INTRODUCTION TO 8086 MICROPROCESSOR AND ASSEMBLER

Microprocessor Architecture

Intel 8086 is a 16 bit integer processor with 16-bit data bus and 20-bit address bus. The lower 16-bit address lines and 16-bit data lines are multiplexed (AD0-AD15). Since 20-bit address lines are available, 8086 can access up to 2^{20} or 1 MB of physical memory.

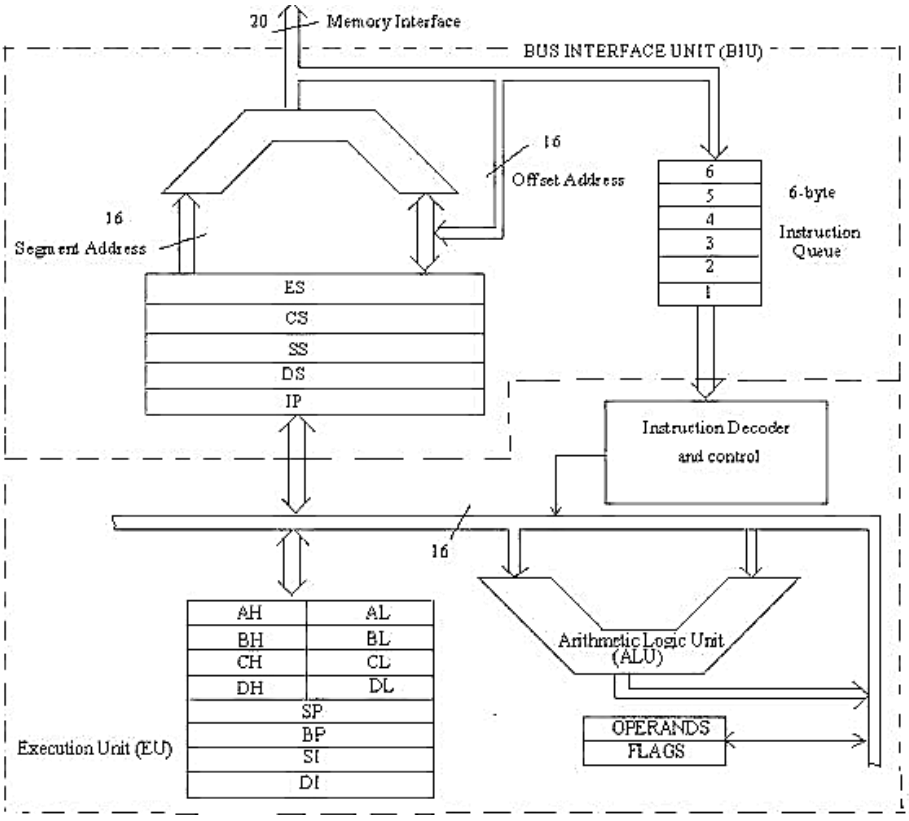

Fig. 1: Internal architecture of 8086 microprocessor

8086 CPU is divided into two independent function parts.

1. Bus Interface Unit (BIU)
2. Execution Unit (EU)

Functions of BIU and EU

EU has three parts namely:

- *Control Circuitry:* Generates control signals.

- *Instruction Decoder:* Translates instructions fetched from memory in series of actions.
- *16-bit ALU:* Performs all arithmetic and logical operations or shifts 16-bit binary numbers.

Flag Register: A flag is a flip-flop that indicates some condition produced by the execution of an instruction or controls certain operations of the EU. A 16-bit flag register in the EU contains nine active flags. It includes six conditional flags namely *Carry Flag (CF), Auxiliary carry Flag(AF),Zero Flag (ZF) Parity Flag (PF), Sign Flag (SF) and Overflow Flag (OF).* The three remaining flags in the flag register are used to control certain operations of the processor namely *Trap Flag (TF),* which is used for single stepping through a program, *Interrupt Flag (IF),* which is used to allow or prohibit the interruption of a program and the *Direction Flag (DF),*which is used with string instructions.

General-purpose registers: EU has eight general-purpose registers, labeled AH, AL, BH, BL, CH, CL, DH, and DL. These registers can be used individually for temporary storage of 8-bit data. The AL register is also called the accumulator. It has some features that the other general-purpose registers do not have. The AH-AL pair is referred to as the AX register, the BH-BL pair is referred to as the BX register, the CH-CL pair is referred to as the CX register, and the DH-DL pair is referred to as the DX register.

B**us Interface Unit:**
Queue IU: BIU fetches up to six instruction bytes and stores these pre-fetched bytes in a first-in-first-out register set called a *queue.* When the EU is ready for its next instruction, it simply reads the instruction bytes for the instruction from the queue in the BIU.

Segment Registers: Four segment registers in the BIU are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with, at a particular time. The four segment registers are the *Code Segment (CS) register*, the *Stack Segment (SS) registers, the Extra Segment (ES) register, and the Data Segment (DS) register.*
Instruction Pointer: The instruction pointer register holds the 16-bit address, or offset, of the next code byte within this code segment. The value contained in the IP is referred to as an offset because this value must be offset from (added to) the segment base address in CS to produce the required 20-bit physical address sent out by the BIU.

Stack Segment Register and Stack Pointer Register: A stack is a section of memory set aside to store addresses and data while a subprogram is executing. The *Stack Pointer (SP)* register in the execution unit holds the 16-bit offset from the start of the segment to the memory location where a word was most recently stored is called the *top of stack.*

Pointer and Index Registers in the Execution Unit: In addition to the Stack Pointer register (SP), the EU contains a 16-bit *Base Pointer (BP)* register. It also contains a 16-bit *Source Index (SI)* register and a 16-bit *Destination Index (DI)* register. These three registers can be used for temporary storage of data just as the general-purpose registers. However, their main use is to hold the 16-bit offset of a data word in one of the segments. It can read or write 16-bit or 8-bit at a time. It can address any one of 2^{20} or 1,048,576 memory locations.

ASSEMBLY LANGUAGE PROGRAMMING

Machines can understand only binary numbers and hence the instructions for computers are written in binary numbers. However human beings find it difficult to manipulate or understand these binary numbers. So designers developed a way to write the instructions in English alphabets but in coded form. These coded English words are called *mnemonics.* This language is called *Assembly Language*. *Assembler* is a program which converts assemble language into machine language. *Assembler directives* are the pseudo instructions for the assembler. These tell the assembler how to convert assembly language into binary language.

Assembly language

Assembly language is the symbolic representation of machine language.

An assembly statement contains four fields:

- Label: Used to associate a symbolic address with an instruction or data location
- Opcode: Contains either a mnemonic machine instruction or a pseudo-operation code or the name of an assembler directive.
- Operands: It follows opcode field separated by a blank or a tab. Operands are separated by commas.
- Comments: It is written with a semicolon, and causes the Assembler to ignore the remainder of the source line.

An Assembly Language Program is a sequence of statements. There are three classes of statements.

- Instructions: Represent a single machine instruction in symbolic form.

