Table of Contents

Experiment #0

PC Hardware and Operating Systems

Objective:

The objective of this experiment is to introduce the operating systems and different hardware components of a microcomputer.

Equipment: Microcomputer

Introduction:

Microcomputer (PC) operating systems are briefly discussed in this experiment. The two commonly used operating systems in PC are MS-DOS and Windows (95, 98,etc..). In the MS-DOS environment, command line is used to view, copy or interact with stored files. In Windows environment, clicking the mouse performs these operations in a user friendly manner. First part of this experiment introduces the file management in MS DOS mode. In the second part a 386 microcomputer is disassembled and its hardware components are identified. Finally the PC is reassembled in the laboratory.

Pre-lab:

Use any computer with 'Windows', to do the following operations:

1. In Windows operating system, use 'Start' and 'Find' menu to locate the 'Debug' program. Note the address or path.
2. From 'Start' and 'Programs' menu, use 'Windows Explorer' to make a new directory in C drive and name it 'EE 390'.
3. Using 'Copy' and 'Paste' command of 'Windows Explorer' copy the 'Debug.exe' program in to the newly created directory of 'EE390'
4. Execute the 'Debug' program by clicking on it. (type 'q' to quit)
5. Also execute the 'Debug' program, from 'Start' and 'Run' menu.
6. From 'Start' and 'Programs' menu, click on 'MS-DOS prompt' to start the MS-DOS debugger. Type 'Debug' and press <enter> to execute the program. Type 'q' to quit the debug program.

Lab Work[1]:

1. Use the Lab microcomputer to perform the following operations;
    a. In MS-DOS mode, ‘C :\>' means we are in the main directory. Go to this directory and use 'DIR' to check the contents of the directory.
    b. Use 'CD' or change directory command to go to 'DOS' directory.
    c. Use 'DIR' to find Debug program in this directory (type 'DIR D* ')
    d. Execute the Debug program. (Type 'Debug' and press <enter>). To quite the debugger, type 'q' in the debug prompt '-' and press enter.

2. Use the computer hardware to locate the following components;
    a. Hard disk: Find the storage capacity and manufacturer of the disk.
    b. ROM: Find its manufacturer and storage capacity.
    c. RAM: Try to find the total storage capacity of RAM and the capacity of individual RAM circuits.
    d. CPU: Find its manufacturer and the operating speed.

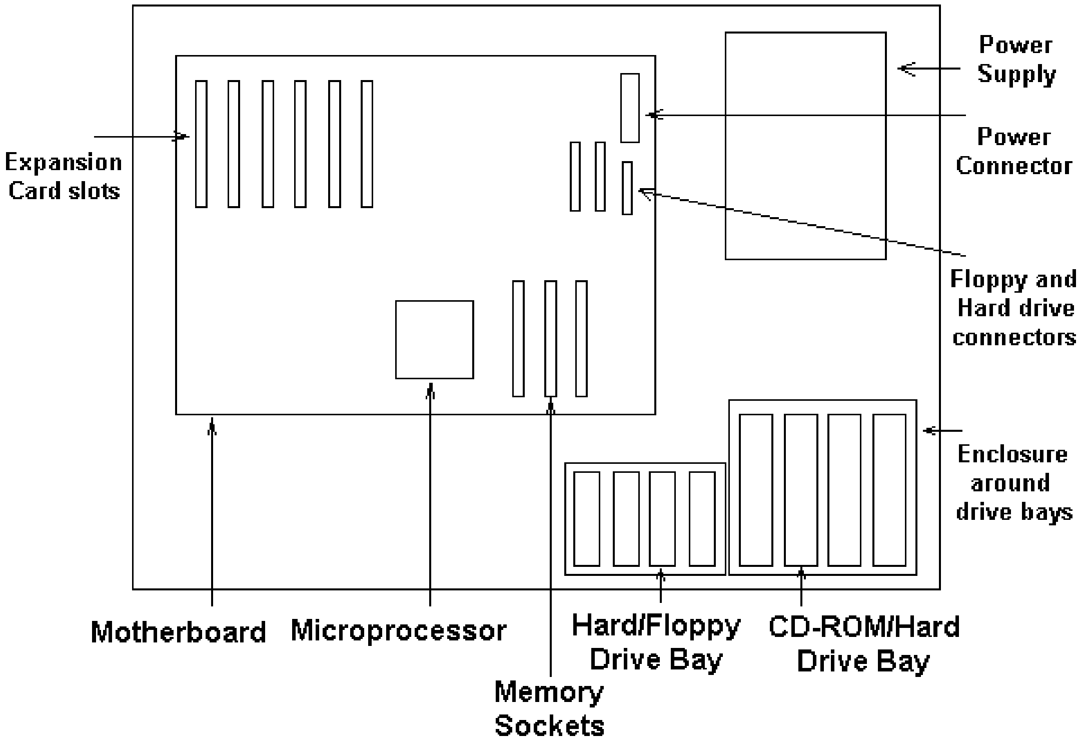
3. What is a BUS.? Can you see any?
4. Find the power supply and what voltages are supplied by it.
5. Where is the Mother board? Locate the Clock in it.
6. Where are ISA and PCI sockets in the mother board? What do they do?
7. Is there any input/output cards attached to the mother board. If so, what external devices can you connect to them?
8. Name the ports at the back of the computer. Write the total number of pins in each port and what devices can be connect to them.

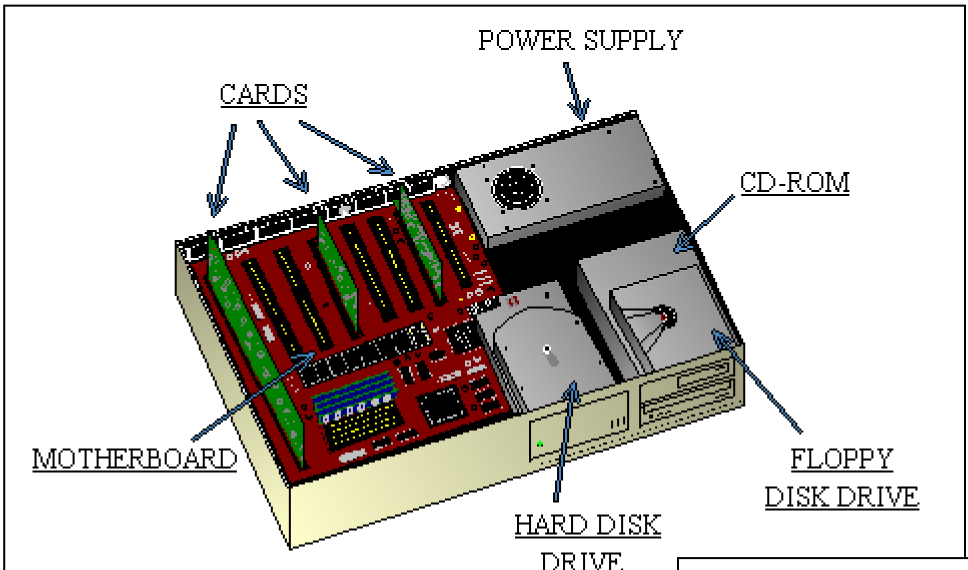
Lab Report:

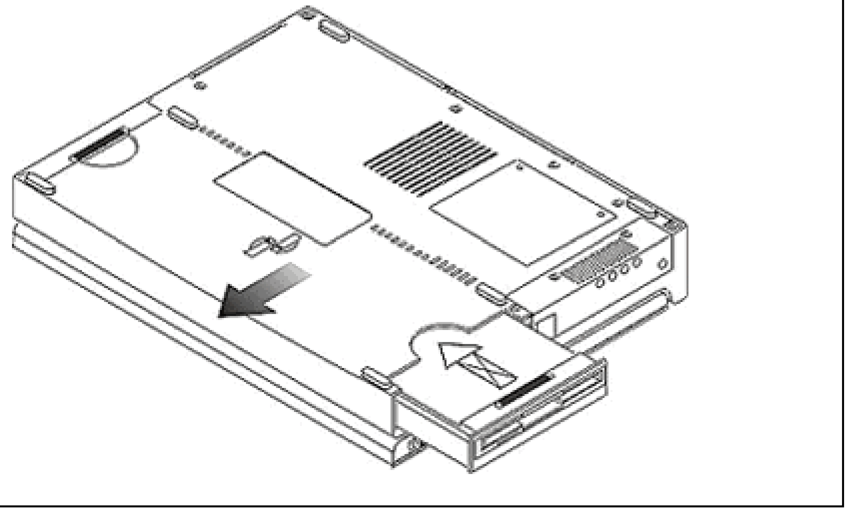
The lab report should contain: 1.OBJECTIVE, 2.INTRODUCTION, 3.RESULTS (observed in the pre-lab and in the experiment) and 4.CONCLUSION.

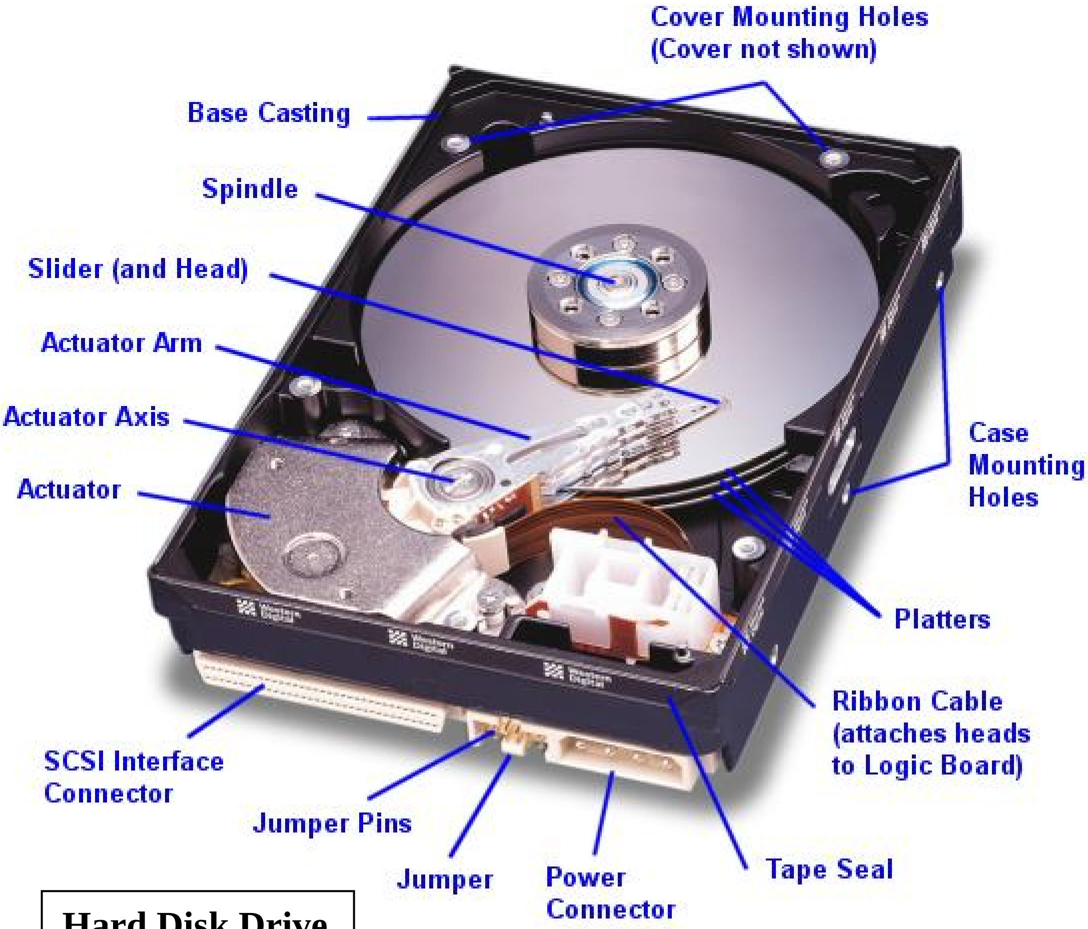
[1] Make sure you know the hardware components as there will be a quiz in this topic.

Inside the Case





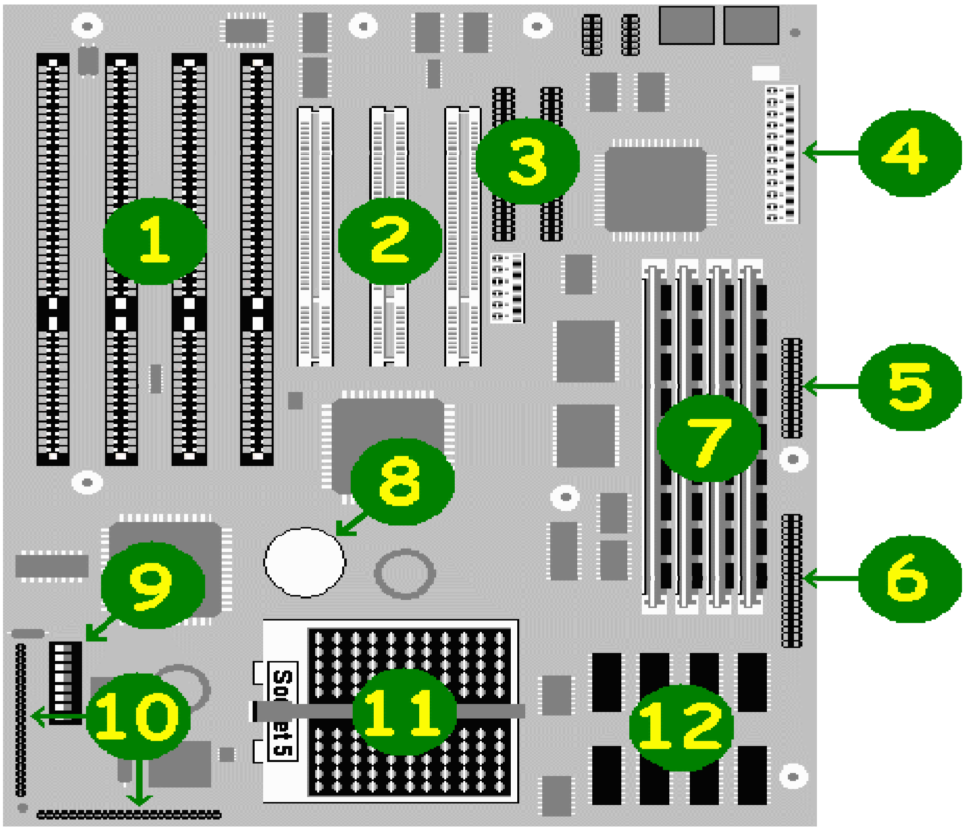




Hard Disk Drive

CD-ROM Drive

Basic Components on a Motherboard



1 ISA (Industry Standard Architecture) bus slots for plugging in older 8 and 16-bit adapter cards.

2 PCI (Peripheral Component Interconnect) bus slots for plugging in newer 32-bit adapter cards.

3 Hard drive controller connectors.

4 Power connector.

5 Parallel port connector.

6 Floppy disk controller connector.

7 SIMM (Single In-line Memory Module) sockets for adding memory.

8 Lithium backup battery for the CMOS.

9 Configuration jumper block for changing the ISA bus clock, clearing a CMOS password, resetting the CMOS to the default settings, etc.

10 Front panel connectors for the internal speaker, keyboard and hard drive lights, +12v fan, etc.

11 Pentium processor in a Socket 5 connector.

12 256K cache (those systems with an external cache only).

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	FUNCTION	EXAMPLE
Register	**R** [Register Name]	Examine or modify the contents of an internal register of the CPU	**-R** AX (AX reg.) **-R** F (flags)
Dump	**D** [Address]	Display the contents of memory locations specified by Address	**-D** DS:100 200 **-D** start-add end-add
Enter	**E** [Register Name]	Enter or modify the contents of the specified memory locations	**-E** DS:100 22 33 **-E** address data data
Fill	**F** [Register name]	Fill a block of memory with data	**-F** DS:100 120 22
Assemble	**A** [Starting address]	Convert assembly lang. instructions into machine code and store in memory	**-A** CS:100 **-A** start-address
Un-assemble	**U** [Starting Address]	Display the assembly instructions and its equivalent machine codes	**-U** CS:100 105 **-U** start-add end-add
Trace	**T** [Address][Number]	Line by line execution of specific number of assembly lang. instructions	**-T**=CS:100 **-T**=starting-address
Go	**G** [Starting Address] [Breakpoint Add.]	Execution of assembly language instructions until Breakpoint address	**-G**=CS:100 117 **-G**=start-add end-add

Table 1: DEBUG commands

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

Flag	Meaning	SET	RESET
CF	Carry	CY	NC
PF	Parity	PE	PO
AF	Auxiliary	AC	NA
ZF	Zero	ZR	NZ

Flag	Meaning	SET	RESET
SF	Sign	NG	PL
IF	Interrupt	EI	DI
DF	Direction	DN	UP
OF	Overflow	OV	NV

AX	**BX**	**CX**	**DX**	**SI**	**DI**	**SP**	**BP**
DS	**CS**	**ES**	**SS**	**IP**	**8086 Internal Registers**		

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.
   - List the values of the following registers:

AX		**SP**	
BX		**CS**	
CX		**DS**	
DX		**SS**	
IP		**ES**	

   - What is the address of the next instruction to be executed?
   - What is the instruction held at this address?
2. Enter the command: R AH (hit <CR>)

   What happens and why?
3. Use a REGISTER command to first display the current contents of BX and then change this value to 0020h.

4. Use a REGISTER command to first display the current contents of IP and then change this value to 0200h.
5. Use a REGISTER command to first display the current contents of the flag register and then ***set*** the parity, zero, and carry 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 100 bytes of the current data segment.
2. Use the DUMP command (D) to display the first 100 bytes of the code segment starting the current value of CS: IP.
3. Use the ENTER command (E) to load locations CS:100, CS:102, and CS:104 with 11, 22, and 33, respectively.
4. Use the ENTER command (E) to load five consecutive byte-wide memory locations starting at CS:105 with data 'FF'.
5. Verify the result of steps 3 and 4 using the DUMP command.
6. Use the FILL command (F) to initialize the 16 storage locations starting at DS:10 with the value AAh, the 16 storage locations starting at address DS:30 with BBh, the 16 storage locations starting at address DS:50 with CCh, and the 16 storage locations starting at address DS:70 with DDh
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.

Instruction	Machine Code	Bytes required	Starting Address
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]			

   d. Why are the starting addresses of the above instructions not consecutive?

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.

F. Music Program

This program generates a musical tone every time a key pressed. It generates 8 tones in total and then stops.

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

```
            LEA SI, TUNE
            CLD
BACK:        MOV AH, 0
            INT 16H
            LODSW
            MOV BX, AX
            CMP AX, 0
            JZ L1
            MOV AL, 0B6H
            OUT 43H, AL
            MOV AL, BL
            OUT 42H, AL
            MOV AL, BH
            OUT 42H, AL
            IN AL, 61H
            OR AL, 3
            OUT 61H, AL
            JMP BACK
L1:         IN AL, 61H
            AND AL, 0FCH
            OUT 61H, AL
            INT 20H

TUNE:
            DW 11D1H
            DW 0FDFH
            DW 0E24H
            DW 0D59H
            DW 0BE4H
            DW 0A98H
            DW 0970H
            DW 08E9H
            DW 0000
```

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

3. 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?

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	Contents after 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.

