
# UNIVERSITY OF HAIL

## College of Engineering

### Department of Electrical Engineering

# LAB MANUAL

## FOR

# EE 390 DIGITAL SYSTEMS ENGINEERING

# Table of Contents

# Experiment #1 MS-DOS Debugger (DEBUG)

## 1.0 Objectives:

The objective of this experiment is to introduce the "DEBUG" program that comes with MS-DOS and Windows operating systems. This program is a basic tool to write, edit and execute assembly language programs.

In this experiment, you will learn DEBUG commands to do the following:

- Examine and modify the contents of internal registers
- Examine and modify the contents of memory
- Load, and execute an assembly language program

## 1.1 Introduction:

DEBUG program which is supplied with both DOS and Windows, is the perfect tool for writing short programs and getting acquainted with the Intel 8086 microprocessor. It displays the contents of memory and lets you view registers and variables as they change. You can use DEBUG to test assembler instructions, try out new programming ideas, or to carefully step through your program. You can step through the program one line at a time (called *tracing*), making it easier to find logic errors.

### 1.2 Debugging Functions

Some of the basic functions that the debugger can perform are the following:

- Assemble short programs
- View a program's source code along with its machine code
- View the CPU registers and flags (See Table 1 below)
- Trace or execute a program, watching variables for changes
- Enter new values into memory
- Search for binary or ASCII values in memory
- Move a block of memory from one location to another
- Fill a block of memory
- Load and write disk files and sectors

The following table shows a list of some commonly used DEBUG commands.

| COMMAND | SYNTAX |
|---|---|
| **Register** | **R** [Register Name] |
| **Dump** | **D** [Address] |
| **Enter** | **E** [Register Name] |
| **Fill** | **F** [Register name] |
| **Assemble** | **A** [Starting address] |
| **Un-assemble** | **U** [Starting Address] |
| **Trace** | **T** [Address][Number] |
| **Go** | **G** [Starting Address] [Breakpoint Add.] |

**Table 1: DEBUG commands**

The Internal Registers and Status Flags of the 8086 uP are shown in the following tables.

| Flag | Flag |
|---|---|
| CF | SF |
| PF | IF |
| AF | DF |
| ZF | OF |

| | | |
|---|---|---|
| **AX** | **BX** | |
| **DS** | **CS** | |

**Table 2: Internal Registers and Status Flags**

## 1.3 Pre-lab:

1. Name a few computer operating systems. Which operating system do you mostly use?
2. What is the full form for MS-DOS?
3. What is the difference between a logical address and a physical address? Show how a physical address is generated from a logical address.
4. What are the following registers used for: DS, CS, SS, SP, IP, AX
5. Define the function each of the following flag bits in the flag register: Overflow, Carry, Sign, and Zero.

## 1.4 Lab Work:

### A. Loading the DEBUG program

1. Load the DEBUG program by typing ***debug*** at the MS-DOS prompt, as shown in the example below: C:\WINDOWS>debug

2. You will see a dash (-) in the left-most column on the screen. This is the DEBUG prompt.

3. Type a (?) to see a list of available commands.

4. Return to MS-DOS by entering Q. What prompt do you see?

***Note***: You have to hit Carriage Return (CR) key (or ENTER key) on the keyboard after you type any **debug** command.

### B. Examining and modifying the contents of the 8086's internal registers

1. Use the REGISTER command to display the current contents of all the internal registers by typing R.

a) List the values of the following registers:

b) What is the address of the next instruction to be executed?

c) What is the instruction held at this address?

| | |
|---|---|
| | **AX** |
| | **SI** |
| | **DI** |
| | **DX** |
| | **IP** |

2. Enter the command: R CL (hit <CR>) What happens and why?

3. Use a REGISTER command to first display the current contents of SI and then change this value to 0102h.

4. Use a REGISTER command to first display the current contents of IP and then change this value to 0300h.

5. Use a REGISTER command to first display the current contents of the flag register and then ***reset*** the overflow, sign, and auxiliary flags.

6. Redisplay the contents of all the internal registers. Compare the displayed register contents with those observed in step 1 above. What instruction is now pointed by CS: IP?

**C. Examining and modifying the contents of memory**

1 Use the DUMP command (D) to display the first 50 bytes of the current data segment.

2 Use the DUMP command (D) to display the first 50 bytes of the code segment starting the current value of CS: IP.

3 Use the ENTER command (E) to load locations CS:50, CS:52, and CS:54 with AA, BB, and CC, respectively.

4 Use the ENTER command (E) to load five consecutive byte-wide memory locations starting at CS:55 with data ‘00’.

5 Verify the result of steps 3 and 4 using the DUMP command.

6 Use the FILL command (F) to initialize the 10h storage locations starting at DS:10 with the value 11h, the 10h storage locations starting at address DS:30 with 22h, the 10h storage locations starting at address DS:50 with 33h, and the 10h storage locations starting at address DS:70 with 44h

7 Verify the result of step 6 using the DUMP command.

**D. Coding instructions in 8086 machine language**

1. Enter each of the following instructions starting at address CS:100 one-by-one using the ASSEMBLE command (A).

| |
|---|
| **MOV AX,BX** |
| **MOV AX, AAAAh** |
| **MOV AX,[BX]** |
| **MOV AX,[0004H]** |
| **MOV AX,[BX+SI]** |
| **MOV AX,[SI+4H]** |
| **MOV AX,[BX+SI+4H]** |

2. Using the UNASSEMBLE command (U), obtain
   a) the machine code of each of the instructions in step 1
   b) the number of bytes required to store each of the machine code instructions in step 1.
   c) the starting address of each instruction.
   d) Why are the starting addresses of the above instructions not consecutive?

| **Instruction** |
|---|
| **MOV BX, CX** |
| **MOV SI, AAAA** |
| **MOV AX,[DI]** |
| **MOV BX,[4]** |
| **MOV DI,[BX+SI]** |
| **MOV IP,[SI+4]** |
| **MOV AX,[BP+DI+F]** |

**E. Coding instructions in 8086 machine language**

1. Using the ASSEMBLE command (A), load the program shown below into memory starting at address CS: 0100.

```
        MOV   SI, 0100H
        MOV   DI, 0200H
        MOV   CX, 010H
BACK:   MOV   AH, [SI]
        MOV   [DI], AH
        INC SI
        INC DI
        DEC   CX
        JNZ   BACK
```

2. Verify the loading of the program by displaying it with the UNASSEMBLE (U) command.

a) How many bytes of memory does the program take up?

b) What is the machine code for the DEC CX instruction?

c) What is the address offset for the label BACK?

