonoverlap26.o
        ;[root@comppl208 nasm-2.10.07]# ./nonoverlap26

        ;**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@comppl208 nasm-2.10.07]# ./nonoverlap26

        ;**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   2

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

;[root@comppl208 nasm-2.10.07]# ./nonoverlap26

;**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   3
;;[;root@comppl208 nasm-2.10.07]#
```

EXPERIMENT NO. 06

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. Nandini Babbar

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 06

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

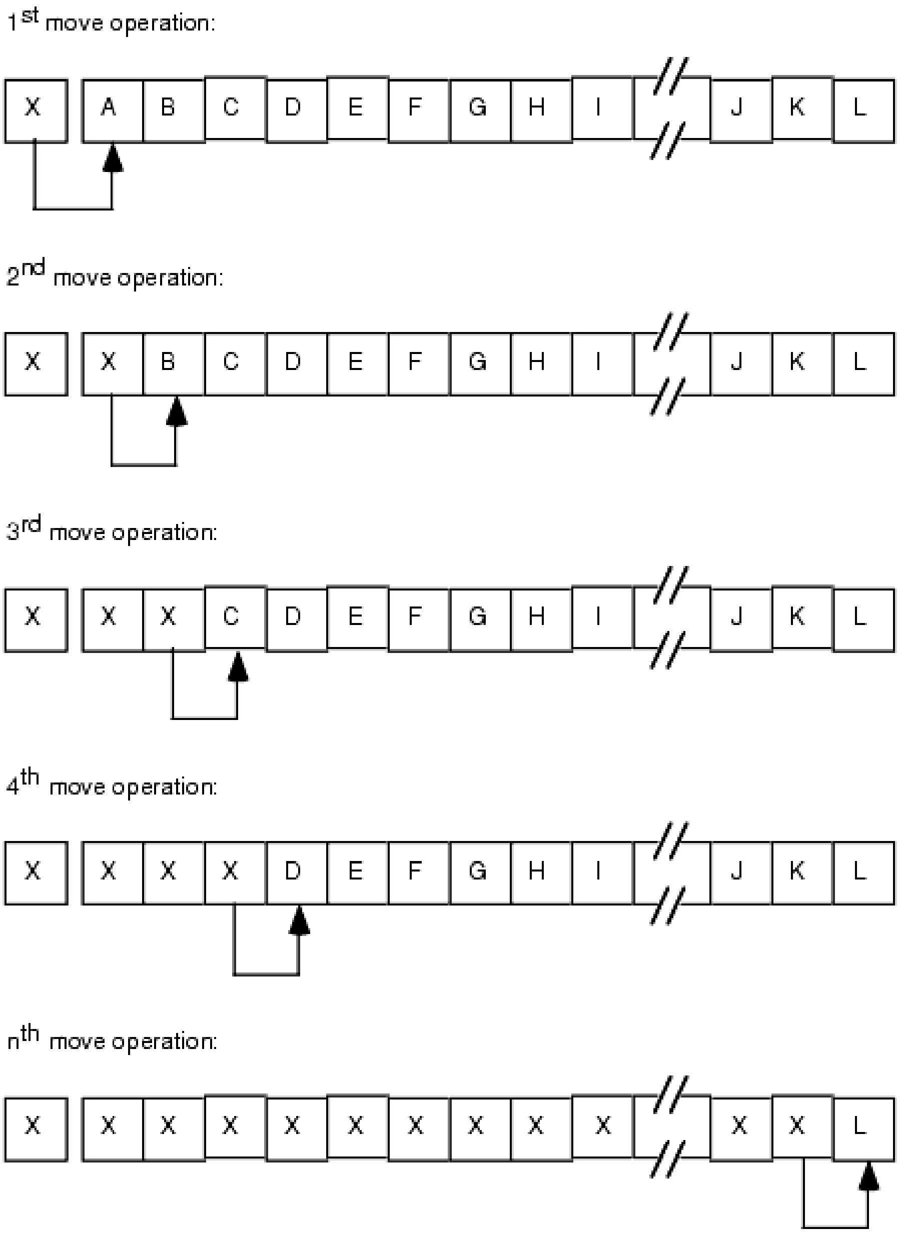
ENVIRONMENT:

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