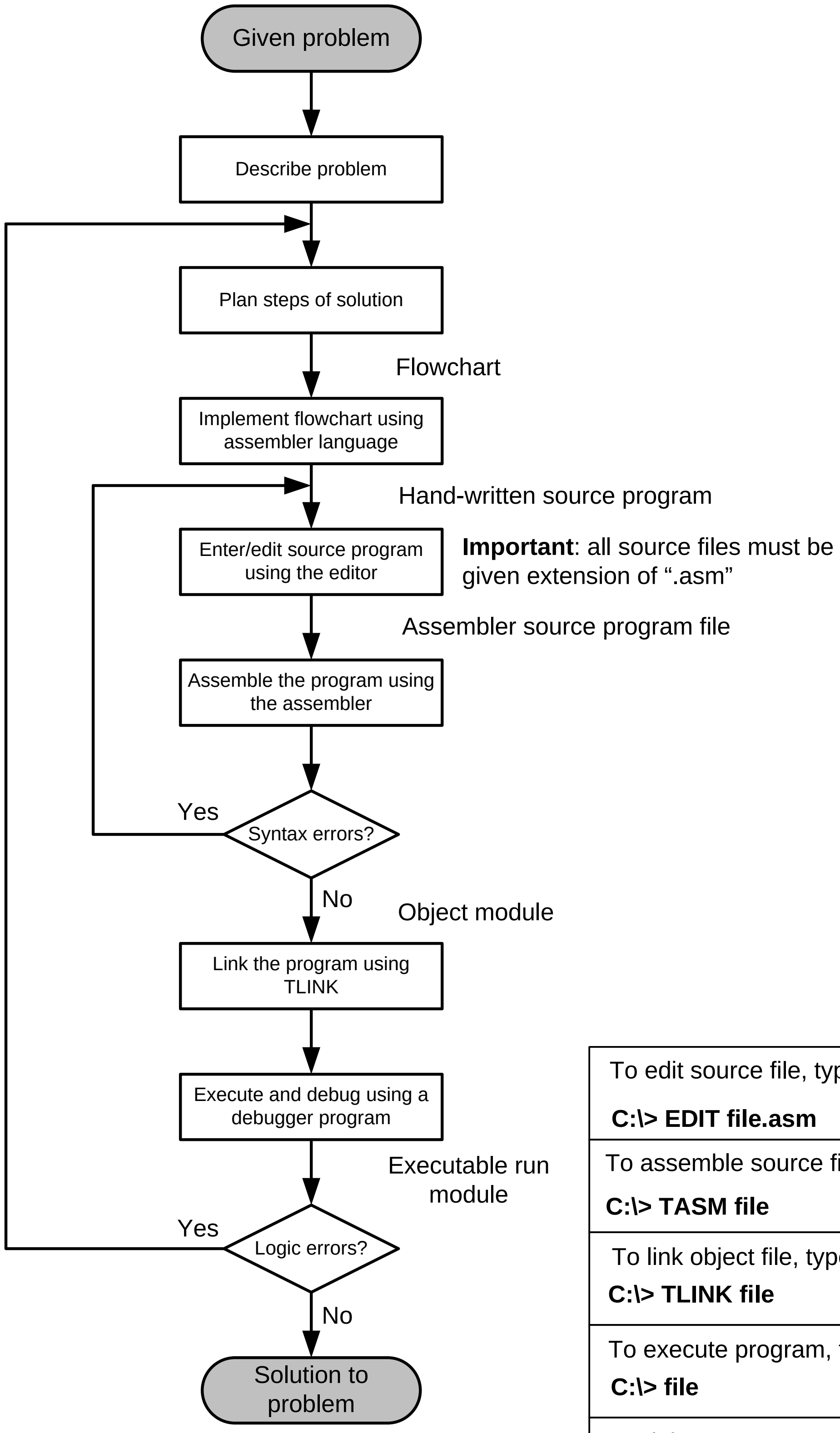
Program Template

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

```
TITLE     "Experiment 2" ←— Short description of program

.MODEL     SMALL         ←—  Assembler directive for memory model (up to 64KB)

.STACK     032h          ←— Assembler directive for stack segment (reserves 50 bytes)

.DATA                    ←— Assembler directive that defines data segment

      …………..            ←— (reserve memory space for constants and variables )

.CODE                    ←— Assembler directive that defines code segment

      …………..            ←— (type your assembly language program here)
      …………..
                             ; (this is a comment starting with ';')

END                      ←— Assembler directive indicates end of program
```

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.


Given problem
Describe problem
Plan steps of solution
Flowchart
Implement flowchart using assembler language
Hand-written source program
Enter/edit source program using the editor
Assembler source program file
Assemble the program using the assembler
Yes
Syntax errors?
No
Object module
Link the program using TLINK
Execute and debug using a debugger program
Executable run module
Yes
Logic errors?
No
Solution to problem

Important: all source files must be given extension of ".asm"

A general program development cycle

To edit source file, type	**C:\> EDIT file.asm**
To assemble source file, type	**C:\> TASM file**
To link object file, type	**C:\> TLINK file**
To execute program, type	**C:\> file**
To debug program, type	**C:\> TD file**

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):

a. **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**".

b. **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).

c. **Link** the object file created in (b). Type "**tlink program1**" at the DOS prompt. This creates an executable file "program1.exe".

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

   num    DB  10,20,30,40 ; store the four numbers in memory

.CODE                     ; this is the code segment

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

        LEA SI,num        ; load the address offset of buffer to store the

        MOV AL,[SI]       ; copy AL with memory contents of ‘SI’, i.e. 10
        MOV AH,[SI+1]     ; copy AH with memory contents of ‘SI+1’, i.e. 20
        MOV CL,[SI+2]     ; copy CL with memory contents of ‘SI+2’, i.e. 30
        MOV CH,[SI+3]     ; copy CH with memory contents of ‘SI+3’, i.e. 40

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

END                               ; end of the program
```

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                           ; this is the code segment

                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 offset address of char_buf

AGAIN:          MOV AH, 01              ; function for character input from keyboard
                INT 21H                 ; ASCII value is returned in the AL register

                MOV [SI], AL            ; transfer the character typed to memory

                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

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

END                             ; end of the program
```

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

        MSG1  DB    ‘Enter a lower case letter: $’
        MSG2  DB    0DH,0AH, ‘The letter in uppercase is: ’
        CHAR  DB    ?, ‘$’

.CODE                     ; this is the code segment

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

                MOV CL,03       ; counter to take 3 inputs

START:          MOV AH,9        ; display string function
                LEA SI,msg      ; get memory location of 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)

                SUB AL,30H      ; subtract 30 to convert from ASCII code to number

                LEA SI,num      ; load the address of memory location num
                MOV [SI],AL          ; and store the first number in this location
                DEC CL               ; decrement CL
                CMP CL,0             ; check if the 3 inputs are complete
                JE ADD_IT            ; if yes then jump to ADD_IT location
                INC SI               ; if no then move to next location in memory
                JMP ADD_IT           ; unconditional jump to get the next number

ADD_IT:         MOV CL,02            ; counter to add the numbers
                LEA SI,NUM           ; get the address of the first stored number
                MOV AL,[SI]          ; store the first number in AL
AGAIN:          ADD AL,[SI+1]        ; add the number with the next number
                CMP CL,0             ; if the numbers are added
                JE DIVIDE            ; then go to the division
                INC SI               ; otherwise keep on adding the next numbers to the
result
                JMP AGAIN            ; unconditional jump to add the next entry

DIVIDE:         MOV AH,0             ; make AX=AL for unsigned division
                MOV CL,03            ; make divisor=3 to find average of three numbers
                DIV CL               ; divide AX by CL
                LEA SI,average       ; get the address of memory location average
                MOV [SI],AX          ; and store the result in the memory

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

END                                  ; end of the program
```

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).

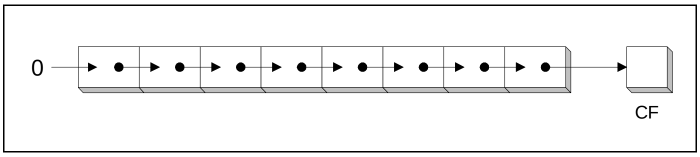
Mnemonic	Meaning	Format
SAL	Shift Arithmetic Left	SAL D, count
SHL	Shift Logical Left	SHL D, count
SAL	Shift Arithmetic Right	SAR D, count
SHL	Shift Logical Right	SHR D, count

Allowed operands

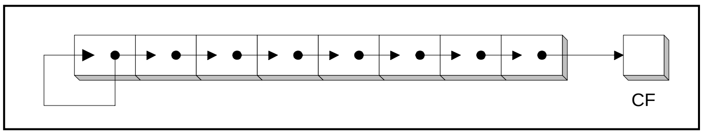
Destination(D)	Count
Register	1
Register	CL
Memory	1
Memory	CL

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.

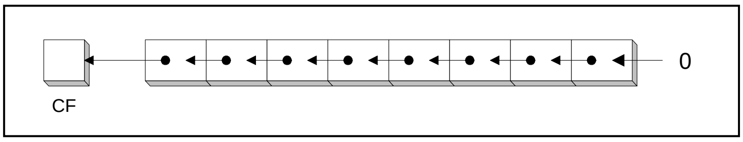
A logical shift fills the newly created bit position with zero:



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



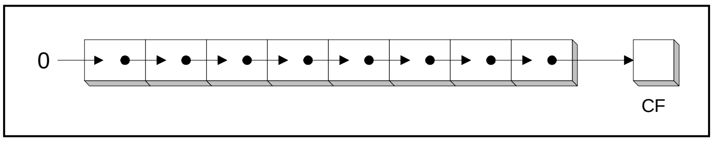
The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.



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 SHL BL, 1	Before:	0 0 0 0 0 1 0 1	= 5
	After:	0 0 0 0 1 0 1 0	= 10

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

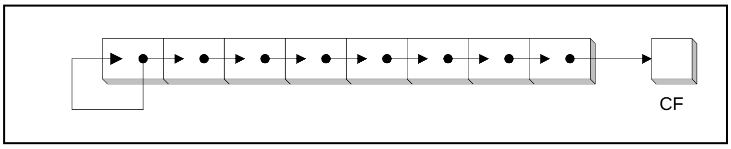


Shifting right 1 bit divides a number by 2 and shifting right *n* bits divides the operand by 2^n.

For example:

MOV DL, 12 SHR DL, 1	Before:	0 0 0 0 1 1 0 0	= 12
	After:	0 0 0 0 0 1 1 0	= 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 SAR BL, 1	BL = -20

Rotate Instructions

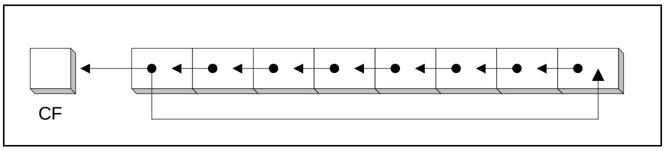
The 8086 can perform two types of rotate operations; the *rotate* without carry and the *rotate* through carry. There are four rotate operations (ROL, ROR, RCL, and RCR).

Mnemonic	Meaning	Format
ROL	Rotate Left	ROL D, count
ROR	Rotate Right	ROR D, count
RCL	Rotate Left through carry	RCL D, count
RCR	Rotate Right through carry	RCR D, count

Allowed operands

Destination(D)	Count
Register	1
Register	CL
Memory	1
Memory	CL

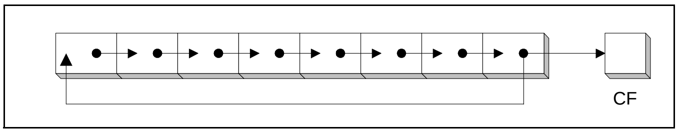
ROL shifts each bit of a register to the left. The highest bit is copied into both the Carry flag and into the lowest bit of the register. No bits are lost in the process.



For example:

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

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

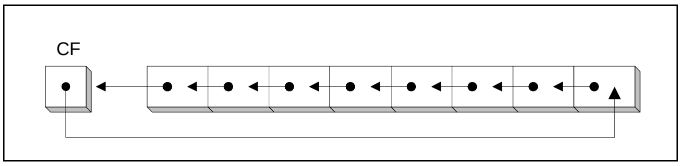
ROR shifts each bit of a register to the right. The lowest bit is copied into both the Carry flag and into the highest bit of the register. No bits are lost in the process.



For example:

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

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

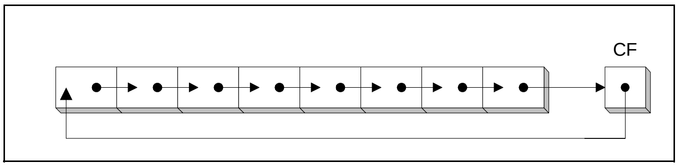
RCL (rotate carry left) shifts each bit to the left. It copies the Carry Flag to the least significant bit and copies the most significant bit to the Carry flag.



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 fill the corresponding column for each Shift or Rotate instruction.

NOTE: Include the status of flags before and after the execution of shift and rotate instructions in Table 1.

```
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
```

TABLE 1

Statement	Source		Destination			Status Flags				
	Register /Memory	Contents	Register/Memory	Contents before execution	Contents after execution	AF	PF	SF	ZF	CF
SHR AL,1										
SHL AL,3										
SHL AX,CL										
SAR AL,CL										
SHL BYTE PTR [DI],CL										
ROR AL,1										
ROL AL,3										
RCR AL,3										
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.

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.

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 right *n* bits divides the operand by 2^n.

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

For example, divide AX by 5. ($5 = 4 + 1 = 2^2 + 1$)

Answer = AX shifted right by 2 - 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

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

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

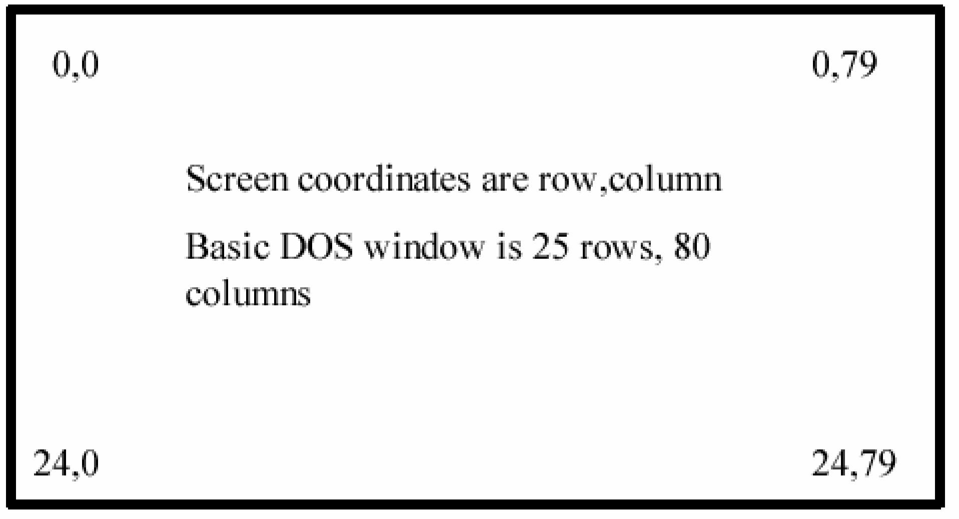
5.1 Introduction:

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 as shown below:

Interrupt Types	Description
0h - 1Fh	BIOS Interrupts
20h - 3Fh	DOS Interrupts
40h - 7Fh	reserved
80h - F0h	ROM BASIC
F1h - FFh	not used

BIOS and DOS interrupt routines provide a number of services that can be used to write programs. These services include formatting disks, creating disk files, reading from or writing to files, reading from keyboard, writing to display monitor, etc. The software interrupt instruction INT is used for calling these services.

5.1.1 Text Mode Programming



Screen coordinates are row,column

Basic DOS window is 25 rows, 80 columns

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.

5.1.2 Commonly used DOS functions

DOS contains many functions that can be accessed by other application programs. These functions are invoked using the assembly language instruction INT XX, where XX is replaced by the number of the appropriate interrupt. Most of the available functions are invoked through the INT 21H instruction.

Character input with echo (INT 21H, Function 01H):

Reads a character from the standard input device (usually the keyboard) and echoes it to the standard output device (usually the display screen), or waits until a character is available**.**

Description: (INT 21H, Function 01H)	**Example**
Invoked with: AH = 01H Returns: AL = character input (ASCII code) and displays the character on the screen	MOV AH, 01H INT 21H MOV [SI],AL ; store char. in memory

Character input without echo (INT 21H, Function 07H):

Reads a character from the standard input device (usually the keyboard) **without** echoing it to the standard output device, or waits until a character is available**.** This function can be used when you don't want the input characters to appear on the display, for example, in the case of password entry.

Description: (INT 21H, Function 07H)	**Example**
Invoked with: AH = 07H Returns: AL = character input (ASCII code)	MOV AH, 07H INT 21H MOV [SI],AL ; store char. in memory

Display Character (INT 21H, Function 02H):

Displays a character at the standard output device (usually the display screen).

Description: (INT 21H, Function 02H)	**Example**
Invoked with: AH = 02H DL = ASCII code for the char. to be displayed Returns: Nothing	MOV DL,'A' ; display character 'A' MOV AH, 02H INT 21H

Display Character String (INT 21H, Function 09H):

Displays a string of characters at the display screen. The string must be terminated with the character ‘$’, which is not displayed.

Description: (INT 21H, Function 09H)	**Example**
Invoked with: AH = 09H DS : DX = segment : offset of string Returns: Nothing	MSG DB “Welcome”,’$’ ; string MOV DX, OFFSET MSG MOV AH, 09H INT 21H

Exit program and return control to DOS (INT 21H, Function 4CH):

Terminates current process and returns control either to the parent process or DOS.

Description: (INT 21H, Function 4CH)	**Example**
Invoked with: AH = 4CH AL = 00H Returns: Nothing	MOV AX, 4C00H INT 21H

5.1.3 BIOS Video I/O Services

The BIOS function requests in this category are used to control text and graphics on the PC's display screen. The function request is chosen by setting the AH register to the appropriate value and issuing interrupt 10H.

Set Video Mode (INT 10H, Function 00H):

Selects the video mode and clears the screen automatically.

Description: (INT 10H, Function 00H)	**Example**
Invoked with: AH = 00H AL = mode number to indicate the desired video mode Returns: Nothing	MOV AH, 00 MOV AL, 03H ; text video mode INT 10H

Set Cursor Position (INT 10H, Function 02H):

Sets the position of the display cursor by specifying the character coordinates.

Description: (INT 10H, Function 02H)	**Example**

Invoked with: AH = 2
BH = video page number (usually 0)
DH = row (0-24)
DL = column (0-79 for 80x25 display)
Returns: Nothing
	MOV AH, 02
	MOV BH, 0
	MOV DH, 12 ; row 12
	MOV DL, 40 ; column 40
	INT 10H

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                    ; 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                           ; this is the code segment

        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):

e. **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**”.

f. **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).

g. **Link** the object file created in (b). Type “**tlink program1**” at the DOS prompt. This creates an executable file “program1.exe”.

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

        LF      EQU   10       ; Line Feed character (0A in Hex)
        CR      EQU   13       ; Carriage Return character (0D in Hex)

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

        row     DB      12          ; define initial row number
        col     DB      39          ; define initial column number

.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

        CALL SETCURSOR              ; set the cursor position

        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 DH, row                 ; load row number
        MOV DL, col                 ; load column number
        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
```

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, as indicated below:

Key pressed	ASCII value read from keyboard	Movement
↑ (Up)	48h	Move up (decrement row)
→ (Right)	4Dh	Move right (increment col)
↓ (Down)	50h	Move down (increment row)
← (Left)	4Bh	Move left (decrement col)

The following can be defined in the data segment:

```
LEFT      EQU   4Bh
RIGHT     EQU   4Dh
UP        EQU   48h
DOWN      EQU   50h
```

The following table shows some 80 x 25 screen positions.

