Kalyani Charitable Trust's
Late G. N. Sapkal College of Engineering, Nashik-422212
DEPARTMENT OF COMPUTER ENGINEERING

Microprocessor Lab

(210257)

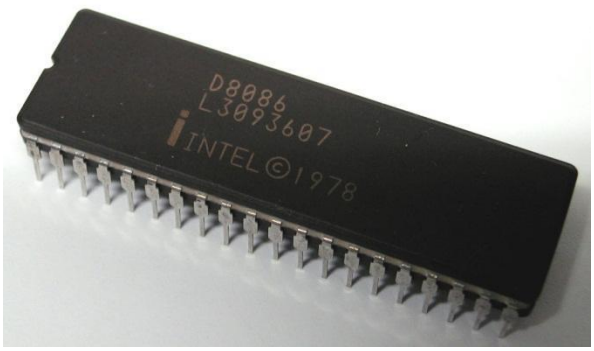
For

SE-2019 Pattern
(Computer Engineering)

Teaching scheme Examination scheme

Practical: 2 Hrs. /week

Oral : 25 Marks
Term work : 25 Marks



SR. No.	List of Laboratory Experiments/Assignments
1.	Write X86/64 ALP to count number of positive and negative numbers from the array
2.	Write an X86/64 ALP to count number of positive and negative numbers from the array.
3.	Write an X86/64 ALP to accept a string and to display its length.
4.	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).
5.	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.
6.	Write X86/64 ALP to perform non-overlapped block transfer without string specificinstructions. Block containing data can be defined in the data segment.
7.	Write X86/64 ALP to perform overlapped block transfer with string specific instructions . Block containing data can be defined in the data segment.
8.	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).
9.	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
10.	Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of aparticular 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.

1. Introduction to Assembly Level Language

Assembly level language is the language which contains machine level instructions to perform certain task.

The following are the list of assembly language development tools:

- ➢ Editor
- ➢ Assembler
- ➢ Linker
- ➢ Debugger

- ➢ **Editor:**
  - Contains assembly language statements (mnemonics)
  - Examples: Notepad, WordPad and DOS editor
  - The editor stores ASCII codes for each mnemonic of the program
  - The program typed in the editor is contained in the text file which is called a source file and will be saved with .asm extension
- ➢ **Assembler:**
  - An assembler is software used to "translate" assembly language mnemonics written in the editor to binary code. The following process takes place in the assembler
  - The assembler reads source file from memory
  - On the first pass, the assembler determines the displacement of labels and put information in "symbol table".
  - On the second pass, it (i.e. assembler) produces binary for each instruction.
  - Once the above process is over, the assemble generates two files:
    1. Object file with .obj extension which contains pure binary code
    2. List file with .lst extension which contains mnemonics, labels and address location of each instruction. The list file is normally used to correct the programs.

- ➢ **Linker:**
  - A linker is a software program used to combine several object files in to one large file. This is useful in case of large programs
- ➢ **Debugger:**
  - The debugger allows us to execute the program and to troubleshoot or "debug" the program.
  - It allows us to look at the contents of registers and memory locations after program execution.
  - It allows us to execute the program step by step or instruction by instruction.

There are several software's available which contain all the above tools. For example MASM (Macro Assembler by Microsoft), TASM (Turbo Assembler by Borland International) andASM6.0 (Assembler by Intel Corporation) these tools are helpful in successfully executing assembly language program statements.

2. Assembler Directives

Assembler directives are special instructions that provide information to the assembler regarding certain aspect of instructions. The directives are part of mnemonics. But they are not converted into binary code when executed

The following are assembler directives:

- **ASSUME:**

Is used to tell the assembler the name of the logical segment it should use for a specified segment. For example

ASSUME CS: CODE, DS: DATA

- **DB: Define Byte**

Is used to declare a byte type variable or to reserve one or more memory locations with byte type. For example

num1 db 42H;

Nums db 42H, 43H, 78H;

String1 db 'tajmahal' (this will declare an array of 8 bytes and load this string in that array)

Array1 db 50dup (?) (Reserve 50 bytes of memory with name array1)

Array2 db 50dup(0) (reserve 50 bytes of memory with name array2and initialize each location with 0)

Like **DB** there are several other directives with similar functionality they are:

- **DD - Define Double Word**
- **DW - Define Word**
- **DQ - Define Quad Word**
- **END - End of the Program**
- **ENDS - End of the Program**
- **ENDP - End of the Procedure**

- **PROC - Procedure**
  - Is used to identify the start of a procedure
  - Ended with ENDP For example:
  - **Addition Proc Near/Far** (Addition is a name of a procedure which can be far or near)
- **SEGMENT**
  - Is used to indicate the start of logical segment
  - Ended with ENDS

3. Structures of Assembly Language Program

Structure 1

```
.Model [Type]
.Stack 100H
.Data
        Data declaration
                .
        Data declaration
                .
        Data declaration
.Code
Start:
        Instructions
        Instructions
        Instructions
        Instructions

        Procedure1
        .
        .
        Procedure ENDP
End Start
```

Structure 2

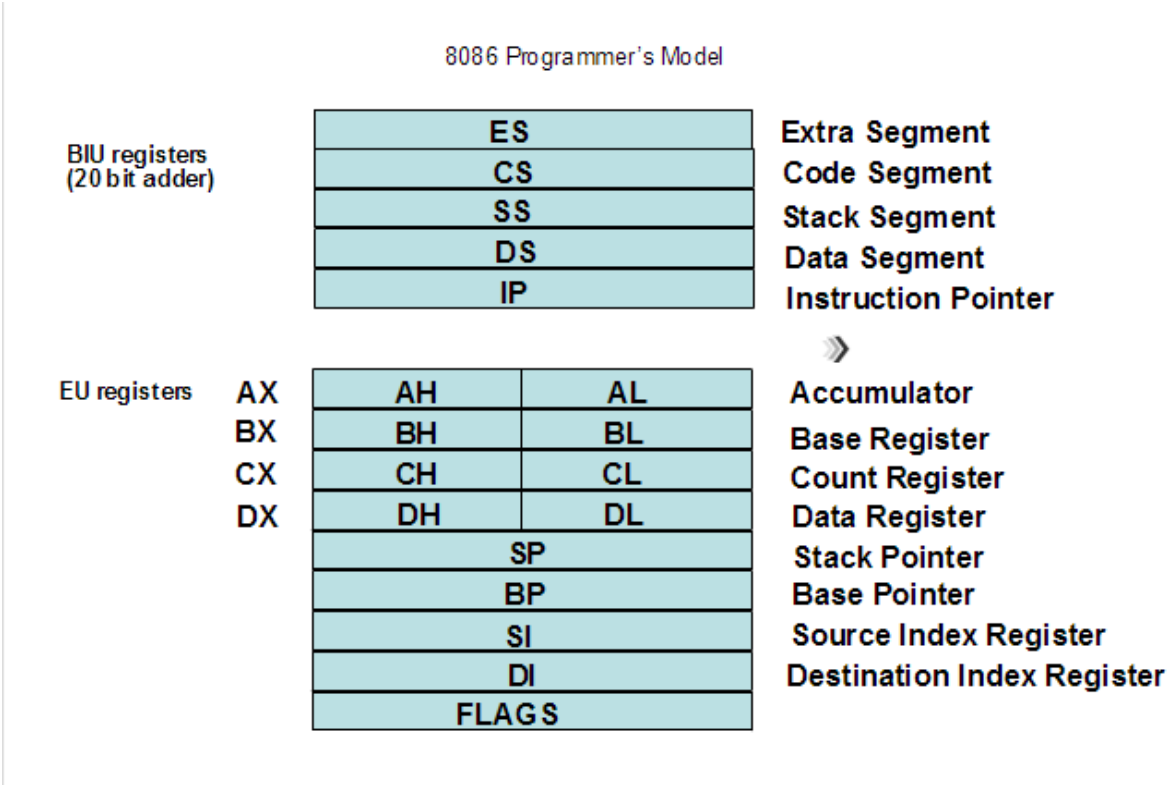
```
Data Segment
        Data declaration
                .
        Data declaration
                .
        Data declaration
Data Ends

Code Segment
Assume cs: code, ds: data
Start:
        Instructions
        Instructions
        Instructions
        Instructions

        Procedure1
        .
        .
        Procedure ENDP
End Start
Code Ends
```