- Directives: Directives communicate information about the program to the assembler, but do not generally cause the assembler to output any machine instructions.
- Pseudo – operations: Pseudo-operations cause the assembler to initialize or reserve one or more words of storage for data, rather than machine instructions.

An assembler reads a single assembly language source file and produces an *object file*. The object file is made up of machine instructions. A program consists of many modules or source files. All these modules are converted to object files independently by the assembler. Then these independent object files are combined together along with the program library with the help of linker. The *linker* then generates an *executable file*. This executable file is finally executed by the processor.

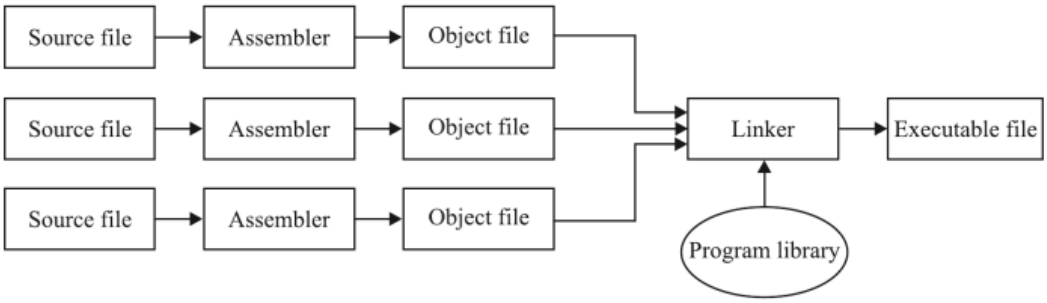

Fig. 2: Assembly Language Programming operations

The steps to develop an Assembly Language Program are the following:

- Write down the source code as per the assembly language.
- Create the object code for the Assembly Language Program.
- Link the object code to create an executable (exe) file.
- Test and debug the program.

Assembler Directives

The assembler decides the address of each label and substitutes the values for each of the constants and variables. It then forms the machine code for the mnemonics and data in the Assembly Language Program. While doing these things, the assembler may find out syntax errors. The logical errors or other programming errors are not found out by the assembler. Some assembler directives and their functions are listed below.

1. **DB (Define Byte):** Used to reserve byte or bytes of memory locations in the available memory.

Ex: LIST DB 0lH, 02H, 03H, 04H. This statement directs the assembler to reserve four memory locations for a list named LIST and initialize them with the above specified four values.

2. **DW (Define Word):** Makes the assembler reserve the number of memory words (16-bit) instead of bytes.
   Ex: WORDS DW 4 DUP (1234H)
   This makes the assembler reserve four words in memory (8 bytes), and initialize all words with 1234H since DUP is used. During initialization, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses.

3. **DQ (Define Quad word):** Used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

4. **ASSUME:** Used to inform the assembler, the names of the logical segments to be assumed for different segments used in the program.
   Ex: If the code segment is given the name CODE. The statement *ASSUME CS: CODE* directs the assembler that the machine codes are available in a segment named CODE. The ASSUME statement is a must at the starting of each Assembly Language Program.

5. **END (END of Program):** The END directive marks the end of an Assembly Language Program. When the assembler comes across this END directive, it ignores the source lines available later on.
6. **ENDP (END of Procedure):** Used to indicate the end of a procedure. A procedure is usually assigned a name, i.e. label. To mark the end of a particular procedure, the name of the procedure, i.e. label may appear as a prefix with the directive ENDP. The statements, appearing in the same module but after the ENDP directive, are neglected from that procedure. The structure given below explains the use of ENDP.

```
PROCEDURE STAR
  ; Write the code here
STAR ENDP
```

7. **ENDS (END of Segment):** Marks the end of a logical segment. The names appear with the ENDS directive as prefixes to mark the end of those particular segments.

Whatever are the contents of the segments, they should appear in the program before ENDS. The structure shown below explains the fact more clearly.

```
DATA SEGMENT
; Initialize data values
DATA ENDS
ASSUME DS: DATA
```

8. **EXTRN (External and PUBLIC):** The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive.

   If one wants to call a procedure FACTORIAL appearing in MODULE 1 from MODULE 2; in MODULE1, it must be declared PUBLIC using the statement PUBLIC FACTORIAL and in module 2, it must be declared external using the declaration EXTRN FACTORIAL. The statement of declaration EXTRN must be accompanied by the SEGMENT and ENDS directives of the MODULE 1, before it is called in MOBULE 2. Thus the MODULE 1 and MODULE 2 must have the following declarations.

```
MODULEl SEGMENT
PUBLIC FACTORIAL FAR
MODULEl ENDS
MODULE2 SEGMENT
EXTRN FACTORIAL FAR
MODULE2 ENDS
```

9. **LABEL (Label):** Used to assign a name to the current content of the location counter. During the assembly process, whenever the assembler comes across the LABEL directive, it assigns the declared label with the current contents of the location counter. The type of the label must be specified, i.e. whether it is a NEAR or a FAR label, BYTE or WORD label, etc. A LABEL directive may be used to make a FAR jump as shown below.

```
CONTINUE LABEL FAR
```

   The LABEL directive can be used to refer to the data segment along with the data type, byte or word as shown.

```
DATA SEGMENT
DATAS DB 50H DUP (?)
DATA-LAST LABEL BYTE FAR
DATA ENDS
```

After reserving 50H locations for DATAS, the next location will be assigned a label DATALAST and its type will be byte and far.

10. **OFFSET (Offset of a Label):** When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement.
    This operator is used with arrays, strings, labels and procedures to decide their offsets in their default segments. The examples of this operator are as follows:

    ```
    CODE SEGMENT
    MOV SI, OFFSET LIST
    CODE ENDS
    ```

11. **ORG (Origin):** The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement. If the ORG statement is not written in the program, the location counter is initialized to 0000.
    If an ORG 200H statement is present at the starting of the code segment of that module, then the code will start from 200H address in code segment). The ORG directive can even be used with data segments similarly.

12. **PROC (Procedure):** Marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e whether it is to be called by the main program located within 64K of physical memory or not.
    The example statements are as follows:

    ```
    RESULT PROC NEAR
    ROUTINE PROC FAR
    ```

13. **PTR (Pointer):** Used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either BYTE or WORD. If the prefix is BYTE, then the particular label, variable or memory operand is treated as an 8-bit quantity
    Ex: MOV AL, BYTE PTR [SI]
    The above instruction moves content of memory location addressed by SI to AL
    In case of JMP instructions, the PTR operator is used to specify the type of the jump, i.e. near or far. *Ex: JMP WORD PTR [BX] -NEAR Jump*

14. **PUBLIC:** Used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes.

15. **SEGMENT (Logical Segment):** Marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement. The SEGMENT and ENDS directive must bracket each logical segment of a program. The program structure given below explains the use of the SEGMENT directive.
    EXE .CODE SEGMENT GLOBAL ; Start of segment named EXE.CODE, that can be accessed by any other module.

