ACA – R - 81	**Lab Manual**	Academic Year: 2020- 21
Rev : 00		Semester: II
Date:01.07.2014		

Ref: SPVP/SBPCOE/COMP/2020-21/F23 Date: 05/08/2020

Microprocessor Laboratory
(ML) (210257)
(SE - 2019 Course)

Shahajirao Patil Vikas Pratishthan's
S.B. PATIL COLLEGE OF ENGINEERING, INDAPUR
(Approved by AICTE New Delhi, and Affiliated to Pune University)
Gat No. 58, Village-Vangali, Pune –Sholapur Highway, Tal-Indapur, Dist- Pune 413106
Phone-7410002240, 7410002239 Fax- (020-25667777),Email- sbpcoeprincipal@gmail.com,
Website-www.spvp.edu.in

CERTIFICATE

This is to certify that, Mr/ Miss_______________________________________

Roll No: ______of ***Second*** Semester of ~~FE~~/**SE**/~~TE/BE~~ in ***Computer Engineering*** has completed the term work satisfactorily in ***Microprocessor Lab (210257)*** for the Academic Year ***2020-2021*** as prescribed in the curriculum of Savitribai Phule Pune University, Pune.

Place: ***Indapur***
Date: ____________ **Exam Seat No:** _________________

Subject Teacher **Head of Department** **Principal**

INDEX

Sr. No.	List of Experiment in Index	Page No	Date	Remark
1	Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers.			
2	Write an X86/64 ALP to accept a string and to display its length.			
3	Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.			
4	Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,*, /) using suitable macros. Define procedure for each operation.			
5	Write an X86/64 ALP to count number of positive and negative numbers from the array.			
6	Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5- digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for: (a) HEX to BCD b) BCD to HEX (c) EXIT. Display proper strings to prompt the user while accepting the input and displaying the result. (Wherever necessary, use 64-bit registers).			
7	Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction.			
8	Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.			
9	Write X86/64 ALP to perform overlapped block transfer with string specific instructions Block containing data can be defined in the data segment.			
10	Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (Use of 64-bit registers is expected).			
11	Write X86 Assembly Language Program (ALP) to implement following OS commands i) COPY, ii) TYPE Using file operations. User is supposed to provide			

	command line arguments			
12	Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program_1 execution and write FAR PROCEDURES in Program_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.			
13	Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.			
14	Write an X86/64 ALP password program that operates as follows: a. Do not display what is actually typed instead display asterisk (“*”). If the password is correct display, “access is granted” else display “Access not Granted”			
15	**Study Assignment:** Motherboards are complex. Break them down, component by component, and Understand how they work. Choosing a motherboard is a hugely important part of building a PC. Study- Block diagram, Processor Socket, Expansion Slots, SATA, RAM, Form Factor, BIOS, Internal Connectors, External Ports, Peripherals and Data Transfer, Display, Audio, https://www.intel.in/content/www/in/en/support/articles/000006014/boards-and- kits/desktop-boards.html			

Assignment No: 1

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

Introduction to Assembly Language Programming:

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

Advantages of Assembly Language

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

Other advantages of using assembly language are:

- It requires less memory and execution time;
- It allows hardware-specific complex jobs in an easier way;
- It is suitable for time-critical jobs;

ALP Step By Step:

Installing NASM:

If you select "Development Tools" while installed Linux, you may NASM installed along with the Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps:

- Open a Linux terminal.
- Type ***whereis nasm*** and press ENTER.
- If it is already installed then a line like, *nasm: /usr/bin/nasm* appears. Otherwise, you will see just*nasm:*, then you need to install NASM.

To install NASM take the following steps:

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

Assembly Basic Syntax:
An assembly program can be divided into three sections:

- The **data** section
- The **bss** section
- The **text** section

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

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

The .bss Section
Not all data items need to have values before the program begins running. When you're reading data from a disk file, for example, you need to have a place for the data to go after it comes in from disk. Data buffers like that are defined in the .bss section of your program. You set aside some number of bytes for a buffer and give the buffer a name, but you don't say what values are to be present in the buffer. There's a crucial difference between data items defined in the .data section and data items defined in the .bss section: data items in the .data section add to the size of your executable file. Data items in the .bss section do not.

The .text Section

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

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

Assembly Language Statements

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

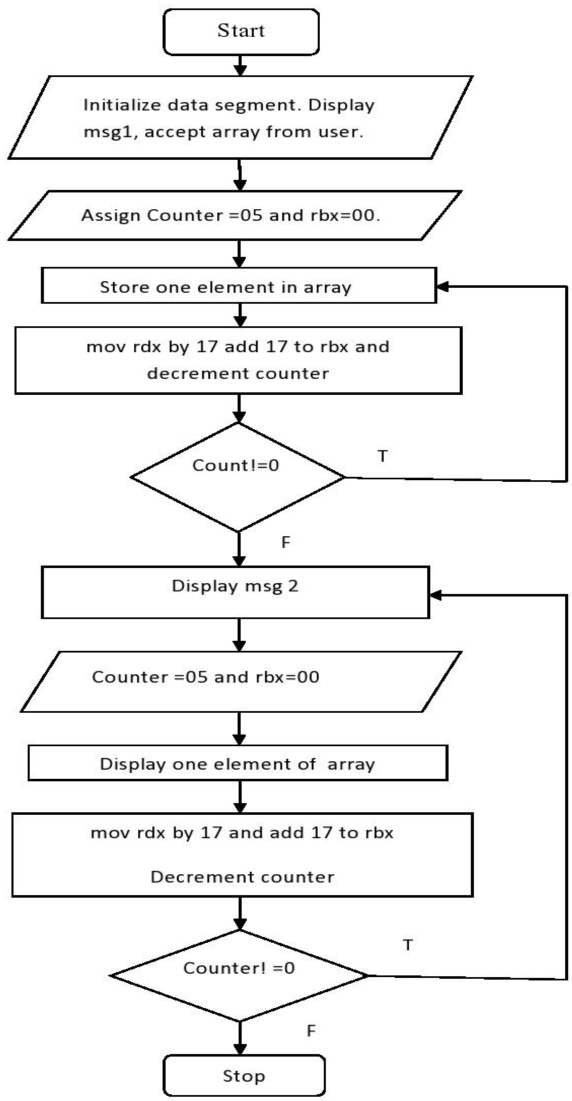
Syntax of Assembly Language Statements

[label]   mnemonic   [operands]   [;comment]

LIST OF INTERRRUPTS USED: NA
LIST OF ASSEMBLER DIRECTIVES USED: EQU, DB
LIST OF MACROS USED: NA
LIST OF PROCEDURES USED: NA
ALGORITHM:
INPUT: ARRAY
OUTPUT: ARRAY
STEP 1: Start.
STEP 2: Initialize the data segment.
STEP 3: Display msg1 “Accept array from user. “
STEP 4: Initialize counter to 05 and rbx as 00
STEP 5: Store element in array.
STEP 6: Move rdx by 17.
STEP 7: Add 17 to rbx.

STEP 8: Decrement Counter.
STEP 9: Jump to step 5 until counter value is not zero.
STEP 9: Display msg2.
STEP 10: Initialize counter to 05 and rbx as 00
STEP 11: Display element of array.
STEP 12: Move rdx by 17.
STEP 13: Add 17 to rbx.
STEP 14: Decrement Counter.
STEP 15: Jump to step 11 until counter value is not zero.
STEP 16: Stop

FLOWCHART:



PROGRAM:

```
section .data
        msg1 db 10,13,"Enter 5 64 bit numbers"
        len1 equ $-msg1
        msg2 db 10,13,"Entered 5 64 bit numbers"
        len2 equ $-msg2
section .bss
        array resd 200
        counter resb 1
section .text
        global _start
        _start:
;display
        mov Rax,1
        mov Rdi,1
        mov Rsi,msg1
        mov Rdx,len1
        syscall
;accept
mov byte[counter],05
mov rbx,00
                loop1:
                        mov rax,0               ; 0 for read
                        mov rdi,0               ; 0 for keyboard
                        mov rsi, array          ;move pointer to start of array
                        add rsi,rbx
                        mov rdx,17
                        syscall
                add rbx,17                      ;to move counter
                        dec byte[counter]
                        JNZ loop1
;display
        mov Rax,1
        mov Rdi,1
        mov Rsi,msg2
        mov Rdx,len2
        syscall
;display
mov byte[counter],05
mov rbx,00
                loop2:
                        mov rax,1               ;1 for write
                        mov rdi, 1              ;1 for monitor
```

```
            mov rsi, array
            add rsi,rbx
            mov rdx,17          ;16 bit +1 for enter
            syscall
            add rbx,17
            dec byte[counter]
            JNZ loop2
        ;exit system call
        mov rax ,60
        mov rdi,0
        syscall
```

;output

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

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

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

CONCLUSION:

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

Assignment No: 2

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

OBJECTIVES:

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

ENVIRONMENT:

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

THEORY:

MACRO:

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

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

The Syntax for macro definition −

```
%macro macro_name  number_of_params
<macro body>
%endmacro
```

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

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

PROCEDURE:

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

Syntax

Following is the syntax to define a procedure −

```
proc_name:
```

```
procedure body
...
ret
```

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

CALL proc_name

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

LIST OF INTERRRUPTS USED: NA
LIST OF ASSEMBLER DIRECTIVES USED: EQU, PROC, GLOBAL, DB,
LIST OF MACROS USED: DISPMSG
LIST OF PROCEDURES USED: DISPLAY
ALGORITHM:
INPUT: String
OUTPUT: Length of String in hex
STEP 1: Start.
STEP 2: Initialize data section.
STEP 3: Display msg1 on monitor
STEP 4: accept string from user and store it in Rsi Register (Its length gets stored in Rax register by default).
STEP 5: Display the result using "display" procedure. Load length of string in data register.
STEP 6: Take counter as 16 int cnt variable
STEP 7: move address of "result" variable into rdi.
STEP 8: Rotate left rbx register by 4 bit.
STEP 9: Move bl into al.
STEP 10: And al with 0fh
STEP 11: Compare al with 09h
STEP 12: If greater add 37h into al
STEP 13: else add 30h into al
STEP 14: Move al into memory location pointed by rdi
STEP 14: Increment rdi
STEP 15: Loop the statement till counter value becomes zero
STEP 16: Call macro dispmsg and pass result variable and length to it. It will print length of string.
STEP 17: Return from procedure
STEP 18: Stop
FLOWCHART:
PROGRAM:

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

section .bss
```

```
        str1 resb 200           ;string declaration
        result resb 16

section .text

global _start
        _start:

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

;store string

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

call display

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


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



display:
        mov rbx,rax             ; store no in rbx
        mov rdi,result          ;point rdi to result variable
```

```
    mov cx,16               ;load count of rotation in cl

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

OUTPUT:

```
sbpcoe
Enter a string:0000000000000006
[Program exited with exit code 0]
```

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

Assignment No: 3

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

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

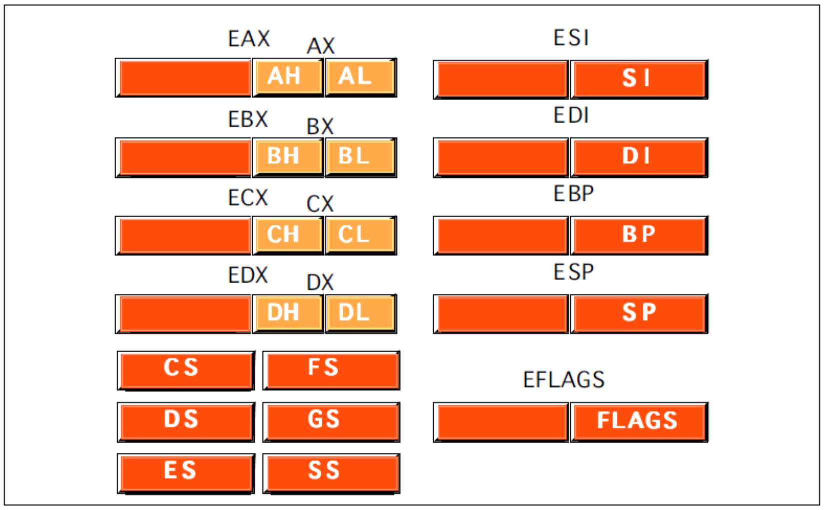
ENVIRONMENT:

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

THEORY:
Datatype in 80386:
The 80386 supports the following data types they are

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

Registers in 80386:



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

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

Size of operands in an Intel assembler instruction

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

Addressing modes in 80386:
The purpose of using addressing modes is as follows:

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

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

ALGORITHM:
FLOWCHART:

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

section .bss
        cnt resb 1
        result resb 16

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

section .text
        global _start
        _start:
                mov byte[cnt],5
                mov rsi,array
                mov al,0
        LP: cmp al,[rsi]
                jg skip
                xchg al ,[rsi]
```

```
        skip: inc rsi
        dec byte[cnt]
        jnz LP


    ;display al

    call display

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


  dispmsg result,16      ;call to macro

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



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

        up1:
            rol rbx,04          ;rotate no of left by four bits
            mov al,bl               ; move lower byte in dl
            and al,0fh          ;get only LSB
            cmp al,09h            ;compare with 39h
            jg add_37            ;if greater than 39h skip add 37
            add al,30h
            jmp skip1             ;else add 30
        add_37:
            add al,37h
        skip1:
            mov [rdi],al          ;store ascii code in result variable
            inc rdi             ; point to next byte
            dec cx              ; decrement counter
            jnz up1             ; if not zero jump to repeat

    ret
```

Output:

Largest no in an array is:0000000000000055
[Program exited with exit code 0]

Assignment No: 4

Title: Write a switch case driven X86/64 ALP to perform 64-bit arithmetic operation (+, -, *, /) using suitable macros.

Program:

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

section .data
	arr dq 000000000000003h,0000000000000002h
	n equ 2
	menu db 10d,13d,"**********MENU**********"
	    db 10d,13d,"1. Addition"
	    db 10d,13d,"2. Subtraction"
	    db 10d,13d,"3. Multiplication"
	    db 10d,13d,"4. Division"
	    db 10d,13d,"5. Exit"
	    db 10d,13d,"Enter your Choice: "
	menu_len equ $-menu

	m1 db 10d,13d,"Addition: "
	l1 equ $-m1
	m2 db 10d,13d,"Substraction: "
	l2 equ $-m2
	m3 db 10d,13d,"Multiplication: "
	l3 equ $-m3
	m4 db 10d,13d,"Division: "
	l4 equ $-m4

section .bss
	answer resb 16		;to store the result of operation
	choice resb 2

section .text
	global _start:
	_start:
```

```
up:     scall 1,1,menu,menu_len
        scall 0,0,choice,2

cmp byte[choice],'1'
je case1
cmp byte[choice],'2'
je case2
cmp byte[choice],'3'
je case3
cmp byte[choice],'4'
je case4
cmp byte[choice],'5'
je case5



case1:  scall 1,1,m1,l1
      call addition
      jmp up

case2:  scall 1,1,m2,l2
        call substraction
        jmp up

case3:  scall 1,1,m3,l3
        call multiplication
        jmp up

case4:  scall 1,1,m4,l4
        call division
        jmp up

case5:  mov rax,60
        mov rdi,0
      syscall



;procedures for arithmetic and logical operations
addition:
    mov rcx,n
    dec rcx
    mov rsi,arr
    mov rax,[rsi]
```

```
up1:    add rsi,8
        mov rbx,[rsi]
        add rax,rbx
        loop up1
        call display
ret


substraction:

        mov rcx,n
        dec rcx
        mov rsi,arr
        mov rax,[rsi]
up2:    add rsi,8
        mov rbx,[rsi]
        sub rax,rbx
        loop up2
        call display
ret


multiplication:
        mov rcx,n
        dec rcx
        mov rsi,arr
        mov rax,[rsi]
up3:    add rsi,8
        mov rbx,[rsi]
        mul rbx
        loop up3
        call display
ret

division:
        mov rcx,n
        dec rcx

        mov rsi,arr
        mov rax,[rsi]
up4:    add rsi,8
        mov rbx,[rsi]
        mov rdx,0
        div rbx
```

```
        loop up4
        call display
ret

or:
        mov rcx,n
        dec rcx
        mov rsi,arr
        mov rax,[rsi]
up6:    add rsi,8
        mov rbx,[rsi]
        or rax,rbx
        loop up6
        call display
ret

xor:
        mov rcx,n
        dec rcx
        mov rsi,arr
        mov rax,[rsi]
up7:    add rsi,8
        mov rbx,[rsi]
        xor rax,rbx
        loop up7
        call display
ret

and:
        mov rcx,n
        dec rcx
        mov rsi,arr
        mov rax,[rsi]
up8:    add rsi,8
        mov rbx,[rsi]
        and rax,rbx
        loop up8
        call display
ret


display:
        mov rsi,answer+15
        mov rcx,16
```

```
cnt:   mov rdx,0
       mov rbx,16
       div rbx
       cmp dl,09h
       jbe add30
       add dl,07h
add30:  add dl,30h
       mov [rsi],dl
       dec rsi
       dec rcx
       jnz cnt
       scall 1,1,answer,16
ret
```