3. Fill 16 bytes of memory locations starting at DS: 0200 with value 45H and verify.

4. Execute the above program one instruction at a time using the TRACE command (T). Observe how the values change for registers: AX, CX, SI, DI flag register, and IP.

5. Run the complete program by issuing a single GO command (G).

a) What is the starting address for this command?

b) What is the ending address for this command?

6. What are the final values of registers: AX, CX, SI, and DI?

7. Describe the function of the above program.

# Experiment #2 Addressing Modes and Data

# Transfer using TASM

## 2.0 Objective

The objective of this experiment is to learn various addressing modes and to verify the actions of data transfer.

## 2.1 Introduction

Assembly language program can be thought of as consisting of two logical parts: data and code. Most of the assembly language instructions require specification of the location of the data to be operated on. There are a variety of ways to specify and find where the operands required by an instruction are located. These are called addressing modes. This section is a brief overview of some of the addressing modes required to do basic assembly language programming.

The operand required by an instruction may be in any one of the following locations

- in a register internal to the CPU
- in the instruction itself
- in main memory (usually in the data segment)

Therefore the basic addressing modes are register, immediate, and memory addressing modes

**1. Register Addressing Mode**

Specification of an operand that is in a register is called register addressing mode. For example, the instruction

MOV AX,CX requires two operands

and both are in the CPU registers.

**2. Immediate Addressing Mode**

In this addressing mode, data is specified as part of the instruction. For example, in the following instruction

MOV BX,1000H the immediate value

1000H is placed into the register BX.

**3. Memory Addressing Mode**

A variety of modes are available to specify the location of an operand in memory. These are direct, register-indirect, based, indexed and based-indexed addressing modes

## 2.2 Pre-lab:

Using turbo debugger, initialize the registers and memory locations before executing the following statements and fill the corresponding columns in Table 1.

Example: Initialize AL=10H, SI=30H, BX=1000H, memory location DS:1030H=2AH

**MOV AL,[BX+SI]**

(see TABLE 1 for the results after execution of this instruction)

a. Initialize AX=200H; DI=50H; memory location DS:58H=9C, DS:59H=9C

**MOV AX,[DI+8]**

b. Initialize BX=1111H;

**MOV BX,2000H**

c. Initialize BX=1010H; CX=2222H

**XCHG BX,CX**

d. Initialize AX=2222H; DI=80H; memory location DS:80H=55H, DS:81H=55H

**MOV [DI],AX**

e. Initialize AX=1000H; BX=200H; SI=10H; memory location DS:215H=2222H

**MOV AX,[BX+SI+5]**

f. Initialize AX=0H; BP=100H; memory location DS:102H=11H, DS:103H=11H

MOV AX,[BP+2]

| Statement | Source | | Destination | | | Addressing Mode |
|---|---|---|---|---|---|---|
| | Register/ Memory | Contents | Register/ Memory | Contents before execution | | |
| **MOV AL,[BX+SI]** | Memory | 2A | Memory | 10 | 2A | Based indexed |
| **MOV AX,[DI+8]** | | | | | | |
| **MOV BX,2000H** | | | | | | |
| **XCHG BX,CX** | | | | | | |
| **MOV [DI],AX** | | | | | | |
| **MOV AX,[BX+SI+5]** | | | | | | |
| **MOV AX,[BP+2]** | | | | | | |

**TABLE 1**

## 2.3 Lab Work:

**USING AN ASSEMBLER**

In Experiment 1, we learned to use the DEBUG program development tool that is available in the PC's operating system. This DEBUG program has some limitations. Program addresses must be computed manually (usually requiring two phases – one to enter the instructions and a second to resolve the addresses), no inserting or deleting of instructions is possible, and symbolic addresses cannot be used. All of these limitations of DEBUG can be overcome by using the proper assembly language tools.

Assembly language development tools, such as Microsoft's macro-assembler (MASM), Borland's Turbo assembler (TASM) together with the linker programs, are available for DOS. An assembler considerably reduces program development time.

Using an assembler, it is very easy to write and execute programs. When the program is assembled, it detects all syntax errors in the program – gives the line number at which an error occurred and the type of error.

We will be using the Turbo assembler (TASM) and linker (TLINK) programs in this lab.

