Lab Manual

for

Microprocessor Lab

5138

Diploma In Computer Engineering

5th Semester

by

SITTTR

Kalamassery

STATE INSTITUTE OF TECHNICAL TEACHERS TRAINING AND RESEARCH

GENERAL INSTRUCTIONS

Rough record and Fair record are needed to record the experiments conducted in the laboratory. Rough records are needed to be certified immediately on completion of the experiment. Fair records are due at the beginning of the next lab period. Fair records must be submitted as neat, legible, and complete.

INSTRUCTIONS TO STUDENTS FOR WRITING THE FAIR RECORD

In the fair record, the index page should be filled properly by writing the corresponding experiment number, experiment name , date on which it was done and the page number.

On the **right side** page of the record following has to be written:

1. **Title**: The title of the experiment should be written in the page in capital letters.
2. In the left top margin, experiment number and date should be written.
3. **Aim**: The purpose of the experiment should be written clearly.
4. **Apparatus/Tools/Equipments/Components used:** A list of the Apparatus/Tools /Equipments /Components used for doing the experiment should be entered.
5. **Principle**: Simple working of the circuit/experimental set up/algorithm should be written.
6. **Procedure:** steps for doing the experiment and recording the readings should be briefly described(flow chart/programs in the case of computer/processor related experiments)
7. **Results**: The results of the experiment must be summarized in writing and should be fulfilling the aim.
8. **Inference** : Inference from the results is to be mentioned.

On the **Left side** page of the record following has to be recorded:

1. **Circuit/Program**: Neatly drawn circuit diagrams/experimental set up.
2. **Design**: The design of the circuit/experimental set up for selecting the components should be clearly shown if necessary.
3. **Observations:** i) Data should be clearly recorded using Tabular Columns.

ii) Unit of the observed data should be clearly mentioned

iii) Relevant calculations should be shown. If repetitive calculations are needed, only show a sample calculation and summarize the others in a table.

4. **Graphs :** Graphs can used to present data in a form that show the results obtained, as one or more of the parameters are varied. A graph has the advantage of presenting large

amounts of data in a concise visual form. Graph should be in a square format.

GENERAL RULES FOR PERSONAL SAFETY

1. Always wear tight shirt/lab coat , pants and shoes inside workshops.

2. REMOVE ALL METAL JEWELLERY since rings, wrist watches or bands, necklaces, etc. make excellent electrodes in the event of accidental contact with electric power sources.

3. DO NOT MAKE CIRCUIT CHANGES without turning off the power.

4. Make sure that equipment working on electrical power are grounded properly.

5. Avoid standing on metal surfaces or wet concrete. Keep your shoes dry.

6. Never handle electrical equipment with wet skin.

7. Hot soldering irons should be rested in its holder. Never leave a hot iron unattended.

8. Avoid use of loose clothing and hair near machines and avoid running around inside lab .

TO PROTECT EQUIPMENT AND MINIMIZE MAINTENANCE:

DO: 1. SET MULTIRANGE METERS to highest range before connecting to an unknown source.

2. INFORM YOUR INSTRUCTOR about faulty equipment so that it can be sent for repair.

DO NOT: 1. Do not MOVE EQUIPMENT around the room except under the supervision of an instructor.

CONTENTS

EXP NO. 1 FAMILIARIZATION OF ASSEMBLER, DIRECTIVES AND SYSTEM INTERRUPTS

AIM

To familiarize with the NASM assembler, its directives, programming environment and system interrupts.

OBJECTIVES

- To understand the NASM assembler and its directives.
- To understand the syntax of the assembly language statements.
- To understand the assembling and linking process.
- To understand the x86 programming model.
- To understand the system calls.

PROCEDURE

INTRODUCTION

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. These set of instructions are called 'machine language instructions'. A processor understands only machine language instructions, which are strings of 1's and 0's. 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.

BASIC SYNTAX

An assembly program can be divided into three sections:

- The .data section,
- The .bss section, and
- The .text section.

The .data Section

The data section is used for declaring initialized data or constants. This data does not change at runtime. We 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 variables. The syntax for declaring bss section is:

```
section .bss
```

The .text section

The text section is used for keeping 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:
```

Comments

Assembly language comment begins with a semicolon (;). It may contain any printable character including blank. It can appear on a line by itself, like:

```
; This program displays a message on screen
```

or, on the same line along with an instruction, like:

```
add eax ,ebx    ; adds ebx to eax
```

Assembly Language Statements

Assembly language programs consist of three types of statements:

- Executable instructions or instructions,
- Assembler directives or pseudo-ops, and
- 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.

Syntax of Assembly Language Statements

Assembly language statements are entered one statement per line. Each statement follows the following format:

[label] mnemonic [operands] [;comment]

The fields in the square brackets are optional. A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic), which is to be executed, and the second are the operands or the parameters of the command.

Compiling and Linking an Assembly Program

Make sure to set the path of **nasm** and **ld** binaries in the PATH environment variable. Now, take the following steps for compiling and linking:

1. Type the program code using a text editor and save it as **filename.asm**.
2. Make sure that you are in the same directory as where you saved **filename.asm**.
3. To assemble the program, type

   nasm -f elf filename.asm

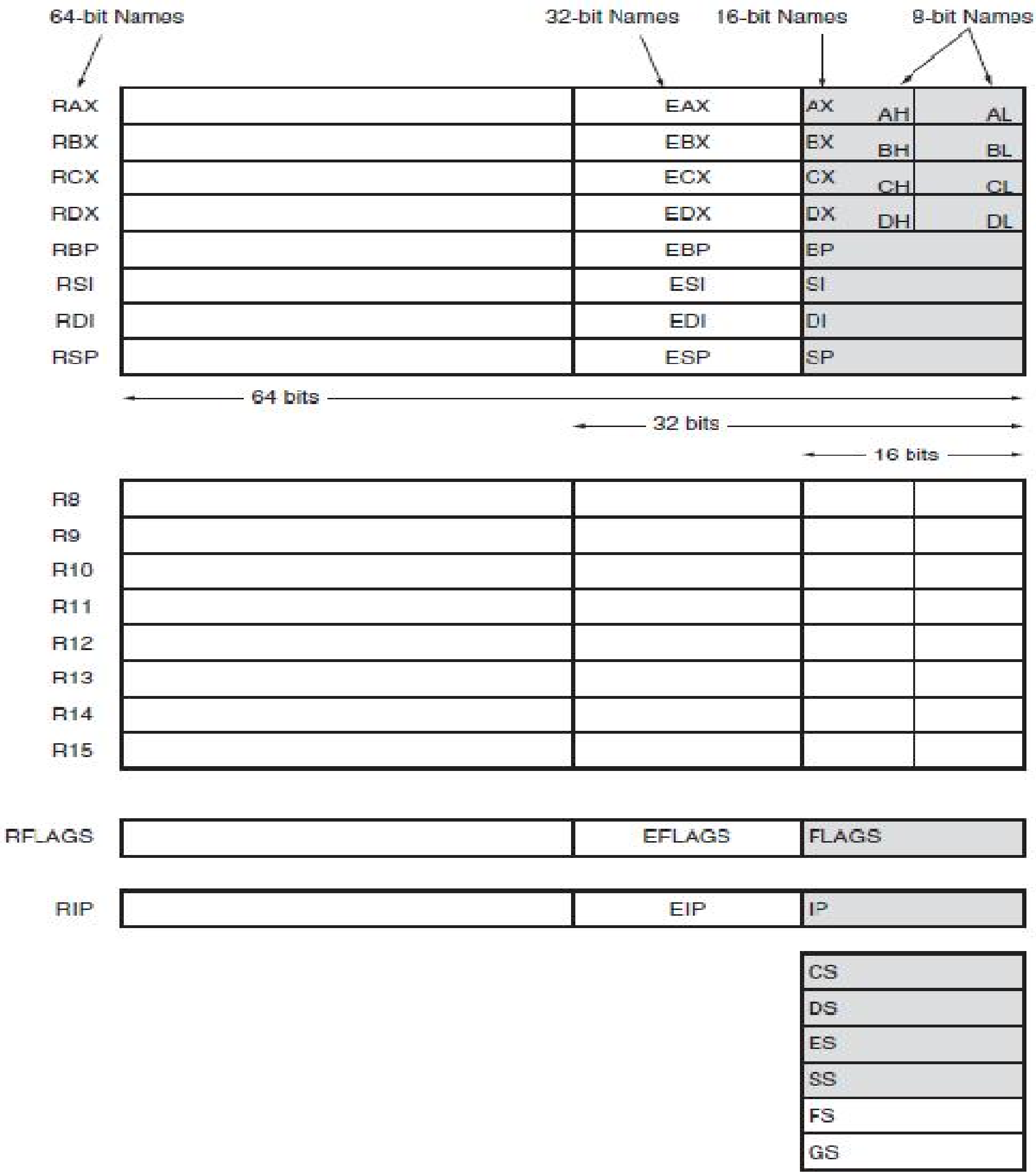
   or

   nasm –f elf –o filename.o filename.asm
4. If there is any error, it will be prompted about that at this stage. Otherwise, an object file of the program named **filename.o** will be created.
5. To link the object file and create an executable file, type

   ld --dynamic-linker /lib/ld-linux.so.2 –lc –o filename filename.o
6. Execute the program by typing **./filename**

THE PROGRAMMING MODEL

The programming model of the 8086 through the Core2 is considered to be **program visible** because its registers are used during application programming and are specified by the instructions. Other registers are considered to be **program invisible** because they are not addressable directly during applications programming, but may be used indirectly during system programming. Only the 80286 and above contain the program-invisible registers used to control and operate the protected memory system and other features of the microprocessor. Figure illustrates the programming model of the 8086 through the Core2 microprocessor including the 64-bit extensions.



Multipurpose Registers

RAX RAX is referenced as a 64-bit register (RAX), a 32-bit register **(accumulator)** (EAX), a 16-bit register (AX), or as either of two 8-bit registers (AH and AL). Note that if an 8- or 16-bit register is addressed, only that portion of the 32-bit register changes without affecting the remaining bits. The accumulator is used for instructions such as multiplication, division, and some of the adjustment instructions. For these instructions, the accumulator has a special purpose, but is generally considered to be a multipurpose register. In the 80386 and above, the EAX register may also hold the offset address of a location in the memory system. In the 64-bit Pentium 4 and Core2, RAX holds a 64-bit offset address, which allows 1T (terra) byte of memory to be accessed through a 40-bit address bus.

RBX RBX is addressable as RBX, EBX, BX, BH, or BL. The BX register **(base index)** sometimes holds the offset address of a location in the memory system in all versions of the microprocessor. In the 80386 and above, EBX also can address memory data. In the 64-bit Pentium 4 and Core2, RBX can also address memory data.

RCX RCX, which is addressable as RCX, ECX, CX, CH, or CL, is a **(count)** general-purpose register that also holds the count for various instructions. In the 80386 and above, the ECX register also can hold the offset address of memory data. In the 64-bit Pentium 4, RCX can also address memory data. Instructions that use a count are the repeated string instructions (REP/REPE/REPNE); and shift, rotate, and LOOP/LOOPD instructions. The shift and rotate instructions use CL as the count, the repeated string instructions use CX, and the LOOP/LOOPD instructions use either CX or ECX. If operated in the 64-bit mode, LOOP uses the 64-bit RCX register for the loop counter.

RDX RDX, which is addressable as RDX, EDX, DX, DH, or DL, is a **(data)** general-purpose register that holds a part of the result from a multiplication or part of the dividend before a division. In the 80386 and above, this register can also address memory data.

RBP RBP, which is addressable as RBP, EBP, or BP, points to a memory **(base pointer)** location in all versions of the microprocessor for memory data transfers.

RDI RDI, which is addressable as RDI, EDI, or DI, often addresses **(destination index)** string destination data for the string instructions.

RSI RSI is used as RSI, ESI, or SI. The source index register often **(source index)** addresses source string data for the string instructions. Like RDI, RSI also functions as a general-purpose register. As a 16-bit register, it is addressed as SI; as a 32-bit register, it is addressed as ESI; and as a 64-bit register, it is addressed as RSI.

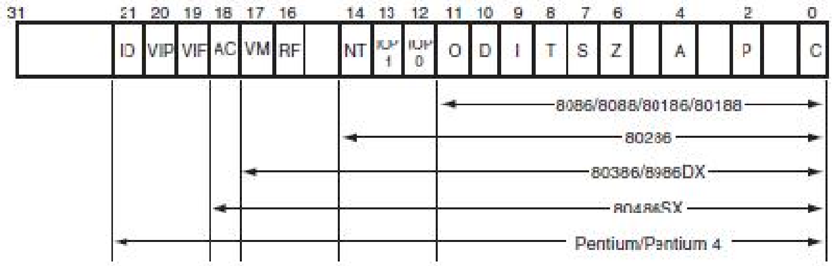
R8 through R15 These registers are only found in the Pentium 4 and Core2 if 64-bit extensions are enabled. As mentioned, data in these registers are addressed as 64-, 32-, 16-, or 8-bit sizes and are of general purpose. Most applications will not use these registers until 64-bit processors are common. Please note that the 8-bit portion is the rightmost 8-bit only; bits 8 to 15 are not directly addressable as a byte.

Special-Purpose Registers. The special-purpose registers include RIP, RSP, and RFLAGS; and the segment registers include CS, DS, ES, SS, FS, and GS.

RIP RIP addresses the next instruction in a section of memory defined as **(instruction pointer)** a code segment. This register is IP (16 bits) when the microprocessor operates in the real mode and EIP (32 bits) when the 80386 and above operate in the protected mode. Note that the 8086, 8088, and 80286 do not contain an EIP register and only the 80286 and above operate in the protected mode. The instruction pointer, which points to the next instruction in a program, is used by the microprocessor to find the next sequential instruction in a program located within the code segment. The instruction pointer can be modified with a jump or a call instruction. In the 64-bit mode, RIP contains a 40-bit address at present to address a 1T flat address space.

RSP RSP addresses an area of memory called the stack. The stack memory **(stack pointer)** stores data through this pointer and is explained later in the text with the instructions that address stack data. This register is referred to as SP if used as a 16-bit register and ESP if referred to as a 32-bit register.