Output:

**********MENU**********

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exit

Enter your Choice: 1

Addition: 0000000000000005

**********MENU**********

1. Addition

2. Subtraction

3. Multiplication

4. Division

5. Exit

Enter your Choice: 2

Substraction: 0000000000000001

**********MENU***********

1. Addition

2. Subtraction

3. Multiplication

4. Division

5. Exit

Enter your Choice: 3

Multiplication: 0000000000000006

Assignment No: 5

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

OBJECTIVES:

1. To be familiar with the format of assembly language program structure and instructions.
2. To study the format of assembly language program along with different assembler directives and different functions of the NASM.
3. To learn the procedure how to store N hexadecimal number in memory.

PROBLEM DEFINITION:

Write X86/64 Assembly language program (ALP) to count number of positive and negative numbers from array.

S/W AND H/W REQUIREMENT:

Software Requirements

1. CPU: Intel I5 Processor
2. OS:- Linux Ubuntu 14 (64 bit Execution)
3. Editor: gedit, GNU Editor
4. Assembler: NASM (Netwide Assembler)
5. Linker:-LD, GNU Linker

INPUT: Hexadecimal number.

OUTPUT: Number of Negative numbers, Number of Positive numbers.

THEORY:

Assembly language Program is mnemonic representation of machine code. Three assemblers available for assembling the programs for IBM-PC are:

1. Microsoft Micro Assembler (MASM)
2. Borland Turbo Assembler (TASM)
3. Net wide Assembler (NASM)

Assembly Basic Syntax

An assembly program can be divided into three sections:

1. The **data** section
2. The **bss** section
3. The **text** section

The *data* Section

The **data** section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names or buffer size etc. in this section.

The syntax for declaring data section is:

section.data

The bss Section

The bss section is used for declaring uninitialized data or variables. The syntax for declaring bss section is:
section .bss

The text section

The text section is used for writing the actual code. This section must begin with the declaration global _start, which tells the kernel where the program execution begins.

The syntax for declaring text section is:

section .text
global _start
_start:

Assembly Language Statements

Assembly language programs consist of three types of statements:

1. Executable instructions or instructions
2. Assembler directives or pseudo-ops
3. Macros

The **executable instructions** or simply **instructions** tell the processor what to do. Each instruction consists of an **operation code** (opcode). Each executable instruction generates one machine language instruction.

The **assembler directives** or **pseudo-ops** tell the assembler about the various aspects of the assembly process.

These are non-executable and do not generate machine language instructions.

Macros are basically a text substitution mechanism.

Assembly System Calls

System calls are APIs for the interface between user space and kernel space. We are using the system calls sys_write and sys_exit for writing into the screen and exiting from the program respectively.

Linux System Calls (32 bit)

You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program:

- Put the system call number in the EAX register.

- Store the arguments to the system call in the registers EBX, ECX, etc.
- Call the relevant interrupt (80h)
- The result is usually returned in the EAX register

There are six registers that stores the arguments of the system call used. These are the EBX, ECX, EDX, ESI, EDI, and EBP. These registers take the consecutive arguments, starting with the EBX register. If there are more than six arguments then the memory location of the first argument is stored in the EBX register.

The following code snippet shows the use of the system call sys_exit:

```
MOV EAX, 1          ; system call number (sys_exit)
INT 0x80            ; call kernel
```

The following code snippet shows the use of the system call sys_write:

```
MOV EAX,4; system call number (sys_write)
MOV EBX,1; file descriptor (stdout)
MOV ECX, MSG   ; message to write
MOV EDX, 4; message length
INT0x80; call kernel
```

Linux System Calls (64 bit)

Sys_write:

```
MOV RAX,1
MOV RDI,1
MOV RSI,message
MOV RDX,message_length
SYSCALL
```

Sys_read:

```
MOV RAX,0
MOV RDI,0
MOV RSI,array_name
MOV RDX,array_size
SYSCALL
```

Sys_exit:

```
MOV RAX,60
MOV RDI,0
SYSCALL
```

Assembly Variables

NASM provides various **define directives** for reserving storage space for variables. The define Assembler directive is used for allocation of storage space. It can be used to reserve as well as initialize one or more bytes.

Allocating Storage Space for Initialized Data

There are five basic forms of the define directive:

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

Allocating Storage Space for Uninitialized Data

The reserve directives are used for reserving space for uninitialized data. The reserve directives take a single operand that specifies the number of units of space to be reserved. Each define directive has a related reserve directive.

There are five basic forms of the reserve directive:

Directive	Purpose
RESB	Reserve a Byte
RESW	Reserve a Word
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Instructions needed:

1. **MOV**-Copies byte or word from specified source to specified destination
2. **ROR**-Rotates bits of byte or word right, LSB to MSB and to CF
3. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word

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

MATHEMATICAL MODEL:

Let S = { s , e , X, Y, Fme, mem │ Φs **} be the programmer's perspective of Array Addition Where**

S = System
s =Distinct Start of System
e = Distinct End Of System
X = Set of Inputs
Y= Set Of outputs
Fme = Central Function
Mem= Memory Required
Φs = Constraits

Let X be the input such that
X= { X1, X2,X3,-------}
Such that there exists function fx_1 : X1 ⟶ {0,1}

X2 Source Array
Let X2 = {{b7-----b0}{ b7-------b0}--------{b7 b6 b5 b4 b3 b2 b1 b0}} where □ bi Є X1
There exists a function f_{X2} :X2 ⟶ { {00h----FFh}{00h----FFh }{00h----FFh}------}

X3 is the two digit count value
Let X3 = b7 b6 b5 b4 b3 b2 b1 b0 where □ bi Є X1
There exists a function $f_{X3:}$ X3 ⟶ { 01h,02h,-,- FFh}

Let Y is the Output
Let Y= b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
Where □bi Є X1
Y ⟶ { 0000h,0001h,------ FFFFh}

Let Fme = { F1 ,F2,F3 }
Where F1 = Accept
F2 = Count
F3 = Display

F1 ⟶ 1) Accept the number stored in array.

F2 ⟶ Y = ∑ (□ X2)

F3 ⟶ Display Output

ALGORITHM:

1 Start

2. Declare & initialize the variables in .data section.

3. Declare uninitialized variables in .bss section.

4. Declare Macros for print and exit operation.

5. Initialize pointer with source address of array.

6. Initialize count for number of elements.

7. Set RBX as Counter register for storing positive numbers count.

8. Set RCX as Counter register for storing negative numbers count.

9. Get the number in RAX register.

10. Rotate the contents of RAX register left 1 bit to check sign bit.

11. Check if MSB is 1. If yes, goto step 12,elsegoto step 13.

12. Increment count for counting negative numbers.

13. Increment count for counting positive numbers.

14. Increment pointer.

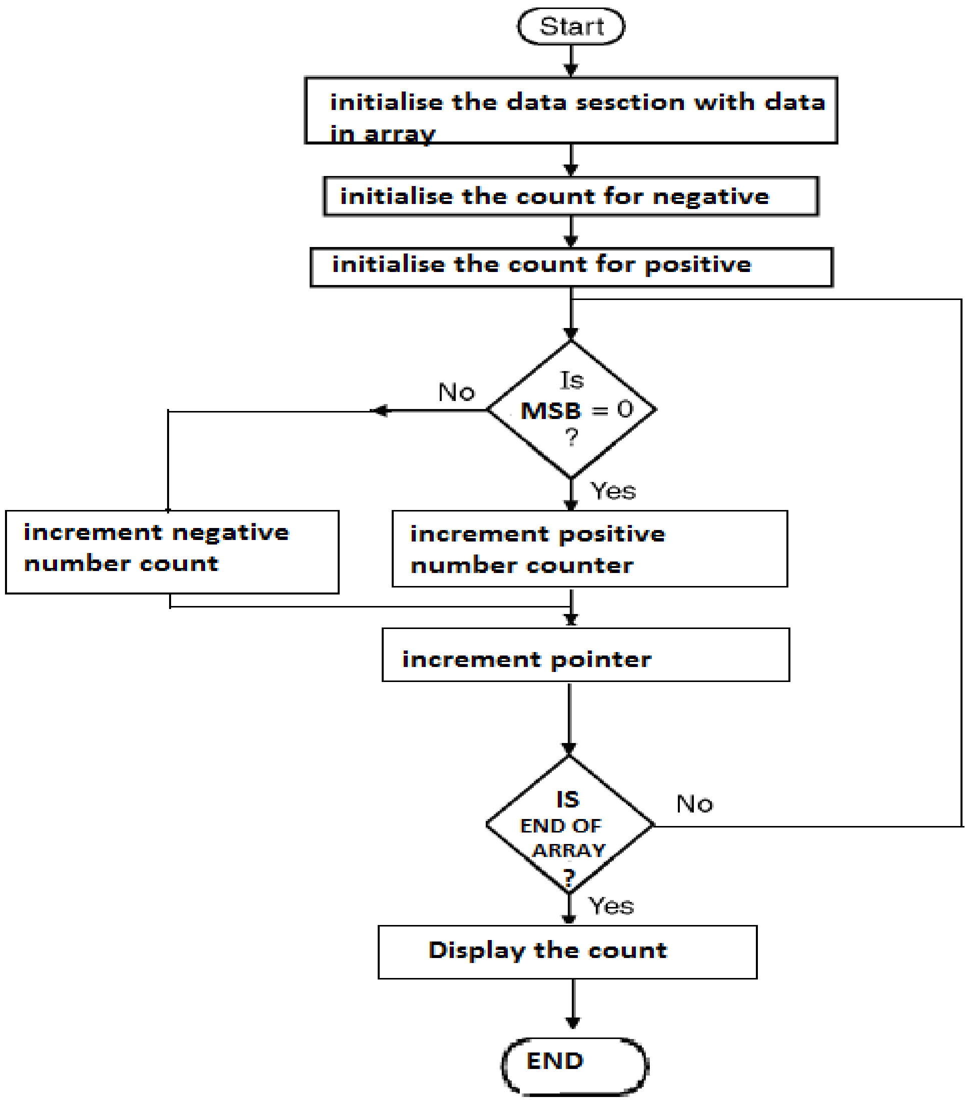
15. Decrement count

16. Check for count = 0. If yes, goto step 17 else goto step 9.

17. Store the positive numbers count and negative numbers count in buffer.

18. Display first message. Call the procedure to display the positive count.
19. Display second message. Call the procedure to display the negative count.
20. Add newline.
21. Terminate the process.
22. Declare the Procedure.
23. Stop.

Flowchart:



CONCLUSION

Here we count the 32-bit numbers of positive and negative numbers in the array in the assembly language.

Program:

```
section .data
    welmsg db 10,'Welcome to count +ve and -ve 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 9000h,8001h,9002h,4553h,8006h,9004h,022h
    arrcnt equ 7

    pcnt db 0
    ncnt db 0

section .bss
    dispbuff resb 2

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


section .text
    global _start
_start:
    print welmsg,welmsg_len

    mov rsi,array
    mov ecx,arrcnt
up1:
    bt word[rsi],15
    jnc pnxt
    inc byte[ncnt]
    jmp pskip

pnxt:   inc byte[pcnt]

pskip:  inc rsi
```

```
    inc rsi
    loop up1

    print pmsg,pmsg_len
    mov bl,[pcnt]
    call disp8num


    print nmsg,nmsg_len
    mov bl,[ncnt]
    call disp8num

    print nwline,1
exit:
    mov rax,60
    syscall

disp8num:
    mov rcx,2
    mov rdi,dispbuff
dup1:
    rol bl,4
    mov al,bl
    and al,0fh      ;Mask upper digit
    cmp al,09       ;Compare with 9
    jbe dskip       ;If number below or equal to 9 go to add only 30h
    add al,07h      ;Else first add 07h
dskip:  add al,30h      ;Add 30h
    mov [rdi],al        ;Store ASCII code in temp buff
    inc rdi         ;Increment 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 assg1.asm
;[root@comppl2022 ~]# ld -o assg1 assg1.o
;[root@comppl2022 ~]# ./ assg1
;Welcome to count +ve and -ve numbers in an array
;Count of +ve numbers::02
;Count of -ve numbers::05
;[root@comppl2022 ~]#
```

Assignment No. 6

TITLE: Hex to BCD & BCD to Hex Conversion.

OBJECTIVES:

To learn the implementation of ALP for conversion of Hex to BCD & vice a versa.

PROBLEM STATEMENT:

To write 64 bit 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.