**Program Template**

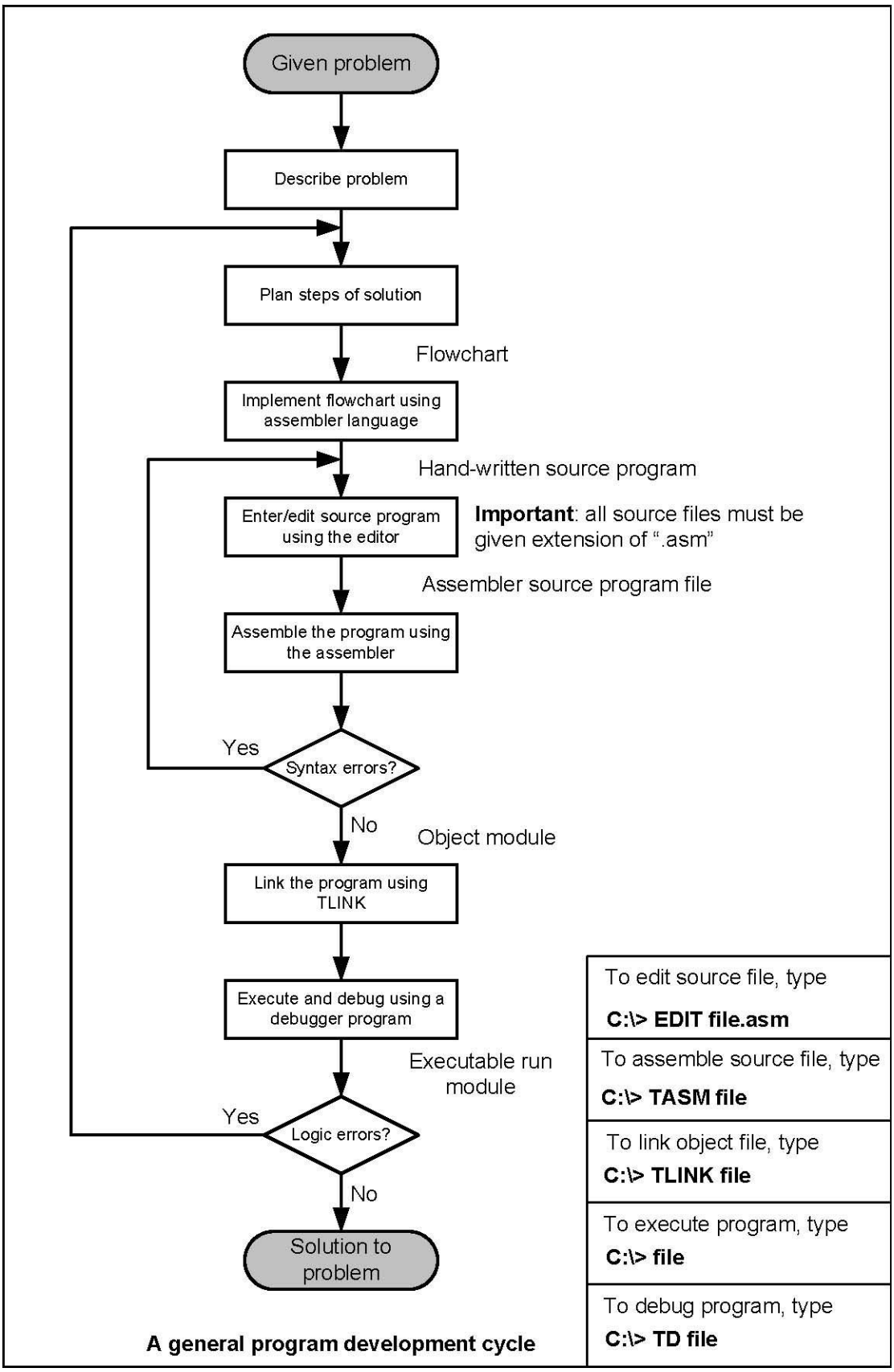
The following program template must be followed when using the Turbo assembler to write programs.

```
TITLE                    "Experiment 2"
.MODEL                   SMALL
.STACK                   032h
.DATA
                         ◄
……..

.CODE …….. ……..
END
```

Any line starting with a ';' (semi-colon) is considered a comment and is ignored by the assembler.

A typical program development cycle using an assembler as a development tool is illustrated in the flowchart below.




**A general program development cycle**

## 2.4 EXAMPLES

**Program 1:** Enter the following program in an editor. Save the program as "program1.asm". Assemble and link the program. Since the program does nothing except for transferring the contents from one register to another, view and verify the action of each statement using turbo debugger.

TITLE "Program to verify register and immediate addressing modes"

```
.MODEL SMALL              ; this defines the memory model
.STACK 100                ; define a stack segment of 100 bytes
.DATA                     ; this is the data segment

.CODE                     ; this is the code segment

        MOV AX,10         ;copy AX with hex number 10
        MOV BX,10H        ;copy BX with hex number 10
        MOV CL,16D        ;copy CL with decimal number 16
        MOV CH,1010B      ;copy CH with binary number 1010
        INC AX            ;increment the contents of AX register
        MOV SI,AX         ;copy SI with the contents of AX
        DEC BX            ;decrement the contents of BX register
        MOV BP,BX         ;copy BP with the contents of BX register

        MOV AX,4C00H      ; Exit to DOS function
        INT 21H
END     ; end of the program
```

In assembler we have to explicitly perform many functions which are taken for granted in high level languages. The most important of these is exiting from a program. The last two lines

```
MOV  AX,4C00H
INT 21H
```

in the code segment are used to exit the program and transfer the control back to DOS.

Procedure (to be followed for all programs):

- **Edit** the above program using an editor. Type "**edit program1.asm**" at the DOS prompt. Save your file and exit the editor. Make sure your file name has an extension of "**.asm**".
- **Assemble** the program created in (a). Type "**tasm program1**" at the DOS prompt. If errors are reported on the screen, then note down the line number and error type from the listing on the screen. To fix the errors go back to step (a) to edit the source file. If no errors are reported, then go to step (c).
- **Link** the object file created in (b). Type "**tlink program1**" at the DOS prompt. This creates an executable file "program1.exe".
- Type "**program1**" at the DOS prompt to run your program.

**Note**: You have to create your source file in the same directory where the TAMS.exe and TLINK.exe programs are stored.

**Program 2:** Write a program for TASM that stores the hex numbers 20, 30, 40, and 50 in the memory and transfers them to AL, AH, BL, and BH registers. Verify the program using turbo debugger; specially identify the memory location where the data is stored.

```
TITLE "Program to verify memory addressing modes"

.MODEL SMALL .STACK 100 .DATA                                                    ;


num                          DB                  10,20,30,40

.CODE

MOV AX,@DATA MOV DS,AX

LEA SI,num


MOV AL,[SI] MOV AH,[SI+1] MOV CL,[SI+2] MOV CH,[SI+3]


MOV AX, 4C00H INT 21H

END
```

The directive DB 'Define Byte' is used to store data in a memory location. Each data has a length of byte. (Another directive is DW 'Define Word' whose data length is of two bytes) The label 'num' is used to identify the location of data. The two instructions

```
MOV   AX,@DATA
MOV DS,AX
```

together with LEA SI,num

are used to find the segment and offset address of the memory location 'num'. Notice that memory addressing modes are used to transfer the data.