4. Programmers Model of 80386

The 8086 programmer's model is the picture of the processor available to the programmer. Those are the registers used to hold the temporary data and addresses as well as indicate status and act as controls

➢ Data Registers:

- Four 16 bit data registers are available and can be accessed by names AX, BX, CX and DX
- Individual 8 bit registers are also can be used as AH, AL, BH, BL, CH, CL, DH and DL



➢ Address registers

- Hold the address of an instruction or data element
- Segment registers (CS, DS, ES, SS)
- Pointer registers (SP, BP, IP)
- Index registers (SI, DI)

5. Addressing Modes of 80386

Addressing mode indicates a way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes, or some instruction may not belong to any of the addressing modes. Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction. Here, we will present the addressing modes of the instructions depending upon their types. According to the flow of instruction execution, the instructions may be categorized as

(i) Sequential control flow instructions and

(ii) Control transfer instructions.

Sequential control flow instructions are the instructions, which after execution, transfer control to the next instruction appearing immediately after it (in the sequence) in the program. For example, the arithmetic, logical, data transfer and processor control instructions are sequential control flow instructions. The control transfer instructions, on the other hand, transfer control to some predefined address somehow specified in the instruction after their execution. For example, INT, CALL, RET and JUMP instructions fall under this category.

The addressing modes for sequential control transfer instructions are explained as follows:

1. Immediate: In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

Example: MOV AX, 0005H

In the above example, 0005H is the immediate data. The immediate data may be 8-bit or 16-bit in size.

2. Direct: In the direct addressing mode, a 16-bit memory address (offset) is directly specified in the instruction as a part of it.

Example: MOV AX, [5000H]

Here, data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the offset address and content of DS as segment address. The effective address, here, is 10H*DS+5000H.

3. Register: In register addressing mode, the data is stored in a register and it is referred using the particular register. All the registers, except IP, may be used in this mode.

Example: MOV BX, AX.

4. Register Indirect: Sometimes, the address of the memory location, which contains data or operand, is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or DI registers. The default segment is either DS or ES. The data is supposed to be available at the address pointed to by the content of any of the above registers in the default data segment.

Example: MOV AX, [BX]

Here, data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H*DS+ [BX].

5. Indexed: In this addressing mode, offset of the operand is stored in one of the index registers. DS and ES are the default segments for index registers SI and DI respectively. This mode is a special case of the above discussed register indirect addressing mode.

Example: MOV AX, [SI]

Here, data is available at an offset address stored in SI in DS. The effective address, in this case, is computed as 10H*DS+ [SI].

6. Register Relative: In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment. The example given before explains this mode.
   Example: MOV Ax, 50H [BX]
   Here, effective address is given as 10H*DS+50H+ [BX].

7. Based Indexed: The effective address of data is formed, in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

Example: MOV AX, [BX] [SI]

Here, BX is the base register and SI is the index register. The effective address is computed as 10H*DS+ [BX] + [SI].

8. Relative Based Indexed: The effective address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the bases registers (BX or BP) and any one of the index registers, in a default segment.

Example: MOV AX, 50H [BX] [SI]

Here, 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as 160H*DS+ [BX] + [SI] + 50H.

For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or a different one. It also depends upon the method of passing the destination address to the processor. Basically, there are two addressing modes for the control transfer instructions, viz. inter-segment and intra-segment addressing modes.

If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called inter-segment mode. If the destination location lies in the same segment, the mode is called intra-segment.

Addressing Modes for Control Transfer Instruction

9. Intra-segment direct mode: In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8 bits (i.e. –128<d<+128), we term it as short jump and if it is of 16 bits (i.e. –32768<+32768), it is termed as long jump.

10. Intra-segment Indirect Mode: In this mode, the displacement to which the control is to be transferred is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a

register or a memory location. This addressing mode may be used in unconditional branch instructions.

11. Inter-segment Direct Mode: In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

12. Inter-segment Indirect Mode: In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e. contents of a memory block containing four bytes, i.e. IP (LSB), IP (MSB), CS (LSB) and CS (MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

Assignment No: 1
Assignment Name: Write X86/64 ALP to count number of positive and negative numbers from the array .

Objective: 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.

Software Requirement: OS: Ubuntu Assembler: NASM version 2.10.07 Linker: ld

Theory :
Mathematical numbers are generally made up of a sign and a value (magnitude) in which the sign indicates whether the number is positive, (+) or negative, (–) with the value indicating the size of the number, for example 23, +156 or -274. Presenting numbers is this fashion is called "sign-magnitude" representation since the left most digit can be used to indicate the sign and the remaining digits the magnitude or value of the number.
Sign-magnitude notation is the simplest and one of the most common methods of representing positive and negative numbers either side of zero, (0). Thus negative numbers are obtained simply by changing the sign of the corresponding positive number as each positive or unsigned number will have a signed opposite, for example, +2 and -2, +10 and -10, etc.
But how do we represent signed binary numbers if all we have is a bunch of one's and zero's. We know that binary digits, or bits only have two values, either a "1" or a "0" and conveniently for us, a sign also has only two values, being a "+" or a "–".
Then we can use a single bit to identify the sign of a signed binary number as being positive or negative in value. So to represent a positive binary number (+n) and a negative (-n) binary number, we can use them with the addition of a sign.
For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is "0", this means the number is positive in value. If the sign bit is "1", then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number in the usual unsigned binary number format way.
Then we can see that the Sign-and-Magnitude (SM) notation stores positive and negative values by dividing the "n" total bits into two parts: 1 bit for the sign and n–1 bits for the value which is a pure binary number.