16. **SHORT: I**ndicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within -128 to +127 bytes from the address of the byte next to the jump opcode).
    Syntax: JMP SHORT LABEL

LAB NO: 1 **Date:**

FAMILIARIZATION OF ASSEMBLER, LINKER AND DEBUGGER

Objectives

- To familiarize MASM.
- Write a simple Assembly Language Program

Introduction to MASM

The microprocessor lab experiments are designed using Microsoft **Macro Assembler MASM**. Assembly level programs are generally abbreviated as ALP.

The process to debug with MASM is

1. Open command prompt and set the path to MASM directory.
   i.e. D:**cd masm**
2. Type the program by opening an editor using Edit command
   i.e. D:\masm\> **edit sample.asm**
   Save it with .asm extension. Note that the length of the file name must be preferably less than or equal to 8 characters.
3. Then create the 'OBJ' file from the ASM file using command MASM. This is referred as assembling
   D:\MASM> **masm sample;**
4. After assembling, link the file to generate EXE file from the OBJ file using command
   D:\MASM> **link sample;**
5. Run turbo debugger to execute the program using executable file
   D:\MASM>**td sample.exe**
6. Execute the program step by step by pressing 'F7' or execute the entire program by pressing 'F9'. Ensure that to execute the program using F9, the code must end with INT 21H before which AH register is loaded with 4CH.

Solved exercise

Move the immediate data 05H to AX register.

```
CODE SEGMENT              ; Begin Code segment
        ASSUME CS: CODE ; Tell assembler that code segment corresponds to 'Code'
START:      MOV AX, 05H ; Move immediate data 05H to AX register
```

```
CODE ENDS  ; Terminate code segment
END START   Terminate the label
```

Note: MOV AH,4CH

INT 21H is to be included before ending the code segment to terminate the program.

When the above code is executed on TD, the window appears as shown in Fig. 3

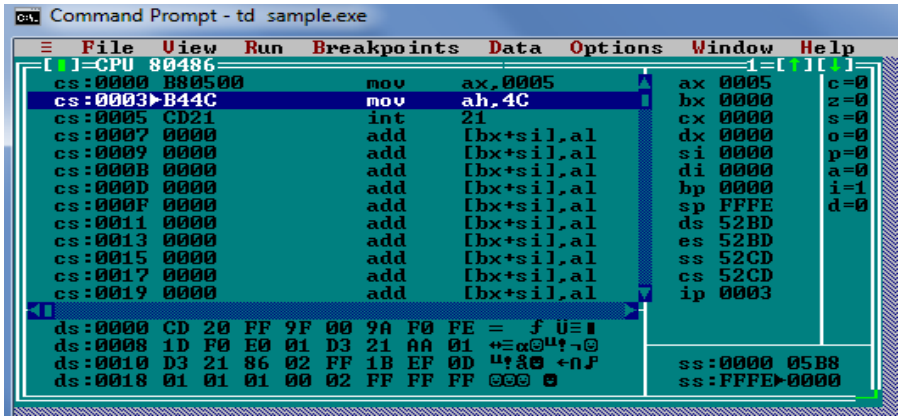

Fig. 3 Screen shot of TD for above program

Output :

	AX
Before Execution	0000
After Execution	0005

Solved exercise

Move the data stored in data segment from one location to another.

```
DATA SEGMENT                ; Begin data segment
        NUM DB 10H          ; Initialize data segment with value 10H
        RES DB ?
DATA ENDS                   ; Terminate data segment


CODE SEGMENT                ; Begin code segment
ASSUME CS:CODE, DS:DATA     ; Indicate assembler the name of code and data
```

```
                                    ; segment

START:MOV AX,DATA      ; Initialize data segment register DS with the base
      MOV DS,AX        ;address of data segment

      MOV AL,NUM       ; AL register is loaded with value in data segment
      MOV RES, AL      ; Move the value of AL to memory location

      MOV AH,4CH       ; Interrupt to terminate the program
      INT 21H
CODE ENDS              ; Terminate Code segment
END START              ; Indicate end of  label
```

Explanation: Define the data segment with values by mentioning whether the value is a byte or word using corresponding assembler directives. Fig. 4 given below depicts the above program when executed on turbo debugger. Note that when the first instruction is executed, data segment is loaded with corresponding values. To check this select 'CPU' under drop down bar of 'VIEW' menu. When the next MOV instruction is executed, the value stored in data segment with label 'NUM' is copied to AL register. Further the next instruction value of AL register is moved to memory location 'RES' (DS: 0001). Note that 8086 always stores the LS byte of word in lower address and MS byte of word at higher address. All registers are available at the right side of the Turbo Debugger.

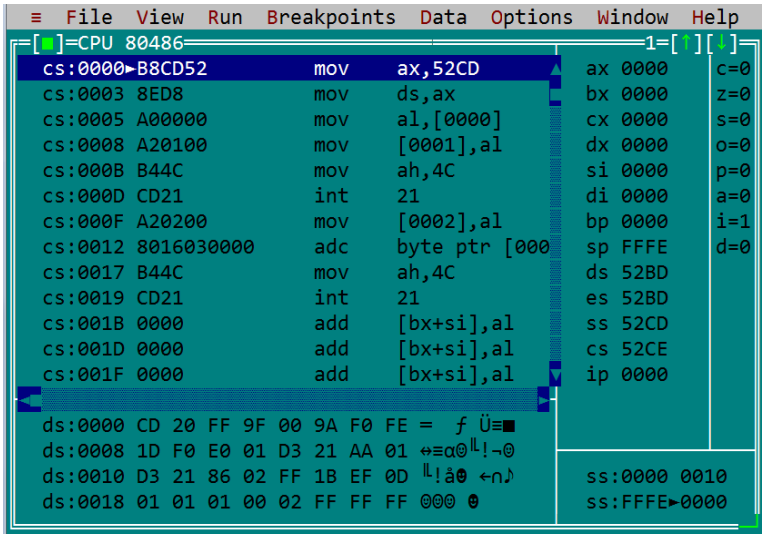

Fig. 4: Screenshot to illustrate features available in Debugger

Output

DS:0000	**DS:0001**

Before Execution	10	00
After Execution	10	10

Lab exercises

Write an Assembly Language Program

1. To copy the contents (1 byte) stored in one register to another register.
2. To copy the contents (1 word) stored in one register to another register.
3. To copy the contents (1 word) stored in one memory location to another memory location

LAB NO: 2 **Date:**

DATA TRANSFER OPERATIONS

Objectives

Write an Assembly Language Program to

- Transfer data from one location to another using different addressing modes
- Perform block transfer of HEX or decimal numbers stored in arrays.
- Learn the usage of instructions like XCHG, INC, DEC and LOOP instructions.

8086 Addressing Modes

Various methods in which processor can access data are called Addressing modes.
There are four major categories of addressing modes as listed below:

1. Immediate Addressing
2. Register Addressing
3. Memory Addressing
4. I/O Port addressing

Data transfer is done by using any of these addressing modes. For this we make use of instructions like MOV, PUSH, POP, XLAT, LEA, LDS, LES, LAHF, SAHF and XCHG. These instructions can be used to operate on data of different sizes. Assembler directives like DB, DW, DD and DT are used to define the size of data. For example, if data is stored in memory locations 00H to 09H, and when the next block of data is to be stored from memory location 06H to 13H, data from memory location 06H to 09H will be overwritten. This lab illustrates usage of pointers to transfer data between overlapped memory locations.