RFLAGS RFLAGS indicate the condition of the microprocessor and control its operation. Figure shows the flag registers of all versions of the microprocessor. The 8086–80286 contain a FLAG register (16 bits) and the 80386 and above contain an EFLAG register (32-bit extended flag register). The 64-bit RFLAGS contain the EFLAG register, which is unchanged in the 64-bit version.



The rightmost five flag bits and the overflow flag change after many arithmetic and logic instructions execute. The flags never change for any data transfer or program control operation.

C (carry) Carry holds the carry after addition or the borrow after subtraction. The carry flag also indicates error conditions, as dictated by some programs and procedures

P (parity) Parity is a logic 0 for odd parity and a logic 1 for even parity. Parity is the count of ones in a number expressed as even or odd.

A (auxiliary carry) The auxiliary carry holds the carry (half-carry) after addition or the borrow after subtraction between bit positions 3 and 4 of the result. This highly specialized flag bit is tested by the DAA and DAS instructions to adjust the value of AL after a BCD addition or subtraction. Otherwise, the A flag bit is not used by the microprocessor or any other instructions.

Z (zero) The zero flag shows that the result of an arithmetic or logic operation is zero. Ifv1, the result is zero; if 0, the result is not zero.

S (sign) The sign flag holds the arithmetic sign of the result after an arithmetic or logic instruction executes. If 1, the sign bit (leftmost bit of a number) is set or negative; if 0, the sign bit is cleared or positive.

T (trap) The trap flag enables trapping through an on-chip debugging feature. If the T flag is enabled (1), the microprocessor interrupts the flow of the program on conditions as indicated by the debug registers and control registers. If the T flag is a logic 0, the trapping (debugging) feature is disabled.

I (interrupt) The interrupt flag controls the operation of the INTR (interrupt request) input pin. If 1, the INTR pin is enabled; if 0, the INTR pin is disabled. The state of the I flag bit is controlled by the STI (set I flag) and CLI (clear I flag) instructions.

D (direction) The direction flag selects either the increment or decrement mode for the DI and/or SI registers during string instructions. If 1, the registers are automatically decremented; if 0, the registers are automatically incremented. The D flag is set with the STD (set direction) and cleared with the CLD (clear direction) instructions.

O (overflow) Overflows occur when signed numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of the machine. The result represents an overflow condition indicated by the overflow flag for signed addition. For unsigned operations, the overflow flag is ignored.

Segment Registers. Additional registers, called segment registers, generate memory addresses when combined with other registers in the microprocessor. There are either four or six segment registers in various versions of the microprocessor. A segment register functions differently in the real mode when compared to the protected mode operation of the microprocessor. In the 64-bit flat model, segment registers have little use in a program except for the code segment register. Following is a list of each segment register, along with its function in the system:

CS (code) The code segment is a section of memory that holds the code (programs and procedures) used by the microprocessor. The code segment register defines the starting address of the section of memory holding code. In real mode operation, it defines the start of a 64Kbyte section of memory; in protected mode, it selects a descriptor that describes the starting address and length of a section of memory holding code. The code segment is limited to 64K bytes in the 8088–80286, and 4G bytes in the 80386 and above when these microprocessors operate in the protected mode. In the 64-bit mode, the code segment register is still used in the flat model, but its use differs from other programming modes.

DS (data) The data segment is a section of memory that contains most data used by a program. Data are accessed in the data segment by an offset address or the contents of other registers that hold the offset address. As with the code segment and other segments, the length is limited to 64K bytes in the 8086–80286, and 4G bytes in the 80386 and above.

ES (extra) The extra segment is an additional data segment that is used by some of the string instructions to hold destination data.

SS (stack) The stack segment defines the area of memory used for the stack. The stack entry point is determined by the stack segment and stack pointer registers. The BP register also addresses data within the stack segment.

FS and GS The FS and GS segments are supplemental segment registers available in the 80386–Core2 microprocessors to allow two additional memory segments for access by programs. Windows uses these segments for internal operations, but no definition of their usage is available.

DIRECTIVES

Allocating Storage Space for Initialized Data

The syntax for storage allocation statement for initialized data is:

[variable-name] define-directive initial-value [,initial-value]...

Where, *variable-name* is the identifier for each storage space. The assembler associates an offset value for each variable name defined in the data segment. 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. It 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

Defining Constants

The EQU Directive

The **EQU** directive is used for defining constants. The syntax is as follows:

CONSTANT_NAME EQU expression

The %assign Directive

The **%assign** directive can be used to define numeric constants like the EQU directive. This directive allows redefinition. The syntax is:

%assign CONSTANT_NAME expression

This directive is case-sensitive.

The %define Directive

The **%define** directive allows defining both numeric and string constants. This directive is similar to the #define in C. The syntax is:

%define CONSTANT_NAME expression

This directive also allows redefinition and it is case-sensitive.

SYSTEM CALLS

System calls are APIs for the interface between the user space and the kernel space.

Linux System Calls

We can make use of Linux system calls in our assembly programs. The following steps are needed for using Linux system calls in our 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 store 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 edx,4 ; message length
mov ecx,msg ; message to write
mov ebx,1 ; file descriptor (stdout)
mov eax,4 ; system call number (sys_write)
int 0x80 ; call kernel
```