Position	Decimal Value	Hexadecimal
Upper left corner	(0,0)	(0,0)
Lower left corner	(0,24)	(0,18)
Upper right corner	(79,0)	(4F,0)
Lower right corner	(79,24)	(4F,18)
Center screen	(39,12)	(27,C)

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.

6.1.1 BIOS Video I/O Services

The BIOS function requests in this category are used to control graphics on the PC's display screen. The function request is chosen by setting the AH register to the appropriate value and issuing and interrupt 10H.

Set Video Mode (INT 10H, Function 00H)**:**

Selects the video mode and clears the screen automatically.

Input:

AH = 00H

AL = mode number to indicate the **video mode** desired

Returns: Nothing

Example:

```
MOV AH, 00
MOV AL, 03H   ; text video mode
INT 10H
```

Table: Possible video mode settings.

Mode	Type	Max Colors	Size	Resol.
00	Text	16	40x25	- -
01	Text	16	40x25	- -
02	Text	16	80x25	- -
03	Text	16	80x25	- -
04	Graphics	4	40x25	320x200
05	Graphics	4	40x25	320x200
06	Graphics	2	80x25	640x200
07	Text	Mono	80x25	- -
08	Graphics	16	20x25	? ?
09	Graphics	16	40x25	? ?
0A	Graphics	4	80x25	? ?
0B	- -	-	- -	- -
0C	- -	-	- -	- -
0D	Graphics	16	40x25	320x200
0E	Graphics	16	80x25	640x200
0F	Graphics	Mono	80x25	640x350
10	Graphics	16	80x25	640x350
11	Graphics	2	80x25	640x480
12	Graphics	16	80x25	640x480
13	Graphics	256	40x25	320x200

Scroll the Screen or a Window Up (INT 10H, Function 06H):

Input:

AH = 6

AL = number of lines to scroll (0 => whole screen)

BH = attribute for blank lines

CH, CL = row, column for upper left corner

DH, DL = row, column for lower right window

Returns: Nothing

Scrolling the **screen up one line** means to move each display line UP one row and insert a blank line at the bottom of the screen. The previous top row disappears from the screen.

The whole screen or any rectangular area (window) may be scrolled. AL contains the number of lines to scroll. If AL = 0, all the lines are scrolled and this clears the screen or window.

Example: Clear the screen to black for the 80x25 display.

```
MOV AH, 6          ; scroll up function
XOR AL, AL         ; clear entire screen
XOR CX, CX         ; upper left corner is (0,0)
MOV DX, 184FH      ; lower right corner is (4Fh, 18H)
MOV BH, 7          ; normal video attribute
INT 10H            ; clear screen
```

Scroll the Screen/Window down (INT 10H, Function 07H):

Input:

AH = 7

AL = number of lines to scroll (0 => whole screen)

BH = attribute for blank lines

CH, CL = row, column for upper left corner

DH, DL = row, column for lower right corner

Returns: Nothing

Same as function 6, but lines are scrolled down instead of up.

16-Color Display

Attribute Byte:

Bit#	7	6	5	4	3	2	1	0
Attr	BL	R	G	B	IN	R	G	B

Attributes:

Bit #	Attribute
0-2	character color (**foreground color**)
3	intensity
4-6	**background** color
7	blinking

E.g., to display a red character on a blue background, the attribute byte would be:

0001 0100 = 14h

If the attribute byte is: 0011 0101 = 35h

Uses blue + green (cyan) in the background and red + blue (magenta) in the foreground, so the character displayed would be magenta on a cyan background.

If the ***intensity bit*** **(bit 3)** is 1, the foreground color is lightened (brightened). If the ***blinking bit*** **(bit 7)** is 1, the character turns on and off.

Write Pixel (INT 10h Function 0Ch):

Draws the smallest unit of graphics display, also called a dot, a point or a pixel (picture element) on the display at specified graphics coordinates. This function operates only in graphics modes.

Input

AH = 0Ch

AL = pixel value

(if bit 7 is 1, the new pixel color bits will be EX-ORed with the color bits of the current pixel.

BH = video display page

CX = column (graphics x coordinate)

DX = row (graphics y coordinate)

Returns: Nothing

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

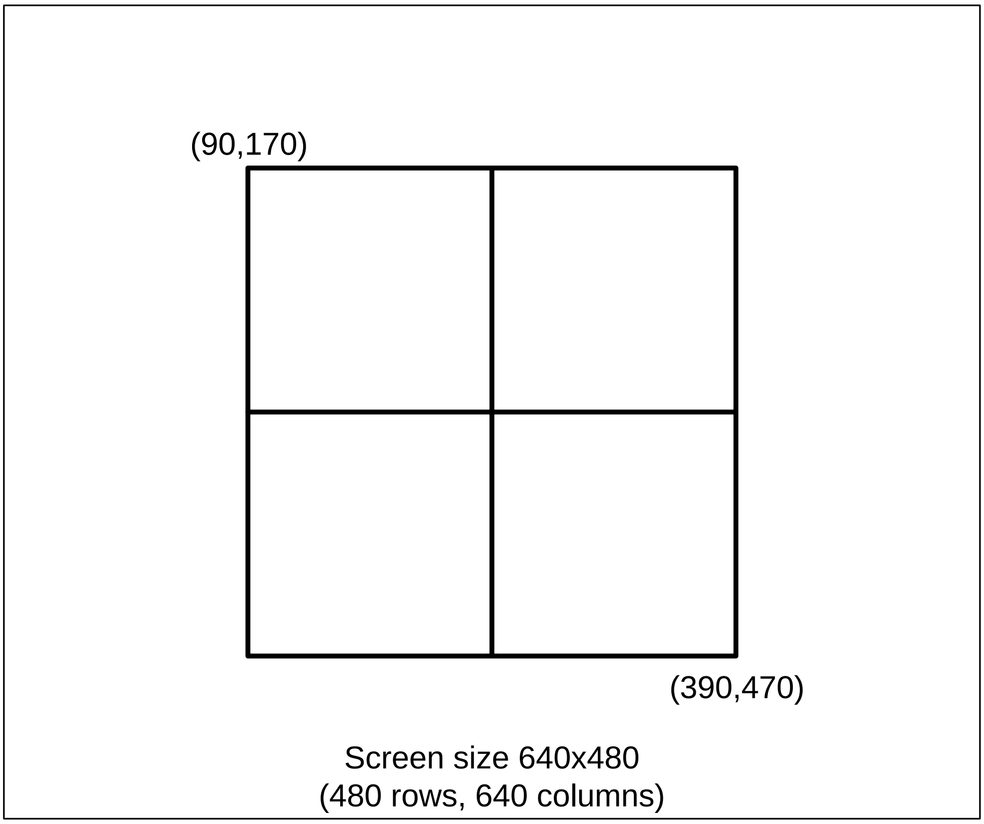
Using the procedure followed in part 2 (drawing a horizontal line), draw a vertical line on the screen from location (90, 320) to (390, 320). Run the program after assembling and linking it.

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.



Experiment #7

Introduction to Flight86 Microprocessor Trainer and Application Board

7.0 Objectives:

The objective of this experiment is to introduce the Flight86 Microprocessor training kit and application board.

In this experiment, you will do the following:

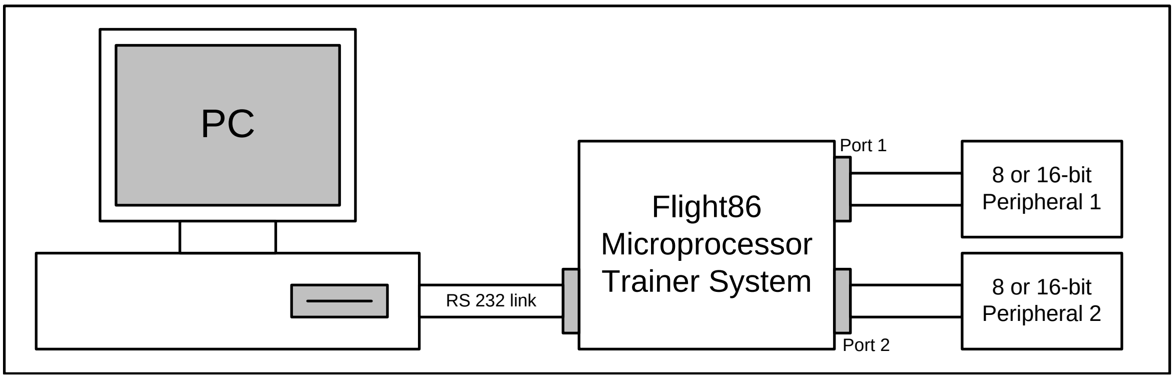
- Study the hardware specifications of the training and application boards
- Learn monitor commands to communicate with the Flight86 trainer
- Assemble, download, and test a program on the trainer board

7.1 Equipment and Software

- Flight86 Trainer and Application Board
- PC with Flight86 Monitor program

7.2 Introduction:

The Flight86 trainer system together with an application board can be used to perform interesting experiments using the 8086 on-board microprocessor. The Flight86 trainer system can be connected to a PC (through its serial port) which allows code to be assembled and debugged in a supportive software environment before being downloaded into the RAM on the board. The block diagram below shows such a setup:



Once the code is downloaded, it can be executed and tested in a system which is accessible to the user. Data may be modified on the board and the change in results can be viewed on the PC display. A monitor program stored on the board in an EEPROM is the software that allows communication between the PC and the trainer system. The monitor program allows code to be downloaded from an Intel Hex file into the controller RAM. The monitor code also enables development activity such as register and memory management and program execution to take place.

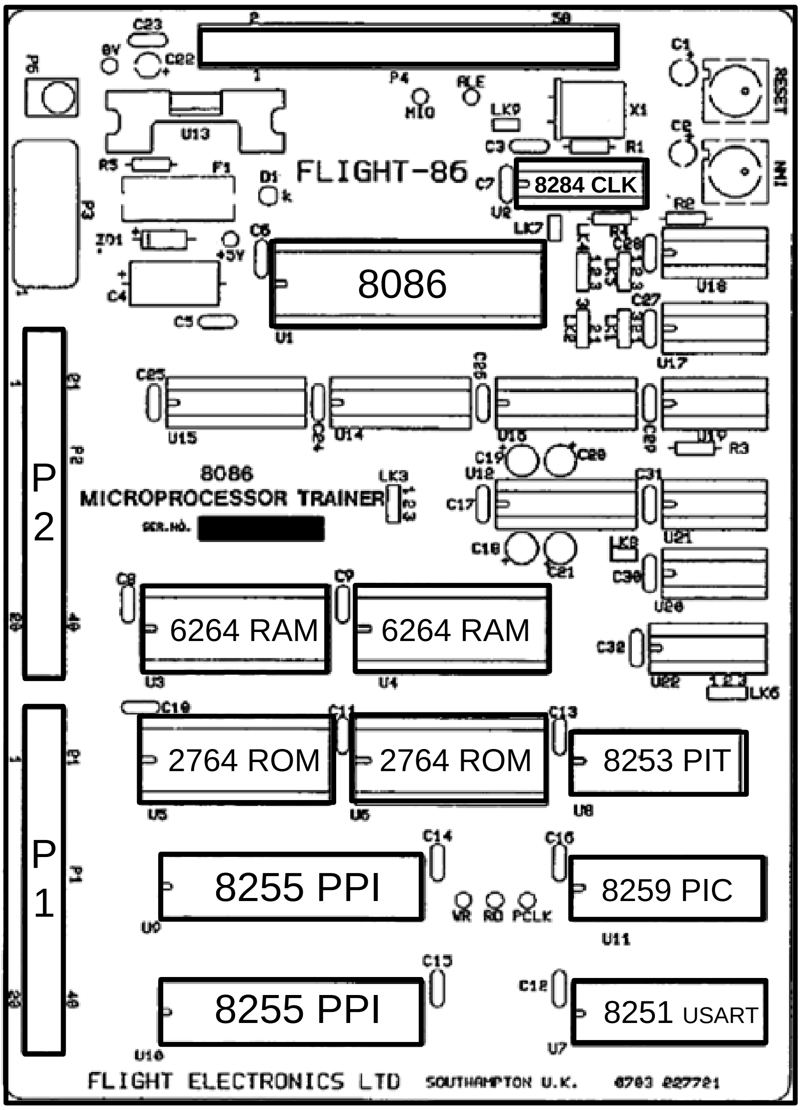
The basic components of the trainer system are described below:

Microprocessor	
CPU 8086	Operating in Min. mode
CGD 8284A	Clock Generator Device Oscillator source: 14.7456 MHz; CPU Clock, CLK: 4.9152 MHz Peripheral clock, PCLK: 2.4576 MHZ
Memory	
EPROM (2) 2764	16k byte (expandable to 64k byte)
RAM (2) 6264	16k byte (expandable to 64k byte)
On-board Peripherals	
PPI (2) 8255A	Programmable Peripheral Interface providing four 8-bit parallel ports with handshake lines
PIT 8253	Programmable Interval Timer providing three 16-bit counter/timer channels
USART 8251A	Universal Synchronous/Asynchronous Receiver/Transmitter
PIC 8259A	Programmable Interrupt Controller providing 8 levels of priority for above devices

Register	Address	Register	Address
8255 PPI (U10) connected to P1		8255 PPI (U9) connected to P2	
Port A	00h	Port A	01h
Port B	02h	Port B	03h
Port C	04h	Port C	05h
Control	06h	Control	07h
8253 PIT (U8)		8259 PIC (U11)	
Count 0	08h	ICW1, OCW2-3	10h
Count 0	0Ah	ICW2-4, OCW1	12h
Count 0	0Ch	8251 USART U7	
Mode Word	0Eh	Data	18h
		Status/Control	1Ah

Table: I/O Map

Layout of the Flight86 Trainer system:

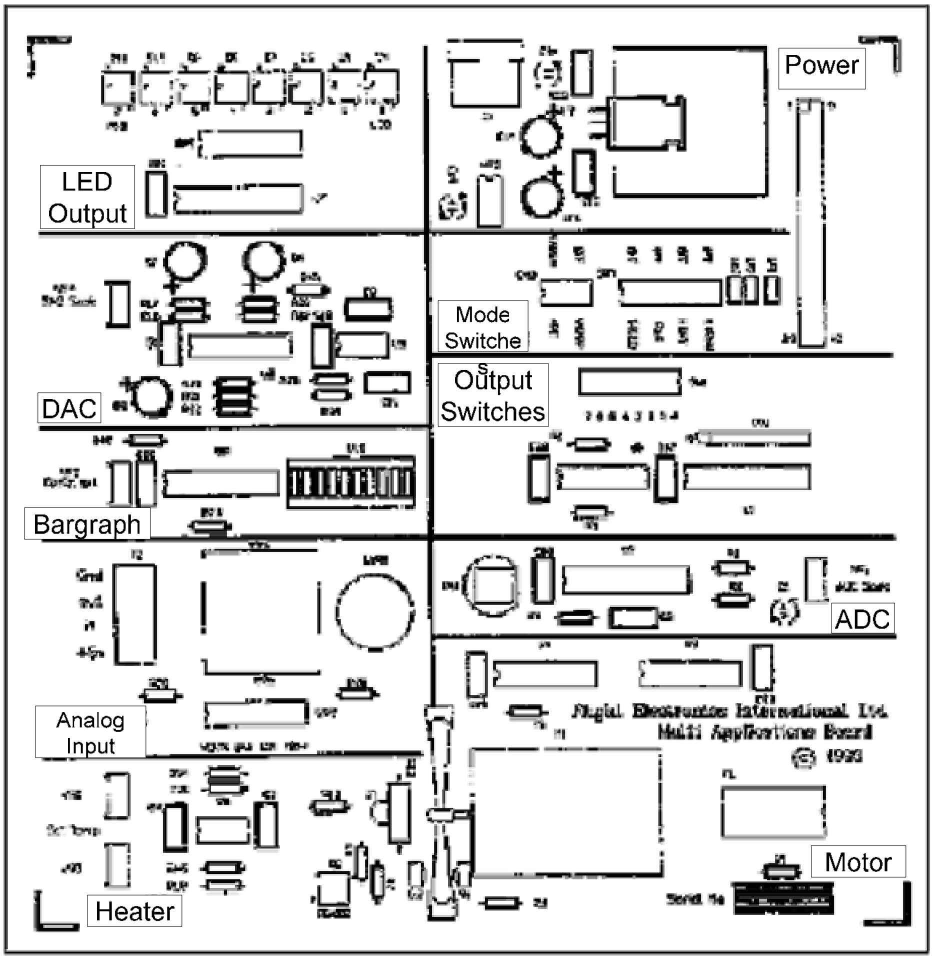


7.2.1 Application Board

The application board is useful for microprocessor interfacing from simple switch and lamp input/output through to more complex closed-loop and open-loop control systems. This board includes the following components:

- Eight digital switches
- Temperature sensor
- Optical speed/position sensor
- Light sensor
- Potentiometer
- External analogue input
- DC motor
- Eight LED's
- Bargraph
- Heater
- Analogue output

Layout of the Application Board



7.2.2 Host (PC) to Controller Board (Flight 86 Trainer) Communication

To establish Host to Controller (Trainer) Board Communication follow this set-up procedure:

1. Turn on Host PC. Ensure you have the DOS prompt showing C:\FLIGHT86>
2. Turn OFF Flight86 board power supply.
3. Connect the serial lead (cable) between the serial port (COM1) on the PC and socket P3 on the Flight86 board.
4. Turn ON Flight86 board DC power supply. [Note: the board indicates that power is actually applied by illuminating the green LED, D1.
5. Type flight86 at the DOS prompt C:\FLIGHT86>, i.e., C:\FLIGHT86>flight86
6. After a few seconds you should see the following messages:

```
FLIGHT86 Controller Board. Host Program Version 2.0.
Press ? and Enter for help – Waiting for controller board response …
ROM found at F000:C000 to F000:FFFF Flight Monitor ROM version 2.0
RAM found at 0000:0000 to 0000:FFFF
-
```

7. The "-" is the host prompt.
8. You have control over the controller board once this prompt is displayed.

If you do not see the prompt, you do not have communications with the controller board.

If the above message stopped at "Waiting for controller board response …" turn the Controller board power supply OFF, wait a few seconds, and turn it ON again. You should then see the "Controller Reset" message followed by the memory test messages.

7.2.3 Host Commands

The command line is executed immediately after the Enter or Return key is pressed. Before this it may be edited using the DEL key.

All data must be specified in hexadecimal, although leading zeros may be omitted. Spaces are ignored for flexible entry, although the first character of a line must be a valid command letter.

The command line syntax uses squared brackets, [ ], to indicate parameter options. The pipe symbol, |, is used to indicate a choice of parameters, one of which must be used.

If an invalid parameter is entered, the full command syntax and help line is displayed to assist.