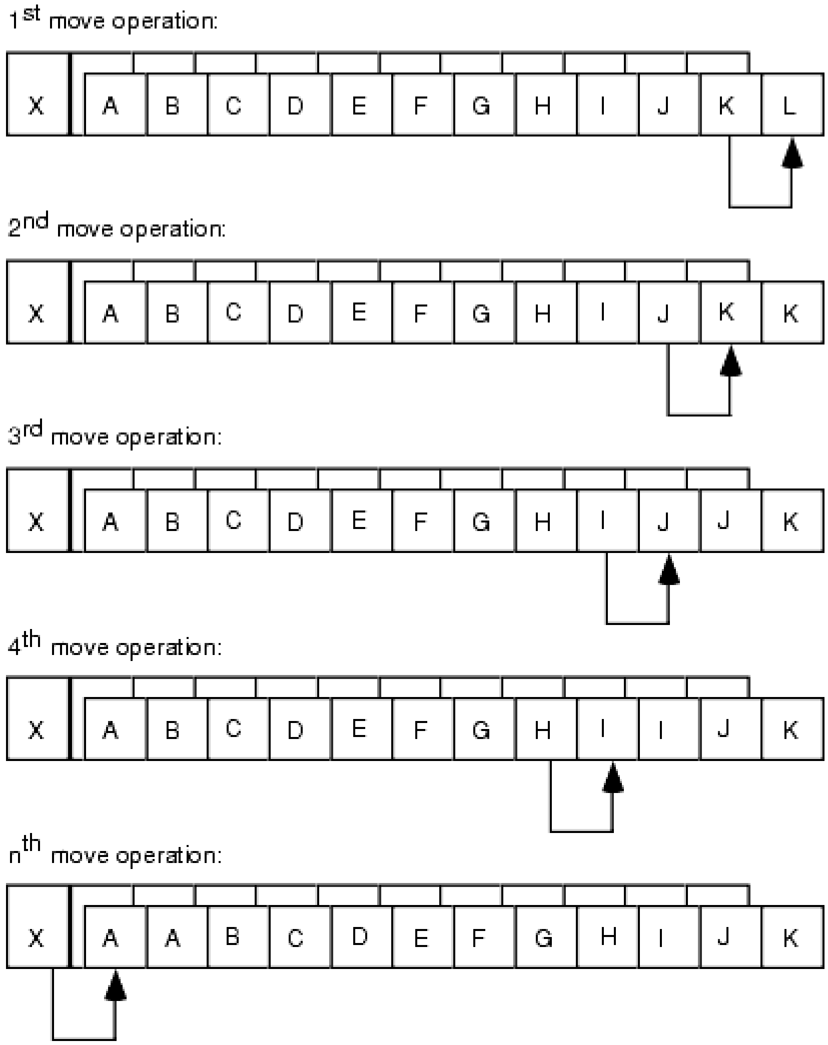
THEORY:

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 [1] 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:
X A B C D E F G H I J K L
2nd move operation:
X X B C D E F G H I J K L
3rd move operation:
X X X C D E F G H I J K L
4th move operation:
X X X X D E F G H I J K L
nth move operation:
X X X X X X X X X X X X L

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.


1st move operation:
X A B C D E F G H I J K L
2nd move operation:
X A B C D E F G H I J K K
3rd move operation:
X A B C D E F G H I J J K
4th move operation:
X A B C D E F G H I I J K
nth move operation:
X A A B C D E F G H I J K

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:

(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.

PROGRAM:

```
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
        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. 7

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. Nandini Babbar

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 7

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

Assembler 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 10,10,'***Multiplication by successive addition***'
        msg1_len equ $-msg1
        msg2 db 10,10,'Enter two digit number: '
        msg2_len equ $-msg2
        msg3 db 10,10,'Multiplication is: '
        msg3_len equ $-msg3

section .bss
        numascii resb 03
        multi1 resb 02
        resl resb 02
        resh resb 01
        dispbuff resb 04

%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 msg1,msg1_len
        dispmsg msg2,msg2_len
        accept numascii,03
        call packnum
        mov [multi1],bl
        dispmsg msg2,msg2_len
        accept numascii,03
        call packnum
        mov ecx,00h
        mov eax,[multi1]
        add1:
                add ecx,eax
                dec bl
                jnz add1                        ;checks bl is 0 or not
                mov [resl],ecx

                dispmsg msg3,msg3_len
                mov ebx,[resl]
                call disp16
                mov eax,01
                mov ebx,00
                int 80h

        packnum:
                mov bl,0
                mov ecx,02
                mov esi,numascii
                up1:
                        rol bl,04
                        mov al,[esi]
                        cmp al,39h
                        jbe skip1
                        sub al,07h
```

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

    disp16:
        mov ecx,4
        mov edi,dispbuff
        dub1:
            rol bx,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,4
    ret


;****OUTPUT****
;[root@comppl208 nasm-2.10.07]# nasm -f elf64 muladd26.asm
;[root@comppl208 nasm-2.10.07]# ld -o muladd26 muladd26.o
;[root@comppl208 nasm-2.10.07]# ./muladd26


;***Multiplication by successive addition***

;Enter two digit number: 05

;Enter two digit number: 20

;Multiplication is: 00A0

;[root@comppl208 nasm-2.10.07]# nasm -f elf64 muladd26.asm
;[root@comppl208 nasm-2.10.07]# ld -o muladd26 muladd26.o
;[root@comppl208 nasm-2.10.07]# ./muladd26

;***Multiplication by successive addition***

;Enter two digit number: 10

;Enter two digit number: 05
```

```
;Multiplication is: 0050

;[root@comppl208 nasm-2.10.07]#




;*-*-Multiplication by add & shift-*-*


section .data
        msg1 db 10,10,'***Multiplication by add & shift***'
        msg1_len equ $-msg1
        msg2 db 10,'Enter two digit number: '
        msg2_len equ $-msg2
        msg3 db 10,'Multiplication is: '
        msg3_len equ $-msg3

section .bss
        numascii resb 03
        multi1 resb 02
        multi2 resb 02
        resl resb 02
        dispbuff resb 04

%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 msg1,msg1_len
        dispmsg msg2,msg2_len
        accept numascii,03
        call packnum
        mov [multi1],bl
        dispmsg msg2,msg2_len
        accept numascii,03
```

```
call packnum
mov [multi2],bl
mov al,[multi1]
mov cl,00
mov edx,00
mov edx,08

add1:
        rcr al,01
        jnc next1
        mov bh,00h
        shl bx,cl                   ;shl=shift left
        add [resl],bx
        mov bl,[multi2]
        next1:
                inc cl
                dec edx
                jnz add1
                dispmsg msg3,msg3_len
                mov bx,[resl]
call disp16
mov eax,01
mov ebx,00
int 80h

packnum:
        mov bl,00
        mov ecx,02
        mov esi,numascii
        up1:
                rol bl,04
                mov al,[esi]
                cmp al,39h
                jbe skip1
                sub al,07h

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

disp16:
        mov ecx,4
        mov edi,dispbuff
        dub1:
                rol bx,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,4
        ret


;****OUTPUT****
;[root@comppl208 nasm-2.10.07]# nasm -f elf64 multi26.asm
;[root@comppl208 nasm-2.10.07]# ld -o multi26 multi26.o
;[root@comppl208 nasm-2.10.07]# ./multi26

;***Multiplication by add & shift***
;Enter two digit number: 50

;Enter two digit number: 02

;Multiplication is: 00A0
;[root@comppl208 nasm-2.10.07]# ./multi26

;***Multiplication by add & shift***
;Enter two digit number: 03

;Enter two digit number: 04

;Multiplication is: 000C
;[root@comppl208 nasm-2.10.07]#
```