Algorithm:
1. Define variables in data segment
2. Display message on screen "Enter 10 numbers to be added" using function 09 of INT 21H
3. Accept 5 numbers from user by calling accept procedure
4. Store those numbers in an array in memory
5. Initialize loop counter to zero
6. Read a number with respect to loop counter from memory; store it temporarily in some register
7. Increment loop counter
8. Read next number from memory
9. Check if number is less than zero then increase negative count.
10. Else if number is greater than zero then increase positive count.
11. Check whether all 5 numbers are read & checked; if yes go to step 12; if no go to step 7
12. Display result by calling display procedure

13. Stop.

Procedure for accept numbers: (ASCII to HEX)

1. Read a single character/digit from keyboard using function 01 of INT 21H
2. Convert ASCII to HEX as per following:
3. Compare its ASCII with 39
4. If it is <= 39 then go to d else go to step c
5. Subtract 07 from it
6. Subtract 30 from it
7. Store the resultant digit
8. Check whether four digits (16-bit number) or two digits (8-bit number) are read; if yes then go to step g else go to step 1 for next digit
9. Return to main routine
10. End of accept procedure.

Procedure for display Result: (HEX to ASCII)

1. Compare 4 bits (one digit) of number with 9
2. If it is <= 9 then go to step 4 else go to step 3
3. Add 07 to that number
4. Add 30 to it
5. Display character on screen using function 02 of INT 21H
6. Return to main routine
7. End of display procedure.

Interrupts Used:

INT 21H with following functions:
09H – Display Messages
01H – Accept Single Character from User
02H – Display Single Character of the Result
4CH – Terminate the program.

PROGRAM

```
%macro scall 4
mov eax,%1
mov ebx,%2
mov ecx,%3
mov edx,%4
int 80h
%endmacro

section .data
arr dd -18888888h,18888888h,22222222h,11111111h,-33333333h
n equ 5
pmsg db 10d,13d,"The Count of Positive No: ",10d,13d
plen equ $-pmsg
nmsg db 10d,13d,"The Count of Negative No: ",10d,13d
nlen equ $-nmsg
```

```
nwline db 10d,13d

section .bss
pcnt resq 1
ncnt resq 1
char_answer resb 8

section .text
global _start
_start:
mov esi,arr
mov edi,n
mov ebx,0
mov ecx,0

up:
mov eax,[esi]
cmp eax,00000000h
js negative

positive:       inc ebx
jmp next
negative:       inc ecx

next:    add esi,4
dec edi
jnz up

mov [pcnt],ebx
mov [ncnt],ecx

scall 4,1,pmsg,plen
mov eax,[pcnt]
call display

scall 4,1,nmsg,nlen
mov eax,[ncnt]
call display
scall 4,1,nwline,1
mov eax,1
mov ebx,0
int 80h


;display procedure for 32bit
display:
mov esi,char_answer+7
mov ecx,8

cnt:    mov edx,0
mov ebx,16h
```

```
div ebx
cmp dl,09h
jbe add30
add dl,07h
add30:    add dl,30h
mov [esi],dl
dec esi
dec ecx
jnz cnt
scall 4,1,char_answer,8
ret
```

Conclusion:

Assignment No: 2
Assignment Name: Write an X86/64 ALP to accept a string and to display its length.
Objective: To learn and understand the operation of accept and display string.
Software Requirement: Ubuntu , NASM etc.
Theory :
The x86 String Instructions: All members of the x 86 families support five different string instructions: MOVS, CMPS, SCAS, LODS and STOS. They are the string primitives since you can build most other string operations from these five instructions.
How the String Instructions Operate: The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the MOVS instruction moves a sequence of bytes from one memory location to another. The CMPS instruction compares two blocks of memory. The SCAS instruction scans a block of memory for a particular value. These string instructions often require three operands a destination block address a source block address and (optionally) an element count. For example, when using the MOVS instruction to copy a string you need a source address a destination address and a count (the number of string elements to move).
Unlike other instructions which operate on memory the string instructions are single-byte instructions which don't have any explicit operands. The operands for the string instructions include
1. The SI (source index) register
2.The DI (destination index) register
3.The CX (count) register
4.The AX register and
5.The direction flag in t e FLAGS register.

For example one variant of the MOVS (move string) instruction copies a string from the source address specified by DS:SI to the destination address specified by ES:DI of length CX. Likewise, the CMPS instruction compares the string pointed at by DS:SI of length CX to the string pointed at by ES: DI.
Not all instructions have source and destination operands (only MOVS and CMPS support them).For example, the SCAS instruction (scan a string) compares the value in the accumulator to values in memory. Despite their differences the 80x86's string instructions all have one thing in common - using them requires that you deal with two segments the data segment and the extra segment.
Field: Label repeat mnemonic operand; comment
For MOVS:
REP MOVS
For CMPS:
REPE CMPS REPZ
REPNE REPNZ CMPS
For SCAS:
REPE SCAS
REPZ SCAS
REPNE SCAS
REPNZ SCAS
For STOS:
REP STOS {operands}

Program:

```
section .data
m2 db 10,10,'Enter the string:'              ;enter message
m2_len equ $-m2                  ;calculate length of m2 message
m3 db 'Length of the string is:'        ;length
m3_len equ $-m3

section .bss
srcstr resb 20   ; resb:reserve a byte
count resb 1
dispbuff resb 2

%macro disp 2    ; 2 means number of parameter
mov eax,4     ;4=Write System call
mov ebx,1     ;file descriptor:1:Stdout
mov ecx,%1     ;%1 means parameter
mov edx,%2   ;%2 Means parameter
int 80h       ;call to kernel and tell about write system call and descriptor
%endmacro

%macro acceptstr 1
mov eax,03  ;3=read system call
mov ebx,0    ;file descriptor:0:stdin
mov ecx,%1
int 80h    ;store user string in al register call to kernel and tell about read system call
%endmacro

section .text
global _start
_start:

disp m2,m2_len    ;call disp macro
acceptstr srcstr     ;call acceptstr
dec al                    ;decrement pointer i.e. pointer goes to first location
mov [count],al       ;store al contents in count location

disp m3,m3_len   ;call disp macro
mov bl,[count]    ;
call display

mov eax,01      //start execution after ret instruction
int 80h

display:
mov cl,2                  ; 2 means length of output number
mov edi,dispbuff

d1:
rol bl,4      ;rotated by 4 bits
mov al,bl    ;move content from bl to al
and al,0Fh  ;anding with al contents and 0Fh:0000 1111 h
cmp al,09    ;compare al contents with 09
jbe d2    ; jump below equal to d2
add al,07  ;otherwise add al and 07
```

```
d2:
add al,30h   ;add al and 30h,30h=48 decimal=0
mov [edi],al
inc edi   ;edi is incremented
dec cl   ;cl is 2 ,then,1 then 0
jnz d1

disp dispbuff,2
ret  ;return to after display call label
```

Output : The output shows us actual length of string without using inbuilt string Function.