**Program 3:** Write a program that allows a user to enter characters from the keyboard using the character input function. This program should also store the characters entered into a memory location. Run the program after assembling and linking. Verify the program using turbo debugger, specially identify the location where the data will be stored.

```
TITLE "Program to enter characters from keyboard" .MODEL SMALL ; this
defines the memory model .STACK 100 ; define a stack segment of 100 bytes
.DATA    ; this is the data segment


        char_buf DB 20 DUP(?) ; define a buffer of 20 bytes


  .CODE

                                        MOV AX,@DATA  MOV DS, AX

                                        LEA SI, char_buf
  AGAIN:
                                        MOV AH, 01 INT 21H

                                        MOV [SI], AL


                                        INC SI CMP AL, 0DH JNE AGAIN


                                        MOV AX, 4C00H  INT 21H

   END
```

The directive DB when used with DUP allows a sequence of storage locations to be defined or reserved. For example

```
DB 20 DUP(?)
```

reserves 20 bytes of memory space without initialization. To fill the memory locations with some initial value, write the initial value with DUP instead of using 'question mark'. For example DB 20 DUP(10) will reserve 20 bytes of memory space and will fill it with the numbers 10.

The Keyboard input function waits until a character is typed from the keyboard. When the following two lines

```
MOV   AH,01
INT 21H
```

are encountered in a program, the program will wait for a keyboard input. The ASCII value of the typed character is stored in the AL register. For example if 'carriage return' key is pressed then AL will contain the ASCII value of carriage return i.e. 0DH

## 2.5 EXERCISE

Write a program in TASM that reserves a memory space ‘num1’ of 10 bytes and initializes them with the hex number ‘AA’. The program should copy the 10 bytes of data of ‘num1’ into another memory location ‘num2’ using memory addressing mode. Verify the program using turbo debugger.

Hint : Use DB instruction with DUP to reserve the space for ‘num1’ of 10 bytes with the initialized value of ‘AA’. Again use DB with DUP to reserve another space for ‘num2’, but without initialization. Use memory content transfer instructions to copy the data of ‘num1’ to ‘num2’.

# Experiment #3

# Arithmetic Instructions

## 3.0 Objective

The objective of this experiment is to learn the arithmetic instructions and write simple programs using TASM

## 3.1 Introduction

Arithmetic instructions provide the micro processor with its basic integer math skills. The 80x86 family provides several instructions to perform addition, subtraction, multiplication, and division on different sizes and types of numbers. The basic set of assembly language instructions is as follows

| | |
|---|---|
| Addition: | ADD, ADC, INC, DAA |
| Subtraction: | SUB, SBB, DEC, DAS, NEG |
| Multiplication: | MUL, IMUL |
| Division: | DIV, IDIV |
| Sign Extension: | CBW, CWD |

Examples:

ADD AX,BX

adds the content of BX with AX and stores the result in AX register.

ADC AX,BX adds the content of BX, AX and the carry flag and store it in the AX register. It is commonly used to add multibyte operands together (such as 128-bit numbers)

DEC BX decreases the content of BX register by one

MUL CL multiplies the content of CL with AL and stores the result in AX register

MUL CX multiplies the content of CX with AX and stores the 16-bit upper word in DX and 16-bit lower word in the AX register

IMUL CL is same as MUL except that the source operand is assumed to be a signed binary number

## 3.2 Pre-lab:

1. Write a program in TASM that performs the addition of two byte sized numbers that are initially stored in memory locations ‘num1’ and ‘num2’. The addition result should be stored in another memory location ‘total’. Verify the result using turbo debugger.

[Hint: Use DB directive to initially store the two byte sized numbers in memory locations called ‘num1’ and ‘num2’. Also reserve a location for the addition result and call it ‘total’]

2. Write a program in TASM that multiplies two unsigned byte sized numbers that are initially stored in memory locations ‘num1’ and ‘num2’. Store the multiplication result in another memory location called ‘multiply’. Notice that the size of memory location ‘multiply’ must be of word size to be able to store the result. Verify the result using turbo debugger.

## 3.3 Lab Work:

**Example Program 1:**Write a program that asks to type a letter in lowercase and then convert that letter to uppercase and also prints it on screen.

```
TITLE "Program to convert lowercase letter to uppercase"

 .MODEL SMALL .STACK 100 .DATA



 MSG1 DB MSG2 DB  CHAR DB                                ‘Enter0DH,0?, ‘$’


.CODE
        MOV AX,@DATA ; get the address of the data segment
        MOV DS,AX ; and store it in register DS

        MOV AH,9 ; display string function LEA SI,MSG1 ; get
        memory location of first message MOV DX,[SI] ; and store it
        in the DX register INT 21H ; display the string


        MOV AH,01 ; single character keyboard input function INT 21H ; call the function,
        result will be stored in AL (ASCII code)

        SUB AL,20H ; convert to the ASCII code of upper case LEA SI,CHAR ; load the
        address of the storage location MOV [SI],AL ; store the ASCII code of the
        converted letter to memory
```

```
MOV AH,9 ; display string function
        LEA SI,MSG2 ; get memory location of second message
        MOV DX,[SI] ; and store it in the DX register
        INT 21H ; display the string

        MOV AX, 4C00H ; Exit to DOS function INT
        21H
```

String output function is used in this program to print a string on screen. The effective address of string must first be loaded in the DX register and then the following two lines are executed

```
MOV AH,09     INT 21H
```