Command	Key	Parameters	Description
Escape	Esc		Press the Escape button to stop the current command
Reset	X		Resets the training board
Help	?	[*command letter*]	Help on the command
Quit	Q		Terminates running of the board software and returns control to the operating system
Register	R	[*register*]	Allows the user to display or change the content of a register
Memory	M	[W][*segment*:] *address*1 [*address*2]	Allows the user to display or edit one or more memory locations
Assembly	A	[[*segment*:] *address*]	Allows the user to write 8086 assembly code directly into the training board
Disassemble	Z	[[V] [*segment*:] *address*1 [*address*2]]	Reverse assembles the contents of memory
Go	G	[[*segment*:] *address*]	Allows the user to execute code that has been downloaded into RAM
Breakpoint	B	? | R | S [*segment*:] *address*	Allows the user to Display/Clear/Set break points inside his code
Single step	S	[R][[*segment*:] *address*]	Allows the user to step through code one instruction at a time
Download	:	[*drive*:\path\\] *filename*	Loads an Extended Intel Hex file from disk into the memory of the training board
Upload	U	[*drive*:\path\\] file [seg:] add1 add2	Allows a block of memory to be saved to a disk file in Extended Intel Hex format

The **Assemble** command (A) is used to enter 8086 assembly code commands, and these will be assembled and the bytes stored directly into memory. There is a short delay as the bytes of code are sent to the board.

Note: The origin address for user RAM on the FLIGHT86 system is 0050:0100.

The **Disassemble** command (Z) allows the contents of memory to be reverse assembled to 8086 mnemonic codes. If the V option is specified, then the ASCII codes for the disassembled bytes are displayed alongside each line. For example,

```
- Z V 100 130
     displays the disassembled code between the addresses specified with ASCII codes
```

The **Download** command (D) loads an Extended Intel Hex file from disk and puts it into the appropriate memory locations. The original file may have been generated using an assembler, compiler or the Upload command. For example,

```
- : C:\FLIGHT86\program.hex
    downloads file program.hex from drive C subdirectory FLIGHT86
```

The **Upload** command (U) allows a block of memory to be saved to a disk file. The data is saved in the Extended Intel Hex format. If the file extension is not specified, then it defaults ti .HX. For example,

```
- U C:\FLIGHT86\program.hex 0050:0100 150
    saves contents of block 0050:0100 to 0050:0150 to file program.hex on drive C
    subdirectory FLIGHT86
```

Note: This command is useful for saving your program to be shown to your lab instructor later while you continue to work on another program. So, the next time you don't have to assemble the program again but simply download it using the Download command ":".

7.2.4 I/O Interfacing using 8255 PPI

The FLIGHT86 controller board has two 8255 PPI chips that provide ports for I/O interfacing. To be able to use the 8255 for I/O interfacing, a control byte has to be written to the Control register of the 8255 so that any of the ports A, B or C can be set up as input or output ports. The format of the Control byte is as shown below:

b7	b6	b5	b4	b3	b2	b1	b0
1	Mode bits for port A, and port C_{Upper}		Port A	Port C_{Upper}	Mode bit for port B, and port C_{Lower}	Port B	Port C_{Lower}

Port bit:
0 → output
1 → input

b6	b5	Mode	b2	Mode
0	0	Mode 0 - Simple I/O	0	Mode 0: Simple I/O
0	1	Mode 1 - Strobed I/O	1	Mode 1: Strobed I/O
1	x	Mode 2*- Bidirectional I/O		

* - Only port A can be configured for Mode 2 whereas the other ports (B, and C) can be used either for input or output only.

Example:

b7	b6	b5	b4	b3	b2	b1	b0	Configuration
1	0	0	0	1	0	0	1	Mode 0; ports A and B: **output**; port C: **input**
1	0	0	0	0	0	1	0	Mode 0; ports A and C: **output**; port B: **input**

The Flight86 controller board communicates with the Application board using the 8255 PPI. The addresses of these ports are given below:

Register	Address	Register	Address
8255 PPI (U10) connected to P1		8255 PPI (U9) connected to P2	
Port A	00h	Port A	01h
Port B	02h	Port B	03h
Port C	04h	Port C	05h
Control	06h	Control	07h

Note that the Flight86 controller board has two 8255 devices one connected to port P1 and the other to port P2 through which it can connect to the Application board. The addresses of the 8255 registers depend on which port is used.

Output Port:

The Controller board output port connects to **Port B** on the Applications Board, and the state of the 8 lines will always be displayed on the 8 colored LED's.

By means of on board mode switches, this port can be used to control the motor (forward and reverse) and/or the heater.

When not in use for these functions, the output port can be used to drive the Digital to Analogue Converter (D/A).

Input Port:

The processor input port connects to **Port A** on the Applications Board, and by selection via mode switches can be used to read the 8 bit DIL switch, or the output of the Analogue to Digital Converter (A/D), or the output of the D/A comparator, and/or the output of the speed sensing infra-red detector.

7.3 Pre-lab:

1. Read all the above sections of this experiment.
2. Read the topic on 8255 PPI from your text book.
3. Read about the IN and OUT instructions from your text book.

7.4 Lab Work:

Part 1: Familiarization with the Flight86 trainer system

The lab instructor will give a demonstration and introduce you to the Flight86 Controller board, the monitor program, and the Application board.

Part 2: Initialization

Before the Flight86 Controller board can communicate with the Application board, the 8255 on the Controller board must be initialized. Telling the 8255 how to perform is known as INITIALIZATION, so the first program we run after power up or reset, must be one which initializes the 8255.

```
MOV AL, 99          ; set up Port A IN, Port B OUT, Port C IN
OUT 07, AL          ; and output this word  to control Port
MOV AL,0            ; data zero
OUT 03,AL           ; output to Port B to ensure all LEDs off
INT 5               ; Return to Monitor program and prompt
```

Note: This program without the last line INT 5 will be required at the start of every program you write for the Flight86 system to ensure the 8255 is set up before doing any thing else.

Procedure

1. Start up the Flight86 board as described in section **1.2 Host (PC) to Controller Board Communication (Flight86 Trainer)** obtaining the sign on message, followed by the ‘-‘ prompt. (Ask your lab instructor to show you how).
2. Make sure the ribbon cable connects port P2 on the Flight86 board to the port on the Application board.
3. Turn ON the Application board power supply.
4. Now enter **A 0050:0100** at the ‘-‘ prompt. This starts the line assembler at the desired address. The monitor program responds by echoing the address 0050:0100. Now enter the program as shown above. Press the Enter or Return key after each line of code. The monitor program goes to the next address automatically. The screen will look like:

```
0050:0100 MOV AL, 99
0050:0102 OUT 07, AL
0050:0104 MOV AL,0
0050:0106 OUT 03,AL
0050:0108 INT 5
0050:010A
```

5. Press the ESC key at the last address 0050:010A to return to the '-' prompt.
6. Enter **Z 0050:0100** at the prompt. The code just entered will be listed on the screen and can be checked for accuracy.
7. Now enter **G 0050:0100** at the prompt and press the Enter key. The program will now run, initialize the 8255, and return to the prompt. Any LEDs that were lit on the application board will now turn off.
8. Enter **U C:\FLIGHT86\init.hex 0050:0100 10A** at the prompt to upload the above program from the Controller board memory to a file init.hex on drive C subdirectory FLIGHT86.
9. Now press the RESET button on the Flight86 Controller board.
10. Enter the following command at the prompt to download init.hex to the Controller board memory.

: C:\FLIGHT86\init.hex

11. Now enter **G 0050:0100** at the prompt to run the program again.

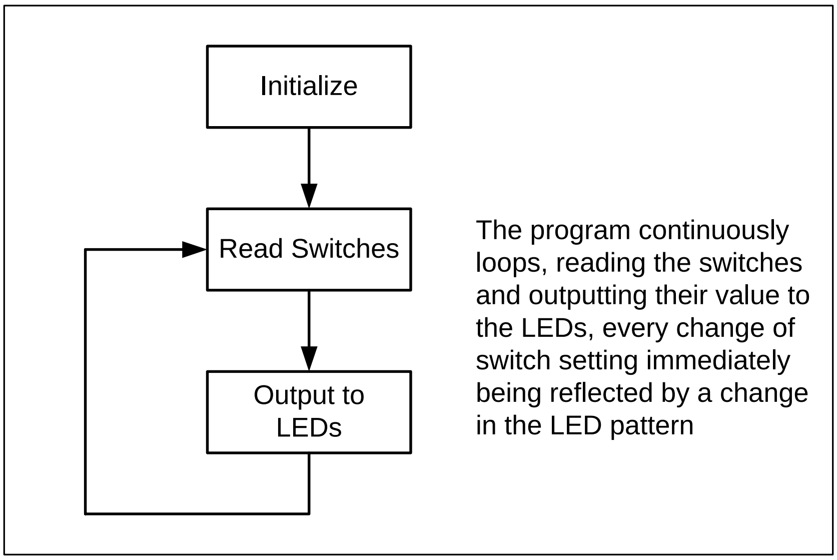
Note that you don't have to enter the whole program again when the controller board is reset for some reason, if you have saved it to a file using the Upload command. This is particularly useful when writing large programs.

Part 3: SWITCHES and LEDs

The Application board has an 8 bit DIL switch and 8 colored LEDs. For example, if switch 2 is set, then LED 2 (the green LED on the right side) will turn ON, and so on.

Write a program that will read the state of the 8 bit DIL switch and output data to the 8 colored LEDs. If a switch is ON, then an LED in the corresponding position will turn ON.

The program logic is illustrated in the following flowchart:



The program continuously loops, reading the switches and outputting their value to the LEDs, every change of switch setting immediately being reflected by a change in the LED pattern

The switches are connected to Port A (address 01), and the LEDs are connected to Port B (address 03) when mode switch SW2A is pushed up to SWITCH position. All the other mode switches must remain in the OFF position.

Procedure

1. Follow steps 1, 2, and 3 of the procedure of Part 1 above.
2. Include the initialization code (without the last line) at the beginning of your program.
3. Assemble your program at 0050:0100.
4. Disassemble your program using the Z command to check for accuracy.
5. Run your program using the G command.
6. Change the settings of the DIL switch and test your program.
7. Check your program again if it does not work.
8. If your program works correctly, then repeat steps 8, 9, 10, and 11 of the procedure of Part 1 above, this time name the file as SW_to_LEDS.hex. The end address to be specified with the Upload command will be the address after the last instruction of your program.
9. Alter the program above so that when a switch is set, the respective LED goes off. Then, repeat steps 3 through 8 above.

Experiment #8

Flight86 Application I – Traffic Lights

8.0 Objectives:

The objective of this experiment is to simulate a traffic lights system.

In this experiment, you will do the following:

- Create software time delays
- Write programs to simulate a traffic lights system
- Assemble, download, and test your program on the trainer board

8.1 Equipment and Software

- Flight86 Trainer and Application Boards
- PC with Flight86 Monitor program
- Assembler and conversion utilities (exe2bin, bin2hex)

8.2 Introduction:

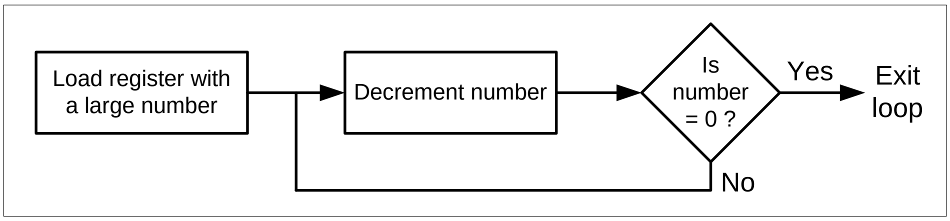
It is often necessary to control how long certain actions last, this can be achieved using software delays, or more accurately by the use of a timer.

In this experiment we will simulate a traffic lights system that requires use of software time delay.

8.2.1 Creating Software Delays

In the various states of the traffic lights sequence, lights have to be ON or OFF for a clearly defined time in seconds, so our program must contain a means of measuring one second. The easiest way, which does not need any further hardware devices, is a software delay.

If we create a program that loops around itself, and does this for a fixed number of times, for a given processor, running at a given clock rate, this process will always take the same time. All we have to do is write such a multiple loop so that it takes one second to complete. This process is illustrated in the flowchart below:



Now the question is: how do we calculate the ‘large number’ to be loaded in the register for the loop?

To calculate a specific time delay we need to calculate the number of times the program will loop around itself. To do this, we need to know how many clock cycles are required to carry out a particular instruction(s), and the processor clock rate which ultimately decides how long an instruction takes to execute.

Let’s examine the code below. This code can be used to produce a certain delay value. We will try to find the value of N such that this code produces a delay of approximately 100ms.

```
DELAY:  MOV CX, N          ; Load CX with a fixed value
DEL1:   DEC CX             ; decrement CX
        JNZ DEL1           ; and loop if not zero
        RET                ; when CX=0, then exit
```

We can see in the code above that instructions which get repeatedly executed (inside the loop) are **DEC CX** and **JNZ DEL1**. The number of clock cycles required to execute these two instructions once, are:

DEC CX	2 clock cycles
JNZ DEL1	16 clock cycles
Total :	18 clock cycles

Note: If we also consider the time required to execute the instructions outside the loop, i.e., the first and the last instructions above, then we can get a more accurate time delay. But we will ignore this, since these instructions are executed only once.

Number of times this loop is executed is: **N**
CPU clock rate for the Flight86 Trainer system is: **CLK = 4.9152 MHz**
Time period for one clock cycle, $\mathbf{T_{CLK}}$ = 1/(4.9152 x 10^6) = **203.5ns**

(Total clock cycles) x (Number of times loop is executed) x T_{CLK} = 100 ms

18 x N x 203.5ns **=** 100 ms; ⟶ Solving for N, we get **N = 27300** = 6AA4H

Therefore, if the above loop is executed N = 27300 times we get a delay of 100ms approximately. Now, this loop can be used to produce a delay of **1 second** if it is executed 10 times. That means, there will be two loops – one that produces 100 ms delay, and the other that executes this loop 10 times to produce 10 x 100ms = 1 second.

8.3 Exercise: Simulating a Traffic lights system

The LED’s on the application board are arranged in two groups of 4 - Red, Amber, Green, and a Yellow. Using these two sets of four lights we can easily simulate the traffic lights at a busy cross road, one set representing the main road, the other set the side road.

Red1	Amber1	Green1	Yellow1	Red2	Amber2	Green2	Yellow2
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

The traffic lights system must be simulated according to the following sequence which must be **repeated** continuously (the Yellow LED’s are not used):

Main Road		Side Road									
Signal	Duration	Signal	Duration	Bit value							
Red 1	15 sec	Green 2	15 sec	1	0	0	0	0	0	1	0
Red 1	03 sec	Amber 2	03 sec								
Red & Amber 1	03 sec	Red 2	03 sec								
Green 1	25 sec	Red 2	25 sec								
Amber 1	03 sec	Red 2	03 sec								
Red 1	03 sec	Red & Amber 2	03 sec								

Table 1: Sequence of lights

The bit value represents the output pattern to turn on the required LED’s.

An easy way to implement the above sequence of lights repetitively is to set all the data up in a table, and advance through the data step by step, until the end of the table is reached, when it can be repeated. The table can contain both the required LED pattern, and the time that pattern is to be manipulated. For the above case, the table would be:

Data, time
82h,150

LED pattern 82h (10000010), maintained for 150 x 100ms (= 15 sec).

The delay is implemented in multiples of 100 ms.

Table 2: Data for traffic lights sequence

8.4 Review

1. Review the hardware specifications of the Flight86 system described in the last experiment.
2. Read about instruction clock cycles from your text book.

8.5 Pre-lab:

1. Complete the bit-value output pattern in Table 1.
2. Complete Table 2 for the corresponding bit-value pattern in Table 1.

   Table 2 can be implemented in assembly language using the "DB" (define byte) assembler directive, as shown below:

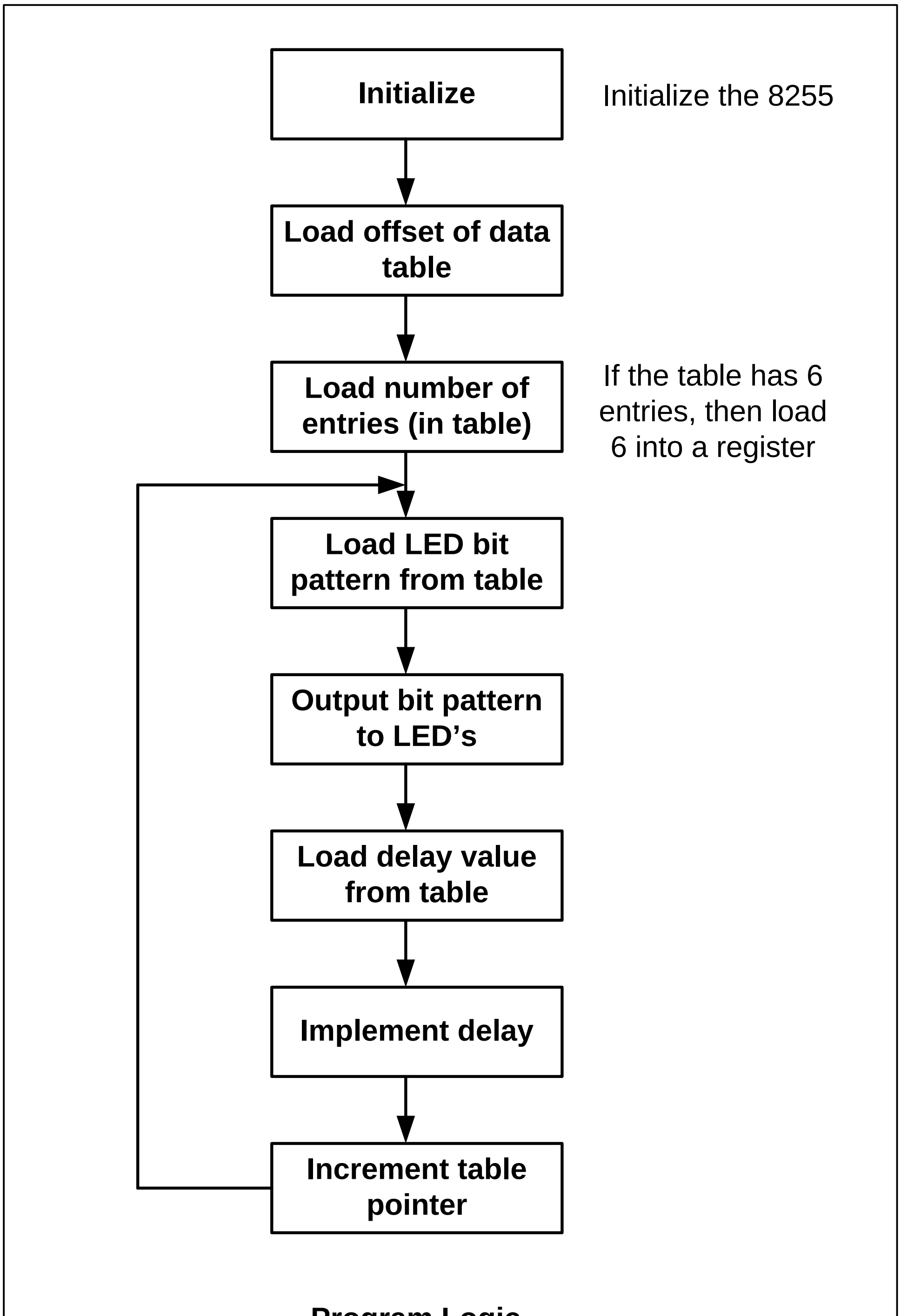
```
    TABLE    DB 82h, 150        ; RED1  GREEN2            15 sec
             DB      ,          ; RED1  AMBER2            03 sec
             DB      ,          ; RED/AMBER1 RED2         03 sec
             DB      ,          ; GREEN1 RED2             25 sec
             DB      ,          ; AMBER1 RED2             03 sec
             DB      ,          ; RED1 RED/AMBER2         03 sec
```

3. Complete the remaining entries of this table and place it after the last instruction of your program that you will write for the Lab Work.
4. Write a program to produce a delay of 5 seconds using the code shown below which produces a delay of 100ms.

```
DELAY:   MOV CX, 27300      ; Load CX with a fixed value
DEL1:    DEC CX             ; decrement CX
         JNZ DEL1           ; and loop if not zero
         RET                ; when CX=0, then exit
```

8.6 Lab Work:

1. Write your program according to the flow chart shown below.


Initialize
Initialize the 8255
Load offset of data table
Load number of entries (in table)
If the table has 6 entries, then load 6 into a register
Load LED bit pattern from table
Output bit pattern to LED's
Load delay value from table
Implement delay
Increment table pointer

Program Logic

2. Include the following initialization code at the beginning of your program. (See previous experiment).

```
INIT:
        MOV AL, 99          ; set up Port A & Port C IN, Port B OUT
        OUT 07, AL          ; and output this word  to control Port
        MOV AL,0            ; data zero
        OUT 03,AL           ; output to Port B to ensure all LED's off
```

3. Use an editor to write to your program. Name your file as ***traffic.asm***
4. Assemble and link your program using the TASM assembler to produce ***traffic.exe***.
5. Convert the ***traffic.exe*** file to binary format using the exe2bin.exe program by typing ***exe2bin traffic.exe*** at the DOS prompt.
6. Convert the ***traffic.bin*** file ***to traffic.hex*** using the bin2hex.exe program by typing ***bin2hex traffic.bin*** at the DOS prompt.
7. Start the Flight86 monitor program.
8. Download ***traffic.hex*** to the Flight86 controller board by typing at the '-' prompt

   : C:\FLIGHT86\traffic.hex

9. Before you run your program make sure the mode switches are in the correct position.

Switch SW2A – Switch position
All other switches - OFF

10. Now enter **G 0050:0100** at the '-' prompt to run the program.
11. Check the sequence of lights and the time duration generated by your program and make sure it is working correctly.