RESULT

Familiarized with the assembler, directives and system calls.

EXP NO. 2 BYTE AND WORD DATA TRANSFER

AIM

Write a program to implement byte and word data transfer in different addressing modes.

OBJECTIVES

- To understand the use of data transfer instructions in various addressing modes.
- To understand the assembly language programming steps.

ALGORITHM

```
SECTION .data                ;  Section containing initialized data
        Data1   db   25h
        Data2   dw   1234h
        Data3   db   0h
        Data4   dw   0h
        Data5   dw   2345h,6789h

SECTION .bss                 ;  Section containing uninitialized data

SECTION .text                ;  Section containing code

        global _start        ;  Linker needs this to find the entry point!

_start:

        mov al,25            ;  copy 25h into 8 bit al register
        mov ax,2345          ;  copy 2345h into 16 bit ax register
        mov bx,ax            ;  copy the content of ax into bx register(16 bit)
        mov cl,al            ;  copy the content of al into cl register
        mov al,Data1         ;  copies the byte contents of data segment memory location
                             ;  Data1 into 8 bit al
```

```
mov ax,Data2            ; copies the word contents of data segment memory
                        ; location Data2 into 16 bit ax
mov Data3,al            ; copies the al content into the byte contents of data
                        ; segment memory location Data3
mov Data4,ax            ; copies the ax content into the word contents of data
                        ; segment memory location Data4
mov bx,offset Data5     ; the 16 bit offset address of ds memeory location Data5 is
                        ; copied into bx
mov ax,[bx]             ; copies the word content of data segment memory location
                        ; addressed by bx into ax(register indirect addressing)
mov di,02h              ; address element
mov ax,[bx+di]          ; copies the word content of data segment memory location
                        ; addressed by bx+di into ax(base plus indirect addressing)
mov ax,[bx+0002h]       ; copies the word content of data segment (16 bit)
mov al,[di+2]           ; register relative addressing
mov ax,[bx+di+0002h]    ; copies the word content of data segment memory location
                        ; addressed by bx+di+0002h into ax(16 bit)

mov eax,1               ; Specify Exit syscall
mov ebx,0               ; Return a code of zero
int 80H                 ; Make syscall to terminate the program
```

RESULT

Studied the use of data transfer instructions in various addressing modes.

EXP NO. 3 BLOCK TRANSFER

AIM

Write a program to transfer a block of data from one location to another.

OBJECTIVES

- To understand the use of data transfer instructions.
- To understand the looping in assembly language.

ALGORITHM

```
SECTION .data                       ;  Section containing initialized data
        X DB 01H,02H,03H,04H,05H    ;  Initialize Data Segments Memory Locations
        Y DB 05 DUP(0)

SECTION .text                       ;  Section containing code
        global _start               ;  Linker needs this to find the entry point!
_start:
        mov cx,05h          ;  Load counter
        lea si,X            ;  SI pointer pointed to top of the memory block
        lea di,Y            ;  DI pointed to the top of the destination block

Up:     mov bl,[si]         ;  Move the SI content to BL register
        mov [di],bl         ;  Move the BL register to content of DI
        inc si              ;  Update SI and DI
        inc di
        dec cx              ;  Decrement the counter till it becomes zero
        jnz Up

        mov eax,1           ;  Make syscall to terminate the program
        mov ebx,0
        int 80h
```

OBSERVATIONS

Before execution

X	**01**
	02
	03
	04
	05
Y	**00**
	00
	00
	00
	00

After execution

X	**01**
	02
	03
	04
	05
Y	**01**
	02
	03
	04
	05

RESULT

A block of data transferred from one location to another.

EXP NO. 4 ARITHMETIC OPERATIONS

AIM

Write a program to implement the basic arithmetic operations.

OBJECTIVES

- To understand the use of arithmetic instructions.

ALGORITHM

```
SECTION .data        ; Section containing initialized data
        Data1        dw   1234h
        Data2        dw   5678h
        Sum          dw   0h
        Diff         dw   0h
        Prod_Low     dw   0h
        Prod_High    dw   0h
        Quotient     dw   0h
        Reminder     dw   0h

SECTION .bss         ; Section containing uninitialized data

SECTION .text        ; Section containing code
        global _start
_start:
        mov ax, Data1          ; Copy Data1 to ax
        mov bx,  Data2         ; Copy Data2 to bx
        add ax,  bx            ; Perform addition
        mov Sum,  ax           ; Copy result to Sum

        mov ax,  Data1         ; Perform subtraction
        sub ax,  bx
        mov Diff,  ax
```

```
mov ax,  Data1                ; Perform multiplication
xor dx,dx
mul bx
mov Product_Low,  ax
mov Product_High,  dx

mov ax,  Data1                ; Perform division
xor dx,dx
div bx
mov Quotient,  ax
mov Reminder,  dx

mov eax,1                    ; Make system call to terminate the program
mov ebx,0
int 80H
```

OBSERVATIONS

Before execution

Data1	5678h
Data2	1234h
Sum	0h
Diff	0h
Prod_Low	0h
Prod_High	0h
Quotient	0h
Reminder	0h

After execution

Data1	5678h
Data2	1234h
Sum	68ach

Diff	4444h
Prod_Low	0060h
Prod_High	0626h
Quotient	0004h
Reminder	0da8h

RESULT

Studied the use of arithmetic instructions.