Conclusion: In this way we studied about accepting and displaying strings using 80386 microprocessors.

Assignment No: 3
Assignment Name: Write 8086 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.

Objective: To learn the implementation stack for its conversion of number.

Software Requirement: Ubuntu ,NASM etc.

Theory :
1. Memory: model directives
These directives instruct the assembler as to how large the various segment (code, data, stack, etc) can be and what sort of segmentation register will be required.
.MODEL
<model >
Where <model>is one of the following options:
Tiny- code and data fit into single 64k and accessed via near pointers.
Small- code and segment both less than 64k and accessed via near pointers. Compact – Code segment is <64K (near ptr) and data segment is <1MB (far ptr) Medium- Code segment is <1Mb (far ptr) and data segment is <64K (near ptr) Large- code and data segment both less than 1MB and accessed via far pointers. Huge- Like "large" model, but also permits arrays larger then 64K.

Segment directives

These directives indicate to assembler the order in which to load segment. When it encounter one of these directives, it interprets all subsequent instructor as belonging to the indicated segment (until the next directives is encounter).
. data
.stack <size>; specified .code

Data type declaration

As has been previously discussed, data can be of several different lengths and assembler must be able to decide what length a specific constant (or variable) is. This can be down using data type declaration in conjunction with a constant declaration or variable assignment .this is akin to strong typing of variable in high level language .the data types are:
Byte (8 bit quantity)—synonyms are byte and db
Word (16) -- synonyms are word dw
Dword (32bit) -- synonyms are dword and dd
Qword (64bit) -- synonyms with dq
Tword (128bit) -- synonyms with dt An
example of their use is:
MOV AX, word VAR; moves a 16-bit Variable VAR into AX

Instruction Used:

1. PUSH:-Push word onto stack.
2. POP:-Pop word off stack.
3. DIV:-Divide byte/word

4. XOR: - Exclusive or byte/word
5. JA/JNBE:-Jump if above/not below or equal
6. JB/JNAE:-Jump if below /not above or equal
7. JG/JNLE:-Jump if /not less nor equal
8. JL/JNGE:-Jump if less /not greater nor equal
9. INT:-It allows ISR to be activated by programmer & external H\W device

New interrupt used:

1 INT 21h, function 0AH:- Read from keyboard and place into a memory buffer a row of character, until<CR>is pressed.

New directives used:

1. .MODEL
2. .STACK
3. .DATA
4. .CODE
5. .OFFSET: - It informs the assembler to determine the offset/displacement of a named data item.
6. .PTR: - assign a specific type to variable/label

Algorithm:

Title program to find out the BCD number of the given hexadecimal number

;procedures:- input (to accept the hex number),
;display(for displaying the number)
;input to the program:- hexadecimal number
;output of the program:-equivalent BCD number

Program:

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

section .data
      menu db 10d,13d,"            MENU"
          db 10d,"1. Hex to BCD"
          db 10d,"2. BCD to Hex"
          db 10d,"3. Exit"
          db 10d,"Enter your choice: "
      menulen equ $-menu
      m1 db 10d,13d,"Enter Hex Number: "
      l1 equ $-m1
      m2 db 10d,13d,"Enter BCD Number: "
```

```
        l2 equ $-m2

        m3 db 10d,13d,"Equivalent BCD Number: "
        l3 equ $-m3
        m4 db 10d,13d,"Equivalent Hex Number: "
        l4 equ $-m4

section .bss
        choice resb 1
        num resb 16
        answer resb 16
        factor resb 16

section .code
        global _start
_start:

        scall 1,1,menu,menulen
        scall 0,0,choice,2

        cmp byte[choice],'3'
        jae exit
        cmp byte[choice],'1'
        je hex2bcd
        cmp byte[choice],'2'
        je bcd2hex

;**********Hex to BCD Conversion*************************
hex2bcd:
        scall 1,1,m1,l1
        scall 0,0,num,17
        call asciihextohex

        mov rax,rbx
           mov rbx,10
           mov rdi,num+15
loop3:
           mov rdx,0
           div rbx
           add dl,30h
           mov [rdi],dl
           dec rdi
           cmp rax,0
           jne loop3

        scall 1,1,m3,l3
        scall 1,1,num,16
jmp _start

;**********BCD to Hex Conversion*************************
bcd2hex:
```

```
        scall 1,1,m2,l2
        scall 0,0,num,17

        mov rcx,16
           mov rsi,num+15
           mov rbx,0
           mov qword[factor],1

loop4:
           mov rax,0
           mov al,[rsi]
        sub al,30h
           mul qword[factor]
           add rbx,rax
           mov rax,10
           mul qword[factor]
           mov qword[factor],rax
           dec rsi
           loop loop4

           scall 1,1,m4,l4
           mov rax,rbx
           call display
jmp _start

exit:
        mov rax,60
        mov rdx,0
        syscall

;**********************PROCEDURES**************************
asciihextohex:
           mov rsi,num
           mov rcx,16
           mov rbx,0
           mov rax,0