Solved exercise

Write an Assembly Language Program to copy five 8-bit binary numbers from an ARRAY1 to ARRAY2. Assume ARRAY1 and ARRAY2 are in data segment

Code	**Comment**
DATA SEGMENT	
ARRAY1 DB 11H,22H,33H,44H,55H	; Initialize ARRAY1 with HEX numbers
ARRAY2 DB 5 DUP (0)	; Allocate 5 bytes in array 2
COUNT DB 05H	; Load the number of elements in array1

```
DATA ENDS
CODE SEGMENT
      ASSUME CS: CODE, DS: DATA
START: MOV AX,DATA
          MOV DS,AX
          LEA SI, ARRAY1          ; Load SI with EA of ARRAY1 in ds
          LEA DI, ARRAY2          ; Load DI with EA of ARRAY2 in ds
          MOV CH, 00H             ; Clear the contents of CH register
          MOV CL, COUNT           ; Load CX register with count value
  NEXT:MOV AL, [SI]               ; Copy contents of DS:SI to AL register.

          MOV [DI], AL            ; Copy contents of AL register to DS:DI.

          INC SI                  ;Increment SI register to point to second
                                   element in array 1
          INC DI                  ;Increment DI register to point to second
                                   position
          LOOP NEXT               ;Decrement CX register and if it is not
                                   equal to zero, jump to label 'NEXT'

           MOV AH,4CH             ;Terminate program
           INT 21H
CODE ENDS                         ;Terminate code segment
END START
```

Explanation

To transfer five HEX numbers from *'*ARRAY1*'* to *'*ARRAY2*'* in data segment, load the number of elements to be copied into CX register. This is because LOOP instruction is used to transfer the array contents. LOOP instruction always looks at contents of CX register to decide whether to continue or terminate the instructions within the loop. The Effective Address indicating the location of each element in the ARRAY1 is pointed by SI and that of ARRAY2 is pointed by DI. LEA instruction is used for this purpose. Every time the value stored in the memory location pointed by SI is copied to destination memory location pointed by DI through AL register. The pointer is then incremented to point to the next location. This is repeated until the count or contents of CX register becomes '0'.

Output

DS:	DS:0005	DS:0006	DS:0007	DS:0008	DS:0009
Before Execution	00	00	00	00	00
After Execution	11	22	33	44	55

Note:

- *If each number is 1 word in length, then the pointer is incremented twice.*
- *To exchange the contents of two arrays, XCH instruction can be used.*
- *To transfer block of data with overlapping, point to the desired location at source and destination. Increment the location from which value is to be copied. Decrement the destination location.*

Lab exercises

Write an Assembly Language Program to

1. Transfer 10 bytes of data in data segment, where blocks are overlapping.
2. Transfer 5 words of data in data segment, where blocks are a). overlapping and b).non-overlapping.
3. Exchange the contents of two arrays, where each array has 10 bytes of data in the data segment.

Additional exercises

1. Illustrate different addressing modes available in 8086 programming.
2. Write an Assembly Language Program to reverse the contents of an array of 10 words stored in the data segment.

LAB NO: 3 **Date:**

ARITHMETIC OPERATIONS: ADDITION AND SUBTRACTION

Objectives

- To learn the usage of arithmetic instructions.
- Write ALPs to perform operations on 8 bits, 16 bits, 32 bits, 64 bits and 10 byte numbers.

Introduction

8086 has very powerful instructions in the arithmetic group. It can perform add, subtract, multiply and divide operations on 8 bit or 16 bit numbers which can be signed or unsigned. Addition is performed using ADD instruction. Carry generated during addition operation is stored in the Carry flag. If addition is to be performed along with Carry flag then ADC instruction is used. There are two types of BCD numbers namely Packed BCD and Unpacked BCD numbers. DAA instruction is used when addition is performed on decimal numbers and the result is expected to be in packed decimal. AAA instruction is used to get the unpacked BCD values when the numbers to be added are in ASCII. Similarly, SUB instruction is used to perform subtraction of two numbers and Carry flag indicates borrow. To perform subtraction with borrow bit, SBB instruction is used. DAS and AAS instructions are used for decimal and ASCII subtraction adjustments respectively.

Solved exercises

Write an Assemble Language Program to add two integers stored in the data segment and store the result in data segment

```
DATA SEGMENT
        NUM1 DB 0FFH
        NUM2 DB 0FEH
        SUM DB 00H
        CARRY DB 00H
DATA ENDS

CODE SEGMENT
        ASSUME CS: CODE, DS: DATA
START: MOV AX, DATA             ; segment address into AX
         MOV DS, AX             ; initialize DS register

          MOV AH, 00H           ; Clear AH register
         MOV AL, NUM1           ; First operand copied into AL register
```

```
        MOV BL, NUM2        ; Second operand copied into BL register
        CLC                 ; Clear Carry flag
        ADC AL, BL          ; Add contents of AL and BL registers
        MOV SUM, AL         ; Copy sum obtained to memory location
        ADC CARRY,0         ; Copy the carry obtained to memory location

        MOV AH, 4CH         ; Exit to DOS
        INT 21H
CODE ENDS
END START
```

Output

	DS:0002 (SUM)	**DS:0003 (CARRY)**
Before Execution	00	00
After Execution	FD	01

Lab exercises

Write an Assembly Language Program to:

1. Add two 16 - bit hexadecimal numbers stored in registers and store the result in data segment
2. Add two 8 - bit decimal numbers stored in data segment and store the result in data segment
3. Add two 32 - bit hexadecimal numbers in data segment and store the result in data segment.
4. Subtract two 16 - bit numbers in memory locations and store the result in data segment
5. Subtract two 32 - bit numbers in data segment and store the result in data segment.
6. Subtract two 10 - byte numbers in data segment and store the result in data segment.
7. Repeat questions 3 - 6 for decimal numbers
8. Find the sum of 10 unsigned bytes in an array and store the result in data segment.

Additional Exercise:

Add twenty 2- digit decimal numbers stored in the data segment and store the decimal result in data segment.

LAB NO.: 4 **Date:**

ARITHMETIC OPERATIONS: MULTIPLICATION AND DIVISION

Objectives

- To learn the usage of arithmetic instructions like MUL, IMUL, AAM, DIV, IDIV and AAD.
- Write ALP to perform addition and subtraction operations of 8 bit, 16 bit and 32 bit signed and unsigned numbers.

Introduction

Multiplication and division are two basic operations required in every application. In 8086 processor, the numbers to be multiplied can be of word size or byte size. For multiplication of byte valued numbers, one operand has to be in AL and other is specified to be in a byte register or a byte location in memory. The 16 bit product will be stored in AX. Similarly, for multiplying words, one operand must be in AX and the product will be stored in register pair DX-AX. IMUL instruction is to be used for multiplication of signed numbers. AAM instruction has to be used to obtain the unpacked BCD values after multiplying two BCD numbers. DIV, IDIV and AAD instructions are used for performing division of unsigned numbers, signed numbers and to obtain unpacked values before division respectively.