SOFTWARE REQUIRED:

6. CPU: Intel I5 Processor
7. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
8. Editor: gedit, GNU Editor
9. Assembler: NASM (Netwide Assembler)
10. Linker:-LD, GNU Linker

INPUT:

1. Hexadecimal number
2. BCD Number

OUTPUT:

1. Conversion of hex to BCD number
2. Coversion of BCD to hex number

MATHEMATICAL MODEL:

Let S = { s , e , X, Y, Fme, mem | Φs } be the programmer's perspective of Hex to BCD & BCD to hex conversion **.**

Let X be the input such that
X= { X1, X2,X3,-------}

Such that there exists function fx_1 : X1 ⟶ {0,1}

X2 is the four Digit Hex Number.
Let X2 = b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 where
 bi Є X1. There exists a function f_{X2} : X2 ⟶ { 0000h,0001h,0002h,-,- FFFFh}

X3 is the Five digit BCD number.
Let X3 = b19 b18 b17 b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5b4 b3 b2 b1 b0
where bi Є X1
There exists a function $f_{X3:}$ X3 ⟶ { 00000,00001,-------99999}

X4 is the Single digit choice.
Let X4 = b3 b2 b1 b0 where bi Є X1
There exists a function $f_{X4:}$ X4 ⟶ { 1,2,3}

Let Y1 is the 5 Digit BCD Output
Let Y= b19 b18 b17 b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
Where bi Є X1
Y 1⟶{ 00000,00001,------ 99999}

Let Y2 is the 4 Digit Hex Output
Let Y= b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
Where bi Є X1
Y 1⟶{ 0000h,0001h,------ FFFFh}

Fme 1(Hex to bcd) = { F1 ,F2,F3 }
Where F1 = Accept X2
F2 = Repeat (X2/10) till quotient =0 & push Remainder on stack
F3 = Pop remainder from stack & display

Fme 2(bcd to hex) = { F1 ,F2,F3,F4 }
Where F1 = Accept BCD digit $X3_i$
F2 = Result= Result*10 + X3i & i=i-1 (Initially Result=0)
F3= repeat F1 onwards untill i=0
F4= Display Result

Fme 3(exit)= {F1}
Where F1= Call Sys_exit Call

Fme = { F1,F2,F3,F4}
Where F1= Accept X4
F2= If X4=1, Call Fme1
F3= If X4=2,Call Fme2
F4= If X4=3, Call Fme3

THEORY:

1. Hexadecimal to BCD conversion:

Conversion of a hexadecimal number can be carried out in different ways e.g. dividing number by 000Ah and displaying quotient in reverse way.

2. BCD to Hexadecimal number:

Conversion of BCD number to Hexadecimal number can be carried out by multiplying the BCD digit by its position value and the adding it in the final result.

Special instructions used:

DIV: Unsigned Divide. Result Quotient in AL and Remainder in AH for 8-bit division and for 16-bit division Quotient in AX and Remainder in DX

MUL: Unsigned Multiply. For 8-bit operand multiplication result will be stored in AX and for 16-bit multiplication result is stored in DX:AX

Commands

- To assemble

    nasm –f elf 64 hello.nasm -o hello.o

- To link

    ld –o hello hello.o

- To execute -

    ./hello

ALGORITHM:

1. Start
2. Initialize data section
3 Display the Menu Message.
4 Accept the choice from the user.

5 If choice=1 then call Hex to bcd procedure. If choice=2 then call Bcd to Hex Procedure If choice=3 then call exit procedure

6. Hex to BCD Procedure:

a) Accept 4 Digit Hexadecimal Number.

b) Number=Number/10 & Number=Quotient

c) Push the remainder on the stack

d) If Number=0, Go to next step otherwise go to step b

e) Pop remainder , Convert into ascii & display untill all remainder are popped out

8. Bcd to hex Procedure:

a) RESULT=0

b) Accept BCD Digit

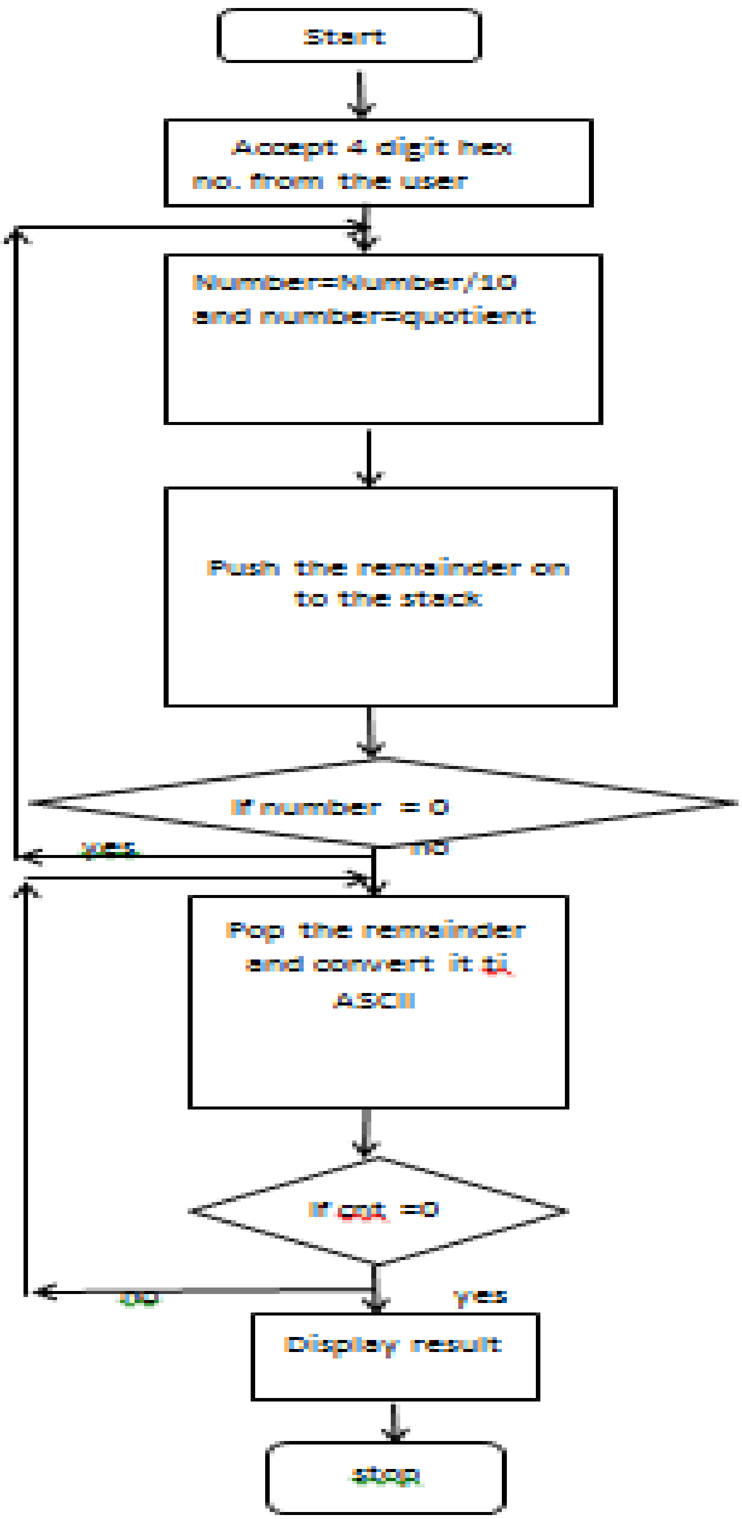
c) Check whether all BCD Digits are accepted. If YES then go to step e otherwise go to next step

d) RESULT= RESULT*10 + BCD Digit accepted

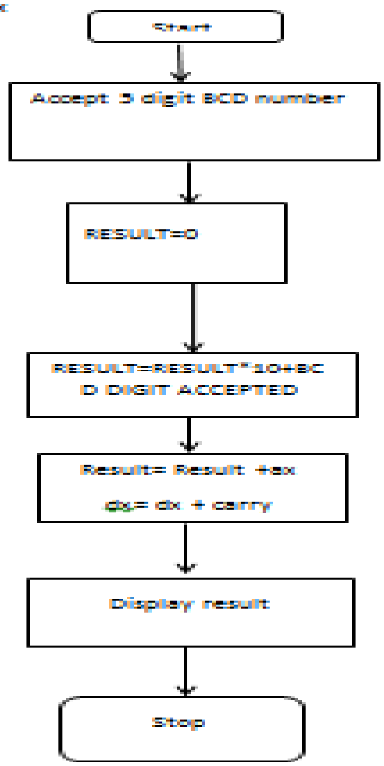
e) Display RESULT

FLOW CHART:

Flowchart of Hex to BCD



Flow chart for BCD to hex



CONCLUSION:

Hence we conclude that we can perform the Hex to BCD conversion & BCD to hex Conversion.

Program:

```
section .data
   nline       db  10,10
   nline_len:  equu $-nline



   menu    db      10,"-----------Menu----------"
         db  10,"1. Hex to BCD "
         db  10,"2. BCD to Hex"
         db  10,"3. Exit "
         db  10
         db  10,"Enter your choice: "
   menu_len:   equ     $-menu

   h2bmsg      db  10,"Hex to BCD "
               db     10,"Enter 4-digit Hex number: "
   h2bmsg_len: equ     $-h2bmsg

   b2hmsg      db  10,"BCD to Hex "
```

```
        db  10,"enter 5-digit BCD number: "
    b2hmsg_len: equ     $-b2hmsg

    hmsg        db  10,13,"Equivalent Hex number is: "
    hmsg_len:   equ     $-hmsg

    bmsg        db  10,13,"Equivalent BCD number is: "
    bmsg_len:   equ     $-bmsg

    emsg        db  10,"You entered Invalid Data!!!",10
    emsg_len:   equ     $-emsg

section .bss
    buf         resb    6
    buf_len:    equ $-buf

    digitcount  resb    1

    ans     resw    1
    char_ans    resb    4
;macros as per 64-bit convensions

%macro  print   2
    mov     rax,1       ; Function 1 - write
    mov     rdi,1       ; To stdout
    mov     rsi,%1      ; String address
    mov     rdx,%2      ; String size
    syscall         ; invoke operating system to WRITE
%endmacro

%macro  read   2
    mov     rax,0       ; Function 0 - Read
    mov     rdi,0       ; from stdin
    mov     rsi,%1      ; buffer address
    mov     rdx,%2      ; buffer size
    syscall         ; invoke operating system to READ
%endmacro

%macro  exit    0
    print   nline, nline_len
    mov rax, 60     ; system call 60 is exit
    xor rdi, rdi        ; we want return code 0
    syscall         ; invoke operating system to exit
%endmacro

section .text
    global _start
_start:
    print   menu, menu_len

    read    buf,2       ; choice + enter
    mov al,[buf]
```

```
c1: cmp     al,'1'
    jne     c2
    call    hex_bcd
    jmp     _start

c2: cmp     al,'2'
    jne     c3
    call    bcd_hex
    jmp     _start

c3: cmp     al,'3'
    jne     err
    exit

err:    print   emsg,emsg_len
    jmp _start

hex_bcd:
    print   h2bmsg, h2bmsg_len

    call    accept_16
    mov     ax,bx

    mov     rbx,10
back:
    xor     rdx,rdx
    div     rbx;rax contains quotient and rdx the remainder

    push    dx
    inc     byte[digitcount]

    cmp     rax,0h
    jne     back;repeat the loop until no remainder is left

    print   bmsg, bmsg_len
print_bcd:
    pop     dx
    add     dl,30h      ; possible digits are 0-9 so add 30H only
    mov [char_ans],dl   ; store character in char_ans

    print   char_ans,1  ; print on screen in reverse order

    dec     byte[digitcount]
    jnz     print_bcd

    ret

bcd_hex:
    print   b2hmsg, b2hmsg_len
    read    buf,buf_len     ; buflen = 6

    mov rsi,buf     ; load bcd pointer
    xor rax,rax     ; sum
```

```
    mov     rbx,10
    mov     rcx,05      ; digit_count

back1:  xor rdx,rdx
    mul     ebx     ; previous digit * 10 = ans (rax*rbx = rdx:rax)

    xor rdx,rdx
    mov     dl,[rsi]    ; Take current digit
    sub     dl,30h      ; accepted digit is Decimal, so Sub 30H only
    add     rax,rdx
    inc     rsi

    dec rcx
    jnz back1

    mov [ans],ax

    print   hmsg, hmsg_len
    mov ax,[ans]
    call    display_16

    ret
;-----------------------------------------------------------------
accept_16:
    read    buf,5       ; buflen = 4 + 1

    xor     bx,bx
    mov     rcx,4
    mov     rsi,buf
next_digit:
    shl bx,04
    mov al,[rsi]
        cmp     al,"0"      ; "0" = 30h or 48d
        jb  error       ; jump if below "0" to error
        cmp   al,"9"
        jbe  sub30      ; subtract 30h if no is in the range "0"-"9"

        cmp al,"A"      ; "A" = 41h or 65d
        jb   error          ; jump if below "A" to error
        cmp al,"F"
        jbe   sub37         ; subtract 37h if no is in the range "A"-"F"

        cmp     al,"a"          ; "a" = 61h or 97d
        jb error        ; jump if below "a" to error
        cmp al,"f"
        jbe  sub57          ; subtract 57h if no is in the range "a"-"f"

error:  print   emsg,emsg_len   ; "You entered Invalid Data!!!"
        exit

sub57:  sub al,20h          ; subtract 57h if no is in the range "a"-"f"
sub37:  sub   al,07h        ; subtract 37h if no is in the range "a"-"f"
sub30:  sub   al,30h        ; subtract 30h if no is in the range "0"-"9"
```

```
    add bx,ax       ; prepare number
    inc rsi     ; point to next digit
    loop    next_digit
ret
;-----------------------------------------------------------------
display_16:
    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,16      ; divisor=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
```

Output:

```
;[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:2


;Enter 5 digit BCD number::65535


;Hex Equivalent::0FFFF
```

Assignment No. 07

TITLE: Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers.

OBJECTIVES:

1. To be familiar with the format of assembly language program structure and instructions.
2. To study GDTR, LDTR and IDTR.

PROBLEM DEFINITION:

Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers.

SOFTWARE HARDWARE REQUIRED:

CPU: Intel i5 Processor

OS: Ubuntu 14 (64 bit) & 64 bit execution

Editer: gedit, GNU Editor

Assembler: NASM (Netwide Assembler)

Linker: GNU Linker

INPUT: system file

OUTPUT: contents of GDTR, LDTR and IDTR.

THEORY:

Four registers of the 80386 locate the data structures that control segmented memory management called as memory management registers:

1. GDTR: Global Descriptor Table Register

These register point to the segment descriptor tables GDT. Before any segment register is changed in protected mode, the GDT register must point to a valid GDT. Initialization of the GDT and GDTR may be done in real-address mode. The GDT (as well as LDTs) should reside in RAM, because the processor modifies the accessed bit of descriptors. The instructions LGDT and SGDT give access to the GDTR.

2. LDTR: Local Descriptor Table Register

These register point to the segment descriptor tables LDT. The LLDT instruction loads a linear base address and limit value from a six-byte data operand in memory into the LDTR. The SLDT instruction always store into all 48 bits of the six-byte data operand.

3. IDTR Interrupt Descriptor Table Register

This register points to a table of entry points for interrupt handlers (the IDT). The LIDT instruction loads a linear base address and limit value from a six-byte data operand in memory into the IDTR. The SIDT instruction always store into all 48 bits of the six-byte data operand.

4. **TR Task Register**

This register points to the information needed by the processor to define the current task.