EXP NO. 5 ODD OR EVEN

AIM

Write a program to check whether the given number is odd or even.

OBJECTIVES

- To understand the use of rotate instruction.

ALGORITHM

```
SECTION .data                    ;  Section containing initialized data
        Num     db      0h
        Msg1    db      "Number is Odd$"
        Msg2    db      "Number is Even$"
SECTION .bss

SECTION .text
        global _start
_start:
;  Write code to read Num

;  Check odd or even using RCR
Check:
        mov al,  Num
        rcr al,1
        JC Print_Odd
Print_Even:
        ;  Write code to print Msg2
        jmp Exit
Print_Odd:
        ;  Write code to print Msg1
Exit:
        ;  System call to terminate the program.
```

OBSERVATIONS

1) Num 10
   Number is Even

2) Num 5
   Number is Odd

RESULT

Verified the given number is odd or even.

EXP NO. 6 MAXIMUM OF THREE NUMBERS

AIM

Write a program to find the maximum of three numbers.

OBJECTIVES

- To understand the use of compare instruction.
- To understand the use of conditional branch instructions.

ALGORITHM

```
; Data section begins
SECTION .data
        Value1      dd      40
        Value2      dd      20
        Value3      dd      30
        MaxValue    dd      0

SECTION .text
        global _start
_start:

;Write code here to read Value1,  Value2,  Value3

; Move the contents of variables
        mov ecx, [Value1]     ;  Copy first number into ecx
        cmp ecx, [Value2]     ;  Compare it with second number
        jg check_third_var    ;  If first number is large, go to next check
        mov  ecx, [Value2]    ;  Otherwise copy second number into ecx
check_third_var:
        cmp ecx, [Value3]     ;  Compare the largest with third number
        jg _exit              ;  Keep the largest in ecx
        mov ecx, [Value3]
```

```
_exit:
        mov MaxValue, ecx     ;   Store the largest in memory

;Write code here to display the largest number

        mov eax, 1            ; Terminate the program
        mov ebx, ecx
        int 80h
```

OBSERVATIONS

Before execution

Value1	40
Value2	20
Value3	30
MaxValue	0h

After execution

Value1	40
Value2	20
Value3	30
MaxValue	40

RESULT

Verified the largest among the given three numbers.

EXP NO. 7 PACKED BCD TO ASCII

AIM

Write a program to convert packed BCD to ASCII

OBJECTIVES

- To understand the use of shift instruction.
- To understand the use of logical instructions.

ALGORITHM

```
; Data section begins
SECTION .data
        Packed_BCD      db      45h
        ASCII_1         db      0h
        ASCII_2         db      0h

SECTION .bss

SECTION .text
        global _start

_start:

;  Write code to read packed BCD number

;  Convert packed BCD to unpacked BCD and then ASCIIa

Pbcd_ascii:
        mov al,Packed_BCD   ; Copy packed BCD to ALa
        and al,0f0h         ; Mask the higher digit
        mov cl,4            ; Convert to unpacked BCD
        shr al,cl
```

```
        or al,30h               ; Convert unpacked BCD to ASCII
        mov ASCII_1,  al

        mov al, Packed_BCD ; Convert lower digit to ASCII
        and al,0fh

        or al,30h
        mov ASCII_2,al

;  Write code here to display ASCII_1 and ASCII_2

;  Write code here to make system call to terminate the program
```

OBSERVATIONS

Before execution

Packed_BCD	45h
ASCII_1	0
ASCII_2	0

After execution

Packed_BCD	45h
ASCII_1	04
ASCII_2	05

RESULT

Verified the conversion of packed BCD to ASCII.

EXP NO. 8 ASCII TO PACKED BCD

AIM

Write a program to convert ASCII to packed BCD

OBJECTIVES

- To understand the use of shift instruction.
- To understand the use of logical instructions.

ALGORITHM

```
; Data section begins
SECTION .data
        ASCII_1         db      04h
        ASCII_2         db      05h
        Packed_BCD      db      0h

SECTION .bss

SECTION .text
        global _start
_start:
;  Write code to read ASCII digits

;  Convert ASCII to packed BCD

Ascii_pbcd:
        mov al,ASCII_1          ; Copy higher digit to AL
        sub al,30h              ; Convert to unpacked BCD
        mov cl,4                ; Shift the digit to higher nibble
        shl al,cl
        mov Packed_BCD, al
```

```
mov al, ASCII_2        ; Copy lower digit to AL
sub al,30h             ; Convert to unpacked BCD
or Packed_BCD, al      ; Convert to Packed BCD

; Write code here to display packed BCD

; Write code here to make system call to terminate the program
```

OBSERVATIONS

Before execution

ASCII_1	04
ASCII_2	05
Packed_BCD	00

After execution

ASCII_1	04
ASCII_2	05
Packed_BCD	45h

RESULT

Verified the conversion of ASCII to packed BCD.

EXP NO. 9 FACTORIAL

AIM

Write a program to find the factorial of a number.

OBJECTIVES

- To understand the use of branch/loop instruction.

ALGORITHM

```
SECTION .data
        N           dd      5h
        Fact_Low    dd      0h
        Fact_High   dd      0h

SECTION .bss

SECTION .text
        global _start
_start:
; Write code here to read N

        mov eax, N              ; Copy N into accumulator
        mov ecx, eax            ; Copy N-1 into counter register
        dec ecx
find_fact:
        mul ecx                 ; Find factorial as N!=Nx(N-1)x(N-2)x...x1
        loop find_fact
        mov Fact_Low, eax       ; Store factorial
        mov Fact_High, edx

;  Write code here to display the factorial

;  Write code here to terminate the program
```

OBSERVATIONS

Before execution

N	5
Fact_Low	0
Fact_High	0

After execution

N	5
Fact_Low	120
Fact_High	0

RESULT

Verified factorial of given number.