**Exercise 1:** Modify the above program so that it asks for entering an uppercase letter and converts it to lowercase.

**Example Program 2:** The objective of this program is to enter 3 positive numbers from the keyboard (0-9), find the average and store the result in a memory location called 'AVG'. Run the program in turbo debugger and verify the result.

```
TITLE "Program to calculate average of three numbers" .MODEL
SMALL ; this defines the memory model .STACK 100 ; define a
stack segment of 100 bytes .DATA ; this is the data segment


        msg 'Enter the number: ',0DH,0AH,'$'
        num DB 3 DUP(?)
        average DW ?



.CODE                                   ; this is the code segment

                                        MOV AX,@DATA MOV DS,AX

                                        MOV CL,03
START:

                                        MOV AH,9 LEA SI,msg MOV DX,[SI] INT 21H


                                         MOV AH,01   INT 21H


(ASCII)

                                        SUB AL,30H

                                         LEA SI,num
```

```
MOV [SI],AL
  DEC CL
CMP CL,0
JE ADD_IT
  INC SI
JMP ADD_IT

ADD_IT:                    MOV CL,02
                             LEA SI,NUM
                           MOV AL,[SI]
AGAIN:                     ADD AL,[SI+1]
                           CMP CL,0
                           JE DIVIDE
                             INC SI
result
                           JMP AGAIN

DIVIDE:                    MOV AH,0
                             MOV CL,03
                           DIV CL
                           LEA SI,average
                           MOV [SI],AX

                           MOV AX, 4C00H
                            INT 21H

  END
```

**Exercise 2:** Write a program in TASM that calculates the factorial of number 5 and stores the result in a memory location. Verify the program using turbo debugger [Hint: Since 5! = 5x4x3x2x1, use MUL instruction to find the multiplication. Store 5 in a register and decrement the register after every multiplication and then multiply the result with the decremented register. Repeat these steps using conditional jump instruction]

**Exercise 3:** Modify the factorial program such that it asks for the number for which factorial is to be calculated using string function and keyboard input function. Assume that the number will be less than 6 in order to fit the result in one byte.

# Experiment #4 Shift and

# Rotate Instructions

## 4.0 Objectives:

The objective of this experiment is to write programs demonstrating the applications of Shift and Rotate instructions. In this experiment, you will do the following:

- Learn to use Shift and Rotate instructions
- Write programs demonstrating the applications of Shift/Rotate instructions
- Execute programs using Turbo Debug and TASM

## 4.1 Introduction:

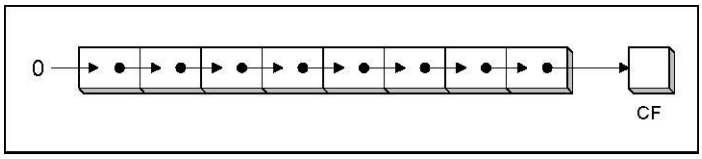
**Shift Instructions**

The 8086 can perform two types of Shift operations; the *logical* shift and the *arithmetic* shift. There are four shift operations (SHL, SAL, SHR, and SAR).

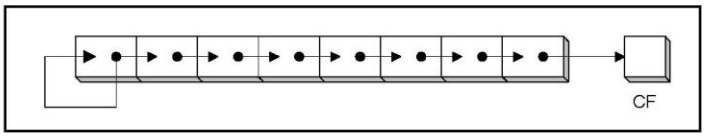
| Mnemonic |
|---|
| SAL |
| SHL |
| SAL |
| SHL |

If the source operand is specified as CL instead of 1, then the count in this register represents the number of bit positions the contents of the operand are to be shifted. This permits the count to be defined under software control and allows a range of shifts from 1 to 255 bits.

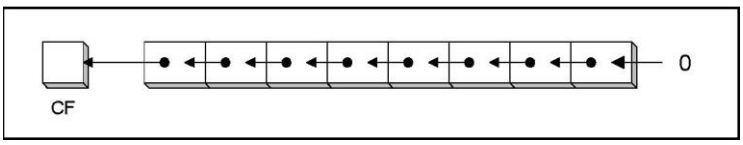
A logical shift fills the newly created bit position with zero:



An arithmetic shift fills the newly created bit position with a copy of the number's sign bit.



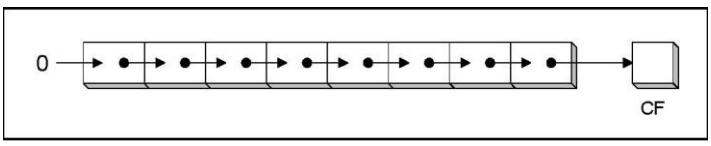

The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.




Shifting left 1 bit multiplies a number by 2 and shifting left *n* bits multiplies the operand by $2^n$. **For example**:

| | | | |
|---|---|---|---|
| MOV BL, 5<br>SHL BL, 1 | Before:<br>After: | 00000101<br>00001010 | = 5<br>= 10 |

The SHR (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero.

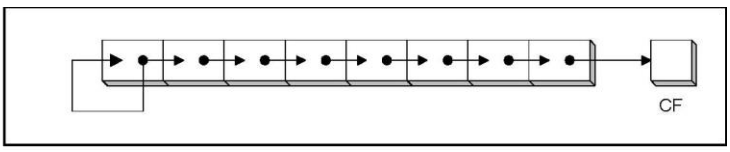



Shifting right 1 bit divides a number by 2 and shifting right *n* bits divides the operand by $2^n$.

**For example**:

| | | | |
|---|---|---|---|
| MOV DL, 12<br>SHR DL, 1 | Before:<br>After: | 00001100<br>00000110 | = 12<br>= 6 |

SAL is **identical** to SHL**.** SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand. An arithmetic shift preserves the number's sign.




**For example:**

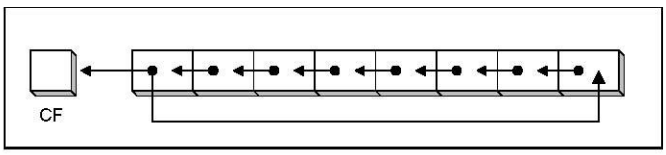
| | |
|---|---|
| MOV BL, -40<br>SAR BL, 1 | BL = -20 |

**Rotate Instructions**

The 8086 can perform two types of rotate operations; the *rotate* without carry and the *rotate* through carry. There are four rotate operations (ROL, ROR, RCL, and RCR).

| Mnemonic |
|---|
| ROL |
| ROR |
| RCL |
| RCR |

ROL shifts each bit of a register to the left. The highest bit is copied into both the Carry flag and into the lowest bit of the register. No bits are lost in the process.