Experiment #9

Flight86 Application II – Motor Control

9.0 Objectives:

The objective of this experiment is to control the operation and speed of a DC motor.

In this experiment, you will do the following:

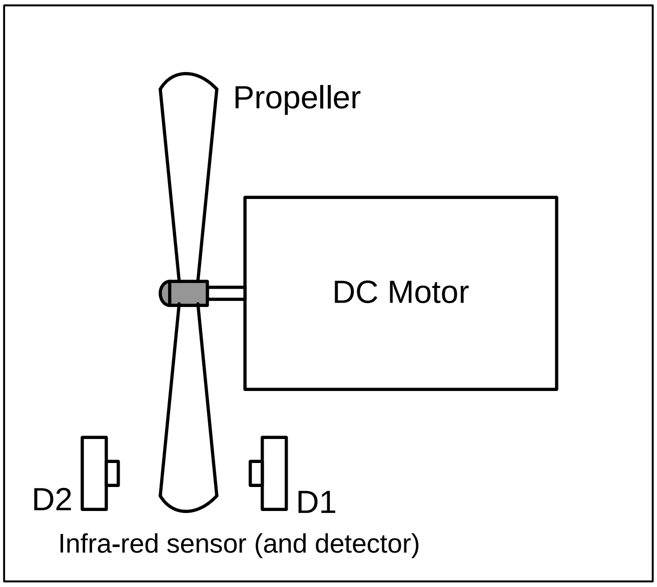
- Write a program to control the DC motor operation
- Write a program to vary the speed of the DC motor with a potentiometer
- Assemble, download, and test your programs on the trainer board

9.1 Equipment and Software

- Flight86 Trainer and Application Boards
- PC with Flight86 Monitor program
- Assembler and conversion utilities (exe2bin, bin2hex)

9.2 Introduction:

There is a small DC motor on the Application board. This motor with 3 bladed-propeller is limited to approx. 8000 RPM by the two current limiting resistors connecting it to the board. The polarity of the voltage applied to the motor and hence forward and reverse is selected by a relay. The motor is driven from the unregulated supply obtained from the mains adaptor (approx. 9V).



As the motor rotates, the propeller blades pass between D1, an infra-red source, and D2 an infra-red detector, the change in current through the detector provides a signal 3 times per revolution back to Port A bit 4. However, for this to function, SW4A must be set to 'SPEED'.

9.2.1 Motor Operation

Motor ON forward/reverse selection is by output bits 6 and 7 on Port B, their value is decoded by U4 (74LS139 – 2 x 4 decoder).

Port B		
Bit 7	**Bit 6**	**Motor Operation**
0	0	Stop
0	1	Forward motion
1	0	Reverse motion
1	1	Stop

Table 1: Motor Operation

To decode and drive any output of U4, first U4 must be enabled by placing SW2B in 'MOTOR' position.

In this experiment, we will write a program to control the operation of the Motor with two DIL switches available on the Applications Board.

9.2.2 Motor Speed Control

The speed controller works by varying the average voltage sent to the motor. It could do this by simply adjusting the voltage sent to the motor, but this is quite inefficient to do. A better way is to switch the motor's supply on and off very quickly. If the switching is fast enough, the motor doesn't notice it, it only notices the average effect.

Now imagine a light bulb with a switch. When you close the switch, the bulb goes on and is at full brightness, say 100 Watts. When you open the switch it goes off (0 Watts). Now if you close the switch for a fraction of a second, and then open it for the same amount of time, the filament won't have time to cool down and heat up, and you will just get an average glow of 50 Watts. This is how lamp dimmers work, and the same principle is used by speed controllers to drive a motor.

The speed of the Motor can be varied by turning the Motor *ON* and *OFF* very quickly.

If the supply voltage is switched fast enough, it won't have time to change speed much, and the speed will be quite steady. This is the principle of switch mode speed control. This principle of controlling the speed of the motor by turning the supply on and off very quickly is called Pulse Width Modulation (PWM).

As the amount of time that the voltage is *on* increases compared with the amount of time that it is *off*, the average speed of the motor increases.

9.2.3 Speed Control with a Potentiometer

The Potentiometer is an analogue input which provides a linear voltage between 0 and 2.55V. We will convert this voltage to a digital value with the help of the Analog-to-Digital Converter (ADC) provided on the board. The specifications of this ADC are given below:

Clock rate 400 KHz
Conversion time 180 us
Input 0.00 V for 00 Hex output Input 2.50 V for 80 Hex output Input 5.00 V for FF Hex output

This is an 8-bit ADC, free running at a clock rate of approx. 400 KHz. As the ADC is free running it is completely asynchronous with any 'read' from the microprocessor training board (Controller board), which means if read just as the output of the ADC is being updated, false readings could be obtained. This can be overcome by reading twice or more times, only accepting a value when it is unchanged over two consecutive readings. For most microprocessors, this should provide no problem in taking two readings in between conversions of the ADC.

To obtain a reliable reading, take reading two or more times, only accepting a value when it is unchanged over two consecutive readings.

In this experiment, we will control the speed of the Motor with a Potentiometer provided on the Applications Board. However, we will not use the Potentiometer to vary the voltage supplied to the Motor but to determine how long the motor should be turned OFF. The Potentiometer value will be converted to a digital value which will then be used to index into a table. This table will contain delay values for the duration for which the running motor will be turned OFF.

The delay values will be stored in a decreasing order so that higher Potentiometer voltage corresponds to lower delay value, and vice-versa.

As we know, the speed of the Motor can be varied by turning the Motor ON and OFF very quickly, we will use ON/OFF duration in multiples of 1ms (milliseconds). This will ensure that the speed of the Motor will be quite steady.

To make things simple, the 'ON' duration of the Motor will be kept constant. Only the 'OFF' duration will change depending on the Potentiometer value.

As the Potentiometer is turned to produce increasing voltage (up to 2.55V), the 'OFF' delay value selected from the table will be smaller, and thus, the speed of the Motor will be increased. Similarly, as the Potentiometer is turned to produce decreasing voltage (down to 0V), the 'OFF' delay value selected from the table will be larger, and thus, the speed of the Motor will be decreased.

The ADC will produce values in the range 00H to 80H (0 to approx. 130) corresponding to Potentiometer values (0 to 2.55V). This range of ADC values (0 to approx. 130) requires a table of 130 elements. This is quite large and not practical for a small application like this. To make matters easy, we will scale down the ADC range. If we divide the ADC value by 13, our effective range is scaled down to (0 to 10).

The delay must be implemented in multiples of 1ms. The following code can be used to implement a delay of approx. 1ms.

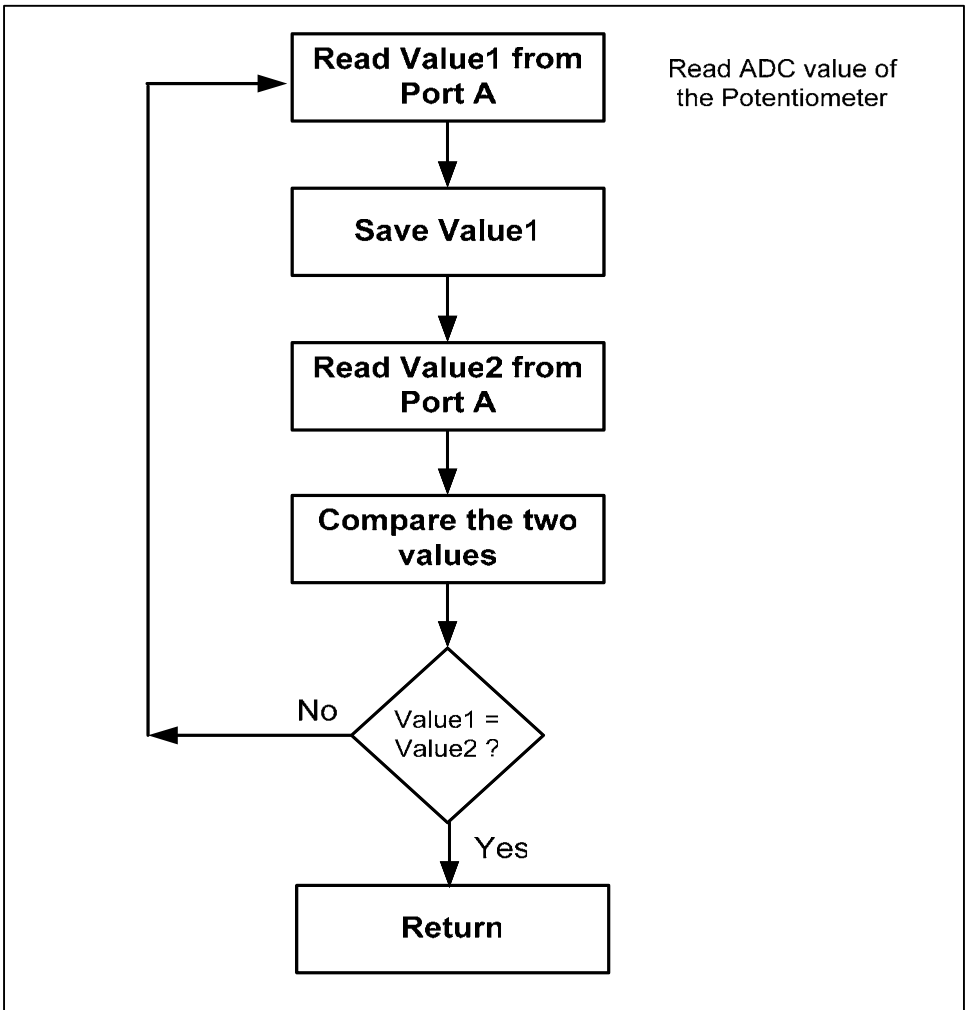
```
DELAY:   MOV CX, 270        ; Load CX with a fixed value
DEL1:    DEC CX             ; decrement CX
         JNZ DEL1           ; and loop if not zero
         RET                ; when CX=0, then exit
```

9.3 Pre-lab

1. Review the hardware specifications of the Flight86 system described in the experiment – **Introduction to Flight86 Trainer and Application Board**.
2. Read all the above sections of this experiment.
3. Write a subroutine to read the ADC value (which provides the Potentiometer value converted to a digital value) on Port A, when Switch SW2B is in Motor position and Switch SW3 is in VOLTS position. This subroutine will be used in Part 2 of the Lab Work of this experiment for Motor Speed Control.

To obtain a reliable reading, you program must take reading two or more times, only accepting a value when it is unchanged over two consecutive readings.

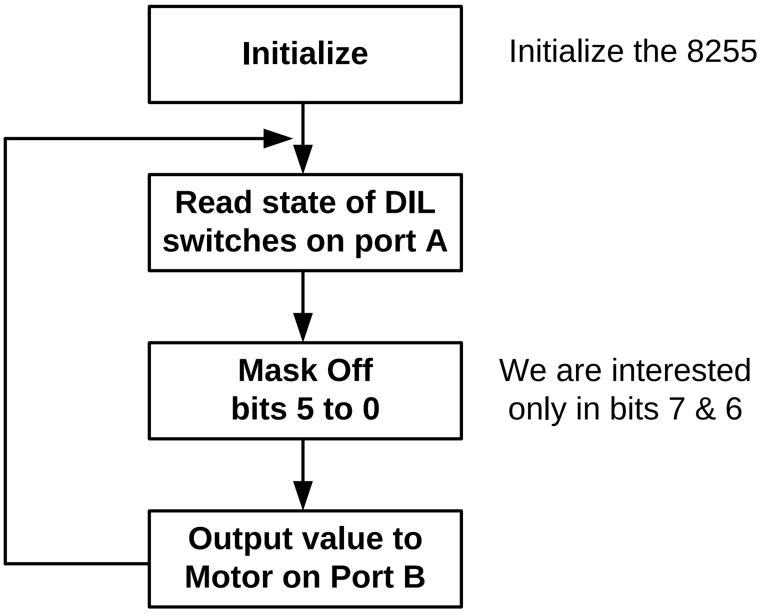
The logic for the read operation is illustrated in the flowchart below:



Program Logic: Reading ADC value

9.4 Lab Work – Part 1: Motor Operation

Write a program to control the operation of the DC Motor. The operation of the motor must be controlled by bits 7 and 6 of the DIL switches (according to Table 1 shown above). The logic of the program is illustrated by the flowchart shown below:


Initialize
Initialize the 8255
Read state of DIL switches on port A
Mask Off bits 5 to 0
We are interested only in bits 7 & 6
Output value to Motor on Port B

Program Logic: Motor Operation

<u>Procedure</u>:

1. Include the following initialization code at the beginning of your program.

```
INIT:
        MOV AL, 99          ; set up Port A IN, Port B OUT,  Port C IN
        OUT 07, AL          ; and output this word  to control Port
        MOV AL,0            ; data zero
        OUT 03,AL           ; output to Port B to ensure all LEDs off
```

2. Use an editor to write to your program. Name your file as ***motor_oper.asm***
3. Assemble and link your program using the TASM assembler to produce ***motor_oper.exe***.
4. Convert the ***motor_oper.exe*** file to binary format using the exe2bin.exe program by typing ***exe2bin motor_oper.exe*** at the DOS prompt.
5. Convert the ***motor_oper.bin*** file ***to motor_oper.hex*** using the bin2hex.exe program by typing ***bin2hex motor_oper.bin*** at the DOS prompt.

6. Start the Flight86 monitor program.
7. Download ***motor_oper.hex*** to the Flight86 controller board by typing at the '-' prompt

   : C:\FLIGHT86\ motor_oper.hex

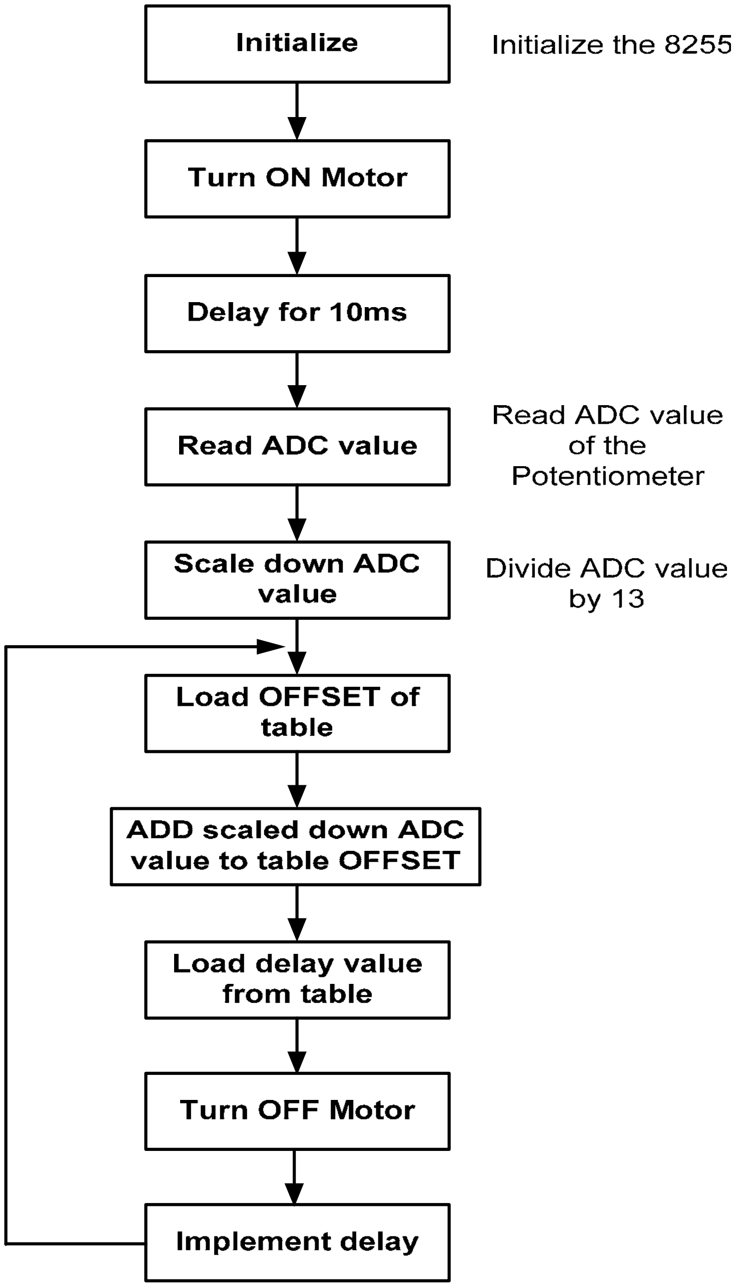
8. Before you run your program make sure the mode switches are in the correct position.

Switch SW2A – Switch position
Switch SW2B – Motor position
All other switches - OFF

9. Now enter **G 0050:0100** at the '-' prompt to run the program.
10. Check the operation of the Motor by changing the state of the two leftmost switches. Make sure the Motor operates according to Table 1.