EXP NO. 10 STRING REVERSE

AIM

Write a program to read a string and find its reverse.

OBJECTIVES

- To understand string read operation.
- To understand string reverse operation.

ALGORITHM

```
extern printf
SECTION .data
        prompt      db   "Enter a string: ", 0
        prompt_len  equ  $-prompt
        format      db   "%s", 10, 0
        LEN         equ  50          ;  constant for string length

SECTION .bss
;; declare space for storing strings
        original_str: resb LEN
        reverse_str: resb LEN

SECTION .text
          global _start
_start:
;; read string from user. just call write and read system calls
;; call write
        mov eax, 4
        mov ebx, 1
        mov ecx, prompt
        mov edx, prompt_len
        int 80H
```

```
;; now read string
        mov eax, 3
        mov ebx, 1              ;1 for stdin
        mov ecx, original_str
        mov edx, LEN – 1
        int 80H
        mov ebx, 0
        mov [original_str + eax], ebx ;null terminate the string
;; push original string to stack, byte by byte,
;; from beginning, untill 0 (null character '\0')
        mov eax, 0
push_loop:
;; check for null character, that is end of string
        cmp dword [original_str + eax], 0
        jz end_push_loop
        push dword [original_str + eax]
        inc eax
        jmp push_loop
end_push_loop:
;; since stack is first in last out, when we pop
;; characters of the string, we will get the string in reverse.
        mov ecx, eax
        mov eax, 0
pop_loop:
        pop dword [reverse_str + eax]
        inc eax
        loop pop_loop
;; end the reverse string with null character
        mov dword [reverse_str + eax], 0
;; call printf to print the string
        push reverse_str
        push format
        call printf
```

```
;; exit the program
        mov eax, 1
        mov ebx, 0
        int 80H
```

OBSERVATIONS

RESULT

Verified reverse of the given string.

EXP NO. 11 STRING COMPARISON

AIM

Write a program to compare two strings.

OBJECTIVES

- To understand the string comparison.

ALGORITHM

```
SECTION .data
        prompt1             db      "Enter first string: ", 0
        prompt1_len         equ     $-prompt1
        prompt2             db      "Enter second string: ", 0
        prompt2_len         equ     $-prompt2
        str_equal           db      "Two strings are equal", 10, 0
        str_equal_len       equ     $-str_equal
        str_not_equal       db      "Two strings are different", 10, 0
        str_not_equal_len   equ     $-str_not_equal
        LEN                 db      50

SECTION .bss
        str1 :   resb    50
        str2 :   resb    50

SECTION .text
          global _start
_start:
        mov eax, 4
        mov ebx, 1
        mov ecx, prompt1
        mov edx, prompt1_len
        int 80h ;display enter first string

        mov eax, 3
        mov ebx, 1
```

```
        mov ecx, str1
        mov edx, LEN
        int 80H

        mov dword [str1 + eax], 0 ;read first string
        mov eax, 4
        mov ebx, 1
        mov ecx, prompt2
        mov edx, prompt2_len
        int 80h ;display enter second string

        mov eax, 3
        mov ebx, 1
        mov ecx, str2
        mov edx, LEN
        int 80H

        mov dword [str2 + eax], 0 ;read second string
        mov eax, 0

loop1:
        mov ebx, [str1 + eax]
        mov ecx, [str2 + eax]
        cmp ebx, ecx
        jnz print_not_equal
        cmp ebx, 0
        jz print_equal
        inc eax
        jmp loop1

print_not_equal:
        mov eax, 4
        mov ebx, 1
        mov ecx, str_not_equal
        mov edx, str_not_equal_len
        int 80H ;display strings are not equal
        jmp finish
```

```
print_equal:
        mov eax, 4
        mov ebx, 1
        mov ecx, str_equal
        mov edx, str_equal_len
        int 80H ;display strings are equal

finish:
        mov eax, 1
        mov ebx, 0
        int 80H
```

OBSERVATIONS

RESULT

Verified the comparison of two strings.

EXP NO. 12 UPPERCASE TO LOWERCASE

AIM

Write a program to convert uppercase characters to lowercase.

OBJECTIVES

- To understand the string case conversion.

ALGORITHM

```
SECTION .data
        Snippet    db    "KANGAROO"
SECTION .text
        global _start
_start:

     mov ebx,Snippet
     mov eax,8

DoMore:
     add byte [ebx],32
     inc ebx
     dec eax
     jnz DoMore

     ;add necessary statements
```

OBSERVATIONS

RESULT

Verified the conversion of uppercase characters to lower case.

EXP NO. 15 LOWERCASE TO UPPERCASE

AIM

Write a program to convert lowercase characters to uppercase.

OBJECTIVES

- To understand the string case conversion.

ALGORITHM

```
SECTION .bss
        Buff    resb    1
SECTION .data

SECTION .text
        global _start
_start:
Read:
        mov eax, 3              ; Specify sys_read call
        mov ebx, 0              ; Specify File Descriptor 0: Standard Input
        mov ecx, Buff           ; Pass address of the buffer to read to
        mov edx, 1              ; Tell sys_read to read one char from stdin
        int 80h                 ; Call sys_read
        cmp eax, 0              ; Look at sys_read's return value in EAX
        je Exit                 ; Jump If Equal to 0 (0 means EOF) to Exit
                                ; or fall through to test for lowercase
        cmp byte [Buff], 61h    ; Test input char against lowercase 'a'
        jb Write                ; If below 'a' in ASCII chart, not lowercase
        cmp byte [Buff], 7Ah    ; Test input char against lowercase 'z'
        ja Write                ; If above 'z' in ASCII chart, not lowercase
        ; At this point, we have a lowercase character
        sub byte [Buff], 20h    ; Subtract 20h from lowercase to give uppercase...
                                ; ...and then write out the char to stdout
```

Write:

```
        mov eax, 4          ; Specify sys_write call
        mov ebx, 1          ; Specify File Descriptor 1: Standard output
        mov ecx, Buff       ; Pass address of the character to write
        mov edx, 1          ; Pass number of chars to write
        int 80h             ; Call sys_write...
        jmp Read            ; ...then go to the beginning to get another character
```

Exit:

```
        mov eax, 1          ; Code for Exit Syscall
        mov ebx, 0          ; Return a code of zero to Linux
        int 80H             ; Make kernel call to exit program
```

OBSERVATIONS

RESULT

Verified the conversion of lowercase characters to uppercase.