These registers store the base addresses of the descriptor tables (A descriptor table is simply a memory array of 8-byte entries that contain Descriptors and descriptor stores all the information about segment) in the linear address space and store the segment limits.

SLDT: Store Local Descriptor Table Register

Operation: DEST ← 48-bit BASE/LIMIT register contents;

Description: SLDT stores the Local Descriptor Table Register (LDTR) in the two-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table. SLDT is used only in operating system software. It is not used in application programs.

Flags Affected: None

SGDT: Store Global Descriptor Table Register

Operation: DEST ← 48-bit BASE/LIMIT register contents;

Description: SGDT copies the contents of the descriptor table register the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 32 bits, the next three bytes are assigned the BASE field of the register, and the fourth byte is written with zero. The last byte is undefined.

Otherwise, if the operand-size attribute is 16 bits, the next 4 bytes are assigned the 32-bit BASE field of the register. SGDT and SIDT are used only in operating system software; they are not used in application programs.

Flags Affected: None

SIDT: Store Interrupt Descriptor Table Register

Operation: DEST ← 48-bit BASE/LIMIT register contents;

Description: SIDT copies the contents of the descriptor table register the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 32 bits, the next three bytes are assigned the BASE field of the register, and the fourth byte is written with zero. The last byte is undefined.

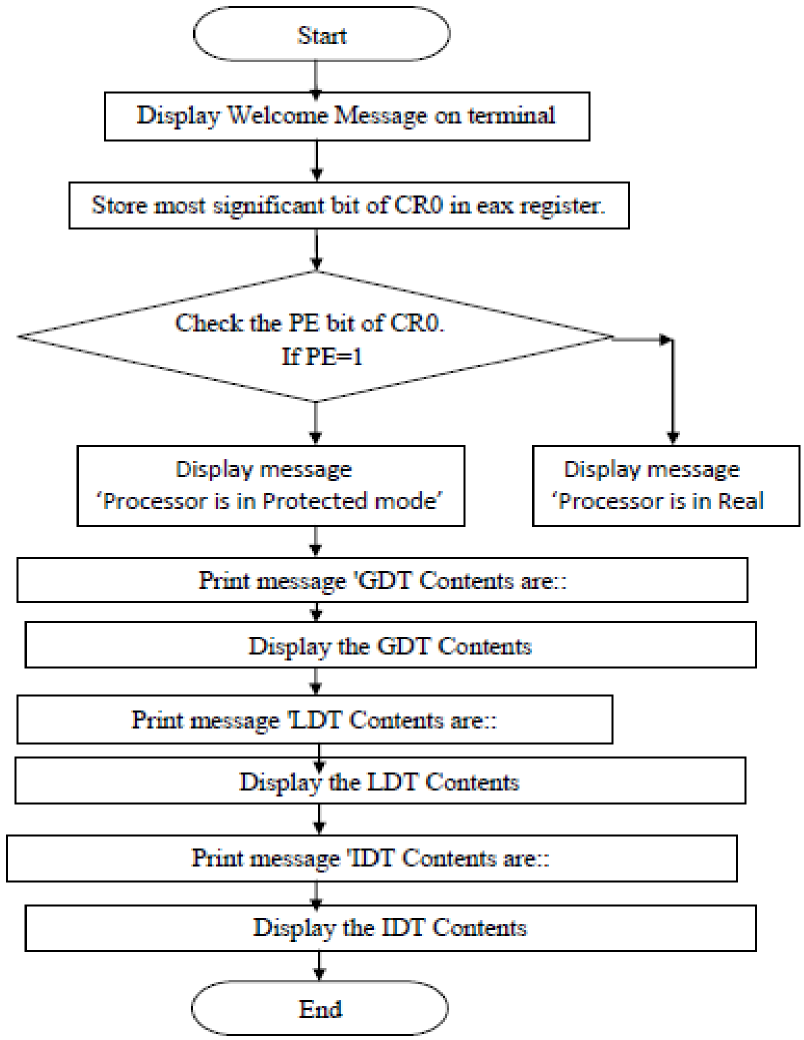
Otherwise, if the operand-size attribute is 16 bits, the next 4 bytes are assigned the 32-bit BASE field of the register. SGDT and SIDT are used only in operating system software; they are not used in application programs.

Flags Affected: None.

ALGORITHM:

1. Display welcome message on terminal using macro disp.
2. Store most significant bit of CR0 in eax register.
3. Check the PE bit of CR0.
4. If PE=1 then display message “Processor is in Protected mode”.
5. And if PE=0 then display message “Processor is in Real mode”.
6. Then copies/stores the contents of GDT, IDT, LDT using sgdt, sidt, sldt instruction.
7. Display their contents using macro function disp and disp_num.

FLOWCHART:



CONCLUSION:

In this way, we use GDTR, LDTR and IDTR in Real mode.

Program:

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

Section .data
title: db 0x0A,"----Assignment 6-----", 0x0A
```

```
title_len: equ $-title
regmsg: db 0x0A,"***** REGISTER CONTENTS *****"
regmsg_len: equ $-regmsg
gmsg: db 0x0A,"Contents of GDTR : "
gmsg_len: equ $-gmsg
lmsg: db 0x0A,"Contents of LDTR : "
lmsg_len: equ $-lmsg
imsg: db 0x0A,"Contents of IDTR : "
imsg_len: equ $-imsg
tmsg: db 0x0A,"Contents of TR : "
tmsg_len: equ $-tmsg
mmsg: db 0x0A,"Contents of MSW : "
mmsg_len: equ $-mmsg
realmsg: db "---- In Real mode. ----"
realmsg_len: equ $-realmsg
protmsg: db "---- In Protected Mode. ----"
protmsg_len: equ $-protmsg
cnt2:db 04H
newline: db 0x0A

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

tr:     resw 1
value :resb 4

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

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

```
call HtoA

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

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

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

;---- Task Register Contents -0-----
scall 1,1, tmsg,tmsg_len
mov bx,word[tr]
call HtoA
;------- Content of MSW ---------
scall 1,1,mmsg,mmsg_len
mov bx, word[msw+2]
call HtoA
mov bx, word[msw]
call HtoA
scall 1,1,newline,1
EXIT:
mov rax,60
mov rdi,0
syscall
;------HEX TO ASCII CONVERSION METHOD ----------------
HtoA:    ;hex_no to be converted is in bx //result is stored in rdi/user defined variable
mov rdi,value
mov byte[cnt2],4H
aup:
rol bx,04
mov cl,bl
and cl,0FH
cmp cl,09H
jbe ANEXT
ADD cl,07H
ANEXT:
add cl, 30H
mov byte[rdi],cl
INC rdi
dec byte[cnt2]
JNZ aup
scall 1,1,value,4
ret
```

Assignment No. 8 and 9

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

OBJECTIVES:

1. To be familiar with the format of assembly language program along with different assembler directives and different functions of the DOS Interrupt.
2. To learn the instructions related to String and use of Direction Flag.
3. To be familiar with data segments.
4. Implement non-overlapped and overlapped block transfer.

PROBLEM DEFINITION:

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

S/W AND H/W REQUIREMENT:

Software Requirements

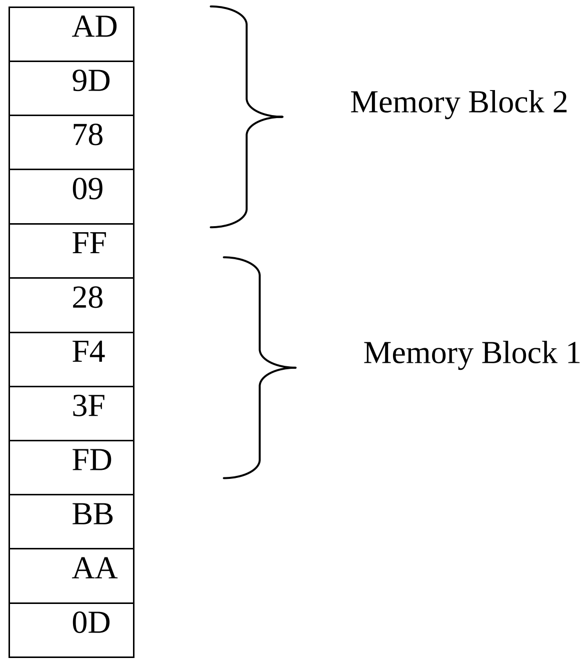
11. CPU: Intel I5 Processor
12. OS:- Linux Ubuntu 14 (64 bit Execution)
13. Editor: gedit, GNU Editor
14. Assembler: NASM (Netwide Assembler)
15. Linker:-LD, GNU Linker

INPUT: Input data specified in program.
OUTPUT: Display the data present in destination block.

THEORY:

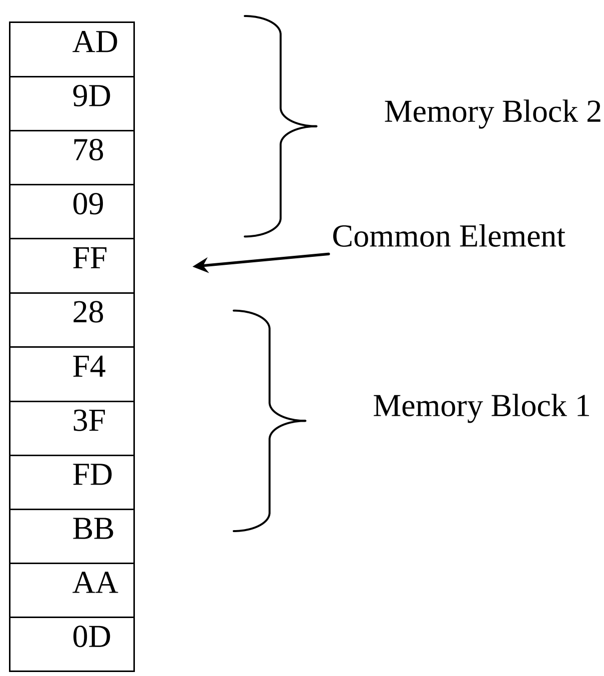
1. **Non-overlapped blocks:**
   In memory, two blocks are known as non-overlapped when none of the element is common. In below example of non-overlapped blocks there is no common element between block 1 & 2.

   While performing block transfer in case of non-overlapped blocks, we can start transfer from starting element of source block to the starting element of destination block and then we can transfer remaining elements one by one.



2. **Overlapped block:**
   In memory, two blocks are known as overlapped when at least one element is common between two blocks.

   While performing block transfer we have to see which element/s of source block is/are overlapped. If ending elements are overlapped then start transferring elements from last and if starting elements are overlapped then start transfer from first element.



Instructions needed:

1. **MOVSB**-Move string bytes.
2. **JNE**-Jump if not equal
3. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word
4. **INC**-Increments specified byte/word by 1
5. **DEC**-Decrements specified byte/word by 1
6. **JNZ**-Jumps if not equal to Zero
7. **CMP**-Compares to specified bytes or words
8. **JBE**-Jumps if below of equal
9. **CALL**-Transfers the control from calling program to procedure.
10. **RET**-Return from where call is made

ALGORITHM:

ALGORITHM:

1) Non –Overlapped Block Transfer:

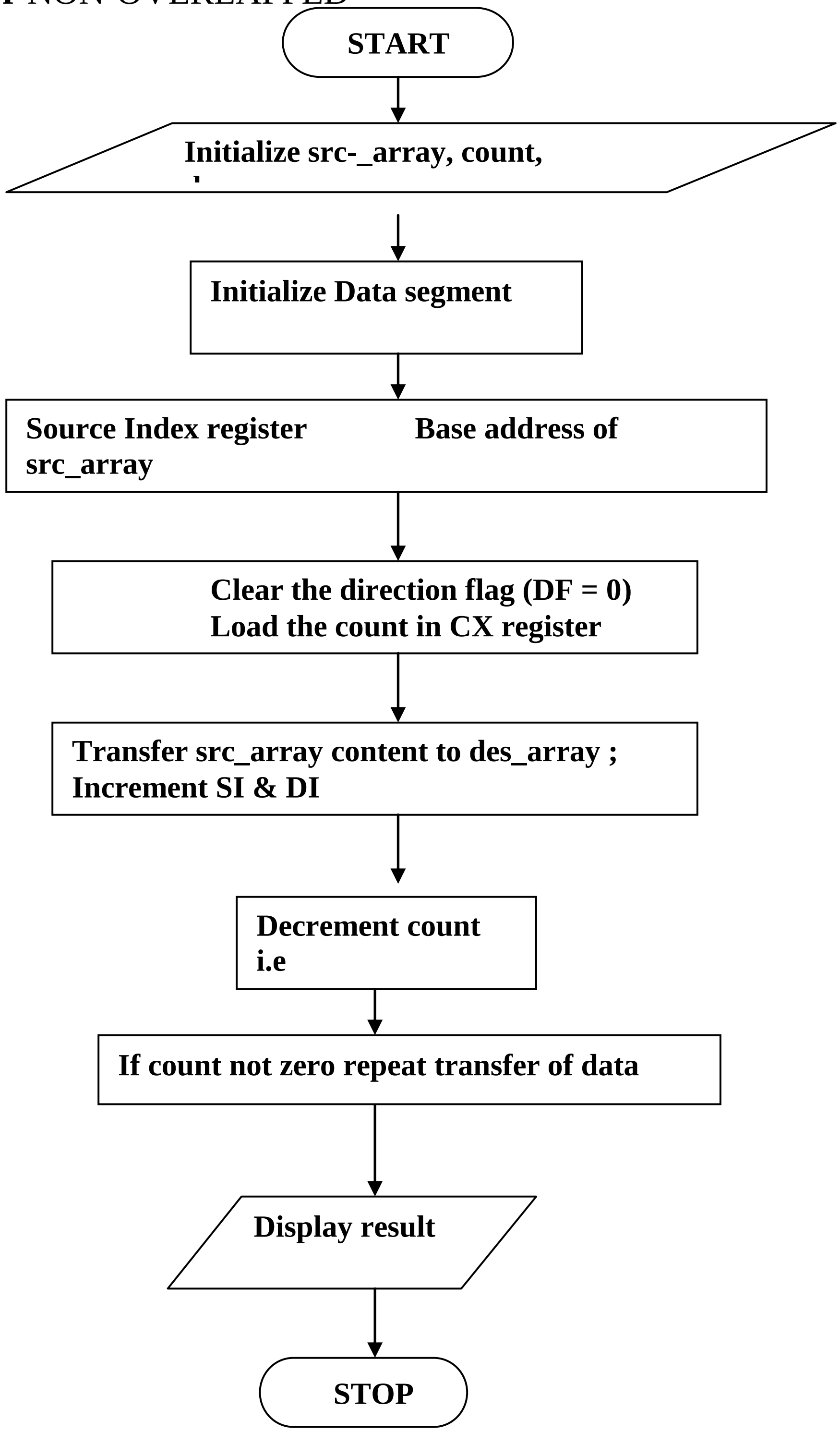
In non-overlapped block transfer Source Block & destination blocks are different. Here we can transfer byte by byte data or word by word data from one block to another block.