9.5 Lab Work – Part 2: Motor Speed Control

Write a program to control the speed of the Motor. The speed of the motor must be controlled by the Potentiometer. The logic of the program is illustrated by the flowchart shown below:



Program Logic: Speed Control

Procedure:

1. Include the initialization code at the beginning of your program.
2. For your program, write a subroutine to read the ADC value. This subroutine can then be called using the CALL instruction. (Refer to section "**Speed Control with a Potentiometer**" for more information).
3. Place this table after the last instruction of your program.

```
TABLE   DB 0AH      ; 10 ms delay
        DB 09H      ; 9 ms delay
        DB 08H      ; 8 ms delay
        DB 07H      ; 7 ms delay
        DB 06H      ; 6 ms delay
        DB 05H      ; 5 ms delay
        DB 04H      ; 4 ms delay
        DB 03H      ; 3 ms delay
        DB 02H      ; 2 ms delay
        DB 01H      ; 1 ms delay
```

4. Use an editor to write to your program. Name your file as ***motor_speed.asm***
5. Assemble and link your program using the TASM assembler to produce ***motor_speed.exe***.
6. Convert the ***motor_speed.exe*** file to binary format using the exe2bin.exe program by typing ***exe2bin motor_speed.exe*** at the DOS prompt.
7. Convert the ***motor_speed.bin*** file ***to motor_speed.hex*** using the bin2hex.exe program by typing ***bin2hex motor_speed.bin*** at the DOS prompt.
8. Start the Flight86 monitor program.
9. Download ***motor_speed.hex*** to the Flight86 controller board by typing at the '-' prompt

   : C:\FLIGHT86\ motor_speed.hex

10. Before you run your program make sure the mode switches are in the correct position.

Switch SW2B – Motor position
Switch SW3 – VOLTS
All other switches – OFF

11. Now enter **G 0050:0100** at the '-' prompt to run the program.
12. Check the speed control of the Motor by turning the Potentiometer both ways.

Experiment #10

Introduction to the 8051 Microcontroller

10.0 Objectives:

The objective of this experiment is to learn to use the 8-bit 89C51 microcontroller to implement a simple LED controlling system.

In this experiment, you will do the following:

- Understand the difference between microprocessors and microcontrollers
- Learn about the MCS-51 (8051) microcontrollers – in particular the ATMEL 89C51
- Implement a LED controlling system using the ATMEL 89C51 microcontroller
- Learn to use the Microcontroller/EEPROM programming tool: WINLV

10.1 Equipment, Software, and Components:

- MicroMaster LV48
- WINLV software
- Assembler and conversion utilities (exe2bin, bin2hex)
- AT89C51 microcontroller
- 11.0592 MHz crystal
- Resistors: 510, 8.2K
- Capacitors: 33pF (2), 10pF
- Proto-board (with 5V supply, LED's, switches)

10.2 Introduction:

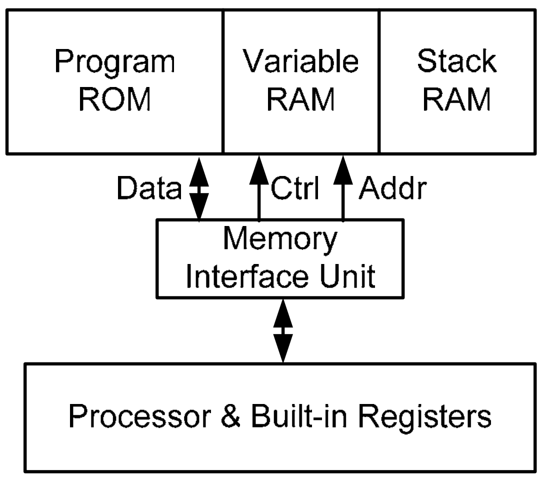
Microprocessors and Microcontrollers

A microprocessor is a **general-purpose** digital computer central processing unit. To make a complete microcomputer, you add memory (ROM and RAM) memory decoders, an oscillator, and a number of I/O devices. The prime use of a microprocessor is to read data, perform extensive calculations on that data, and store the results in a mass storage device or display the results. The design of the microprocessor is driven by the desire to make it as expandable and flexible as possible.

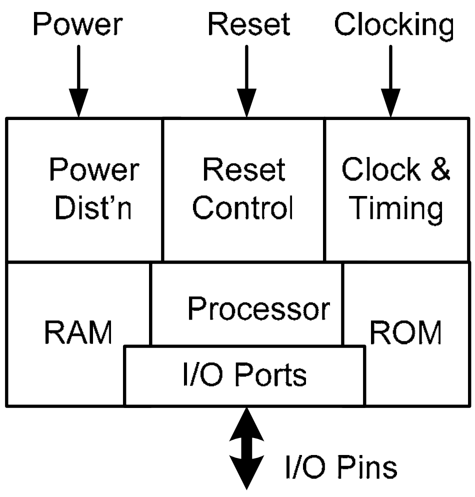
A microcontroller is a true **computer on a chip**. The design incorporates all of the features found in a microprocessor CPU: ALU, PC, SP, and registers, plus ROM, RAM, parallel I/O, serial I/O, counters and a clock circuit – **all in a single IC**. The microcontroller is a general-purpose device meant to read data, perform limited calculations on that data and control its environment based on those calculations. The prime use of a microcontroller is to control the operations of a machine using a fixed program that is stored in ROM and does not change over the lifetime of the system. The microcontroller is concerned with getting data from and to its own pins; the architecture and instruction set are optimized to handle data in bit and byte size.

Therefore, a microcontroller is a **highly integrated device** which includes, on one chip, all or most of the parts needed to perform an application control function.

Microprocessors vs. Microcontrollers	
Microprocessor	**Microcontroller**
• CPU is stand-alone, RAM, ROM, I/O, timer are separate • designer can decide on the amount of ROM, RAM and I/O ports • expansive • versatile • general-purpose • mostly used in microcomputer systems	• CPU, RAM, ROM, I/O and timer, etc. are all on a single chip • fix amount of on-chip ROM, RAM, I/O ports • for applications in which cost, power and space are critical • single-purpose • mostly used in embedded systems



Microprocessor based System



Microcontroller Block Diagram

Microcontrollers are frequently found in home appliances (microwave oven, refrigerators, television and VCRs, stereos), computers and computer equipment (laser printers, modems, disk drives), cars (engine control, diagnostics, climate control), environmental control (greenhouse, factory, home), instrumentation, aerospace, and thousands of other uses. In many items, more than one processor can be found.

Microcontrollers come in many varieties. Depending on the power and features that are needed, one might choose a 4, 8, 16, or 32 bit microcontroller. The following table lists some of the commonly used microcontrollers.

4-bit Microcontrollers
Texas Instruments TMS 1000
National COP420
Hitachi HMCS40
Toshiba TLCS47
8-bit Microcontrollers
Intel 8048
Intel 8051
Microchip PIC16C56
National COP820
Motorola 68HC11
Texas Instruments TMS7500
Zilog Z8
16-bit Microcontrollers
Motorola MC68332
Motorola 68HC16
Intel MCS-96 Family of Microcontrollers
National HPC16164
Hitachi H8/532
32-bit Microcontrollers
Intel 80960CA, KA, KB, MC
LR 33000
AMD Am29050
NS 32000

Table: Some of the commonly used microcontrollers

10.2.1 MCS-51 Family of Microcontrollers

The MCS-51 is a family of microcontroller ICs developed, manufactured, and marketed by Intel Corporation. Other IC manufacturers, such as Siemens, AMD, ATMEL, Philips, etc. are licensed "second source" suppliers of devices in the MCS-51 family.

10.2.2 The 8051 Microcontroller

The generic MCS-51 IC is the 8051 8-bit microcontroller, the first device in the family offered commercially. It is the world's most popular microcontroller core, made by many independent manufacturers (truly multi-sourced). There were 126 million 8051s (and variants) shipped in 1993!! Its features are summarized below:

- CPU with Boolean processor
- 4K bytes ROM (factory masked programmed)
- 128 bytes RAM
- Four 8-bit I/O ports
- Two 16-bit timer/counters
- Serial Interface (programmable full-duplex)
- 64K external code memory space
- 64K external data memory space
- Five interrupts (2 priority levels; 2 external)

10.2.3 8051 Flavors

The 8051 has the widest range of variants of any embedded controller on the market. The smallest device is the Atmel 89c1051, a 20 Pin FLASH variant with 2 timers, UART, 20mA. The fastest parts are from Dallas, with performance close to 10 MIPS! The most powerful chip is the Siemens 80C517A, with 32 Bit ALU, 2 UARTS, 2K RAM, PLCC84 package, 8 x 16 Bit PWMs, and other features.

Among the major manufacturers are:

AMD	Enhanced 8051 parts (no longer producing 80x51 parts)
Atmel	FLASH and semi-custom parts, e.g., 89C51
Dallas	Battery backed, program download, and fastest variants
Intel	8051 through 80c51gb / 80c51sl
OKI	80c154, mask parts
Philips	87c748 thru 89c588 - more variants than anyone else
Siemens	80c501 through 80c517a, and SIECO cores

10.2.4 An Architectural Overview of the MCS-51 (8051) Microcontroller

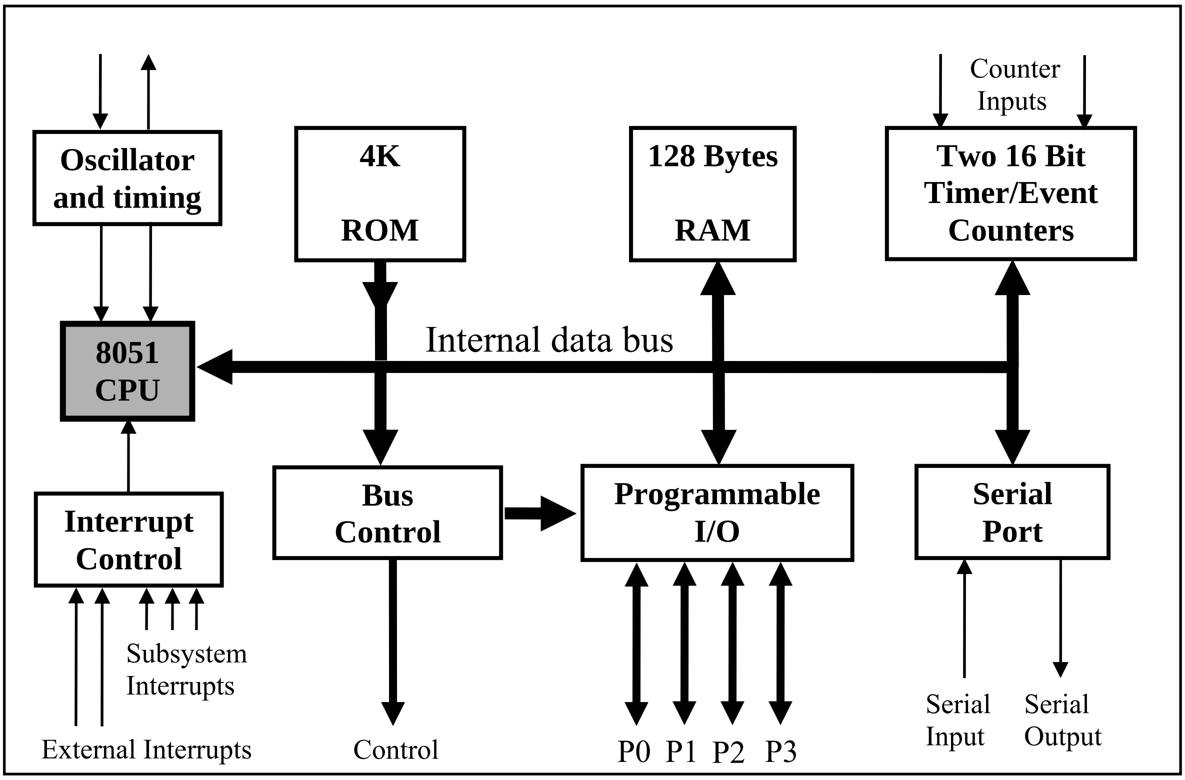
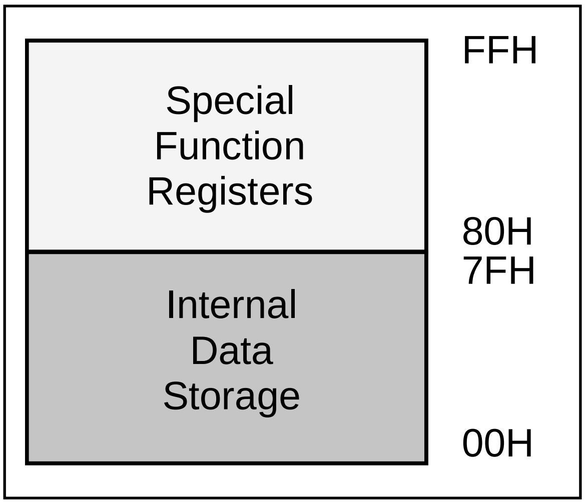


Figure: Block Diagram of the 8051 Core

10.2.5 Data Storage

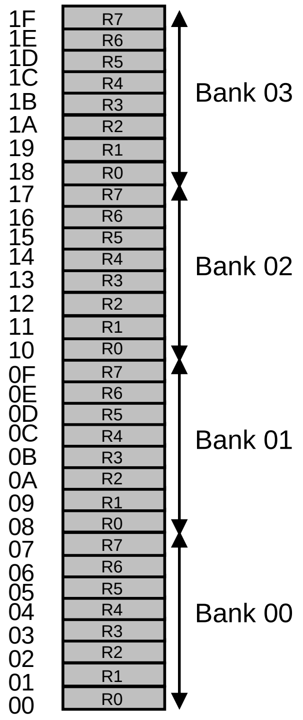
The 8051 has 256 bytes of RAM on-chip. The lower 128 bytes are intended for internal data storage. The upper 128 bytes are the Special Function Registers (SFR). The lower 128 bytes are not to be used as standard RAM. They house the 8051's registers, its default stack area, and other features.



Register Banks

- The lowest 32 bytes of the on-chip RAM form 4 banks of 8 registers each.
- Only one of these banks can be active at any time.
- Bank is chosen by setting 2 bits in PSW
- Default bank (at power up) is bank 0 (locations 00 – 07).
- The 8 registers in any active bank are referred to as R0 through R7.

Given that each register has a specific address; it can be accessed directly using that address even if its bank is not the active one.



10.2.6 Special Function Registers

The upper 128 bytes of the on-chip RAM are used to house special function registers. In reality, only about 25 of these bytes are actually used. The others are reserved for future versions of the 8051.

These are registers associated with important functions in the operation of the MCS-51.

Some of these registers are *bit-addressable* as well as *byte-addressable*. The address of bit 0 of the register will be the same as the address of the register.

- ACC and B registers – 8 bit each
- DPTR : [DPH:DPL] – 16 bit combined
- PC : Program Counter – 16 bits
- Stack pointer SP – 8 bit
- PSW : Program Status Word
- Port Latches
- Serial Data Buffer
- Timer Registers
- Control Registers

See **Appendix B** for a complete list of Special Function Registers and their addresses.

Register A or ACC – Accumulator

This register is commonly used for move operation and arithmetic instructions. It operates in a similar manner to the 8086 accumulator. It is also bit addressable. It can be referred to in several ways:

- Implicitly in op-codes.
- Referred to as ACC (or A) for instructions that allow specifying a register.
- By its SFR address 0E0H.

Register B

It is commonly used as a temporary register. It is also bit addressable.

- Used by two op-codes
  - MUL AB, DIV AB
- B register holds the second operand and will hold part of the result
  - Upper 8 bits of the multiplication result
  - Remainder in case of division.
- Can also be accessed through its SFR address of 0F0H.

DPH and DPL Registers

These are two 8-bit registers which can be combined into a 16-bit DPTR – Data Pointer. The DPTR is used by commands that access external memory.

- Also used for storing 16bit values

```
MOV DPTR, #data16          ; setup DPTR with 16bit ext address
MOVX A, @DPTR              ; copy mem[DPTR] to A
```

- Can be accessed as 2 separate 8-bit registers if needed.
- DPTR is useful for string operations and look up table (LUT) operations.

Port Latches – P0, P1, P2, and P3

These registers specify the value to be output on an output port or the value read from an input port. They are also bit addressable. Each port is connected to an 8-bit register in the SFR.

P0 = 80H, P1 = 90H, P2 = A0H, P3 = B0H

- First bit has the same address as the register.
- Example: P2 has address A0H in the SFR, so
  - P2.7 or A7H refer to the same bit.
- All ports are configured for output at reset.

PSW – Program Status Word

Program Status Word is a bit addressable 8-bit register that has all the status flags.

CY	AC	F0	RS1	RS2	OV	-	P

Symbol	Position	Function
CY	PSW.7	Carry Flag
AC	PSW.6	Auxiliary Carry Flag. For BCD Operations
F0	PSW.5	Flag 0. Available to the user for general purposes.
RS1	PSW.4	Register bank select bits. Set by software to determine which register bank is being used.
RS2	PSW.3	
OV	PSW.2	Overflow Flag
-	PSW.1	Not used
P	PSW.0	Parity Flag. Even Parity.

10.2.7 8051 Instructions

The 8051 has 255 instructions. Every 8-bit op-code from 00 to FF is used except for A5. The instructions are grouped into 5 groups:

- Arithmetic
- Logic
- Data Transfer
- Boolean
- Branching

The following table lists some of the commonly used instructions. See **Appendix A** for a complete list of the 8051 instructions.

Instruction	Description	Execution Cycle
ACALL sub1	Call a subroutine labeled sub1	2
CJNE a, b, addr	Compare a and b, if they are not equal then jump to addr	2
CLR x	Set the bit x to 0	1
DJNZ a, addr	Decrease a by 1, then check if a = 0, then jump to addr	2
INC a	Increase a by 1	1-2
MOV a, b	Move the content in b to a	1-2
MOVC a, addr	Move the content stored in address addr to a	2
RET	Return to main program from a subroutine	2
RETI	Return to main program from an interrupt subroutine	2
SETB x	Set the bit x to 1	1
SJMP addr	Jump to the addr	2
XRL a, b	Perform a XOR b (XOR: Exclusive Or)	1-2

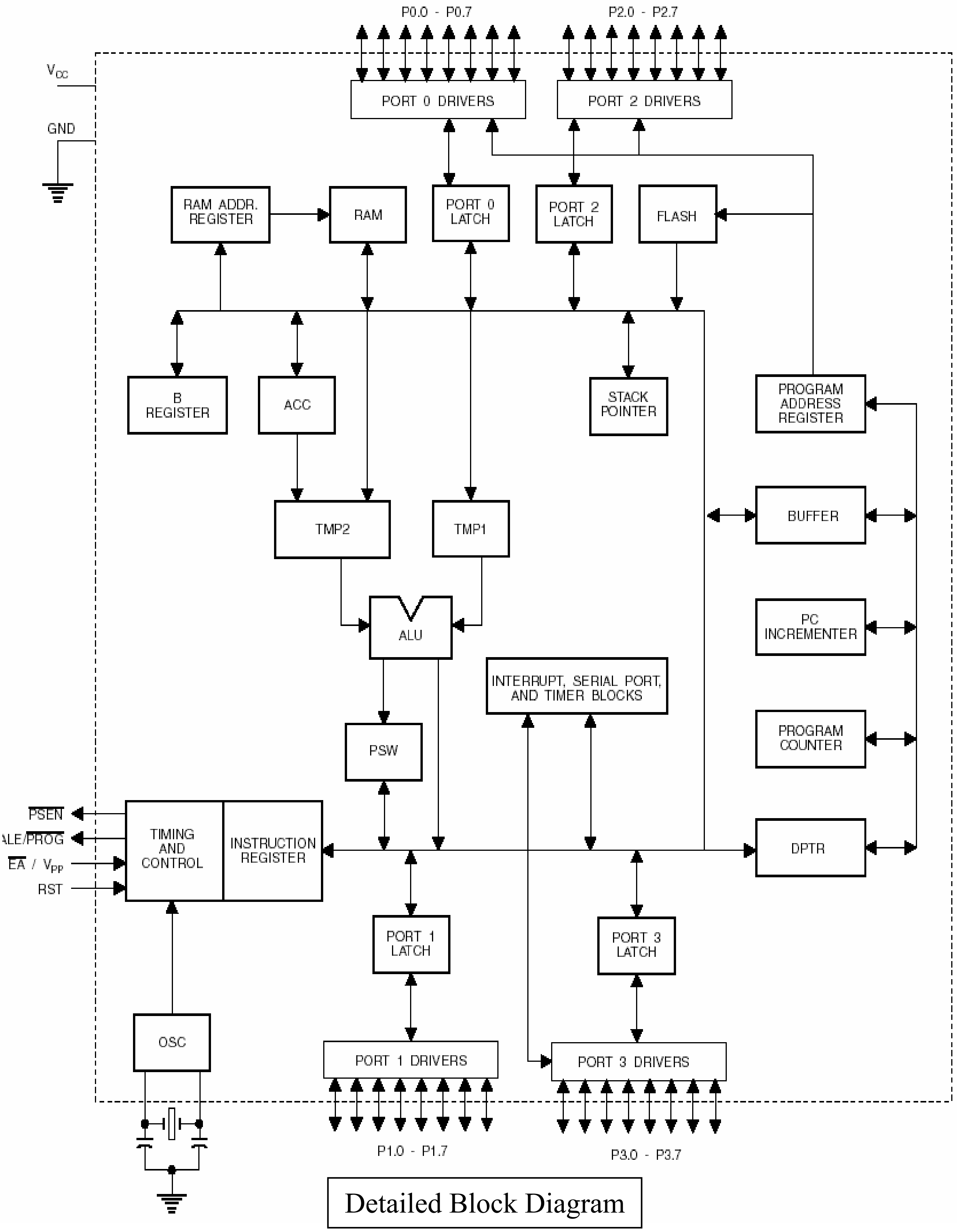
10.2.8 ATMEL 89C51 Microcontroller

The AT89C51 is a low-power, high-performance CMOS 8-bit microcomputer with 4K bytes of **Flash Programmable and Erasable Read Only Memory** (PEROM). The device is manufactured by Atmel and is compatible with the industry-standard MCS-51 instruction set and pin-out. The on-chip Flash allows the program memory to be reprogrammed in-system or by a conventional nonvolatile memory programmer. By combining a versatile 8-bit CPU with Flash on a monolithic chip, the Atmel AT89C51 is a powerful microcomputer which provides a highly-flexible and cost-effective solution to many embedded control applications.