Solved exercise

Write an Assembly Language Program to multiply two 8 bit numbers.

```
Code                        Comment
CODE SEGMENT
ASSUME CS:CODE
START:MOV AL,11H            ; Copy first operand to AL register
      MOV BL,0FFH           ; Copy second operand to BL register
      MUL BL                ; Perform multiplication

      MOV AH,4CH
      INT 21H
CODE ENDS
END START
```

Output

	AX	BL
Before Execution	0000	00
After Execution	10EF	FF

Lab exercise

Given that the numbers and results are stored in memory, write an assembly language program to perform

1. Multiplication of
    i. Two unsigned words
    ii. Two signed bytes
    iii. Two signed words
2. Division of
    i. 16 - bit unsigned number by 8 - bit unsigned number
    ii. 16 - bit signed number by 8-bit signed number
    iii. 32 - bit unsigned number by 16 - bit unsigned number
    iv. 32 - bit signed number by 16 - bit signed number.
3. Multiplication of 16 bit unsigned numbers by repetitive addition.

Additional exercises

1. Find the square of an unsigned TWO digit hexadecimal number
2. Find the cube of an unsigned TWO digit hexadecimal number

LAB NO. 5 **Date:**

LOGICAL OPERATIONS

Objectives

- To learn the usage of logical instructions like AND, OR, XOR, NOT, SHR, SHL, ROL, ROR, RCL and RCR
- Write ALP to convert numbers from one base system to another system.

Introduction

Computers can interpret binary data. However, humans use integers or decimal numbers conveniently. In order to communicate with the computer, it is necessary to convert from one number system to another. Such programs requires some logical operations like AND, OR, shifting and negating. So, 8086 offers powerful logical instructions.

Logical shift and rotate instructions are called bit manipulation operation. These operations are designed for low level operations and are commonly used for low level control of input and output devices. The logical operations are software analogy of logic gates. They are commonly used to separate a bit or a group of bits in register or in a memory location for the purpose of testing, resetting or complementing.

Solved exercises

Write an Assembly Language Program to convert a 2 digit BCD value to its corresponding hexadecimal value and store the result in memory segment.

```
DATA SEGMENT
BCD DB 63H
HEX DB ?
DATA ENDS

CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START: MOV AX, DATA
       MOV DS, AX

       MOV AL, BCD          ; Copy the BCD number to AL register
       MOV BL, 0AH          ; Load BL register with 'A' – the hex value
       MOV CL, 04H          ; Copy 04 to CL register to shift the bits
       SHR AL, CL     ; Shift AL contents by 4 places to swap digits
       MUL BL               ; Multiply AL contents with BL contents
```

```
        MOV CL, BCD          ; Copy th BCD number to CL register
        AND CL, 0FH          ; Perform AND operation to mask higher 4 bits
        ADD AL, CL           ; Add AL contents with CL contents
        MOV HEX, AL          ; Copy the result to memory location 'HEX'

        MOV AH, 4CH
        INT 21H
CODE ENDS
END START
```

Explanation: The number (two digits) to be converted to HEX value is stored in memory location with label 'BCD'. The MS digit of BCD number is multiplied with $(10)_{10}$. This is then added to LS digit. This gives the value of BCD number in HEX.
Ex: Let the number be $63 = 6 \text{ X } (10)_{10} + 3 = 6 \text{ X } (A)_{16} + 03 = 3C + 3 = 3F$.

Output

	DS:[0000]	**DS:[0001]**
Before Execution	00	00
After Execution	63	3F

Lab exercises

Write an Assembly Language Program to
1. Convert four digit BCD number to HEX.
2. Convert two digit number from HEX to BCD.
3. Convert four digit number from HEX to BCD.

Additional exercises

Write an Assembly Language Program
1. To check whether the given number is even or odd using logical instructions
2. To count the number of 1's in the given binary 32 bit binary number store in the memory using logical instructions.

LAB NO. 6 **Date:**

PROCEDURES AND MACROS

Objectives

- Write an assembly language program to compute GCD and LCM of two numbers
- Write an assembly language program to compute factorial of a number
- Learn the usage of procedures and macros

Introduction

Often when writing programs, there will be a need to use a particular sequence of instructions at different points in a program. To avoid writing the sequence of instructions in the program each time it is needed, we can write the sequence as a separate "subprogram" called *procedure. 'CALL'* instruction is used to execute the sequence of instructions contained in the procedure.. Procedures can even be "nested", which means that one procedure calls another procedure as part of its instruction sequence.

The syntax for procedure declaration is given below

```
name PROC
    ; The procedure code
     RET
name ENDP
```

name - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures. A RET instruction at the end of the procedure returns execution to the next instruction in the mainline. PROC and ENDP are compiler directives, so they are not assembled into any real machine code. The Compiler just remembers the address of procedure.

A macro is a group of instructions we bracket and give a name to, at the start of our program. Macro tells the assembler every time we see a macro name in the program replace it with the group of instructions defined as the macro. Example given below shows the format to declare a macro without parameters and the format to call a macro.

```
        PUSH-ALL MACRO      ;Identifies start of macro and gives it a name
        ; Rest of the code to be executed.
  ENDM                      ;Identifies end of macro
 TEST PROC
 PUSH-ALL                   ;macro call
MOV AX, DATA
MOV DS, AX
```

The assembler inserts the machine code for the macro instruction in the object code version of

the program. Given below is the format to define and to call macro with parameters.

```
MOVE_ASCII MACRO NUMBER, SOURCE, DESTINATION
        ;Rest of the code
ENDM

MOVE_ASCII 03DH,BLOCK_START,BLOCK_DESTINATION ; Macro call
```

Solved exercises

Example to display a message using a macro

```
DATA SEGMENT
A DB 'MANIPAL$'
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE,DS:DATA
START:
        MOV DX,DATA
        MOV DS,DX

        DISP A

        MOV AH,4CH
        INT 21H

DISP MACRO MSG
        MOV DX,OFFSET MSG
        MOV AH,09
        INT 21H
        ENDM

CODE ENDS
END START
```

Write an Assembly Program to compute HCF of two numbers.

```
DATA SEGMENT
Num1 DW 0005h
Num2 DW 0002h
Ans DW ?
DATA ENDS
```

```
CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START: MOV AX, DATA
       MOV DS, AX

       MOV AX, Num1        ; Load first number to AX register
       MOV BX, Num2        ; Load second number to BX register
FIRST: CMP AX, BX          ; Compare two numbers
       JA NEXT             ; If AX contents are greater jump to label 'next'
       XCHG AX, BX         ; Otherwise exchange the contents of AX and BX
NEXT: MOV DX, 0000h        ; Initialize DX register with 0000
       DIV BX              ; Divide the AX register contents by BX content
       CMP DX, 0000h       ; Compare whether remainder is 0
       JE LAST             ; If remainder is 0, jump to label 'last'
       MOV AX, DX          ; Otherwise, copy the remainder to AX register
       JMP FIRST           ; Jump to label 'first'
LAST: MOV Ans, BX          ; Move the contents of BX to memory location 'Ans'
       MOV AH, 4CH
       INT 21H
CODE ENDS
END START
```

Explanation

To compute HCF of two numbers, copy the greatest number to AX register. Divide the greatest number with the second number which is copied to BX register. If the remainder is not zero, continue the division process. Once the remainder becomes zero, copy the number stored in BX (Since BX has the smallest value) which becomes the HCF of two numbers.