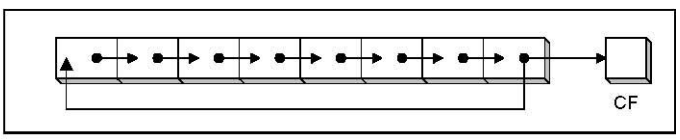



**For example:**

```
MOV AL,11100010B
ROL AL,1                ; AL = 11000101B

MOV BL,0A5H
MOV CL, 4
ROL BL, CL              ; BL = 5AH
```

ROR shifts each bit of a register to the right. The lowest bit is copied into both the Carry flag and into the highest bit of the register. No bits are lost in the process.

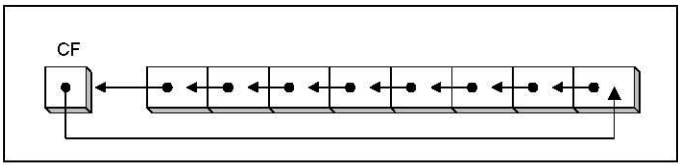



**For example:**

```
MOV AL, 00001011B
ROR AL, 1               ; AL = 10000101B

MOV BL, 90H
MOV CL, 4
ROR BL, CL              ; BL = 09H
```

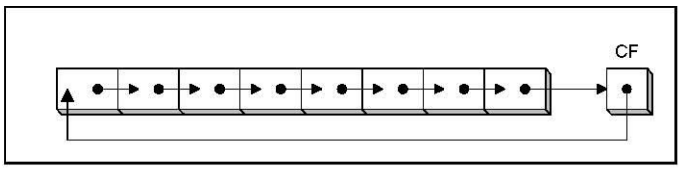
RCL (rotate carry left) shifts each bit to the left. It copies the Carry Flag to the least significant bit and copies the most significant bit to the Carry flag.




**For example:**

```
CLC                 ; clear carry flag, CF = 0
MOV BL,A4H          ; CF = 0, BL = 10100100B
RCL BL,1            ; CF = 1, BL = 01001000B
RCL BL,1            ; CF = 0, BL = 10010001B
```

RCR (rotate carry right) shifts each bit to the right. It copies the Carry Flag to the most significant bit and copies the least significant bit to the Carry flag.




**For example:**

```
STC                 ; set carry flag, CF = 1
MOV AH,14H          ; CF = 1, AH = 00010100B
RCR AH,1            ; CF = 0, AH = 10001010B
```

## 4.2 Pre-lab:

Run the following instructions in Turbo Debugger and state the values of source and destination for each Shift or Rotate instruction.

1. **MOV AL, 6BH  SHR AL,1 SHL AL,3**
2. **MOV AX, 0AAAAH  MOV CL,8 SHL AX,CL**
3. **MOV AL, 8CH  MOV CL,3 SAR AL,CL**
4. **MOV DI, 1000H MOV [DI], 0AAH  MOV CL,3 SHL BYTE PTR [DI],CL**
5. **MOV AL, 6BH  ROR AL,1 ROL AL,3**
6. **STC MOV AL, 6BH  RCR AL,3**
7. **CLC  MOV DI,2000H  MOV [DI],0AAH  MOV CL,1 RCL BYTE PTR [DI],CL**

## 4.3 Lab Work:

**Multiplication and Division using Shift instructions**

We have seen earlier that the SHL instruction can be used to multiply an operand by $2^n$ and the SHR instruction can be used to divide an operand by $2^n$.

The MUL and DIV instructions take much longer to execute than the Shift instructions. Therefore, when multiplying/dividing an operand by a small number it is better to use Shift instructions than to use the MUL/DIV instructions. For example MUL BL where BL = 2 takes many more clock cycles than SHL AL, 1.

In **Exercise 1, and 2**, you will write programs to multiply, and divide respectively, using shift instructions.

Write each of the programs using the TASM assembler format. Programs 1, 2, and 3 must be executed using the Turbo Debugger (TD) program. Program 4 must be directly executable from the DOS prompt.

1. Write a program to multiply AX by 27 using only Shift and Add instructions. You should not use the MUL instruction.

2. Write a program to divide AX by 11 using Shift and Subtract instructions. You should not use the DIV instruction. Assume AX is a multiple of 11.

Recall that shifting left $n$ bits multiplies the operand by $2^n$.

If the multiplier is not an absolute power of 2,
then express the multiplier as a sum of terms which are absolute powers of 2.

For example, multiply AX by 7. ($7 = 4 + 2 + 1 = 2^2 + 2^1 + 1$)

Answer = AX shifted left by 2 + AX shifted left by 1 + AX.

**Note:** Only the original value of AX is used in each operation above.

3. Write a program to check if a byte is a Palindrome. [Hint: Use Rotate instructions]. If the byte is a Palindrome, then move AAh into BL. Otherwise move 00h in BL.

A Palindrome looks the same when seen from the left or the right.

For example, 11011011 is a Palindrome but 11010011 is not a Palindrome

[Hint: Use logical shift instruction to move data bit into the carry flag]

4. Write a program to display the bits of a register or memory location. Use the INT 21H interrupts to display data on the display monitor.

For example, if AL = 55H, then your program must display:

**AL = 0 1 0 1 0 0 1 0 1**

# Experiment #5

# Using BIOS Services and DOS functions
# Part 1: Text-based Graphics

## 5.0 Objectives:

The objective of this experiment is to introduce BIOS and DOS interrupt service routines to be utilized in assembly language programs. In this experiment, you will use BIOS and DOS services to write programs that can do the following:

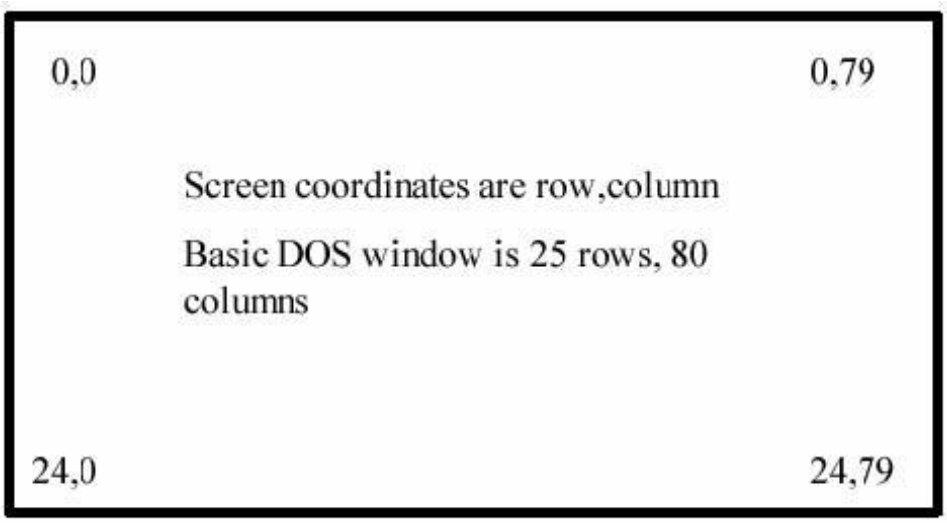
- Read a character/string from the keyboard
- Output a character/string to the display monitor
- Clear the display screen
- and display cursor at a desired location on the screen

## 5.1 Introduction:

The Basic Input Output System (BIOS) is a set of x86 subroutines stored in Read-Only Memory (ROM) that can be used by any operating system (DOS, Windows, Linux, etc) for low-level input/output to various devices. Some of the services provided by BIOS are also provided by DOS. In fact, a large number of DOS services make use of BIOS services. There are different types of interrupts available which are divided into several categories.

.

### 5.1.1 Text Mode Programming




Positions on the screen are referenced using **(row, column)** coordinates. The upper left corner has coordinates (0,0). For an 80 x 25 display, the rows are 0-24 and the columns are 0-79.

## 5.2 Pre-lab:

1. The following program allows a user to enter characters from the keyboard using the character input function (AH=01) of INT 21h. This program also stores the characters entered into a buffer. Run the program after assembling and linking.

TITLE "Program to enter characters from keyboard"

.MODEL SMALL .STACK 100 .DATA

char_buf

.CODE

MOV AX,@DATA ; get the address of the data segment MOV DS, AX ; and store it in register DS

LEA SI, char_buf ; load the address offset of buffer to store the name MOV AH, 01 ; DOS interrupt for character input from keyboard AGAIN: INT 21H ; call the DOS interrupt

MOV [SI], AL ; store character in buffer INC SI ; point to next location in buffer CMP AL, 0DH ; check if Carriage Return <CR> key was hit JNE AGAIN ; if not <CR>, then continue input from keyboard

MOV AX, 4C00H ; Exit to DOS function INT 21H

END ; end of the program

**Procedure (to be followed for all programs):**

- **Edit** the above program using an editor. Type "**edit program1.asm**" at the DOS prompt. Save your file and exit the editor. Make sure your file name has an extension of "**.asm**".
- **Assemble** the program created in (a). Type "**tasm program1**" at the DOS prompt. If errors are reported on the screen, then note down the line number and error type from the listing on the screen. To fix the errors go back to step (a) to edit the source file. If no errors are reported, then go to step (c).
- **Link** the object file created in (b). Type "**tlink program1**" at the DOS prompt. This creates an executable file "program1.exe".
- Type "**program1**" at the DOS prompt to run your program.

**Note**: You have to create your source file in the same directory where the TAMS.exe and TLINK.exe programs are stored.

2. Modify the above program such that the characters entered from the keyboard are not echoed back on the screen (i.e., they are not displayed when keys are pressed). [Hint: use function AH=07 with INT 21h]. After that, add the following lines of code between "**JNE AGAIN**" and MOV AX, 4C00H to display the characters stored in the buffer on the screen.

```
        LEA DI, char_buf ; load the address offset of buffer to store the name MOV DL,
        [DI] ; move character to be displayed in DL MOV AH, 02 ; DOS interrupt for
        character output
BACK: INT 21H ; call the DOS interrupt INC DI ; point to next location in buffer CMP
[DI], 0DH ; check for 0Dh - ASCII value for ENTER key JNE BACK ; if not ENTER key,
then continue output to screen
```

3. The following program clears the screen and positions the cursor at a specified location on the screen using INT 10H functions. The program also displays a message string on the screen using function 09h of INT 21H. Run the program after assembling and linking.

```
TITLE "Program to enter characters from keyboard"

  .MODEL SMALL .STACK 100 .DATA




  LF CR
        msg1     DB "EE 390 Lab, EE Department, KFUPM ", LF, CR, "$"
        msg2  DB "Press any key to exit", LF, CR, "$"

.CODE

MAIN PROC

        MOV AX,@DATA ; get the address of the data segment MOV DS,
        AX ; and store it in register DS

        CALL CLEARSCREEN ; clear the screen

        MOV DH, 10 ; row 10 MOV DL, 13 ; column 13
        CALL SETCURSOR ; set cursor position


        LEA DX, msg1 ; load the address offset of message to be displayed MOV AH, 09h ;
        use DOS interrupt service for string display INT 21H ; call the DOS interrupt


        MOV DH, 20 ; row 20 MOV DL, 13 ; column 13
        CALL SETCURSOR ; set cursor position


        LEA DX, msg2 ; load the address offset of message to be displayed MOV AH, 09h ;
        use DOS interrupt service for string display INT 21H ; call the DOS interrupt


        MOV AX, 4C00H ; exit to DOS
```

```
        INT 21H

MAIN ENDP
CLEARSCREEN PROC

        MOV AH, 00 ; set video mode MOV AL, 03 ; for text 80
        x 25 INT 10H ; call the DOS interrupt RET ; return to
        main procedure


CLEARSCREEN ENDP

SETCURSOR PROC

        MOV AH, 2 ; use DOS interrupt service for positioning screen  MOV BH, 0 ; video
        page (usually 0) INT 10H ; call the DOS interrupt RET ; return to main procedure



SETCURSOR ENDP

END MAIN
```

**Notes**:

1 The above program uses three procedures – MAIN, SETCURSOR, and CLEARSCREEN. The SETCURSOR and CLEARSCREEN procedures are called from the MAIN procedure using the CALL instruction.