loop1:  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 loop1
ret

display:
        mov rsi,answer+15
```

```
        mov rcx,16
loop2:  mov rdx,0
        mov rbx,16
        div rbx
        cmp dl,09h
        jbe skip2
        add dl,07h
skip2:  add dl,30h
        mov [rsi],dl
        dec rsi
        dec rcx
        jnz loop2
        scall 1,1,answer,16
ret
```

Conclusion: In this way we studied about hex to BCD and hex to BCD number conversion.

Assignment No: 4

Assignment Name: Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW registers also identify CPU type using CPUID instruction.

Objective: 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.

Software Requirement: OS: Ubuntu Assembler: NASM version 2.10.07 Linker: ld

Theory : Real Mode:

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

Protected Mode:

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

When a processor that supports x86 protected mode is powered on, it begins executing instructions in real mode, in order to maintain backward compatibility with earlier x86 processors. Protected mode may only be entered after the system software sets up several descriptor tables and enables the Protection Enable (PE) bit in the control register 0 (CR0).

Explanation:

1. **GDTR:** Global Descriptor Table Register
2. **LDTR:** Local Descriptor Table Register
3. **IDTR :** Interrupt Descriptor Table Register
4. **TR:** Task Register
5. **MSW:** MSW Register

Algorithm :

1. Start
2. Display the message using sys_ write call
3. Read CR0
4. Checking PE bit, if 1=Protected Mode
5. Load number of digits to display
6. Rotate number left by four bits
7. Convert the number in ASCII
8. Display the number from buffer
9. Exit using sys_ exit call

Program:

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

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

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

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

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

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

mswmsg db 10,"Machine Status Word::"
mswmsg_len equ $-mswmsg
colmsg db ":"
nwline db 10

section .bss

gdt resd 1
resw 1
ldt resw 1
idt resd 1
resw 1
tr  resw 1
cr0_data resd 1
dispbuff resb 04

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

section .text
global _start
_start:
 smsw eax     ;Stores the MSW (bits 0 through15 of control register CR0) into eax.
```

```
    mov [cr0_data],eax

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

prmode: display pmodemsg,pmsg_len

nxt1:sgdt [gdt]
     sldt [ldt]
     sidt [idt]
     str [tr]
    ;.......display gdt data.....

     display gdtmsg,gdtmsg_len

     mov bx,[gdt+4]
     call display16_proc

     mov bx,[gdt+2]
     call display16_proc

     display colmsg,1

     mov bx,[gdt]
     call display16_proc

    ;......display ldt data.....

     display ldtmsg,ldtmsg_len
     mov bx,[ldt]
     call display16_proc

    ;......display idt data.....

    display idtmsg,idtmsg_len

    mov bx,[idt+4]
    call display16_proc

    mov bx,[idt+2]
    call display16_proc

    display colmsg,1

    mov bx,[idt]
    call display16_proc

    ;....display task register data.....
```

```
    display trmsg,trmsg_len

    mov bx,[tr]
    call display16_proc

    ;....display machine status word data.....

    display mswmsg,mswmsg_len

    mov bx,[cr0_data+2]
    call display16_proc

    mov bx,[cr0_data]
    call display16_proc

    display nwline,1

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

Conclusion: Hence we performed an ALP to program to use GDTR, LDTR and IDTR in Real Mode.

Assignment No: 5
Assignment Name: Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.

Objective: Understand the memory Addressing.
Understand the localization of Data.

Software Requirement: OS: Ubuntu Assembler: NASM version 2.10.07 Linker: ld

➢➢ **Explanation: with string specific instruction**

- Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
- Let the number of bytes N = 10.
- We will have to initialize this as count in the CX register.
- We know that source address is in the SI register and destination address is in the DI register.
- Clear the direction flag.
- Using the string instruction move the data from source location to the destination location. It is assumed that data is moved within the same segment. Hence the DS and ES are initialized to the same segment value.
- Display the contents using display routine.

Theory :
Note that segments can overlap. This means that two different logical addresses can refer to the same physical address (aliasing).
In non-overlapping method, address of source is totally different from address of destination. Therefore, we can directly transfer the data using MOVSB/MOVSW instruction for transferring the data.
The blocks are said to be overlapped if some of the memory locations are common for both the blocks.
Case I: If address in SI is greater than address in DI then start the data transfer from last memory location keeping DF=1.
Case II: If address in SI is less than address in DI, then start the data transfer from first memory location by keeping DF=0.

Algorithm for procedure for non overlapped block transfer without string instruction:

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

Program:

```
section .data
   sourceBlock db 12h,45h,87h,24h,97h
   count equ 05
```

```
    msg db "ALP for non overlapped block transfer using string instructions : ",10
    msg_len equ $ - msg

    msgSource db 10,"The source block contains the elements : ",10
    msgSource_len equ $ - msgSource

    msgDest db 10,10,"The destination block contains the elements : ",10
    msgDest_len equ $ - msgDest

    bef db 10, "Before Block Transfer : ",10
    beflen equ $ - bef

    aft db 10,10 ,"After Block Transfer : ",10
    aftlen equ $ - aft

section .bss
    destBlock resb 5
    result resb 4

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

section .text
    global _start

_start:

     write msg , msg_len

     write bef , beflen

     write msgSource , msgSource_len
     mov rsi,sourceBlock
     call dispBlock

     write msgDest , msgDest_len
     mov rsi,destBlock
     call dispBlock
```

```
    mov rsi,sourceBlock
    mov rdi,destBlock

    mov rcx, count
    cld
    rep movsb

    write aft , aftlen

    write msgSource , msgSource_len
    mov rsi,sourceBlock
    call dispBlock

    write msgDest , msgDest_len
    mov rsi,destBlock
    call dispBlock

    mov rax,60
    mov rdi,0
    syscall

dispBlock:
        mov rbp,count
   next:mov al,[rsi]
        push rsi
        call disp
        pop rsi
        inc rsi
        dec rbp
        jnz next
   ret

disp:
       mov bl,al ;store number in bl
       mov rdi, result ;point rdi to result variable
       mov cx,02 ;load count of rotation in cl
up1:
       rol bl,04 ;rotate number 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 grater 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 the count of digits to display
        jnz up1 ;if not zero jump to repeat

        write result , 4
        ret
```

Assignment No: 6
Assignment Name: Write X86/64 ALP to perform overlapped block transfer with string specific instructions. Block containing data can be defined in the data segment.

Objective: Understand the memory Addressing.
Understand the localization of Data.

Software Requirement: OS: Ubuntu Assembler: NASM version 2.10.07 Linker: ld

Theory : Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
Let the number of bytes N = 05.
We will have to initialize this as count.
Overlap the source block and destination block.

We know that source address is in the ESI register and destination address is in the EDI register.

For block transfer without string instruction, move contents at ESI to accumulator and from accumulator to memory location of EDI and decrement ESI and EDI for next content transfer.
For block transfer with string instruction, set the direction flag. Move the data from source location to the destination location using string instruction.

Algorithm for procedure for overlapped block transfer with string instruction:
1. Initialize ESI and EDI with source and destination address.
2. Move count in ECX register.
3. Move source block's and destination block's last content address in ESI and EDI.
4. Set the direction flag.
5. Move the data from source location to the destination location using string instruction.
6. Repeat string instruction the number of times indicated in the count register.

Program:

```
section .data

msg0 db 10,"1.nonoverlap with string"
db 10,"2.nonoverlap without string"
db 10,"3.overlap with string"
db 10,"4.overlap without string"
db 10,"5.exit"
db 10,"Enter your choice ->"
len0 equ $-msg0
msg1 db 10,"Source block contents are "
len1 equ $-msg1
msg2 db 10,"Destination block contents are "
len2 equ $-msg2
array db 01h,02h,03h,04h,05h,00h,00h,00h,00h,00h
array2 times 5 db 0
space db 20h
cnt equ 05
```