Ex: Numbers are 10, 15 AX = 15, BX = 10 , Remainder – 5, AX = 5, BX = 10

Loop back and swap values in AX and BX – AX= 10, BX = 5, Remainder – 0. Hence HCF is 5.

Output

	DS:[0000]	**DS:[0001]**	**DS:[0002]**	**DS:[0003]**	**DS:[0004]**	**DS:[0005]**
Before Execution	10	00	15	00	00	00
After Execution	10	00	15	00	05	00

Lab exercises

Write an Assembly Language Program to find:

1. GCD of two unsigned words using a

    a. Macro
    b. Procedure
2. GCD of four unsigned words using a
    a. Macro which finds GCD of two unsigned words
    b. Function which finds GCD of two unsigned words
3. LCM of two unsigned bytes.
4. LCM of four unsigned bytes using function which finds LCM of two unsigned bytes.
5. Factorial of unsigned byte using recursion

Additional exercises

1. Write an Assembly Language Program to compute the possible permutation and combination to select 'r' objects out of 'n' objects (nP_r, nC_r) using recursive factorial function.
2. Using procedure BCD_HEX which converts a 2-digit BCD into HEX, write an ALP to convert 4-digit BCD to HEX.

LAB NO. 7 **Date:**

SORTING AND SEARCHING

Objectives

- To understand the usage of conditional jump instructions.
- To search an element in an array
- To sort the elements of array in ascending or descending order

Introduction

Arrays are used to store the data elements. It may be necessary to perform multiple operations on array of elements like searching an element, sorting the array elements in ascending or descending order, inserting or deleting an element from the array etc. In such cases, a set of instructions might be executed repeatedly based on some condition. This is achieved using branch instructions in Assembly Language Program.

Normally 8086 keeps on executing one instruction after another in a sequence. However when a branch group of instructions is encountered a jump is affected, which can be forward or backward jump, if the condition for branching is satisfied. 8086 has very powerful branch instructions. It can perform a branch unconditionally, or based on a single flag value, or based on multiple flag values. There are instructions to branch to a subroutine and from a subroutine.

Conditional jump based on single flag: Some branch instructions perform a branch based on value of a single status flag. The status flag used in these instructions are Zero, Sign, Overflow, Carry and Parity.

Conditional jump based on more than one flag: Some instructions branch based on value of more than one flag. The conditional jump instructions like JBE and JNBE work on unsigned numbers. The instructions JLE, JNLE, JL and JNL work on signed numbers.

Unconditional jump instructions: The mnemonic for the instruction is JMP. The branch could be within the same code segment (near) or an intra segment jump. Near jump instructions can provide the destination address using the label in a relative mode or destination address is provided in a register or a memory location. The intra segment jump is also called the far jump. Here the destination is provided using a label or in two consecutive memory locations.

Solved exercises

Program to find the largest number in an array of elements

```
DATA SEGMENT                    ;start of data segment
X DW 0010H,52H,30H,40H,50H      ;initialize data segment
LAR DW ?
DATA ENDS                       ;end of data segment

CODE SEGMENT                    ;start of code segment
ASSUME CS:CODE,DS:DATA
START: MOV AX,DATA
       MOV DS,AX
       MOV CX,05H               ;load CX register with number of datawords in X
       LEA SI,X                 ;initialize SI to point to the first number
       MOV AX,[SI]              ;make a copy of the number pointed by SI ;in AX
       DEC CX                   ;set count value in CX for comparison
   UP: CMP AX,[SI+2]            ;compare two adjacent numbers(one is in ;AX and
                                    the other is pointed by SI+2)
       JA CONTINUE              ;if contents of AX is greater than the next
       MOV AX,[SI+2]            ;number in array retain the contents of AX
CONTINUE:ADD SI,2               ;if not make a copy of the larger number in ;AX point to the next
                                number
          DEC CX                ;decrement CX to check if all numbers are ;compared
          JNZ UP                ;if no continue to compare
          MOV LAR,AX            ;if yes make a copy of AX(largest number) ;in user defined memory
                                    location LAR
          MOV AH,4CH            ;terminate the process
          INT 21H
          CODE ENDS                     ;end of code segment
          END START
```

Output

	DS:[0000]	**DS:[0001]**	**DS:[0002]**	**DS:[0003]**	**DS:[0004]**	**DS:[0005]**	**DS:[0006]**	**DS:[0007]**	**DS:[0008]**
Before Execution	10	00	52	00	30	00	40	00	50
After Execution	10	00	52	00	30	00	40	00	50

	DS:[0009]	**DS:[0010]**	**DS:[0011]**
Before Execution	00	00	00
After Execution	00	52	00

Lab exercises

Write an Assembly Language Program to perform

1. Linear search in an array of 10 unsigned
   i. Bytes ii. Words
2. Smallest number in an array of N bytes
3. Sorting of a). Unsigned Bytes b). a). Unsigned Words
   c). Signed Bytes d) Signed Words
   Using
   i. Selection sort
   ii. Bubble sort

Additional exercises

Write an Assembly Language Program to perform

1. Binary search 2. Insertion sort

LAB NO: 8 **Date:**

PROGRAMMING USING BIOS AND DOS SERVICES

Objectives

- To understand character input with and without echo, display a character or string.
- To learn the usage of string instructions.
- To familiarize various 8086 interrupts

Introduction

DOS contains many functions that can be accessed by outside programs. These functions are invoked using assembly language instruction INT 21H. Some functions available in this interrupt are listed below

i. **Character input with echo:**
MOV AH, 01H
INT 21H
This instruction reads a character from the standard input device (keyboard) and echoes it to the standard output device (screen). AL register contains the ASCII value of the character input.

ii. **Display character:**
MOV AH, 02H
INT 21H
This instruction displays a character at the standard output device. DL register contains the ASCII value of the character to be displayed.

iii. **Direct Console I/O:**
MOV AH, 06H
INT 21H
This instruction reads a character from the keyboard or writes a character to the standard output device. If the DL register contains a value between 00H-FEH, it displays the character on screen. If the DL register contains FFH, and a character is ready, AL register contains the character input and Zero Flag is SET.
If the character is not ready when DL register contains FFH, Zero Flag is RESET and AL register contains 00H.

iv. **Character input without echo:**
MOV AH, 08H
INT 21H
This instruction reads a character from the keyboard without echoing it on the screen or waits until character is available. AL register contains the character input.

v. **Display character string:**

MOV AH, 09H
INT 21H
This instruction displays a string of characters on the screen. The string must be terminated with the character '$", which is not displayed. DS: DX register pair contains segment: offset of the string.

vi. **Buffered input:**
MOV AH, 0AH
INT 21H
This instruction reads string of characters from the keyboard, echoes the characters on the screen and places the characters into a buffer. DS: DX register pair contains segment: offset of the buffer.