1. Start
2. Initialize data section
3. Initialize RSI to point to source block
4. Initialize RDI to point to destination block
5. Initialize the counter equal to length of block
6. Get byte from source block & copy it into destination block
7. Increment source & destination pointer
8. Decrement counter
9. If counter is not zero go to step vi
10. Display Destination Block
11. Stop

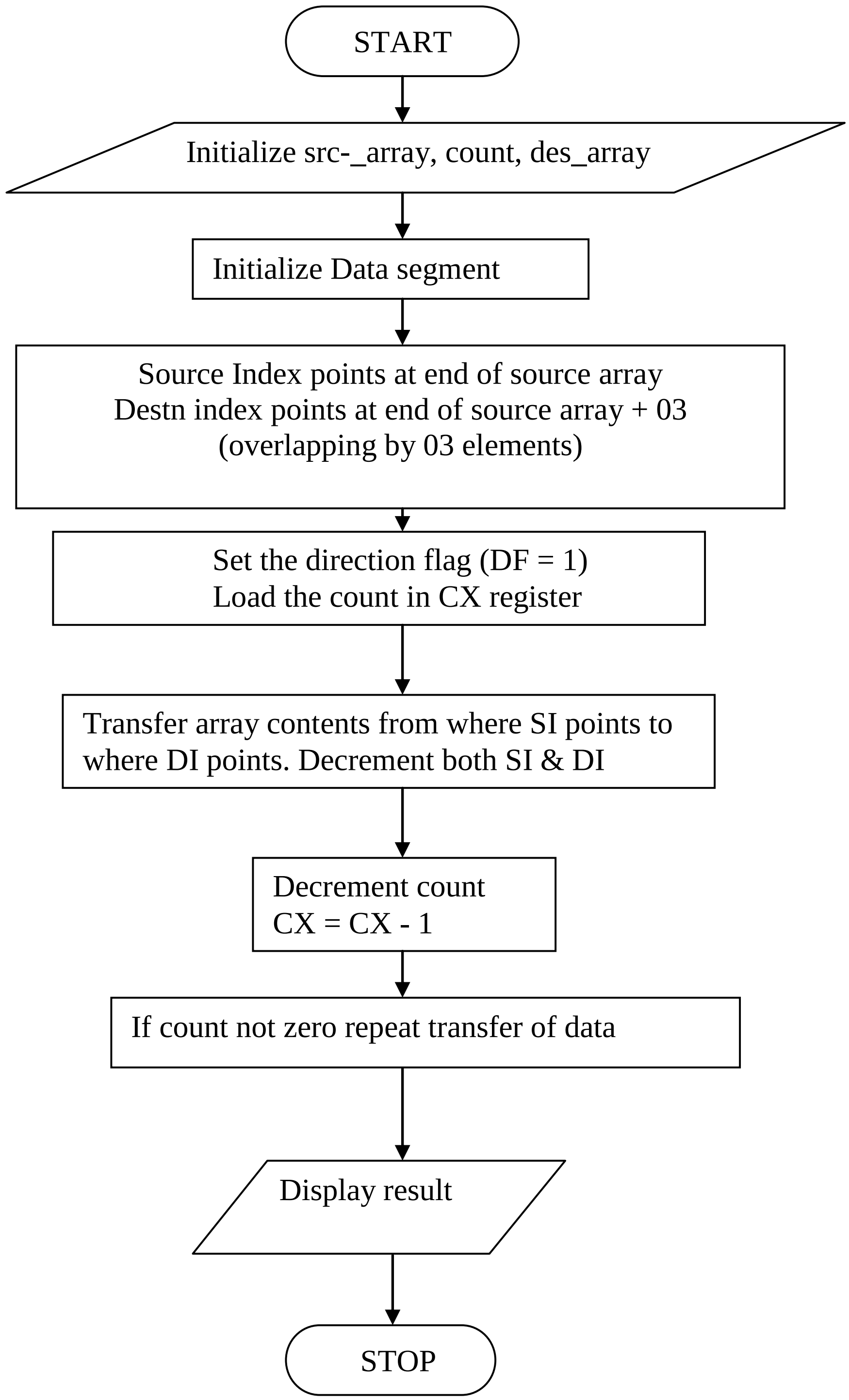
Overlapped Block Transfer: In Overlapped block transfer there is only one block & within the same block.We are transferring the data.

1. Start
2. Initialize data section
3. Accept the overlapping position from the user
4. Initialize RSI to point to the end of source block
5. Add RSI with overlapping Position & use it as pointer to point to End of
6. Destination Block.
7. Initialize the counter equal to length of block
8. Get byte from source block & copy it into destination block
9. Decrement source & destination pointer
10. Decrement counter
11. If counter is not zero go to step vii
12. Display Destination Block
13. Stop

Flowchart: For NON-OVERLAPPED



Flowchart: For OVERLAPPED BLOCK TRANSFER



CONCLUSION

Hence we conclude that we can perform non-overlapped & overlapped block Transfer with & without using string instructions.

Program:

```
%macro scall 4                          ;macro to take input and output
       mov rax,%1
       mov rdi,%2
       mov rsi,%3
       mov rdx,%4
       syscall
%endmacro

section .data

menu db 10d,13d,"                MENU"
     db 10d,"1. Non-Overlapping (Without String Instructions)"
     db 10d,"2. Overlapping (Without String Instructions)"
     db 10d,"3. Non-Overlapping (With String Instructions)"
     db 10d,"4. Overlapping (With String Instructions)"
     db 10d,"5. Exit"
     db 10d
     db 10d,"Enter your choice: "
lnmenu equ $-menu

       m1 db 10d,13d,"Enter Count Of numbers: "
       l1 equ $-m1
       m2 db 10d,13d,"Enter Numbers: ",10d,13d
       l2 equ $-m2
       m3 db 10d,13d,"Array 1: ",10d,13d
       l3 equ $-m3
       m4 db 10d,13d,"Array 2: ",10d,13d
       l4 equ $-m4
       m5 db 10d,13d,"Enter Overlapping Position: "
    l5 equ $-m5
       nwline db 10d,13d

section .bss
    choice resb 2
    answer resb 20
    array1 resb 300
    array2 resb 300
    count resb 20
    count1 resb 20
    count2 resb 20
    temp resb 20
    posn resb 20

section .text
       global _start
       _start:
```

```
;*********************MAIN LOGIC****************************
main:
        scall 1,1,menu,lnmenu
        scall 0,0,choice,2

        cmp byte[choice],'5'
        je exit

        call inputarray

        mov rax,qword[count1]
        mov qword[count],rax
        mov qword[count2],rax
        cmp byte[choice],'1'
        je case1
        cmp byte[choice],'2'
        je case2
        cmp byte[choice],'3'
        je case3
        cmp byte[choice],'4'
        je case4


back:
        mov rax,qword[count2];rax=5
        mov qword[count],rax;count=5

        call displayarray
        jmp main

;******************CASE1 to CASE4******************************
case1:
;number moving from arr1 to arr2
    mov rsi,array1
    mov rdi,array2

loop6:  mov rax,[rsi]
    mov [rdi],rax

    add rsi,8
    add rdi,8
    dec qword[count];count=5
    jnz loop6

jmp back

case2:
;position enter
    scall 1,1,m5,l5
    scall 0,0,temp,17
    call asciihextohex
```

```
    mov qword[posn],rbx;posn=2
    add qword[count1],rbx;count1=5+2=7
;number moving from arr1 to arr2
    mov rsi,array1
    mov rdi,array2

loop7:  mov rax,[rsi]
    mov [rdi],rax

    add rsi,8
    add rdi,8
    dec qword[count]
    jnz loop7

    mov rax,qword[count2];count2=5
    mov qword[count],rax;count=5

    mov rsi,array1
    mov rdi,array2

loop8:  add rdi,8
    dec qword[posn];posn=2
    jnz loop8

loop9:mov rax,[rsi]
    mov [rdi],rax
    add rsi,8
    add rdi,8
    dec qword[count];count=5
    jnz loop9
jmp back

case3:
    mov rsi,array1
    mov rdi,array2
    mov rcx,[count];count=5
    cld                     ;CLEAR DIRECTION FLAG
    rep movsq;repeat unitl counter becomes zero

jmp back

case4:
;position enter
    scall 1,1,m5,l5
    scall 0,0,temp,17
    call asciihextohex

    mov qword[posn],rbx;posn=2
    add qword[count1],rbx;count1=5+2=7

;number moving from arr1 to arr2
    mov rsi,array1
    mov rdi,array2
```

```
    mov rcx,[count]
    cld                  ;CLEAR DIRECTION FLAG;if DF=0 increment rsi and rdi by one byte else decrement by one byte
    rep movsq

    mov rax,qword[count2]
    mov qword[count],rax

    mov rsi,array1
    mov rdi,array2

loop10: add rdi,8
    dec qword[posn]
    jnz loop10

    mov rcx,qword[count1]
    rep movsq
jmp back

exit:                          ;exit code
    mov rax,60
    mov rdi,0
    syscall


;*********************PROCEDURES**********************************

inputarray:
    scall 1,1,m1,l1
    scall 0,0,temp,17
    call asciihextohex

    mov qword[count],rbx;count=5
    mov qword[count1],rbx;count1=5

    scall 1,1,m2,l2
    mov rbp,array1

loop1:  scall 0,0,temp,17
    call asciihextohex
    mov qword[rbp],rbx

    add rbp,8;to point to next element after 8 byte(one quad word)
    dec qword[count]
    jnz loop1
ret

displayarray:
        scall 1,1,m3,l3
    mov rbp,array1
loop2:  mov rax,[rbp]
    call display
    scall 1,1,nwline,1
```

```
    add rbp,8
    dec qword[count]
    jnz loop2

    mov rax,qword[count1]
    mov qword[count],rax;value given to it again as count becomes zero in above instruction

    scall 1,1,m4,l4
    mov rbp,array2
loop3:  mov rax,[rbp]
    call display
    scall 1,1,nwline,1
    add rbp,8
    dec qword[count]
    jnz loop3

ret

asciihextohex:
    mov rsi,temp
    mov rcx,16
    mov rbx,0
    mov rax,0


loop4:  rol rbx,04
    mov al,[rsi]
    cmp al,39h
    jbe skip1
    sub al,07h
skip1:  sub al,30h

    add rbx,rax

    inc rsi
    dec rcx
    jnz loop4
ret

display:
        mov rsi,answer+15;since in 1 byte 2 packed bcd no can be stored
        mov rcx,16;total count=no.bytes X 2=8*2=16

loop5:  mov rdx,0
        mov rbx,16;divisor=10 for decimal and 16 for hex
        div rbx
        cmp dl,09h
        jbe skip2

        add dl,07h
skip2:  add dl,30h
        mov [rsi],dl
```

```
        dec rsi
        dec rcx
        jnz loop5
        scall 1,1,answer,16
ret
```

Assignment No. 10

TITLE: Multiplication of two 8 bit nos. using Successive addition and Shift and add method

OBJECTIVES:

1. Understand the implementation.
2. To interpret the Microprocessor Interfacing paradigms.
3. To express and apply the method of odd, add and shift method.
4. Understand implementation of arithmetic instruction of 8086.

PROBLEM STATEMENT:

Write 8086/64 ALP to perform multiplication of two 8 bit hexadecimal nos. Use successive addition & shift & add method, Accept i/p from the user.

SOFTWARE HARDWARE REQUIRED:

CPU: Intel i5 Processor

OS: Ubuntu 14 (64 bit) & 64 bit execution

Editer: gedit, GNU Editor

Assembler: NASM (Netwide Assembler)

Linker: GNU Linker

INPUT: Two hex nos.

For e.g. AL=12H, BL= 10H

OUTPUT:

Result : D120H

THEORY:

There are 5 basic form of define reverse directives.

Directives	**Purpose**
DD	Define byte
DW	Define word
DD	Define doubleword
DQ	Define quad word

DT	Define Ten byte
RESB	Reserve byte
RESW	Reserve word
RESQ	Reserve quad word
REST	Reserve ten word

Instructions Needed:

MOV : Move or copy word

ROR : Rotate to right

AND : Logical AND

INC : Increment

DEC : Decrement

JNZ : Jump if not zero

CMP : Compare

JNC : Jump if no carry

JBE : Jump if below

Shift & Add method:

The method taught in school for multiplying decimal no. is based on calculated partial products, shifting it to the left & then adding them together. Shift & add multiplication is similar to the multiplication performed by paper & pencil. This method adds the multiplicand X to itself Y times where Y denotes the multiplier. To multiply two nos. by paper & pencil placing the intermediate product in the appropriate positions to the left of earlier product.

1. Consider 1 byte is in AL & another in BL
2. We have to multiply byte in AL with byte in BL
3. In this method, you add 1 with itself & rotate other no. each times &shift it by 1 bit n left along with carry
4. If carry is present add 2 NOS.
5. Initialize count to n as we are scanning for n digit decrement counter each time, the bits are added

The result is stored in AX, display the result.

Eg., AH=11H, BL=10H, Count=n

Step 1:

AX=11 + 11 = 22H

Rotate BL by 1 bit to left along with carry 0001 0000

B1=10H <u>0010</u> <u>0000</u> (<u>20</u>)

Step 2:

Decrement count =3

Check for carry, carry is not there So Add with itself

AX=22+22=44H

Rotate BL to left

BL=0 <u>0000</u> <u>0000</u> (<u>00</u>)

No carry

Step 3:

Decrement count=2

Add no. with itself

AX=44+44=88H

Rotate BL to left

B2=0 (carry) <u>1000</u> <u>0000</u> (<u>80</u>)

Step 4:

Decrement count=0

Add no. with itself,

AX=88+88=110H

Rotate BL to left

BL=0 (carry) <u>1000</u> <u>0000</u> (<u>80</u>)

Step 5:

Decrement count =0, carry is generated

Add Ax, BX

0110+0000=0110H

i.e.,

11H+10H=0110H

MATHEMATICAL MODEL:

Let S={s, e, x, y, time, mem Φs} be program perspective of multiplication of two 8 bit hexadecimal Nos.

Let X be the input such that

X={x1, x2, x3, ……………}

Such that there exists function fx1: x{0, 1}

X2={s the two digit multiplicate }

Let x2= b7 b6 b5 b4 b3 b2 b1 b0

Where xb1 € x1 Here exists a function fx2:x2 {00h, 01h, 02h,………..ffh}

X3 is the single digit choice

Let x4 = b3 b2 b1 b0 where (bi x € x1 there Let y is the output)

Y= b15 b14 b13 b12 bb11 b10 b09 b08 b07 b06 b05 b04 b03 b02 b01 b00 (where bi€ x1)

Y= {0000b,0001b…………………….ffffb}

Time (for successive addition) = {f1,f2,f3,f4}

Where f1 = Accept x2 & x3

F2= Addition (Repeat x2+x1+……………………………& till x3=00)

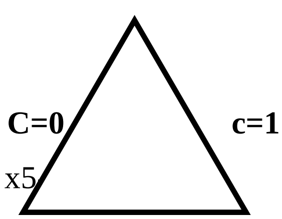
F3= display

Time (for shift and add)= {f1,f2,f3}

Where

F1= Accept x1&x3

F2= shift x3>>



Shift x2 << & x5 = x5

Shift x2 << & add with y, x5=x5-1

Repeat till x5=0

F3 = display y

C=0 c=1

Shift x2 << x5 << x5-1

Shift x2 << & add with y x5 = x5-1

Repeat till x5=0

F3= display

Add it with result & then decrement counter display result.