EXPERIMENT NO. 8

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. Nandini Babbar

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 8

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:-

- 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:

```
%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 ;read,write,execute
            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       ;2bytes , 1 for choice 2nd for enter
        buffer resb 50      ; maximum 50bytes reserve
        name1 resb 15       ;name1 first file name
        name2 resb 15       ;second file name on which data is copied
        cmdlen resb 1
        filehandle1 resq 1
        filehandle2 resq 1

        abuf_len        resq    1                   ; actual buffer length
        dispnum resb 2

        buf resb4096                        ;maximum size of buffer
        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] ;if al=1
        cmp al,31h                  ;       al=31 in ascii
        jbe op1             ;equal, jump to op1
        cmp al,32h                  ;if 2nd choice
        jbe op2         ;jump to op2
        cmp al,33h          ;if 3rd choice, jump to op3
        jbe op3
```

```
    exit
    ret

op1:
        call tproc                           ;call procedure for type
        jmp again
;TYPE means same character one by one is wriiten in other file


op2:
        call cpproc                 ;call procedure for copy
        jmp again
;COPY the contents of file one to file 2

op3:
        call delproc                ;delete the contents from mail file
        jmp again



;type command procedure
tproc:
        cmn 1,1,msg,msglen
        cmn 0,0,buffer,50
        mov byte[cmdlen],al                  ;cmden=4 as character is of 4
        dec byte[cmdlen]                              ;after inserting one dec

        mov rsi,buffer                       ;1st element is moved in rsi =t
        mov al,[rsi]                         ;search for correct type command
        cmp al,'t'             ;compare ascii value of t with al
        jne skipt                                     ;jump if not equal
        inc rsi                              ; if match inc rsi =y
        dec byte[cmdlen]                              ;cmdlen=3
        jz skipt                             ;jump if zero
        mov al,[rsi]                         ;al=y
        cmp al,'y'                                    ;yes
        jne skipt                                     ;no
        inc rsi                              ;rsi=p
        dec byte[cmdlen]                              ;cmdlen=2
        jz skipt                             ;no
        mov al,[rsi]                         ;al=p
        cmp al,'p'                                    ;yes
        jne skipt                                     ;no
        inc rsi                              ;rsi=e
        dec byte[cmdlen]                              ;rsi=1
        jz skipt                             ;no
        mov al,[rsi]                         ;al=e
        cmp al,'e'                                    ;yes
        jne skipt                                     ;no
        inc rsi                              ;rsi= 0
        dec byte[cmdlen]                              ;cmdlen=0
        jnz correctt                         ;jump 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                  ;for displaimg content on screen

        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
        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
```

EXPERIMENT NO. 9

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. Nandini Babbar

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 9

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:

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‟. Ifgreater, goto step (5).
5. Compare the character with „5AH‟. If below or equal, „increment the count for no. of capitalletters‟ & 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. ofcharacters‟.
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.

PROGRAM:

```
;Assignment no.
;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
;-------------------------------------------------------------------------


;---------------------------------------------------------------
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
;--------------------------------------------------------------
```

EXPERIMENT NO. 10

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: Prof. Nandini Babbar

Marks/Grade Obtained: /10

Remark:

Signature of faculty

EXP NO: 10

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

OBJECTIVES:

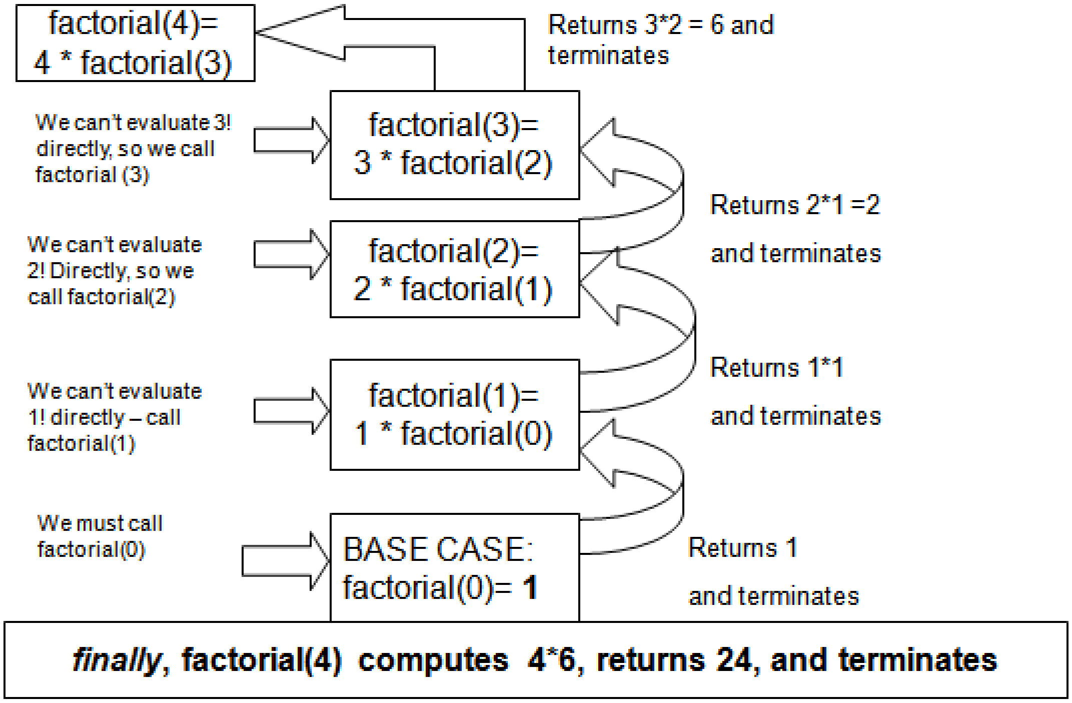
- To understand assembly language programming instruction set.
- To understand different assembler directives with example.
- To apply instruction set for implementing X86/64 bit assembly language programs