To execute I/O operations, after linking the .OBJ file, execute .EXE file without 'TD'. For example to run the file saved by name display.asm, after executing link display.obj; execute 'display.exe'. To display the string 'AH' register must be loaded with '09h' value before calling the interrupt, INT 21H. It then displays the string pointed by DX register in the data segment. The string terminated by '$' is displayed on the COMMAND PROMPT screen.

Solved exercises

Write an Assembly Language Program to display a string 'FOURTH SEM IT" on the screen

```
DATA SEGMENT
MSG DB 'FOURTH SEM IT$'     ; Message to be displayed must terminate with '$'
DATA ENDS

CODE SEGMENT
ASSUME CS: CODE, DS: DATA

START: MOV AX, DATA
       MOV DS, AX

       MOV DX, OFFSET MSG          ;Initialize DX with offset of message
       MOV AH, 09H                 ; Display character string
       INT 21H

       MOV AH, 4CH                 ; Interrupt to terminate Program execution
       INT 21H
```

```
CODE ENDS
END START
```

Output

After executing ‘display.exe’, string is displayed on the screen as follows.
FOURTH SEM IT

Lab exercises

Write an Assembly Language Program to

1. Input a character without echo and with echo and display the character in next line.

2. Prompt the user to input two 2-digit HEX numbers from the keyboard and display the sum on the screen.
3. Repeat question 1 for decimal numbers.
4. Read a two-digit hexadecimal number from the keyboard and display its decimal equivalent on the screen.
    i) Use interrupt to read single character
    ii) Use interrupt to read string

Additional exercises

Write an Assembly Language Program to

1. Accept an array of 10 unsigned hexadecimal numbers from the keyboard and display the sorted array.
2. Accept an array of five hexadecimal numbers from the keyboard and display the GCD.
3. Right adjust each entered name to the right of the screen, one below the other.
4. Input 8 bit binary number from keyboard and display its decimal equivalent on the screen. (*Ex: If the input is 111111110 the result displayed is 254*)

LAB NO: 9 **Date:**

STRING MANIPULATION

Objectives

- Write Assembly Language Program to validate the password.
- Write Assembly Language Program to perform string manipulations.
- Learn the usage and applications of string instructions.

Features of string operations

A string instruction can specify the respective processing of one byte, word, or (80386 and later) double word at a time. Thus we could select a byte operation for a string with an odd number of bytes and a word operation for a string with an even number of bytes. Each string instruction has a byte, word, and double word version and assumes use of the ES:DI or DS:SI registers. The DI and SI should contain valid offset addresses. For the string instructions MOVS, STOS, CMPS and SCAS, ensure that the .EXE programs initialize the ES register. Use the suffixes B, W or D for handling byte, word or double word strings. Initialize CX register for REP to process the required number of bytes, words or double words. This lab illustrates the usage of string instructions.

Solved exercises

Write an Assembly Language Program to transfer 10 bytes from 'SRC' to 'DST', where SRC and DST represent 10 bytes of data stored in data segment.

```
DATA SEGMENT
SRC DB 11H,12H,13H,14H,15H,16H,17H,18H,19H,20H
DST DB 10 DUP (?)
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA
START: MOV AX,DATA
       MOV DS,AX
       MOV ES,AX          ; Initialize ES and DS registers with the base ;address of
                            DATA segment
       CLD                ; Clear the direction flag to increment SI & DI
       MOV CX,0010H       ; Load CX register with number of elements
       LEA SI, SRC        ; Point SI to first element in SRC
```

```
    LEA DI, DST          ; Point DI to first element in DST
    REP MOVSB            ; Copy the string from SI to DI and repeat.

    MOV AH, 4CH
    INT 21H
CODE ENDS
END START
```

Lab exercises

For the following programs use string instructions where ever necessary

Write an Assembly Language Program to

1. Validate the password entered from the keyboard and display an appropriate message.
2. Check whether string entered from the keyboard is palindrome or not, without reversing the string. Display the message accordingly .
3. Read substring and main string from the keyboard. Search for a substring in a main string and display the message accordingly.
4. Accept a sentence and toggle the case of every characters. Display the final message.

Additional exercises

Write an Assembly Language Program to

1. Insert a substring in a main string.
2. Count the number of vowels in a string.

LAB NO: 10 **Date:**

COUNTERS

Objectives

- To write Assembly Language Program for UP/DOWN counters
- To simulate a digital clock.
- Write ‘Delay’ procedures.

Introduction

Automation systems use PC and laptops to monitor different parameters of machines and production data. Counters may count parameters such as the number of pieces produced, the production batch number, and measurements of the amounts of material used. A counter is a device which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal. The most common type is a sequential digital logic circuit with an input line called the "clock" and multiple output lines. The values on the output lines represent a number in the binary, hexadecimal or BCD number system. Each pulse applied to the clock input increments or decrements the number in the counter. Counters are used in many applications like digital clock or tossing a die. In this lab, we will learn to write assembly language programs to simulate these applications using procedures and display the count on the screen.

Solved exercises

Write an Assembly Language Program to count from 00 to 99 in decimal and display the count on the screen.

```
CODE SEGMENT
ASSUME CS:CODE
START: MOV BH,33H

BACK:  ; Extract MS digit of count

    MOV BL,BH
    AND BL,0F0H

; Convert to decimal value
    MOV CL,4
    ROR BL,CL
```

```
        ADD BL,30H
        MOV DL,BL
        CALL DISP

; Extract the LS  digit  and convert to decimal value
        MOV BL,BH
        AND BL,0FH
        ADD BL,30H
        MOV DL,BL
        CALL DISP

        MOV AL,BH
        ADD AL,01H
        DAA
        MOV BH,AL

; Provide delay to display the next count value
    CALL DELAY

; Set the cursor to starting location of the same line
        MOV DL,13
        CALL DISP

        MOV DL,0FFH
        MOV AH,06
        INT 21H
        JNZ DOWN
UPP:    JMP BACK

DOWN:  CMP AL,13
        JNZ BACK
        MOV AH,4CH
        INT 21H

; Procedure to display the value
DISP PROC
        MOV AH,02
        INT 21H
        RET
DISP ENDP
```

```
; Delay function

DELAY PROC
 MOV SI,0FFFFH
  UP1: MOV DI,0FFFH
  UP:   DEC DI
        JNZ UP
        DEC SI
        JNZ UP1
      RET
DELAY ENDP

   CODE ENDS
   END START
```

Output

Decimal count from 00 to 99 is displayed on the screen.

Lab exercises

1. Write Assembly Language Program to simulate following counters which terminate on any key pressed by user.
   a. A two digit HEX
      i. Up ii. Down
   b. two digit decimal
      i. Down ii. UP DOWN counter
   c. two digit HEX Counter
      i. Up counter -N1 to N2 (N1<N2)
      ii. Down counter-N1 to N2 (N1>N2)
      iii. Up down counter-N1 to N2
2. Write Assembly Language Program to simulate digital clock

Additional exercises

1. Write Assembly Language Program
   a. To simulate Digital clock with AM/PM provision
   b. To simulate a die tossing
2. For a TWO digit Hex UP DOWN counter

LAB NO: 11 **Date:**

ADVANCED SCREEN PROCESSING

Objectives

- To learn advanced screen processing features of 8086.
- To program in text mode and graphics mode of 8086.
- Write Assembly Language Program using various BIOS interrupts

Introduction

Setting the video mode

BIOS INT 10H,function 00h,can set the mode for the currently executing program or can switch between text and graphics. Setting the mode also clears the screen. As an example, mode 03 represents text mode, color and screen resolution, depending on the type of monitor.