```
section .bss

dispbuff resb 4
var resb 1

%macro display 2
mov eax,4
mov ebx,1
mov ecx,%1
mov edx,%2
int 80h

%endmacro

section .code
global _start
_start:

display msg0,len0
mov eax,3
mov ebx,0
mov ecx,var
mov edx,2
int 80h
cmp byte [var],'5'
je exit

display msg1,len1
mov ecx,cnt
mov esi,array

bk:
push ecx
mov bl,[esi]
call display1_proc
display space,1
inc esi
pop ecx
loop bk

display msg2,len2

cmp byte [var],'1'
je case1
cmp byte [var],'2'
je case2
cmp byte [var],'3'
je case3
cmp byte [var],'4'
je case4
```

```
case1:

mov esi,array
mov edi,array2
mov ecx,cnt
cld
rep movsb
mov esi,array2
mov ecx,5
jmp end

case2:

mov esi,array
mov edi,array2
mov ecx,cnt

nover:

mov bl,[esi]
mov [edi],bl
inc esi
inc edi
loop nover
mov esi,array2
mov ecx,5
jmp end

case3:

mov esi,array+5
mov edi,array+7
mov ecx,cnt
inc ecx
std
rep movsb
mov esi,array
mov ecx,7
jmp end

case4:

mov esi,array+5
mov edi,array+7
mov ecx,cnt
inc ecx

overl:

mov bl,[esi]
mov [edi],bl
```

```
dec esi
dec edi
loop overl
mov esi,array
mov ecx,7
jmp end

end:

x1:
push ecx
mov bl,[esi]
call display1_proc
display space,1
inc esi
pop ecx
loop x1
jmp _start

exit:

mov eax,01
mov ebx,00
int 80h

display1_proc:

mov ecx,4
mov edi,dispbuff
d1:
rol bx,4
mov al,bl
and al,07h
cmp al,09
jbe dskip
add al,07h

dskip:

add al,30h
mov [edi],al
inc edi
loop d1
display dispbuff,4
ret
```

Conclusion: Hence we performed overlapped block transfer with string specific instructions.

Assignment No: 7

Assignment Name: Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. Accept input from the user. (use of 64-bit registers is expected)

Objective: To learn and understand shift & rotate Instruction.

Software Requirement: Ubuntu ,NASM etc.

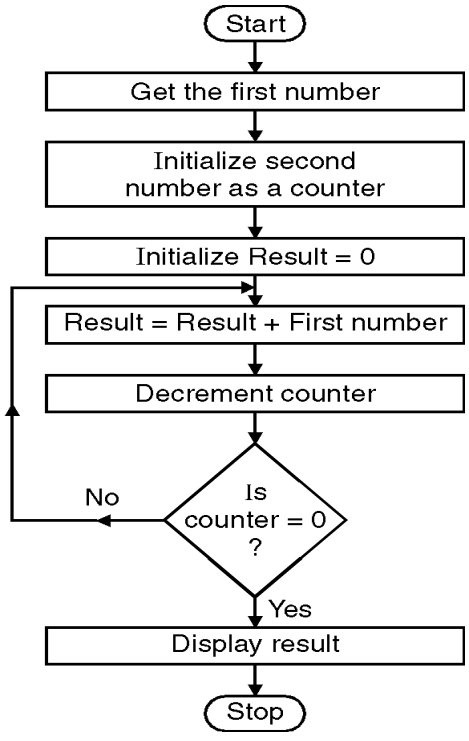
➢ **Explanation: Successive Addition Method**

- Consider that a byte is present in the AL register and second byte is present in the BL register.
- We have to multiply ht byte in AL with the byte in BL.
- We will multiply the numbers using successive addition method.
- In successive addition method, one number is accepted other number is taken as a counter. The first number is added with itself, till the counter decrement to zero.
- Result will store in DX register. Display the result, using display routine.
  For example: AL= 12, BL= 10
  Result=12H+12H+12H+12H+12H+12H+12H+12H+12H+12H
  Result=0120H

Algorithm:

Step I : Initialize the data segment.
Step II: Get the first number.
Step III: Get the second number as counter.
Step IV: Initialize result= 0
Step V: Result= Result + First number.
Step VI: Decrement counter.
Step VII: If counter = 0, go to step V.
Step VIII: Display the result.
Step IX: Stop.

Flow Chart:

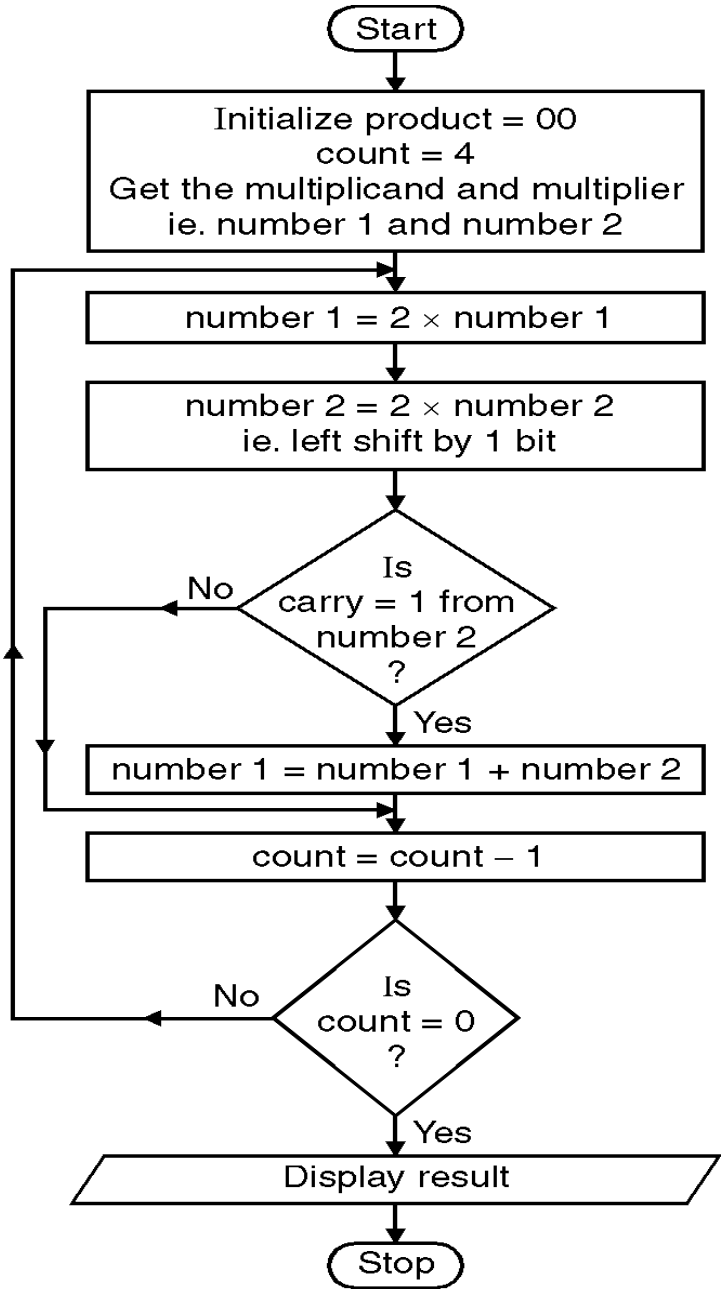


➢➢ **Explanation: Add & shift Method**

- Consider that one byte is present in the AL register and another byte is present in the BL register.
- We have to multiply the byte in AL with the byte in BL.
- We will multiply the numbers using add and shift method. In this method, you add number with itself and rotate the other number each time and shift it by one bit to left along with carry. If carry is present add the two numbers.
- Initialize the count to 4 as we are scanning for 4 digits. Decrement counter each time the bits are added. The result is stored in AX. Display the result.

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

Flowchart:



; sample program of 8-bit multiplier