EXP NO. 13 BINARY TO HEX

AIM

Write a program to convert binary value into hexadecimal strings.

OBJECTIVES

- To understand number format conversion.

ALGORITHM

```
SECTION .bss                    ; Section containing uninitialized data
        BUFFLEN equ 16          ; We read the file 16 bytes at a time
        Buff: resb BUFFLEN      ; Text buffer itself

SECTION .data                   ; Section containing initialized data
        HexStr:  db     " 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00",10
        HEXLEN      equ     $-HexStr
        Digits:  db     "0123456789ABCDEF"

SECTION .text                   ; Section containing code
        global _start           ; Linker needs this to find the entry point!
_start:
; Read a buffer full of text from stdin
Read:
        mov eax,3               ; Specify sys_read call
        mov ebx,0               ; Specify File Descriptor 0: Standard Input
        mov ecx,Buff            ; Pass offset of the buffer to read to
        mov edx,BUFFLEN         ; Pass number of bytes to read at one pass
        int 80h                 ; Call sys_read to fill the buffer

        mov ebp,eax             ; Save # of bytes read from file for later
        cmp eax,0               ; If eax=0, sys_read reached EOF on stdin
        je Done                 ; Jump If Equal (to 0, from compare)
        ; Set up the registers for the process buffer step:
```

```
            mov esi,Buff                ; Place address of file buffer into esi
            mov edi,HexStr              ; Place address of line string into edi
            xor ecx,ecx                 ; Clear line string pointer to 0
            ; Go through the buffer and convert binary values to hex digits:
Scan:
            xor eax,eax                 ; Clear eax to 0
; Here we calculate the offset into HexStr, which is the value in ecx X 3
            mov edx,ecx                 ; Copy the character counter into edx
            shl edx,1                   ; Multiply pointer by 2 using left shift
            add edx,ecx                 ; Complete the multiplication X3
; Get a character from the buffer and put it in both eax and ebx:
            mov al,byte [esi+ecx]       ; Put a byte from the input buffer into al
            mov ebx,eax                 ; Duplicate the byte in bl for second nybble
; Look up low nybble character and insert it into the string:
            and al,0Fh                  ; Mask out all but the low nybble
            mov al,byte [Digits+eax]    ; Look up the char equivalent of nybble
            mov byte [HexStr+edx+2],al ; Write LSB char digit to line string
; Look up high nybble character and insert it into the string:
            shr bl,4                    ; Shift high 4 bits of char into low 4 bits
            mov bl,byte [Digits+ebx]    ; Look up char equivalent of nybble
            mov byte [HexStr+edx+1],bl ; Write MSB char digit to line string
; Bump the buffer pointer to the next character and see if we’re done:
            inc ecx                     ; Increment line string pointer
            cmp ecx,ebp                 ; Compare to the number of chars in the buffer
            jna Scan                    ; Loop back if ecx is <= number of chars in buffer
; Write the line of hexadecimal values to stdout:
            mov eax,4                   ; Specify sys_write call
            mov ebx,1                   ; Specify File Descriptor 1: Standard output
            mov ecx,HexStr              ; Pass offset of line string
            mov edx,HEXLEN              ; Pass size of the line string
            int 80h                     ; Make kernel call to display line string
            jmp Read                    ; Loop back and load file buffer again
; All done! Let’s end this party:
Done:
```

```
mov eax,1           ; Code for Exit Syscall
mov ebx,0           ; Return a code of zero
int 80H             ; Make kernel call
```

OBSERVATIONS

RESULT

Verified the binary to hex conversion.

EXP NO. 14 TRANSLATION

AIM

Write a program to convert all lowercase characters into uppercase and non printable characters into spaces.

OBJECTIVES

- To understand the use of translation using XLAT.

ALGORITHM

```
SECTION .data                ; Section containing initialized data
        StatMsg:   db    "Processing...",10
        StatLen:   equ   $-StatMsg
        DoneMsg:   db    "...done!",10
        DoneLen:   equ   $-DoneMsg

; The following translation table translates all lowercase characters to
; uppercase. It also translates all non-printable characters to spaces,
; except for LF and HT.

UpCase:
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,09h,0Ah,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
        db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
        db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
        db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
        db 60h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
        db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,7Bh,7Ch,7Dh,7Eh,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
```

```
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h

; The following translation table is "stock" in that it translates all
; printable characters as themselves, and converts all non-printable
; characters to spaces except for LF and HT.

Custom:
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,09h,0Ah,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
        db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
        db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
        db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
        db 60h,61h,62h,63h,64h,65h,66h,67h,68h,69h,6Ah,6Bh,6Ch,6Dh,6Eh,6Fh
        db 70h,71h,72h,73h,74h,75h,76h,77h,78h,79h,7Ah,7Bh,7Ch,7Dh,7Eh,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
        db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h

SECTION .bss                    ; Section containing uninitialized data
        READLEN equ 1024            ; Length of buffer
        ReadBuffer: resb READLEN    ; Text buffer itself

SECTION .text                   ; Section containing code
        global _start           ; Linker needs this to find the entry point!
_start:
```

```
; Display the "I'm working..." message via stderr:
        mov eax,4               ; Specify sys_write call
        mov ebx,2               ; Specify File Descriptor 2: Standard error
        mov ecx,StatMsg         ; Pass offset of the message
        mov edx,StatLen         ; Pass the length of the message
        int 80h                 ; Make kernel call

; Read a buffer full of text from stdin:
read:
        mov eax,3               ; Specify sys_read call
        mov ebx,0               ; Specify File Descriptor 0: Standard Input
        mov ecx,ReadBuffer      ; Pass offset of the buffer to read to
        mov edx,READLEN         ; Pass number of bytes to read at one pass
        int 80h
        mov ebp,eax             ; Copy sys_read return value for safekeeping
        cmp eax,0               ; If eax=0, sys_read reached EOF
        je done                 ; Jump If Equal (to 0, from compare)
; Set up the registers for the translate step:
        mov ebx,UpCase          ; Place the offset of the table into ebx
        mov edx,ReadBuffer      ; Place the offset of the buffer into edx
        mov ecx,ebp             ; Place the number of bytes in the buffer into ecx

; Use the xlat instruction to translate the data in the buffer:
; (Note: the commented out instructions do the same work as XLAT;
; un-comment them and then comment out XLAT to try it!
translate:
        ; xor eax,eax           ; Clear high 24 bits of eax
        mov al,byte [edx+ecx]   ; Load character into AL for translation
        ; mov al,byte [UpCase+eax] ; Translate character in AL via table
        xlat                    ; Translate character in AL via table
        mov byte [edx+ecx],al   ; Put the translated char back in the buffer
        dec ecx                 ; Decrement character count
        jnz translate           ; If there are more chars in the buffer, repeat
```

```
; Write the buffer full of translated text to stdout:
write:
        mov eax,4               ; Specify sys_write call
        mov ebx,1               ; Specify File Descriptor 1: Standard output
        mov ecx,ReadBuffer      ; Pass offset of the buffer
        mov edx,ebp             ; Pass the # of bytes of data in the buffer
        int 80h                 ; Make kernel call
        jmp read                ; Loop back and load another buffer full

; Display the "I'm done" message via stderr:
done:
        mov eax,4               ; Specify sys_write call
        mov ebx,2               ; Specify File Descriptor 2: Standard error
        mov ecx,DoneMsg         ; Pass offset of the message
        mov edx,DoneLen         ; Pass the length of the message
        int 80h                 ; Make kernel call

; All done! Let's end this party:
        mov eax,1               ; Code for Exit Syscall
        mov ebx,0               ; Return a code of zero
        int 80H                 ; Make kernel call
```