2 The SETCURSOR procedure sets the cursor at a specified location on the screen whereas the CLEARSCREEN procedure uses the SET MODE function 00H of INT 10H to set the video mode to 80 x 25 text which automatically clears the screen.

3 You can display a string of characters on the screen, without using a loop, by using MOV AH, 09 with INT 21h. But the string must end with '$' character. You must also load the effective address of the string in register DX.

4 To display a string on a new line, you need to put CR after your string and LF and '$' at the end. CR stands for Carriage Return (or Enter key) and LF stands for Line Feed. You can also put 0Dh or 13 instead of CR (or cr), and 0Ah or 10 instead of LF (or lf).

## 5.3 Lab Work:

The following program clears the screen and positions the cursor in the middle of the screen. Two memory locations ‘row’ and ‘col’ are used to keep track of the cursor position.

TITLE "Program to move the cursor on the screen"

```
.MODEL SMALL
.STACK 100
.DATA

row DB 12
col DB 39

.CODE

MAIN PROC

MOV AX,@DATA
MOV DS, AX

CALL CLEARSCREEN

CALL SETCURSOR

MOV AX, 4C00H
INT 21H

MAIN ENDP

CLEARSCREEN PROC

MOV AH, 00
MOV AL, 03
INT 10H
RET

CLEARSCREEN ENDP

SETCURSOR PROC

MOV DH, row
MOV DL, col
MOV AH, 2
MOV BH, 0
INT 10H
RET

SETCURSOR ENDP

END MAIN
```

**Note** that the SETCURSOR procedure shown above gets its row and column positions directly from the memory variables ‘row’ and ‘col’.

Modify the MAIN procedure in the above program to read an arrow key value from the keyboard using the DOS single character input function INT 21h, AH=7 which waits for a character and does not echo the character to the screen. Depending on which arrow key is pressed, the program must move the cursor accordingly

The program must wrap the cursor correctly around to the next boundary, for e.g., if the cursor moves off the right edge it should appear at the left edge and vice-versa. Similarly, if the cursor moves off the bottom edge it should appear at the top edge and vice-versa.

The program must continuously check for a key press (using the ASCII values given above) inside a loop, and move the cursor to a new position only when an arrow key is pressed. The program must exit the loop and return to DOS when the ENTER key (ASCII value 0Dh) is pressed.

# Experiment #6

# Using BIOS Services and DOS functions
# Part 1: Pixel-based Graphics

## 6.0 Objectives:

The objective of this experiment is to introduce BIOS and DOS interrupt service routines to write assembly language programs for pixel-based graphics.

In this experiment, you will use BIOS and DOS services to write programs that can do the following:

- Set graphics video mode
- Write a pixel on the screen
- Draw a line on the screen
- Draw a rectangle on the screen

## 6.1 Introduction:

In text mode, the cursor is always displayed on the screen and the resolution is indicated as number of characters per line and number of lines per screen. In graphics mode, the cursor will not appear on the screen and the resolution is specified as number of pixels per line and number of lines per screen. Text can be used as usual in graphics mode.

## 6.2 Pre-lab:

### 1. Drawing a Pixel

The following program draws a pixel on the screen at location (240, 320) using the “write pixel” function (AH=0Ch) of INT 10h. Run the program after assembling and linking it.

```
TITLE "Program to enter characters from keyboard" .MODEL SMALL ; this
defines the memory model .STACK 100 ; define a stack segment of 100 bytes
.DATA    ; this is the data segment .CODE     ; this is the code segment


        MOV AX,@DATA ; get the address of the data segment MOV DS,
        AX ; and store it in DS register

        MOV AH, 00h ; set video mode MOV AL, 12h ;
        graphics 640x480 INT 10h


        ; draw a green color pixel at location (240, 320) MOV AH, 0Ch
        ; Function 0Ch: Write pixel dot MOV AL, 02 ; specify green
        color MOV CX, 320 ; column 320 MOV DX, 240 ; row 240
        MOV BH, 0 ; page 0 INT 10h



        MOV AH, 07h ; wait for key press to exit program INT 21h


        MOV AX, 4C00H ; Exit to DOS function INT 21H


END     ; end of the program
```

**2. Drawing a horizontal line**

The following program draws a horizontal line on the screen from location (240, 170) to (240, 470) by writing pixels on the screen using function (AH=0Ch) of INT 10h. Run the program after assembling and linking it.

```
TITLE "Program to enter characters from keyboard" .MODEL SMALL ; this
defines the memory model .STACK 100 ; define a stack segment of 100 bytes
.DATA    ; this is the data segment .CODE     ; this is the code segment


        MOV AX,@DATA ; get the address of the data segment MOV DS,
        AX ; and store it in DS register

        MOV AH, 00h ; set video mode MOV AL, 12h ;
        graphics 640x480 INT 10h


        ; draw a green color line from (240, 170) to (240, 470) MOV CX, 170 ;
        start from row 170 MOV DX, 240 ; and column 240 MOV AX, 0C02h ;
        AH=0Ch and AL = pixel color (green)

BACK: INT 10h ; draw pixel INC CX ; go to next column CMP CX, 470 ; check if
column=470  JB BACK ; if not reached column=470, then continue


        MOV AH, 07h ; wait for key press to exit program INT 21h


        MOV AX, 4C00H ; Exit to DOS function INT 21H


END     ; end of the program
```

**3. Drawing a vertical line**

Using the procedure followed in part 2 (drawing a horizontal line), draw a vertical line on the screen from location (90, 320) to (390, 320). Run the program after assembling and linking it.

**4. Drawing a plus (+) sign in the middle of the screen**

Combine the programs written for parts 2 and 3 above to draw a plus sign. All you have to do is to insert the code for drawing the vertical line [from location (90, 320) to (390, 320)] right after the code for drawing the horizontal line [from location (240, 170) to (240, 470)]. Run the program after assembling and linking it.

## 6.3 Lab Work:

Draw the following figure on the screen using function 0Ch of INT 10h. Assemble, link, and run it and show it to your lab instructor for credit.