```
section .data
msg  db 10d,13d,"                MENU For Multiplication"
    db 10d,"1. Successive Addition"
    db 10d,"2. Shift and Add method"
    db 10d,"3. Exit"
    db 10d
    db 10d,"Enter your choice: "
msglen equ $-msg

msg1 db 10,13,"Enter first 4 digit hexadecimal no:"
msg1len equ $-msg1

msg2 db 10,13,"Enter second 4 digit hexadecimal no:"
msg2len equ $-msg2

msg3 db 10,13,"Result by first method is:"
msg3len equ $-msg3

msg4 db 10,13,"Result by second method is:"
msg4len equ $-msg4

section .bss
choice resb 2
abc resb 8
num1 resb 5
num2 resb 5
cnt resb 1
num resb 5
number resb 4

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

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



section .text
```

```
global _start
_start:

display msg,msglen
accept choice,2
mov al,[choice]
sub al,30h

cmp al,01
je successive_addition

cmp al,02
je add_rol

cmp al,03
je exit

successive_addition:

display msg1,msg1len
accept num,5
call ascii_original

mov [num1],rbx

display msg2,msg2len
accept num,5
call ascii_original

mov [num2],rbx

mov rbx,0
mov rax,0  ;used for sum variable
mov rdx,0    ;used for carry

mov rbx,[num1]
mov rcx,[num2]
l11:
add rax,rbx
jnc l12
inc rdx    ;ie carry=carry+1

l12:
dec rcx
jnz l11

mov rbx,0

call original_ascii
jmp _start
```

```
add_rol:
display msg1,msg1len

accept num,5
call ascii_original
mov [num1],bx

display msg2,msg2len
accept num,5
call ascii_original
mov [num2],bx

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

mov cx,16

mov ax,[num1]
mov bx,[num2]

l15:
;add rax,rax

shl ax,1
jnc l66
add ax,bx
l66:    dec cl
jnz l15
;mov bx,ax
call original_ascii

exit:
mov rax,60
mov rbx,0
syscall

ascii_original:
mov esi,num
mov ecx,4
mov bx,0

l2:
rol bx,4
mov al,[esi]
cmp al,39h
jbe l3
sub al,07h

l3:
sub al,30h
```

```
mov ah,0
add bx,ax
inc esi
loop l2

ret

original_ascii:
mov ecx,4
mov esi,number

l4:
rol ax,4
mov dl,al
and dl,0fh
cmp dl,09h
jbe l5
add dl,07h
l5:
add dl,30h
mov [esi],dl
inc esi
loop l4
display number,4
ret
```

Conclusion: In this way we studied shifting operation and multiplication using successive addition and add-shift method.

Assignment No: 8

Assignment Name: Write X86 menu driven Assembly Language Program (ALP) to implement OS (DOS) commands TYPE, COPY using file operations. User is supposed to provide command line arguments in all cases.

Objective: To learn and understand TYPE, COPY commands.

Software Requirement: Ubuntu ,NASM etc.

Program:

```
section .data

fname1: db 'abc.txt',0
fname2: db 'def.txt',0
fname3: db 'efg.txt',0

menu:  db "------ MENU ------",0x0A
       db "1.TYPE",0x0A
       db "2.COPY",0x0A
       db "3.DELETE",0x0A
       db "4.Exit",0x0A
menulen: equ $-menu

choice: db "Enter Your Choice",0x0A
ch_len: equ $-choice

msg: db "file1 opened successfully",0x0A
len: equ $-msg

msg1: db "file1 closed successfully",0x0A
len1: equ $-msg1

msg2: db "error in opening file",0x0A
len2: equ $-msg2

msg3: db "Content copied in destination file",0x0A
len3: equ $-msg3

msg4: db "file2 opened successfully",0x0A
len4: equ $-msg4

msg5: db "file2 closed successfully",0x0A
len5: equ $-msg5

msg6: db "file3 deleted successfully",0x0A
len6: equ $-msg6

new: db "",0x0A
new_len: equ $-new
```

```
;_____________________________________________________________________________
_____________________________________
section .bss

cho: resb 2
fd1: resb 17
fd2: resb 17
buffer1: resb 200
buffer2: resb 200
buf_len1: resb 17
buf_len2: resb 17


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

;_____________________________________________________________________________
_____________________________________________
section .text
global _start
_start:

scall 1,1,menu,menulen
scall 1,1,choice,ch_len
scall 0,1,cho,2

cmp byte[cho],31h
je case1

cmp byte[cho],32h
je case2

cmp byte[cho],33h
je case3

cmp byte[cho],34h
je case4

case1:
call type
jmp _start

case2:
call copy
jmp _start
```

```
case3:
call delete
jmp _start

case4:
jmp exit
;_____________________________________________________________________
_____
type:

mov rax,2
mov rdi,fname1
mov rsi,2
mov rdx,0777
syscall

mov qword[fd1],rax
BT rax,63
jc next
scall 1,1,msg,len
jmp next2
next:
scall 1,1,msg2,len2

next2:
scall 0,[fd1],buffer1, 200
mov qword[buf_len1],rax
dec qword[buf_len1]
scall 1,1,buffer1,200

mov rax,3
mov rdi,[fd1]
syscall
scall 1,1,msg1,len1
ret
;_____________________________________________________________________
_____
copy:

mov rax,2
mov rdi,fname1
mov rsi,2
mov rdx,0777
syscall

mov qword[fd1],rax
BT rax,63
jc next3
scall 1,1,msg,len
jmp next4
next3:
```

```
scall 1,1,msg2,len2

next4:
scall 0,[fd1],buffer1, 200
mov qword[buf_len1],rax
dec qword[buf_len1]

mov rax,2
mov rdi,fname2
mov rsi,2
mov rdx,0777
syscall

mov qword[fd2],rax
BT rax,63
jc next5
scall 1,1,msg4,len4
jmp next6
next5:
scall 1,1,msg,len

next6:
scall 1,[fd2],buffer1,200

scall 1,1,msg3,len3

mov rax,3
mov rdi,[fd1]
syscall
scall 1,1,msg1,len1

mov rax,3
mov rdi,[fd2]
syscall
scall 1,1,msg5,len5
ret
;_____
_____

mov rax,87
mov rdi,fname3
syscall
scall 1,1,msg6,len6
ret

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

Conclusion: In this way we studied implementation of TYPE & COPY OS commands.