Features

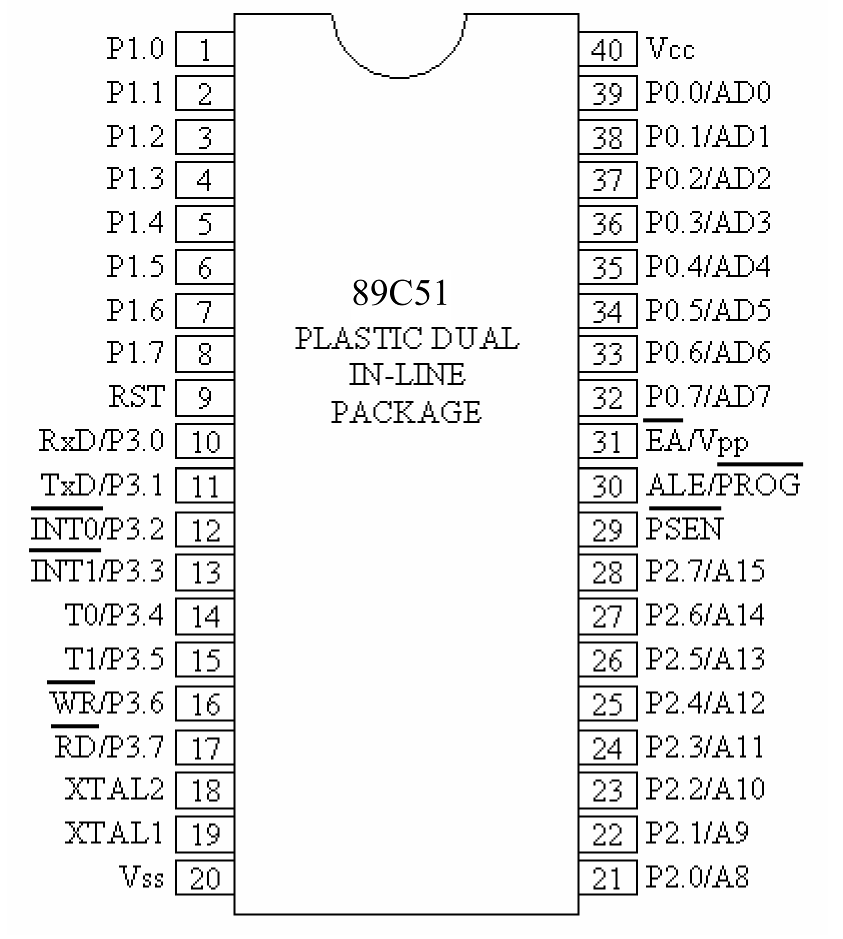
- Compatible with MCS-51™ Products
- 4K Bytes of In-System Reprogrammable Flash Memory Endurance: 1,000 Write/Erase Cycles
- Fully Static Operation: 0 Hz to 24 MHz
- Three-level Program Memory Lock
- 128 x 8-bit Internal RAM
- 32 Programmable I/O Lines
- Two 16-bit Timer/Counters
- Six Interrupt Sources
- Programmable Serial Channel
- Low-power Idle and Power-down Modes

A detailed Block Diagram of the AT89C51 microcontroller is shown in the figure below.



Detailed Block Diagram

The AT89C51 comes in a 40 pin package. The Pin Configuration of the AT89C51 microcontroller is shown in the diagram below.



PDIP Pin Configuration

32 pins are used for the 4 ports - **P0, P1, P2, and P3**
1 pin each for **V_{CC}** and **V_{SS}**
1 pin for the **ALE (**Address Latch Enable)
1 pin for **EA/V_{PP}** • **EA** - External Address • **V_{PP}** - Program Voltage for EPROM based versions of the 8051
1 pin each for **XTAL1** and **XTAL2** - Connections for clock crystal
1 pin for **PSEN** - “Program Store Enable” • Read signal for external program memory
1 pin for **RST -** Reset

10.3 Pre-lab:

1. Review all the above sections of this experiment and **Appendix A, B, and C.**
2. Look for information on the 8051 microcontroller in the library and on the internet.

10.4 Lab Work:

In this experiment, we will implement a simple LED control system on the 89C51 microcontroller. This implementation involved three steps:

1. Software programming 2. Hardware connections 3. Microcontroller programming

Procedure: **Software**

1. Use an editor to write the program shown below. Name your file as ***led_blink.asm***.

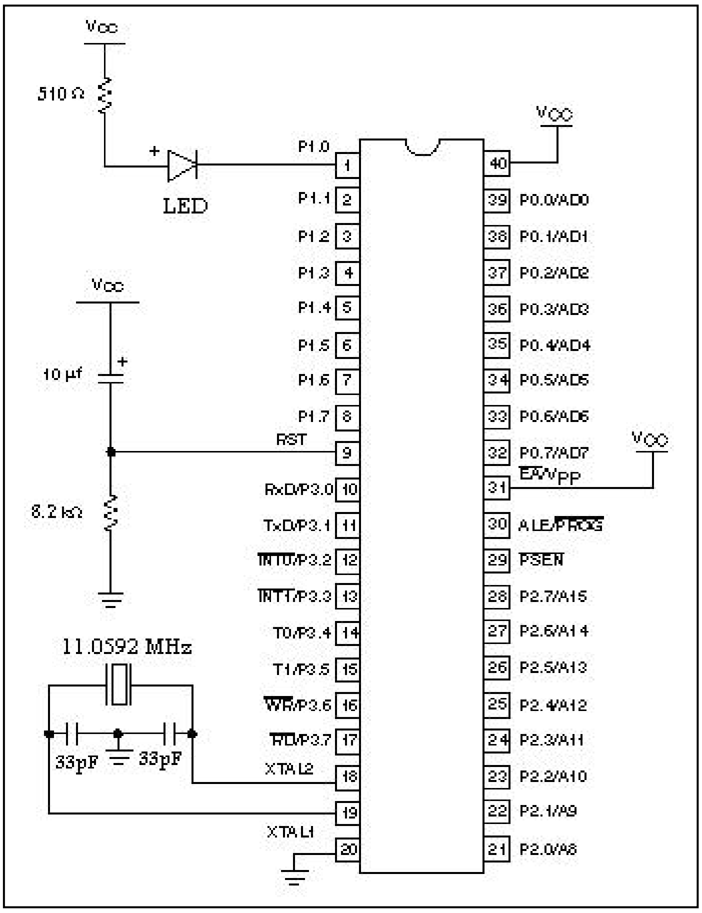
```
; This program blinks LEDs every half second

ORG 00H                        ; After reset, start fetching instructions from 00H

            MOV A, #55H        ; load 0101 0101 in A
AGAIN:
            MOV P1, A          ; move data to Port A

            ACALL DELAY        ; call delay routine

            CPL A              ; invert A to 1010 1010
            SJMP AGAIN         ; repeat blinking

DELAY:                         ; this routine implements a delay 0f 500ms
            MOV R4, #05        ; move 5 in R4
OUTER2: MOV R3, #200           ; move 200 in R3
OUTER1: MOV R2, #0255          ; move 255 in R2
INNER:   DJNZ R2, INNER        ; decrement R2 until 0
            DJNZ R3, OUTER1    ; decrement R3 until 0
            DJNZ R4, OUTER2    ; decrement R4 until 0
            RET

END
```

2. Assemble and link your program using the TASM assembler to produce ***led_blink.exe***.

3. Generate the HEX file ***led_blink.hex*** for your program.

Procedure: **Hardware**

1. Turn OFF the power supply to your board.
2. Obtain the necessary components from your lab instructor and connect the circuit as shown in the schematic below:



3. Connect the rest of the pins of Port 1 to an LED each.
   Note 1: Only P1.0 (Pin 0 of Port 1) is shown connected to an LED.
   Note 2: You may be directly able to connect to LED's on the board provided without using any resistor with the LED's. Check with your lab instructor.

4. Make sure all your connections are properly done and turn ON the power supply.

5. Observe that no LED is blinking.

Procedure: **Microcontroller Programming**

Step 1: Run WinLV
Launch the WinLV program. See **Appendix C** - An Introduction to WinLV.

Note: During start-up, the red light of the programmer will go ON indicating that the software has successfully found the programmer. If the software fails it will display a prompt to enter a demo-mode. If this is the case, double check the cable connected to the parallel port and the power supply of the programmer.

Step 2: Select Device (ATMEL 89C51 microcontroller)
Choose the device that you are using by selecting it from the ***Select Device*** window. To open the ***Select Device*** window – click the left hand mouse button on ***Programmer*** from the top menu bar then click on ***Select Device***. Use the mouse to browse through the Parts Database and select a device from the Programmable Parts section then click on Accept to confirm your choice.

Step 3: Load File into Buffer
Select the file you would like to program into the Microcontroller following these steps:-

- Select the ***File*** menu and then ***Open…***
  Select which file you want to load into the buffer OR type the file name into the ***File Name*** box. Locate the ***led_blink.hex*** file generated from the assembly code.
- If any files that you expecting to see aren't there try using the ***Files of Type*** drop down menu to select which file type is displayed in the main window.
- Use the ***Open as*** drop down menu to select which Format the file will be opened as and loaded into the buffer. Make sure that the value of the field "Open as" is : HEX – Intel" or " HEX - auto recognition ".
- Set the Defaults to pre fill the buffer with 0xFF so that all empty address locations are filled with FF.

Step 4: Place Device in Programmer
Place the device you've selected to program into the ZIF socket on the programmer. Make sure that the device is aligned with the bottom of the ZIF socket.

Step 5: Programme the Device
Follow these final steps to finish programming the device:

- Select ***Operations*** from the ***Programmer*** menu to view the “Operations for Device” dialogue window.
- Click on the ***Programme*** button in the ***Operations*** dialogue window.
- The device in the socket will now be erased, programmed and verified.

Finally, disconnect the IC from the programmer and reconnect it to the circuit. Reset the microcontroller and check if your program is working correctly.

Appendix A – 8051 Instruction Set

The instructions are grouped into 5 groups

- Arithmetic
- Logic
- Data Transfer
- Boolean
- Branching

A complete list of all instructions in each of the above 5 groups is shown from the next page.

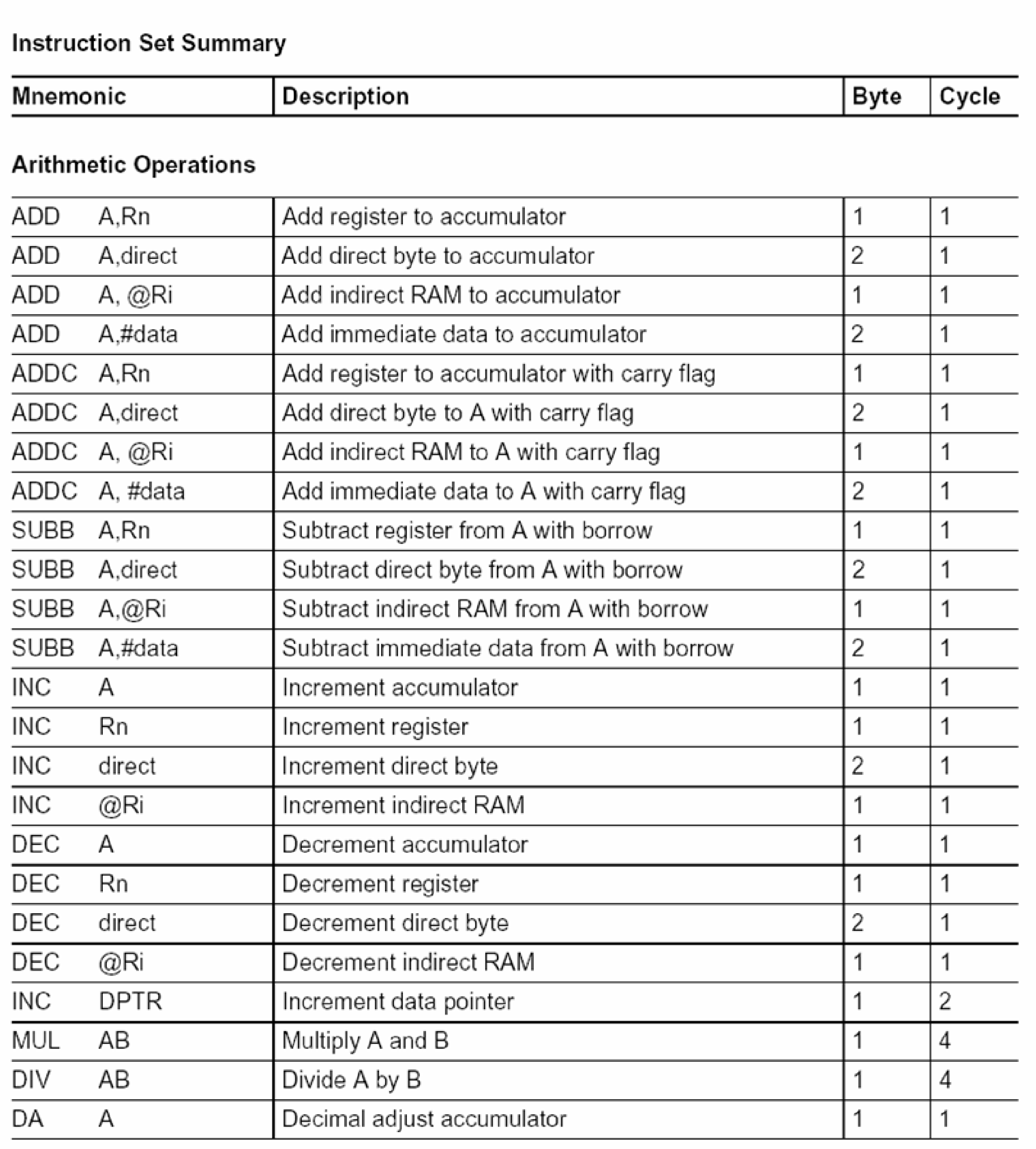
Notes on Data Addressing Modes

Rn - Working register R0-R7
direct - 128 internal RAM locations, any l/O port, control or status register
@Ri - Indirect internal or external RAM location addressed by register R0 or R1
#data - 8-bit constant included in instruction
#data 16 - 16-bit constant included as bytes 2 and 3 of instruction
bit - 128 software flags, any bitaddressable l/O pin, control or status bit
A – Accumulator

Notes on Program Addressing Modes

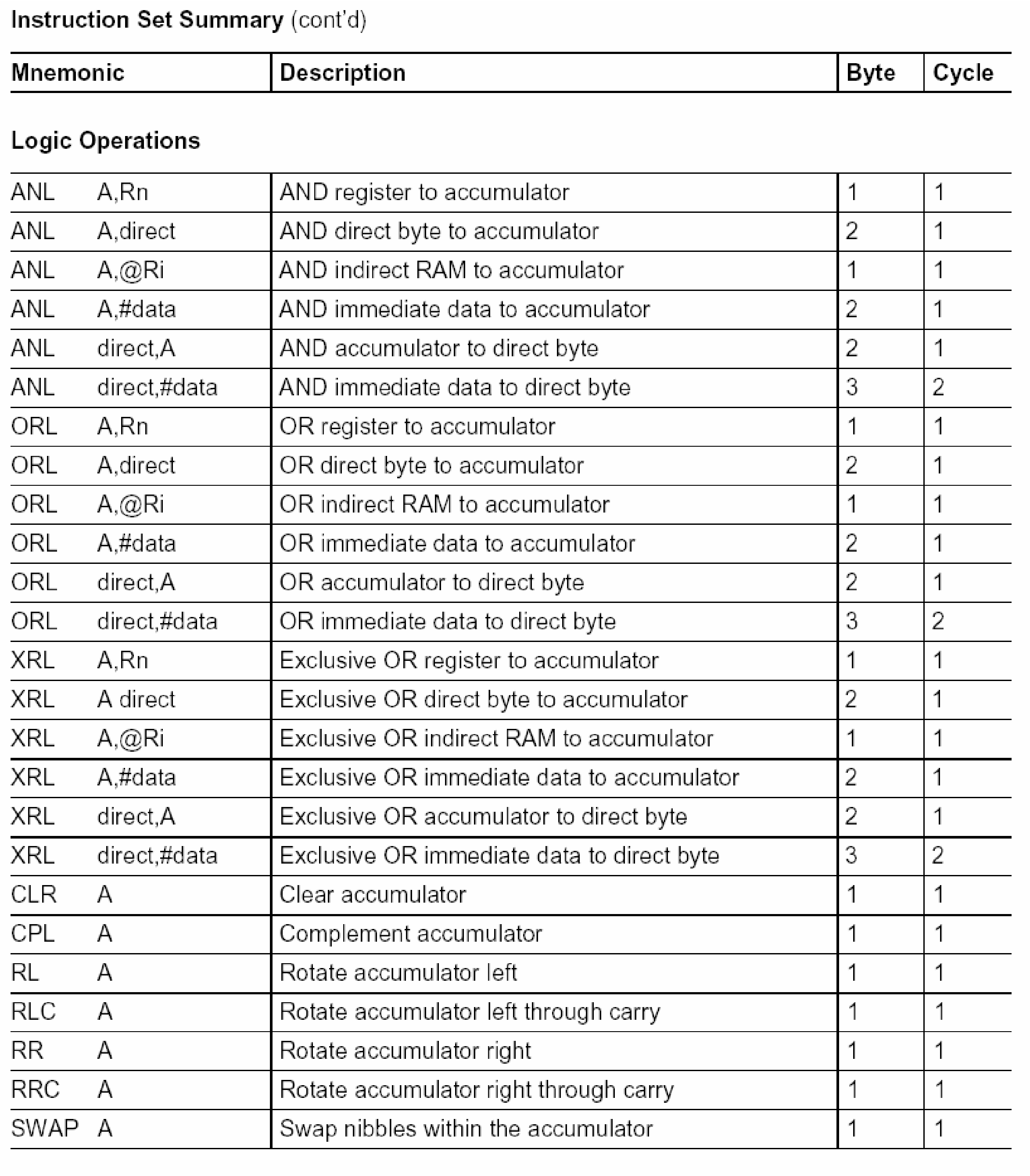
addr16 - Destination address for LCALL and LJMP may be anywhere within the 64-Kbyte program memory address space.
addr11 - Destination address for ACALL and AJMP will be within the same 2-Kbyte page of program memory as the first byte of the following instruction.
rel - SJMP and all conditional jumps include an 8 bit offset byte. Range is + 127/–128 bytes relative to the first byte of the following instruction.