ALGORITHM:

1. **Successive Addition**
    1. Start
    2. Get the 1st no. from user
    3. Get 2nd no. from user the no. will get as counter.
    4. Initialize result=0
    5. Add the 1st no. of itself as multi times
    6. Decrement counter
    7. Compare the counter with ‘0’
        1. If count ≠0
           Goto step 5
        2. Else
            1. Display the result
            2. Stop

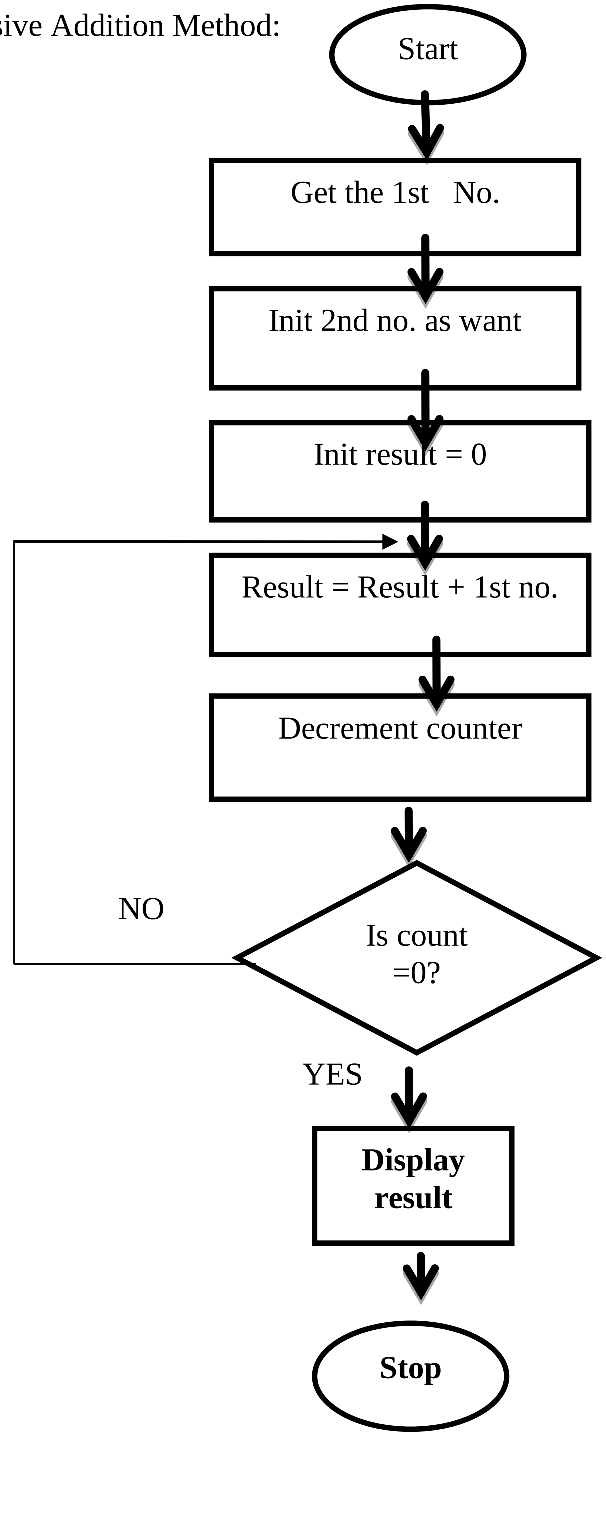
Shift Addition Method

1. Start
2. Get the 1st no. from user
3. Initialize count =0
4. No.1= no*2
5. Get the 2nd no. from user
6. Shift multiplier to left along with carry

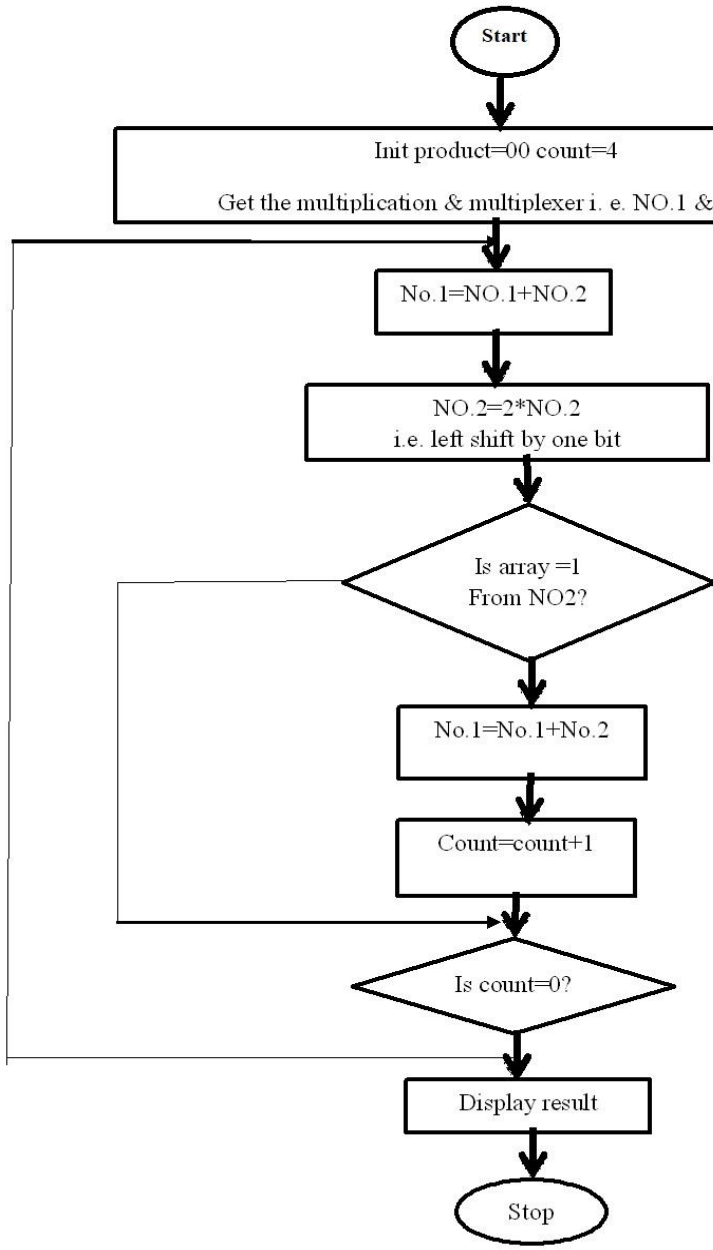
7. Check for carry, if present goto step 4
8. No. 1= no.1 + shifted no.2
9. Decrement counter
10. If not zero, goto step 6
11. Display result
12. Stop

FLOWCHART:

Successive Addition Method:



1. **Add & shift Method**



CONCLUSION:

From this program we have studied the multiplication of 8 bit nos. and in this we have studied and implemented the program of successive addition and shift & add method.

Program:

```
section .data                         ;initialised data section

   aim db 0ah,"Program to perform multiplication of two hexadecimal numbers"
   len equ $-aim

   menu db 0ah,'1. Successive Addition'           ;menu to display
       db 0ah,'2. Add and Shift'
       db 0ah,'3. Exit'
       db 0ah,'   Enter your choice : '
   menul equ $-menu                   ;length of menu

   msg1 db 0ah,'Enter First Number  : '          ;message to display
   msgl1 equ $-msg1                   ;length of message

   msg2 db 0ah,'Enter Second Number : '
   msgl2 equ $-msg2

   msg3 db 0ah,'Result : '
   msgl3 equ $-msg3

section .bss                        ;uninitialised data section
n resb 4                          ;variable to take the input and print the output
num1 resb 1                         ;variable to store multiplicand
num2 resb 1                         ;variable to store multiplier
choice resb 2                         ;variable to take choice as input

section .text                       ;code section
global _start                       ;start of program execution
_start :

%macro io 4                        ;macro for print and scan

   mov rax,%1                      ;first parameter
   mov rdi,%2                     ;second parameter
   mov rsi,%3                    ;third parameter
   mov rdx,%4                      ;fourth parameter
   syscall                     ;interrupt call

%endmacro                          ;end of macro

   io 1,1,aim,len

   men:                        ;user defined flag

      io 1,1,menu,menul            ;printing the menu
```

```
    io 0,0,choice,2              ;taking user choice as input

    case1:                   ;case 1 for successive addition method

        cmp byte[choice],'1'          ;comparing whether user's choice is 1
        jne case2              ;jumping to case2 if user didnot enter 1 as choice
        io 1,1,msg1,msgl1          ;print message1
        io 0,0,n,3             ;taking multiplicand as input
        call asciihex              ;procedure call to convert input ascii to hex hex equivalent
        mov [num1],bl              ;moving contents of bl to num1 variable

        io 1,1,msg2,msgl2          ;print message2
        io 0,0,n,3             ;taking multiplier as input
        call asciihex              ;procedure call to convert input ascii to hex equivalent
        mov [num2],bl              ;moving contents of bl to num2 variable

        call p1               ;procedure call
        jmp men               ;unconditional jump

    case2:                   ;case 2 for add and shift method
        cmp byte[choice],'2'          ;comparing whether user's choice is 1
        jne exi               ;jumping to case2 if user didnot enter 1 as choice
        io 1,1,msg1,msgl1          ;print message1
        io 0,0,n,3             ;taking multiplicand as input
        call asciihex              ;procedure call to convert input ascii to hex hex equivalent
        mov [num1],bl              ;moving contents of bl to num1 variable

        io 1,1,msg2,msgl2          ;print message2
        io 0,0,n,3             ;taking multiplier as input
        call asciihex              ;procedure call to convert input ascii to hex equivalent
        mov [num2],bl              ;moving contents of bl to num2 variable

        call p2               ;procedure call
        jmp men               ;unconditional jump

exi:                       ;user defined flag

mov rax,60                  ;system exit
mov rdi,0                   ;system exit
syscall                    ;system interrupt

p1:                     ;procedure for successive addition method

    mov rcx,0               ;clearing contents of rcx
    mov rbx,0               ;clearing contents of rbx
    mov rax,0               ;clearing contents of rax
    mov al,[num1]               ;storing num1 in al
    mov cl,[num2]               ;storing num2 in cl

    lp2:                   ;user defined label

        add bx,ax             ;add contents of bx and ax and store them in bx
```

```
    loop lp2                ;looping instruction,automatically decrements rcx
    io 1,1,msg3,msgl3             ;printing message3
    call displaynum               ;procedure call

ret                     ;proccedure return

p2:                     ;procedure for add and shift method

    mov rcx,0               ;clearing contents of rcx
    mov rbx,0               ;clearing contents of rbx
    mov rdx,0               ;clearing contents of rdx
    mov al,[num1]                 ;storing num1 in al
    mov bl,[num2]                 ;storing num2 in bl
    mov rcx,8               ;initialising rcx to 8

    lp4:                    ;user defined flag

        shr bl,1              ;shifting bits of bl right by 1 bit,lost bit is copied in carry flag
        jnc flg               ;making conditional jump if carry flag is 1
        add dx,ax               ;adding contents of dx and ax and storing them in dx
        flg:shl ax,1              ;shifting bits of ax left by 1

    loop lp4                ;looping instruction,automatically decrements rcx
    mov rbx,rdx               ;moving contents of rdx to rbx
    io 1,1,msg3,msgl3             ;printing message3
    call displaynum               ;procedure call

ret                     ;procedure return

displaynum:                 ;procedure to display 4-digit number

    mov rcx,4               ;initialising rcx to 4
    mov rax,0               ;clearing contents of rax
    mov rsi,n              ;making rsi point to the number to be displayed

    lp3:                    ;user defined label

        rol bx,4              ;rotating bits of bx to left by 1 bit
        mov al,bl              ;moving contents of bl into al
        and al,0fh             ;anding al and 0fh to get LSB digit
        cmp al,9               ;comparing al and 9
        jbe add30h              ;jump below or equal
        add al,7h              ;adding 7h to al
        add30h : add al,30h         ;adding 30h to al
        mov [rsi],al               ;moving contents of al to location pointed by rsi
        inc rsi               ;incrementing rsi to point to next location

    loop lp3                ;looping instruction,automatically decrements rcx

    io 1,1,n,4              ;printing 4-digit number

ret                     ;procedure return
```

```
asciihex:                ;procedure for ascii to hex conversion

    mov rsi,n              ;making rsi point to variable hex
    mov rbx,0              ;clearing rbx
    mov rax,0              ;clearing rax
    mov rcx,2              ;initialising rcx to 4

    lp1:                   ;user defined label

        rol bl,4           ;rotating contents of bx to left by 4 bits
        mov al,[rsi]           ;moving value pointed by rsi to al
        cmp al,39h         ;comparing value in al to 9
        jbe sub30h         ;jump if below or equal to flag sub30h
        sub al,7h          ;subtract 7h from al
        sub30h: sub al,30h     ;subtract 30h from al
        add bl,al          ;adding value in al as units digit in bl
        inc rsi            ;incrementing rsi to point to next location

    loop lp1               ;looping instruction

ret                      ;procedure return
```