OBSERVATIONS

RESULT

Studied the use of procedure.

EXP NO. 15 SORTING

AIM

Write a program to implement selection sort of an integer array.

OBJECTIVES

- To understand the use of procedure.

ALGORITHM

```
SECTION .data
        array db 89, 10, 67, 1, 4, 27, 12, 34, 86, 3
        ARRAY_SIZE equ $ - array
        array_fmt db "  %d", 0
        usort_str db "unsorted array:", 0
        sort_str db "sorted array:", 0
        newline db 10, 0

SECTION .text
        extern puts
        global _start

_start:
        push  usort_str
        call  puts
        add   esp, 4
        push  ARRAY_SIZE
        push  array
        push  array_fmt
        call  print_array10
        add   esp, 12
        push  ARRAY_SIZE
        push  array
        call  sort_routine20
```

```
; Adjust the stack pointer
        add   esp, 8
        push  sort_str
        call  puts
        add   esp, 4
        push  ARRAY_SIZE
        push  array
        push  array_fmt
        call  print_array10
        add   esp, 12
        jmp   _exit
        extern printf
print_array10:
        push  ebp
        mov   ebp, esp
        sub   esp, 4
        mov   edx, [ebp + 8]
        mov   ebx, [ebp + 12]
        mov   ecx, [ebp + 16]
        mov   esi, 0
push_loop:
        mov   [ebp - 4], ecx
        mov   edx, [ebp + 8]
        xor   eax, eax
        mov   al, byte [ebx + esi]
        push  eax
        push  edx
        call  printf
        add   esp, 8
        mov   ecx, [ebp - 4]
        inc   esi
        loop  push_loop
        push  newline
        call  printf
```

```
        add   esp, 4
        mov   esp, ebp
        pop   ebp
        ret
sort_routine20:
        push  ebp
        mov   ebp, esp
; Allocate a word of space in stack
        sub   esp, 4
; Get the address of the array
        mov   ebx, [ebp + 8]
; Store array size
        mov   ecx, [ebp + 12]
        dec   ecx
; Prepare for outer loop here
        xor   esi, esi
outer_loop:
; This stores the min index
        mov   [ebp - 4], esi
        mov   edi, esi
        inc   edi
inner_loop:
        cmp   edi, ARRAY_SIZE
        jge   swap_vars
        xor   al, al
        mov   edx, [ebp - 4]
        mov   al, byte [ebx + edx]
        cmp   byte [ebx + edi], al
        jge   check_next
        mov   [ebp - 4], edi
check_next:
        inc   edi
        jmp   inner_loop
```

```
swap_vars:
        mov   edi, [ebp - 4]
        mov   dl, byte [ebx + edi]
        mov   al, byte [ebx + esi]
        mov   byte [ebx + esi], dl
        mov   byte [ebx + edi], al
        inc   esi
        loop  outer_loop
        mov   esp, ebp
        pop   ebp
        ret
_exit:
        mov   eax, 1
        mov   ebx, 0
        int   80h
```

OBSERVATIONS

RESULT

Verified the sorting of numbers using procedure.

EXP NO. 16 MACRO

AIM

Write a program to read a string and display a greeting to user using macro.

OBJECTIVES

- To understand the use of macro.

ALGORITHM

```
SECTION .data
        prompt_str  db   'Enter your name: '
        STR_SIZE  equ  $ - prompt_str
        greet_str  db  'Hello '
        GSTR_SIZE  equ  $ - greet_str

SECTION .bss
        buff  resb  32          ;  Reserve 32 bytes of memory

; A macro with two parameters
; Implements the write system call
 %macro write 2
        mov   eax, 4
        mov   ebx, 1
        mov   ecx, %1
        mov   edx, %2
        int   80h
 %endmacro
; Implements the read system call
%macro read 2
        mov   eax, 3
        mov   ebx, 0
        mov   ecx, %1
        mov   edx, %2
        int   80h
%endmacro
```

```
SECTION .text
        global _start
_start:
        write prompt_str, STR_SIZE
        read buff, 32
        ; Read returns the length in eax
        push eax
        ; Print the hello text
        write greet_str, GSTR_SIZE
        pop  edx
        ; edx  = length returned by read
        write buff, edx
  _exit:
        mov   eax, 1
        mov   ebx, 0
        int 80h
```

OBSERVATIONS

Enter your name: MYNAME

Hello MYNAME

RESULT

Verified string read and display greeting to the user using macro.

APPENDIX-A

APPENDIX-B