Assignment No: 9

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

Objective: 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.

Software Requirement: OS: Ubuntu Assembler: NASM version 2.10.07 Linker: ld

Explanation :

1. **Public Directive: (Global Directive)**
   This directive is use for Export the elements from source file to destination file.
2. **Extern Directive:**
   This directive is use for Import the elements from source file to destination file.

Main Algorithm:
A1: Algorithm for program_1
i. Start
ii. Initialize all the sections needed in programming.
iii. Display “Enter file name” message using Print macro expansion.
iv. Accept file name using Accept macro and store in filename buffer.
v. Display “Enter character to search” message with the expansion of Print macro
vi. Read character using Accept macro expansion..
vii. Open file using fopen macro.
viii. Compare RAX with -1H if equal then display error message “Error in Opening File”. with Print macro expansion else go to step ix.
ix. Read content of opened file in buffer.
x. Store file length in abuf_ len.
xi. Call far_ procedure.
xii. xii. Stop.

Program :

Program_1.asm

```
section .data

global msg6,len6,scount,ncount,chacount,new,new_len

fname: db 'abc.txt',0

msg: db "file opened successfully",0x0A
len: equ $-msg

msg1: db "file closed successfully",0x0A
```

```
len1: equ $-msg1

msg2: db "error in opening file",0x0A
len2: equ $-msg2

msg3: db "No of spaces are",0x0A
len3: equ $-msg3

msg4: db "No of enters are",0x0A
len4: equ $-msg4

msg5: db "enter the character",0x0A
len5: equ $-msg5

msg6: db "No of occurance of character",0x0A
len6: equ $-msg6

new: db "",0x0A
new_len: equ $-new

scount: db 0
ncount: db 0
ccount: db 0
chacount: db 0

section .bss

global cnt,cnt2,cnt3,buffer

fd: resb 17
buffer: resb 200
buf_len: resb 17
cnt: resb 2
cnt2: resb 2
cnt3: resb 2
cha: resb 2

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

section .text
global _start
_start:

extern spaces,enters,occ
```

```
mov rax,2
mov rdi,fname
mov rsi,2
mov rdx,0777
syscall

mov qword[fd],rax
BT rax,63
jc next
scall 1,1,msg,len
jmp next2
next:
scall 1,1,msg2,len2

next2:
scall 0,[fd],buffer, 200
mov qword[buf_len],rax
mov qword[cnt],rax
mov qword[cnt2],rax
mov qword[cnt3],rax

scall 1,1,msg3,len3
call spaces
scall 1,1,msg4,len4
call enters

scall 1,1,msg5,len5
scall 0,1,cha,2
mov bl, byte[cha]
call occ
jmp exit
```

exit:

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

Program_2.asm:

```
section .data
extern msg6,len6,scount,ncount,chacount,new,new_len

section .bss
extern cnt,cnt2,cnt3,scall,buffer

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

```
%endmacro

section .text
global _start1
_start1:

global spaces,enters,occ

spaces:
mov rsi,buffer
up:
mov al, byte[rsi]
cmp al,20H
je next3
inc rsi
dec byte[cnt]
jnz up
jmp next4
next3:
inc rsi
inc byte[scount]
dec byte[cnt]
jnz up
next4:
add byte[scount], 30h
scall 1,1,scount, 2
scall 1,1,new,new_len
ret

enters:
mov rsi,buffer
up2:
mov al, byte[rsi]
cmp al,0DH
je next5
inc rsi
dec byte[cnt2]
jnz up2
jmp next6
next5:
inc rsi
inc byte[ncount]
dec byte[cnt2]
jnz up2
next6:
add byte[ncount], 30h
scall 1,1,ncount, 2
scall 1,1,new,new_len
ret

occ:
```

```
mov rsi,buffer
up3:
mov al, byte[rsi]
cmp al,bl
je next7
inc rsi
dec byte[cnt3]
jnz up3
jmp next8
next7:
inc rsi
inc byte[chacount]
dec byte[cnt3]
jnz up3
next8:
add byte[chacount], 30h
scall 1,1,msg6,len6
scall 1,1,chacount, 1
scall 1,1,new,new_len
ret
```

Conclusions:

Assembly Level Program to find,
a) Number of Blank spaces
b) Number of lines
c) Occurrence of a particular character is assembled and executed successfully.

Assignment No: 10

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

Objective: 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.

Software Requirement: OS: Ubuntu Assembler: NASM version 2.10.07 Linker: ld

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, by the equation −

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

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

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

```
procedure:
Recursive
proc
callRecursive
ret
Recursive endp
```

Of course the CPU will never execute the ret instruction at the end of this procedure. Upon entry into Recursive this procedure will immediately call itself again and control will never pass to the ret instruction. In this particular case run away recursion results in an infinite loop. In many respects recursion is very similar to iteration (that is the repetitive execution of a loop). The following code also produces an infinite loop:

```
Recursive
proc jmp
Recursive
ret
Recursive endp
```

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

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"

mov byte[num_cnt],bl
dec byte[num_cnt]   ;decrement count by 1 for stack top

mov rax,00H
mov al,bl
TOP:
push rax ;push every 0<value<number to stack
dec rax
```

```
cmp rax,01H
jnbe TOP

mov al,01H
FACTLOOP:
pop rbx
mul bx ;ax=ax*bx
dec byte[num_cnt]
jnz FACTLOOP

mov bx,ax ;copy result from ax to bx for HEX to ASCII Conversion
mov rdi,factorial
call HtoA_value

scall 1,1,factMsg,factMsg_len ;Print msg and factorial
scall 1,1,factorial,8

;Exit System call
mov rax,60
mov rdi,0
syscall

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

Conclusions:

Assembly Level Program to find the factorial of a given integer number on a command line by using recursion is assembled and executed successfully.