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]# 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
```

Assignment No. 11

TITLE: ALP to implement DOS commands TYPE, COPY, & DELETE using File Operations.

OBJECTIVES:

1. To be familiar with the format of assembly language program along with different assembler directives and different functions of the DOS Interrupt.
2. To be familiar with DOS Commands.
3. Implement file operations.

PROBLEM DEFINITION:

Write x86 menu driven ALP to implement DOS commands TYPE, COPY, & DELETE using File Operations. User is supposed to provide command line arguments in all cases.

SOFTWARE HARDWARE REQUIRED:

CPU: Intel i5 Processor

OS: Ubuntu 14 (64 bit) & 64 bit execution

Editer: gedit, GNU Editor

Assembler: NASM (Netwide Assembler)

Linker: GNU Linker

INPUT: Input data specified in program(TEXT FILE)**.**
OUTPUT: Display the data Present in File.
THEORY:

PSP-:

Whenever DOS loads a program for execution it creates a 256 byte data structure called PSP.

It is available at 1st paragraph of true memory program itself is located and load above psp.

There are two types of disk access method.

1. ASCII method
2. Handles.

Handles-

Use of file handles to file operation the file management function access the file in fashion similar To that used under UNIX.

ASCII Function calls -:

1. 3H- Creates a file DS:DX.
2. 30H- Open a file.
3. 3EH- Close file.
4. 40H- Write into file.
5. 41H- Delete a file.
6. 17H- Rename a file.
7. 4EH- To check if file exists.

File Control Block (FCB)-

As FCB is a 37 file data structure allocated with the application program memory.

Common FCB Record Operation-:

0FH: Open a File.

10H: Close file.

16H: Create a file.

14H: perform segment Read.

15H: perform segment write.
22H: perform random.

ALGORITHM FOR TYPE:

1. Start
2. Get source file name and destination file name from command tail.
3. If file is not present display error message as “File not found” and stop.
4. If present, open the file in read mode.

5. Read the contents of file and print the data on the screen
6. Stop

ALGORITHM FOR COPY:

1. Start.
2. Get source file name and destination file name from command tail.
3. If file is not present display error message as “File not found” and stop.
4. If present, open the file in read mode.
5. Read name of destination file and open it in read mode.
6. Read the contents of file source and write it into destination file.
7. Stop.

Conclusion:

Hence we conclude that we Implement DOS commands like TYPE, COPY, DELET using file operations

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
   syscall
%endmacro

%macro fread 3
   mov rax,0       ;read
   mov rdi,%1  ;filehandle
```

```
    mov rsi,%2  ;buf
    mov rdx,%3  ;buf_len
    syscall
%endmacro

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

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

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


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

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

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

section .text
global _start
_start:

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

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

        exit
        ret

op1:
    call tproc
    jmp again

op2:
    call cpproc
    jmp again


op3:
    call delproc
    jmp again



;type command procedure
tproc:
    cmn 1,1,msg,msglen
    cmn 0,0,buffer,50;read file in buffer using function 2 ,you will get the length in rax including EOF
    mov byte[cmdlen],al
    dec byte[cmdlen];decrement the length by 1 since the length also includes EOF

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

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

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

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

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

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

ret


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

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

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

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

    mov rdi,name2           ;finding second file name
    call find_name

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

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

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

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

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

    jmp again
error:
```

```
    cmn 1,1,err,errlen
    exit
ret



;delete command procedure
delproc:

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

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

skipr:  cmn 1,1,cpywr,cpywrlen
    exit

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

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

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

errord:
```

```
    cmn 1,1,err,errlen
    exit

ret


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

skip2:
ret
```

Output:


enigma@enigma-VirtualBox: ~/Videos/Microprocessor Lab FINAL/as8
enigma@enigma-VirtualBox:~/Videos/Microprocessor Lab FINAL/as8$ nasm -f elf64 8.asm
enigma@enigma-VirtualBox:~/Videos/Microprocessor Lab FINAL/as8$ ld -o a 8.o
enigma@enigma-VirtualBox:~/Videos/Microprocessor Lab FINAL/as8$ ./a
MENU :
1. TYPE
2. COPY
3. DELETE
4. Exit
Enter your choice : 1
Command : type file1.txt
hello this is the content of file1
MENU :
1. TYPE
2. COPY
3. DELETE
4. Exit
Enter your choice : 2
Command : copy file1.txt file2.txt
File copied successfully !!
MENU :
1. TYPE
2. COPY
3. DELETE
4. Exit
Enter your choice : 3
Command : del file3.tx

Assignment No. 12

TITLE:

Program to analyze the difference between near and far procedure to find number of lines, blank spaces & occurance of character using nasm.

OBJECTIVES:

5. To be familiar with the format of assembly language program along with different assembler directives and different functions of the DOS Interrupt.
6. To be familiar with FAR PROCEDURES and PUBLIC and EXTERN directives.
7. Implement file operations.

PROBLEMDEFINITION:

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.

SOFTWARE HARDWARE REQUIRED:

CPU: Intel i5 Processor

OS: Ubuntu 14 (64 bit) & 64 bit execution

Editer: gedit, GNU Editor

Assembler: NASM (Netwide Assembler)

Linker: GNU Linker

INPUT: Input data specified in program (TEXT FILE)**.**
OUTPUT: Display the data Present in File.

THEORY:
In this program the user is going to enter the text file name on command line. Suppose user has enter file named abc.txt, which is present on stack memory in following form :

Arguments	abc.txt
.asm program name	file.asm
no. of arguments in the program	1
Current SP	

By accessing this stack we can get the file name in the program. Then by calling the interrupt for opening the file and closing the file we can access the contents of a file. The file contents are taken into buffer memory for display.

System calls for File –

Open File –

open the file for reading

```
        mov eax,5           ;system call number (sys_open)
        movebx,file_name    ;file name
        mov ecx,0           ;file access mode
        mov edx,0777        ;read,write and execute by all
        int 80h             ;call kernel

        mov [fd_in],eax             ;file descriptor
        bt eax,31           ;negative value in eax indicates error
        jnc conti1

        printfnotmsg,fnmsg_len
        jmp exit
conti1:
        printopenmsg,omsg_len
        printfilemsg,fmsg_len
```

Read File -

read from file

```
readfile:
        mov eax,3           ;system call number (sys_read)
        movebx,[fd_in]              ;file descriptor
        movecx,fbuff        ;pointer to the input buffer
        movedx,fb_len               ;buffer size i.e the number of bytes to read
        int 80h             ;call kernel

        mov [act_len],eax
        cmp eax,0
        je nxt1
        printfbuff,[act_len]
        jmpreadfile
```

nxt1:

Close File -

close the file

```
mov eax,6            ;system call number (sys_close)
movebx,[fd_in]             ;file decriptor
int 80h
```

Instruction:

1) POP –

Description - This instruction going to pop the contents from stack into destination mentioned in the instruction. The stack pointer is first incremented and then the contents are popped.
e. g. pop ebx

2)JNS -

Description-This instruction is going jump on label mentioned if sign flag is not set.
Flag: SF
e.g. jns up

3) JS –

Description-This instruction is going jump on label mentioned if sign flag is set.
Flag: SF
e. g. js up

4)JZ-

Description-This instruction is going jump on label mentioned if zero flag is set.
Flag: ZF
e. g. jz up

- **EXTERN**

Informs the assembler that the names, procedures, labels declared after this directive have already been defined in some other assembly language module while in the other module where the names, procedures & labels actually appear, they must be declared Global using GLOBAL directive

- **GLOBAL**

The labels, variables, constants or procedures declared GLOBAL may be used by other modules of program . Once the variable is declared GLOBAL, it can be used by any module in the program.

- **FAR**

Used to declare the procedure far from the segment from where we are calling it.

- **PUBLIC**

Used to declare procedure publically

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 IP register on the stack.
3. Loads the base address of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the IP 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 IP 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 the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.
4. Resumes execution of the calling procedure.

Commands:
- To assemble

nasm –f elf 64 prog5a.asm

nasm –f elf 64 prog5b.asm

- To link

ld prog5a.o prog5b.o

- To execute -

./a.out

ALGORITHM:

Algorithm to Read a file
1) Call file in the program
2) put a pointer to the stack
3) Get file name from the stack.
4) Call interrupt to open the file.
5) The file descriptor is now available in eax. Test that descriptor.
6) If file descriptor is not valid then close the file and exit from program.
7) If valid file descriptor then copy the file contents into buffer.
8) It will display the no of blank spaces in the file.
9) It will display the no of lines in the file
10) It will display the occurrences of character (o).

11) Again test file descriptor. If it returns null then display the contents from the buffer.
12) Close the file.
10) Exit from the program

A1: Algorithm for Number of Blank spaces in the file

i. Start
ii. Initialize RSI to start of text file and RDI to end of text file,
iii. Start checking for the blank space if space is detected count will increase.
iv. Display Number of blank spaces
v. Exit

A2: Algorithm for Number of lines in the file

ι. Start
ιι. Initialize RSI to start of text file and RDI to end of text file,
ιιι. Start checking the line if enter is detected count of line will increase
ιϖ. Display Number of lines
ϖ. Ret

A3: Algorithm to count occurrences of character

ii. Start
iii. Initialize RSI to start of text file and RDI to end of text file,
iv. Start checking the occurrences of character (o) and count it,
v. Display the count value if character (o) is present.
vi. If character (o) is not present Display the message that character (o) is not present.
vii. Stop

Mathematical Model:
Let S = { s , e , X, Y, Fme, mem | Φs } be the programmer's perspective of String Manupulations Where

Let X be the input such that
X= { X1, X2,X3,-------}
Such that there exists function fx_1 : X1 $\longrightarrow$ {0,1}

X2 is the Ascii Value of String Character
Let X2 = b7 b6 b5 b4 b3 b2 b1 b0 where bi Є X1

There exists a function f_{X2} :X2 $\longrightarrow$ { 41h,42h,-------,61h,------}i.e 41h is ASCII equivalent of A & 61h is Ascii equivalent of a.

X3 is String1 Array
Let X3 = { {b7-----b0}{b7----b0}-----------{ b7 b6 b5 b4 b3 b2 b1 b0}} where bi Є X1
= { $X2_{n1}$,$X2_{n1-1}$,------------X_0} Where n1= Length of String1
e.g X3 = {Abcd} ------Entered String
then how it is stored { 41h,61h,62h,63h}
X4 is String2 Array
Let X4 = { {b7-----b0}{b7----b0}-----------{ b7 b6 b5 b4 b3 b2 b1 b0}} where bi Є X1
= { $X2_{n2}$,$X2_{n2-1}$,------------X_0} Where n2 = Length of String2
X5 is single digit choice
Let X4 = b3 b2 b1 b0 where bi Є X1
There exists a function f_{X5} :X5 $\longrightarrow$ { 1,2,3 }.
Y1 is concatenated String Array
Let Y1 = { {b7-----b0}{b7----b0}-----------{ b7 b6 b5 b4 b3 b2 b1 b0}} where bi Є X1
= {{ $X2_{n1}$,$X2_{n1-1}$,------------X_0}{ $X2_{n2}$,$X2_{n2-1}$,------------X_0} Where n1 = Length of String1 & n2= Length of String2
Let Y2= n1 is length of string1 which is returned by accept macro (Returned in RAX,64 bit Register)
Let Y2= b63 b62 b61------------------- b3 b2 b1 b0
Where bi Є X1
Y 2 $\longrightarrow$ { 0000000000000001h,-------------------------FFFFFFFFFFFFFFFFh}
Let Y3= n2 is length of string2 which is returned by accept macro (Returned in RAX,64 bit Register)
Let Y3= b63 b62 b61------------------- b3 b2 b1 b0
Where bi Є X1
Y 3 $\longrightarrow$ { 0000000000000001h,-------------------------FFFFFFFFFFFFFFFFh}
Y4 is the substring Count
Let Y4 = b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 where bi Є X1
Y4 $\longrightarrow$ {0000H,0001H,---------------------,FFFFH}
Let Fme = { F1 ,F2,F3,F4 }
Where F1 = Accept String1 & store n1
F2 = Accept String2 & store n2
F3= Accept X5 i.e choice.
F4 = If X5=1 call Fme1
F5 = If X5=2 call Fme2
F6 = If X5=3 call Fme3
Fme1 (concatenated String)= { F1,F2,F3}
Where F1= Copy X3 in Y1
F2= Copy X4 in Y1
F3= Display Y1

Fme2 (Substring)= { F1,F2}
Where F1= Compare X4 with X3 for count=n2
if Equal Y4=Y4+1

F2= If not equal increment pointer to X3 &
Repeat F1 till the end of X3
F3 = Display Y4

Fme3 (Exit)= { F1}
Where F1= Call Sys_exit

Commands:

- To assemble

  nasm –f elf 64 prog5a.asm

nasm –f elf 64 prog5b.asm

- To link

  ld prog5a.o prog5b.o

- To execute -

  ./a.out

Program: (Note:All files are save within single folder).

A5_file1.asm

```
;**********************************************************
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:
        display    ano,ano_len        ;assignment no.

        display    filemsg,filemsg_len
        accept     filename,50
        dec    rax
        mov    byte[filename + rax],0        ; blank char/null char

        display    charmsg,charmsg_len
        accept     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:    display    errmsg, errmsg_len

Exit:    display    exitmsg,exitmsg_len

   display nline,nline_len

   mov rax,60
   mov rdi,0
   syscall
```

A5_file2.asm

```
;***********************************************************
global    far_proc

extern    filehandle, char, buf, abuf_len

%include "macro.asm"
;***********************************************************
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
```

```
    dispbuff    resb    4

;************************************************************
section .text
;    global    _main
;_main:

far_proc:                ;FAR Procedure

       mov    rax,0
       mov    rbx,0
       mov    rcx,0
       mov    rsi,0

       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

       display smsg,smsg_len
       mov    rbx,[scount]
       call    display16_proc

       display nmsg,nmsg_len
```

```
        mov    rbx,[ncount]
        call    display16_proc

        display cmsg,cmsg_len
        mov    rbx,[ccount]
        call     display16_proc

    fclose    [filehandle]
    ret


;***********************************************************
display16_proc:
    mov rdi,dispbuff    ;point esi to buffer
    mov rcx,4        ;load number of digits to display
dispup1:
    rol bx,4        ;rotate number left by four bits
    mov dl,bl        ;move lower byte in dl
    and dl,0fh        ;mask upper digit of byte in dl
    add dl,30h        ;add 30h to calculate ASCII code
    cmp dl,39h        ;compare with 39h
    jbe dispskip1        ;if less than 39h akip adding 07 more
    add dl,07h        ;else add 07

dispskip1:
    mov [rdi],dl        ;store ASCII code in buffer
    inc rdi            ;point to next byte
    loop dispup1        ;decrement the count of digits to display
            ;if not zero jump to repeat

    display dispbuff,4    ;

    ret
;***********************************************************
```