Text mode

This mode is used for the normal display of ASCII characters on the screen. Processing is similar for both monochrome and color, except that color does not support the underline attribute. Text mode provides access to the full extended ASCII 256-character set.

Attribute byte

An attribute byte in text (not graphics) mode determines the characteristics of each displayed character. When a program sets an attribute, it remains set: that is, all subsequent displayed characters have the same attribute until another operation changes it. We can use INT 10H functions to generate a screen attribute and perform such actions as scroll up, scroll down, read attribute or character or display attribute or character. If we use DEBUG to view the video display area of our system, we'll see each one-byte character, immediately followed by its one-byte attribute.

The attribute byte has the following format, according to bit position:

		Background			Foreground			
Attributes:	BL	R	G	B	I	R	G	B
Bit Number:	7	6	5	4	3	2	1	0

Text Mode

The letters R, G and B indicate bit positions for red, green, and blue, respectively.

- Bit 7(BL) sets blinking

- Bits 6-4 determine the screen background
- Bits 3(I) sets high intensity
- Bits 2-0 determine the foreground (for the character being displayed)

The REG bits define a color –on both color and monochrome, 000 is BLACK and 111 is WHITE. For example, an attribute set with the value 0000 0111 means black background with white foreground.

Screen pages

Text modes allow us to store data in video memory in pages. Pages numbers are 0 through 3 for normal 80-colum mode (and 0 through 7 for the rarely used 40-column screen). In 80-column mode, page number 0 is the default and begins in the video display area at B800 [0], page 1 begins at B900 [0], page 2 at BA00 [0], and page 3 at BB00 [0].

We may format any of the pages in memory ,although we can display only one page at a time, each character to be displayed on the screen requires two bytes of memory—one byte for the character and a second for its attribute. In this way, a full page of characters for 80 columns and 25 rows requires 80*25*2=4000 bytes. The amount of memory actually allocated for each page is 4k, or 4,096 bytes, so that 96 unused bytes immediately follow each page.

Direct video display

The first two bytes in the video display area represent one screen location, for row 00, column 00, and the last bytes at F9EH and F9FH represent the screen location for row 24, column 79. Simply moving a character; attribute into the video area of the active page causes the character to appear immediately on the screen.

The starting position of a page in the video display area based on the fact that there are 80*2=160 columns in a row. The Effective Address of starting position, then, for row 10, column 15, is (160*10) + (15*2) = 1630).

Graphics mode

Graphics adapters have two basic modes of operation: text (the default) and graphics.
Use BIOS INT 10H, function 00h, to set graphics or text mode, as the following two examples show:

1. Set graphics mode for VGA:

```
MOV AH, 00H   ; Request set mode
MOV AL, 0CH    ; Color graphics
INT 10H          ; CALL BIOS
```

2. Set text mode:

```
MOV AH, OOH       ; Request set mode
MOV AL, 03H       ; Color text
INT 10H           ; Call BIOS
```

Solved exercise

Write an Assembly Language Program to display a string on screen using video RAM.

```
DATA SEGMENT
MSG DB 'HELLO'
COL DB 40
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
      MOV DS, AX



  ;Set text mode and it also clears the screen

     MOV AH,0
     MOV AL,3
     INT 10H
     MOV DL,00
      LEA DI, MSG
      MOV CL,COL



 UP:   MOV DH,[DI]
       PUSH DS
       MOV AX,0B800H
       MOV DS,AX

       MOV BL,12
       MOV AL,160
       MUL BL
       MOV BX,AX
```

```
    MOV AL,2
    MUL CL
    ADD AX,BX
    MOV SI,AX
    MOV [SI],DH
    MOV BYTE PTR [SI+1],47H
    INC DI
    INC CL
    POP DS
    INC DL
    CMP DL,05
    JNZ UP
    MOV AH,4CH
    INT 21H

  CODE ENDS
  END START
```

Output

Text "HELLO" will be displayed in the center of the screen with background as RED color and foreground as WHITE color.

Solved exercise

Write an Assembly Language Program to display a rectangle on the screen using interrupt for scrolling.

```
CODE SEGMENT
ASSUME CS:CODE
START:
        MOV AH,0
        MOV AL,03
        INT 10H

        MOV  AH,07
        MOV AL,7
        MOV BH,01110000B
        MOV CH,10
        MOV CL,2
```

```
        MOV DH,15
        MOV DL,40
        INT 10H

        MOV AH,4CH
        INT 21H
CODE ENDS
END START
```

Output

A rectangle is displayed on the screenshot as shown in the Fig. 5 given below.

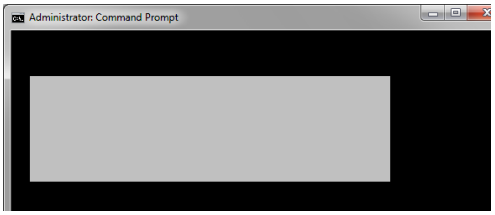

Fig. 5: Screenshot of output

Lab exercises

Write Assembly Language Program to perform

1. Screen saver-concentric colored rectangle such that for every second, the color shifts.
2. Display a string at the center of the screen using video RAM. Move the string UP, DOWN, LEFT, RIGHT using U, D, L, R keys.
3. Display a window at the center of the screen using scrolling. Prompt the user to enter a string. When enter key is pressed, the string occupies the window. Assume that window can accommodate 5 strings. After inputting the 5 string, the window should automatically scroll up.

Additional exercise

Accept string input from the user and display the string at the center of the screen using video RAM.

LAB NO: 12 **Date:**

APPLICATION OF ADVANCED SCREEN PROCESSING

Objective

- To write an Assembly Language Program for menu driven calculator.

Introduction

We have studied how 8086 can be programmed to display a text on the screen using advanced screen processing techniques. This can be used to simulate multiple applications, one of which is a simple calculator. A calculator performs basic arithmetic operations like add, subtract, multiply and divide. In this lab we write an Assembly Language Program to simulate a menu driven calculator. A menu will be displayed on the screen with different background colors and as the user scrolls up/down, the selection has to be highlighted suitably.

Lab exercise

Design a menu driven simple calculator.

REFERENCES

1. K Udaya Kumar, B.S. Umashankar, "Advanced Microprocessors and IBM – PC Assembly Language Programming", Tata McGraw-Hill, 2010.
2. Douglas V. Hall, "Microprocessors and Interfacing – Programming and Hardware", Tata McGraw Hill, 2006.
3. Peter Abel, "IBM PC Assembly Language and Programming", 3rd Edition, Prentice Hall if India, 1999.