ENVIRONMENT:

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

Theory:-

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



factorial(4)=
4 * factorial(3)
Returns 3*2 = 6 and terminates
We can't evaluate 3! directly, so we call factorial (3)
factorial(3)=
3 * factorial(2)
Returns 2*1 =2
and terminates
We can't evaluate 2! Directly, so we call factorial(2)
factorial(2)=
2 * factorial(1)
Returns 1*1
and terminates
We can't evaluate 1! directly – call factorial(1)
factorial(1)=
1 * factorial(0)
We must call factorial(0)
BASE CASE:
factorial(0)= **1**
Returns 1
and terminates
***finally*, factorial(4) computes 4*6, returns 24, and terminates**

Algorithm
1.Accept
Number from
User2.Call
Factorial
Procedure
3.Define Recursive
Factorial Procedure4.Disply
Result.

PROGRAM:

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


%macro exitprog 0
mov rax, 60
mov edi,0
syscall
%endmacro

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

section .data
nwline db 10
m0 db 10,10,"Program to calculate factorial of a given number.",10,10
l0 equ $-m0

m2 db 10,"Enter Number (2 digit HEX no) : "
l2 equ $-m2

m4 db 10,"The factorial is : "
l4 equ $-m4
factorial  dq 1
```

```
section .bss
no1 resq 1
input resb 1
output resb 1


section .text
global _start
_start :

dispmsg m0,l0


dispmsg m2,l2    ; Display message
call getnum

mov [no1],rax     ; Accept number
gtch input ; To read and discard ENTER key pressed.

mov rcx,[no1]

call facto
mov rax,00



dispmsg m4,l4
mov rax,qword[factorial]

call disphx16       ; displays a 8 digit hex number  in rax


exitprog



facto:
push rcx
cmp rcx,01
jne ahead
jmp exit2

ahead:dec rcx
call facto
exit2:pop rcx
mov rax,rcx
mul qword[factorial]
mov qword[factorial],rax
ret
```

```
;; Procedure to get a 2 digit hex no from user; number returned in rax
getnum:
mov cx,0204h
mov rbx,0
ll2:
push rcx            ; syscall destroys rcx. Rest all regs are preserved
gtch input
pop rcx
mov rax,0
mov al,byte[input]
sub rax,30h
cmp rax,09h
jbe skip1
sub rax,7
skip1:
shl rbx,cl
add rbx,rax
dec ch
jnz ll2
mov rax,rbx
ret


disphx16:   ; displays a 16 digit hex number passed in rax
mov rbx,rax
mov cx,1004h                 ;16 digits to display and 04 count to rotate
ll6:
rol rbx,cl
mov rdx,rbx
and rdx,0fh
add rdx,30h
cmp rdx,039h
jbe skip4
add rdx,7
skip4:
mov byte[output],dl
push rcx
dispmsg output,1
pop rcx
dec ch
jnz ll6
ret
```