macro.asm

```
;************************************************************
;macro.asm
;macros as per 64 bit conventions

%macro accept 2
    mov    rax,0    ;read
    mov    rdi,0    ;stdin/keyboard
```

```
    mov    rsi,%1    ;buf
    mov    rdx,%2    ;buf_len
    syscall
%endmacro

%macro display 2
    mov    rax,1    ;print
    mov    rdi,1    ;stdout/screen
    mov    rsi,%1    ;msg
    mov    rdx,%2    ;msg_len
    syscall
%endmacro

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

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

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

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

myfile.txt

"Welcome!!!"
Computer Engineering
Sinhgad Institute of Technology & Science Narhe,Pune

;******************Output***************************************
;[root@localhost A5_Far]# nasm -f elf64 A5_file1.asm
;[root@localhost A5_Far]# nasm -f elf64 A5_file2.asm
;[root@localhost A5_Far]# ld -o A5_file1 A5_file1.o A5_file2.o
;[root@localhost A5_Far]# ./A5_file1

;ML assignment 05 :- String Operation using Far Procedure
;---

;Enter filename for string operation : myfile.txt

;Enter character to search : e

;No. of spaces are : 0007
;No. of lines are : 0003
;No. of character occurances are : 000B

;Exit from program...

;[root@localhost A5_Far]#
;

CONCLUSION

Hence we conclude that we have find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character from a text file.

Assignment No. 13

TITLE: Find factorial of a given integer number on a command line by using recursion.

OBJECTIVES:

8. To be familiar with the format of assembly language program along with different assembler directives and different functions of the DOS Interrupt.
9. To learn the instructions related to 80386.
10. To be familiar with Math Coprocessor.
4. Implement factorial of a integer number.

PROBLEM DEFINITION:

Write 80386 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.

SOFTWARE HARDWARE REQUIRED:

CPU: Intel i5 Processor

OS: Ubuntu 14 (64 bit) & 64 bit execution

Editer: gedit, GNU Editor

Assembler: NASM (Netwide Assembler)

Linker: GNU Linker

INPUT: Input data specified in program**.**
OUTPUT: Display the data present in destination block.
THEORY:

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

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

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

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

Instructions needed:

1. AND-AND each bit in a byte or word with corresponding bit in another byte or word
2. INC-Increments specified byte/word by1
3. DEC-Decrements specified byte/word by1
4. JG - The *command JG* simply means: Jump if Greater.
5. CMP-Compares to specified bytes or words
6. MUL - The MUL (Multiply) instruction handles unsigned data
7. CALL-Transfers the control from callingprogramto procedure.
8. ADD- ADD instructions are used for performing simple addition of binary data in byte, word and doubleword size, i.e., for adding 8-bit, 16-bit or 32-bit operands, respectively.
9. RET-Return from where call is made

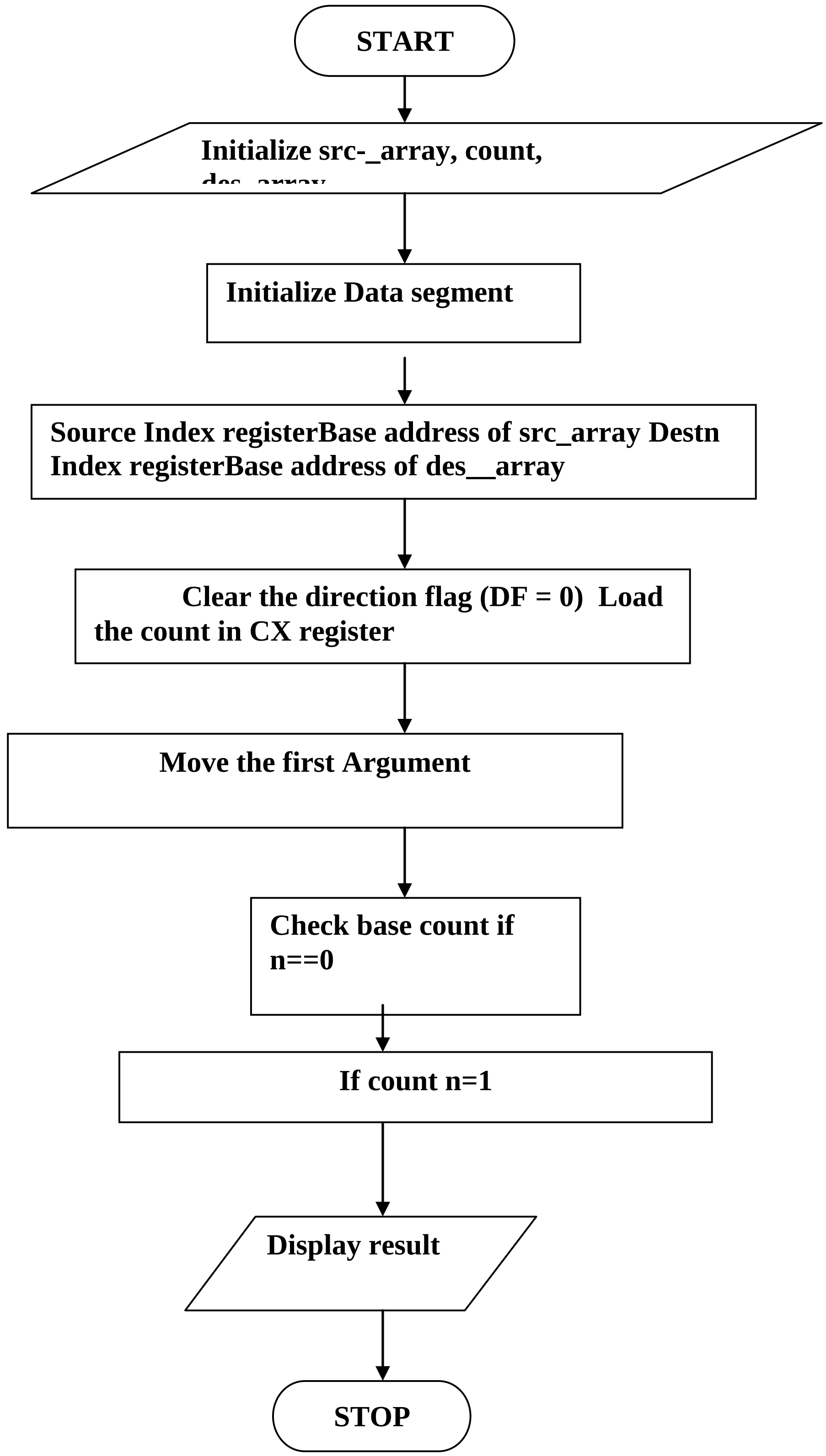
ALGORITHM:

This algorithm use recursive approach to find factorial of N.

1. Start
2. Read: Take input N
3. Retrieve parameter and put it into Register-PUSH
4. Check for base case if n==0
5. move the first argument to %eax
6. If the number is 1, that is our base case, and we simply return.
7. multiply by the result of the last call to factorial.
8. Restore %ebp and %esp to where they were before the function started.
9. return to the function

FLOWCHART:

To find factorial of number


START
Initialize src-_array, count,
Initialize Data segment
Source Index registerBase address of src_array Destn Index registerBase address of des__array
Clear the direction flag (DF = 0) Load the count in CX register
Move the first Argument
Check base count if n==0
If count n=1
Display result
STOP

MATHEMATICAL MODEL:

Let S = { s , e , X, Y, | Φs }

The factorial function is formally defined by the product $n! = \prod_{k=1}^{n} k$

This notation works in a similar way to **summation notation** (Σ), but in this case we multiply rather than add terms. For example, if $n = 4$, we would substitute $k = 1$, then $k = 2$, then $k = 3$ and finally $k = 4$ and write:

$$4! = \prod_{k=1}^{4} k = 1 \times 2 \times 3 \times 4 = 24$$

n math, we often come across the following expression:

$$n!$$

This is "*n* factorial", or the product

$$n(n-1)(n-2)(n-3)\ \dots\ (3)(2)(1).$$

CONCLUSION

Hence we conclude that we can perform ALP to find out factorial of a given integer number.

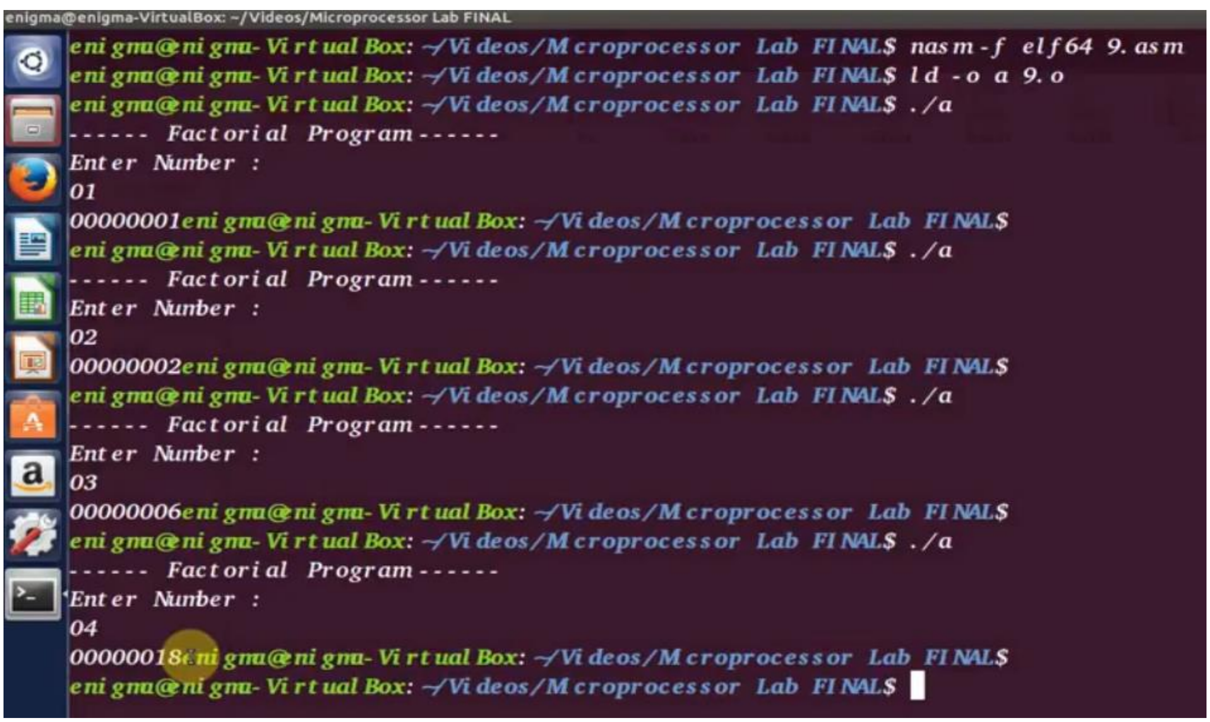
Program:

```
%macro scall 4   ;macro for read/write system call
mov rax,%1
mov rdi,%2
mov rsi,%3
mov rdx,%4
syscall
%endmacro
;--------------- DATA SECTION -----------------------
Section .data
title:db "------ Factorial Program ------",0x0A
     db "Enter Number : ",0x0A
title_len: equ $-title
factMsg: db "Factorial is :", 0x0A
factMsg_len: equ $-factMsg
cnt: db 00H
cnt2:db 02H
num_cnt: db 00H
;--------------- BSS SECTION -------------------------
```

```
Section .bss
number:resb 2
factorial:resb 8
;--------------- TEXT SECTION -------------------------
Section .text
global _start
_start:
scall 1,1,title,title_len
scall 0,0,number,2

mov rsi,number      ;convert no.from ascii to hex
call AtoH   ;converted number is stored in "bl"

FACTORIAL:
call fact_proc
mov rbx,rax
mov rdi,factorial
call HtoA_value
scall 1,1,factorial,8

;Exit System call
exit:
mov rax,60
mov rdi,0
syscall
;------------ FACT PROCEDURE ------
fact_proc:
cmp bl,01H
jne do_calc
mov ax,1
ret
do_calc:
push rbx;
dec bl
call fact_proc
pop rbx
mul bl
ret
;------------- ASCII to HEX Conversion Procedure ---------------------
AtoH:       ;result hex no is in bl
mov byte[cnt],02H
mov bx,00H
hup:
rol bl,04
mov al,byte[rsi]
cmp al,39H
JBE HNEXT
SUB al,07H
HNEXT:
sub al,30H
add bl,al
INC rsi
DEC byte[cnt]
```

```
JNZ hup
ret
;------HEX TO ASCII CONVERSION METHOD FOR VALUE(2 DIGIT) ----------------
HtoA_value: ;hex_no to be converted is in ebx //result is stored in rdi/user defined variable
mov byte[cnt2],08H
aup1:
rol ebx,04
mov cl,bl
and cl,0FH
CMP CL,09H
jbe ANEXT1
ADD cl,07H
ANEXT1:
add cl, 30H
mov byte[rdi],cl
INC rdi
dec byte[cnt2]
JNZ aup1
ret
;------------- END PROGRAM -----------------------------
```

Output:



Assignment No. 14

Titile: Write X86/64 ALP password program that operates as follow: a. Do not display what is actully type instaed displayed asterisk ("*") . If password is correct diplay, "Acess is granted" else display "Acess is not granted".

Program:

```
;References:
;https://www.gnu.org/software/libc/manual/html_node/Local-Modes.html
;https://man7.org/linux/man-pages/man3/termios.3.html
;https://www.csie.ntu.edu.tw/~comp03/nasm/nasmdoc3.html


;syscalls numbers
sys_exit    equ 60
sys_read    equ 0
sys_write   equ 1
sys_ioctl   equ 16

stdin       equ 0
stdout      equ 1


;the flags for the c_lflag member of the struct termios structure
ICANON      equ 1<<1
ECHO        equ 1<<3

section .data       ;Section for initialised variables
        welmsg          db 10,'Welcome to MicroSIG',10
        welmsg_len      equ $-welmsg

        entmsg          db 10,'Please Enter Password: '
        entmsg_len      equ $-entmsg

        WrPass          db 10,10,10,'Access Not Granted',10,10
        WrPass_len      equ $-WrPass

        CrPass          db 10,10,10,'Access Granted',10,10
        CrPass_len      equ $-CrPass

        astr_char       db  "*"

        Password        db '123456'
        Password_len    equ $-Password


section    .bss
        BufIn           resb   2
        PassIn          resb   24
        termios         resb   36
```

```
%macro write 2
        mov rax,sys_write
        mov rdi,stdout
        mov rsi,%1
        mov rdx,%2
        syscall
%endmacro

section .text

  global _start
_start:
        write welmsg,welmsg_len

        call    echo_off
        call    canonical_off

        write entmsg,entmsg_len

        xor     rbx, rbx
.GetCode:
        cmp     rbx, 24
        je      .Continue

        mov     rax, sys_read
        mov     rdi, stdin
        lea     rsi, [BufIn]
        mov     rdx, 1
        syscall

        mov     al, byte [BufIn]
        cmp     al, 10
        je      .Continue

        mov     byte [PassIn + rbx], al
        write astr_char,1
        inc     rbx
        jmp     .GetCode

.Continue:
        cmp rbx,Password_len
        jne .WrongPW

        mov rcx,rbx
        lea rsi,[Password]
        lea rdi,[PassIn]
        repe cmpsb
        je .CorrectPW
```

```
.WrongPW:
        write WrPass, WrPass_len
        jmp .Done
                        ;~ #
.CorrectPW:
        write CrPass,CrPass_len

.Done:
  call    echo_on
  call    canonical_on

  mov     rax, sys_exit
  xor     rdi, rdi
  syscall


;~ ##########################################
canonical_off:
  call read_stdin_termios

  ; clear canonical bit in local mode flags
  and dword [termios+12], ~ICANON

  call write_stdin_termios
  ret

;~ ##########################################
echo_off:
  call read_stdin_termios

  ; clear echo bit in local mode flags
  and dword [termios+12], ~ECHO

  call write_stdin_termios
  ret

;~ ##########################################
canonical_on:
  call read_stdin_termios

  ; set canonical bit in local mode flags
  or dword [termios+12], ICANON

  call write_stdin_termios
  ret

;~ ##########################################
echo_on:
  call read_stdin_termios

  ; set echo bit in local mode flags
```

```
  or dword [termios+12], ECHO

  call write_stdin_termios
  ret

;~ ##########################################
read_stdin_termios:
  mov rax, sys_ioctl
  mov rdi, stdin
  mov rsi, 0x5401
  lea rdx, [termios]
  syscall
  ret

;~ ##########################################
write_stdin_termios:
  mov rax, sys_ioctl
  mov rdi, stdin
  mov rsi, 0x5402
  lea rdx, [termios]
  syscall
     ret
```