All mnemonics copyrighted: Intel Corporation 1980



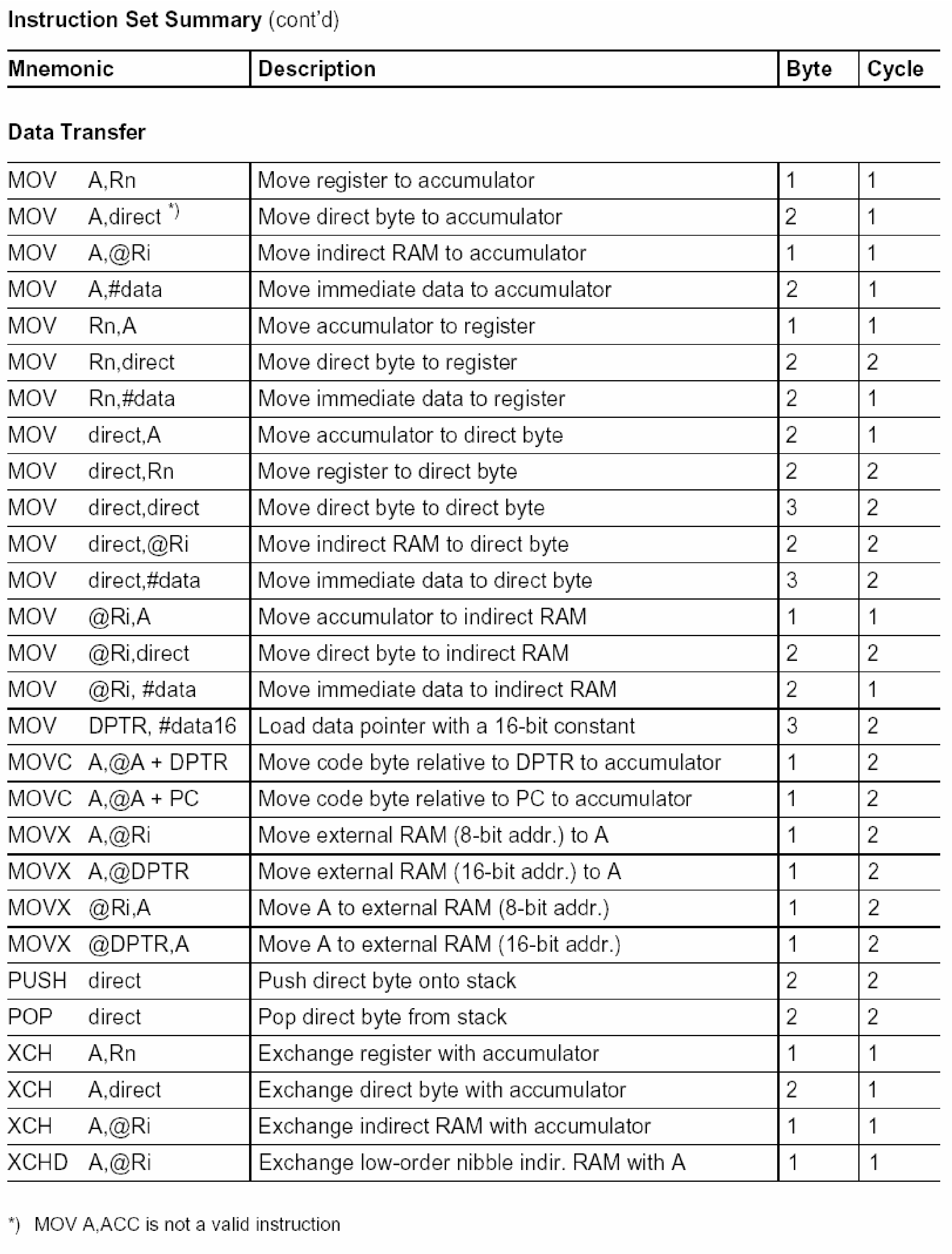
Instruction Set Summary

Mnemonic	Description	Byte	Cycle
Arithmetic Operations			
ADD A,Rn	Add register to accumulator	1	1
ADD A,direct	Add direct byte to accumulator	2	1
ADD A, @Ri	Add indirect RAM to accumulator	1	1
ADD A,#data	Add immediate data to accumulator	2	1
ADDC A,Rn	Add register to accumulator with carry flag	1	1
ADDC A,direct	Add direct byte to A with carry flag	2	1
ADDC A, @Ri	Add indirect RAM to A with carry flag	1	1
ADDC A, #data	Add immediate data to A with carry flag	2	1
SUBB A,Rn	Subtract register from A with borrow	1	1
SUBB A,direct	Subtract direct byte from A with borrow	2	1
SUBB A,@Ri	Subtract indirect RAM from A with borrow	1	1
SUBB A,#data	Subtract immediate data from A with borrow	2	1
INC A	Increment accumulator	1	1
INC Rn	Increment register	1	1
INC direct	Increment direct byte	2	1
INC @Ri	Increment indirect RAM	1	1
DEC A	Decrement accumulator	1	1
DEC Rn	Decrement register	1	1
DEC direct	Decrement direct byte	2	1
DEC @Ri	Decrement indirect RAM	1	1
INC DPTR	Increment data pointer	1	2
MUL AB	Multiply A and B	1	4
DIV AB	Divide A by B	1	4
DA A	Decimal adjust accumulator	1	1



Instruction Set Summary (cont'd)

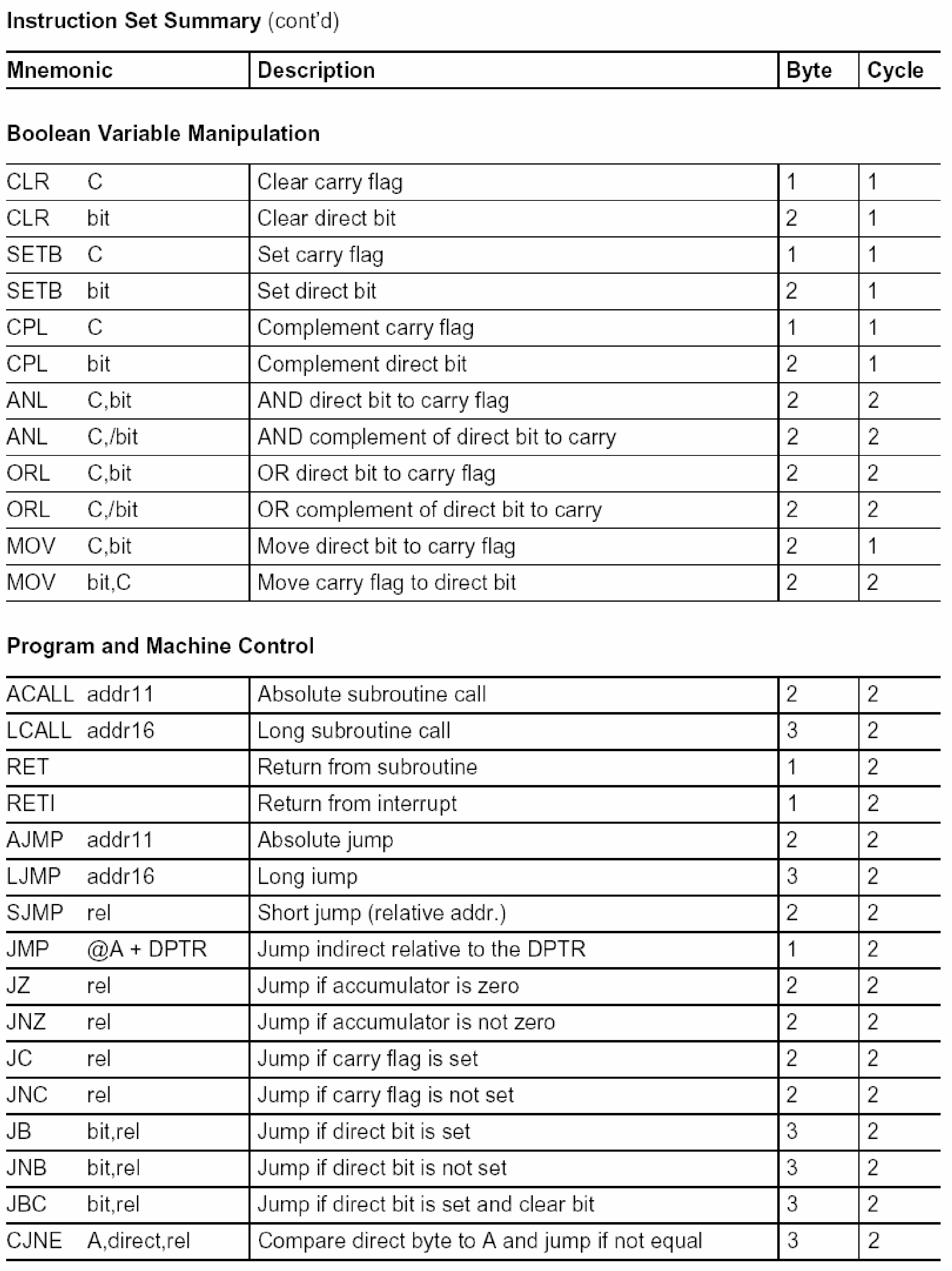
Mnemonic	Description	Byte	Cycle
Logic Operations			
ANL A,Rn	AND register to accumulator	1	1
ANL A,direct	AND direct byte to accumulator	2	1
ANL A,@Ri	AND indirect RAM to accumulator	1	1
ANL A,#data	AND immediate data to accumulator	2	1
ANL direct,A	AND accumulator to direct byte	2	1
ANL direct,#data	AND immediate data to direct byte	3	2
ORL A,Rn	OR register to accumulator	1	1
ORL A,direct	OR direct byte to accumulator	2	1
ORL A,@Ri	OR indirect RAM to accumulator	1	1
ORL A,#data	OR immediate data to accumulator	2	1
ORL direct,A	OR accumulator to direct byte	2	1
ORL direct,#data	OR immediate data to direct byte	3	2
XRL A,Rn	Exclusive OR register to accumulator	1	1
XRL A direct	Exclusive OR direct byte to accumulator	2	1
XRL A,@Ri	Exclusive OR indirect RAM to accumulator	1	1
XRL A,#data	Exclusive OR immediate data to accumulator	2	1
XRL direct,A	Exclusive OR accumulator to direct byte	2	1
XRL direct,#data	Exclusive OR immediate data to direct byte	3	2
CLR A	Clear accumulator	1	1
CPL A	Complement accumulator	1	1
RL A	Rotate accumulator left	1	1
RLC A	Rotate accumulator left through carry	1	1
RR A	Rotate accumulator right	1	1
RRC A	Rotate accumulator right through carry	1	1
SWAP A	Swap nibbles within the accumulator	1	1



Instruction Set Summary (cont'd)

Mnemonic	Description	Byte	Cycle
Data Transfer			
MOV A,Rn	Move register to accumulator	1	1
MOV A,direct *)	Move direct byte to accumulator	2	1
MOV A,@Ri	Move indirect RAM to accumulator	1	1
MOV A,#data	Move immediate data to accumulator	2	1
MOV Rn,A	Move accumulator to register	1	1
MOV Rn,direct	Move direct byte to register	2	2
MOV Rn,#data	Move immediate data to register	2	1
MOV direct,A	Move accumulator to direct byte	2	1
MOV direct,Rn	Move register to direct byte	2	2
MOV direct,direct	Move direct byte to direct byte	3	2
MOV direct,@Ri	Move indirect RAM to direct byte	2	2
MOV direct,#data	Move immediate data to direct byte	3	2
MOV @Ri,A	Move accumulator to indirect RAM	1	1
MOV @Ri,direct	Move direct byte to indirect RAM	2	2
MOV @Ri, #data	Move immediate data to indirect RAM	2	1
MOV DPTR, #data16	Load data pointer with a 16-bit constant	3	2
MOVC A,@A + DPTR	Move code byte relative to DPTR to accumulator	1	2
MOVC A,@A + PC	Move code byte relative to PC to accumulator	1	2
MOVX A,@Ri	Move external RAM (8-bit addr.) to A	1	2
MOVX A,@DPTR	Move external RAM (16-bit addr.) to A	1	2
MOVX @Ri,A	Move A to external RAM (8-bit addr.)	1	2
MOVX @DPTR,A	Move A to external RAM (16-bit addr.)	1	2
PUSH direct	Push direct byte onto stack	2	2
POP direct	Pop direct byte from stack	2	2
XCH A,Rn	Exchange register with accumulator	1	1
XCH A,direct	Exchange direct byte with accumulator	2	1
XCH A,@Ri	Exchange indirect RAM with accumulator	1	1
XCHD A,@Ri	Exchange low-order nibble indir. RAM with A	1	1

*) MOV A,ACC is not a valid instruction



Instruction Set Summary (cont'd)

Mnemonic		Description	Byte	Cycle

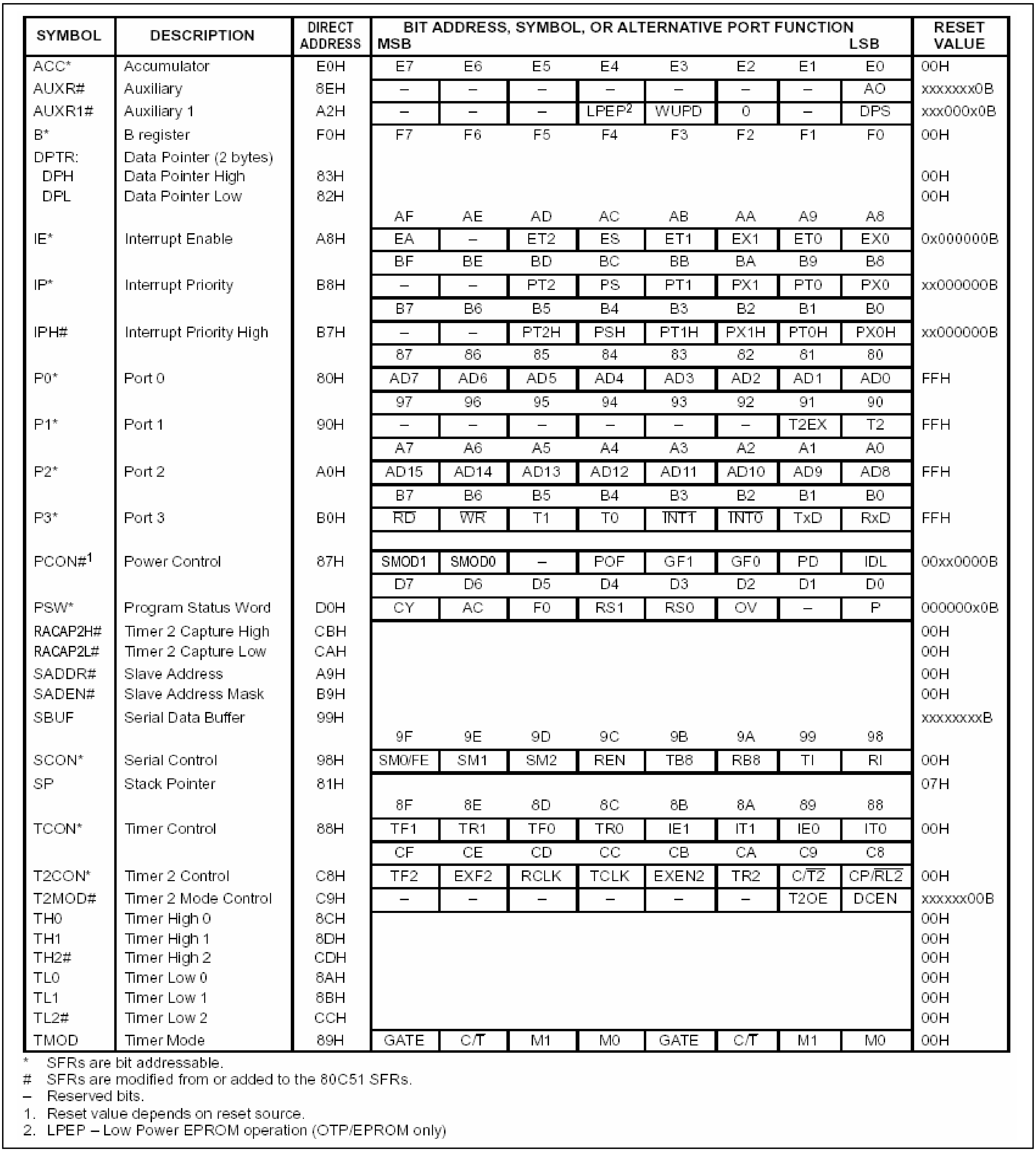
Boolean Variable Manipulation

Mnemonic		Description	Byte	Cycle
CLR	C	Clear carry flag	1	1
CLR	bit	Clear direct bit	2	1
SETB	C	Set carry flag	1	1
SETB	bit	Set direct bit	2	1
CPL	C	Complement carry flag	1	1
CPL	bit	Complement direct bit	2	1
ANL	C,bit	AND direct bit to carry flag	2	2
ANL	C,/bit	AND complement of direct bit to carry	2	2
ORL	C,bit	OR direct bit to carry flag	2	2
ORL	C,/bit	OR complement of direct bit to carry	2	2
MOV	C,bit	Move direct bit to carry flag	2	1
MOV	bit,C	Move carry flag to direct bit	2	2

Program and Machine Control

Mnemonic		Description	Byte	Cycle
ACALL	addr11	Absolute subroutine call	2	2
LCALL	addr16	Long subroutine call	3	2
RET		Return from subroutine	1	2
RETI		Return from interrupt	1	2
AJMP	addr11	Absolute jump	2	2
LJMP	addr16	Long iump	3	2
SJMP	rel	Short jump (relative addr.)	2	2
JMP	@A + DPTR	Jump indirect relative to the DPTR	1	2
JZ	rel	Jump if accumulator is zero	2	2
JNZ	rel	Jump if accumulator is not zero	2	2
JC	rel	Jump if carry flag is set	2	2
JNC	rel	Jump if carry flag is not set	2	2
JB	bit,rel	Jump if direct bit is set	3	2
JNB	bit,rel	Jump if direct bit is not set	3	2
JBC	bit,rel	Jump if direct bit is set and clear bit	3	2
CJNE	A,direct,rel	Compare direct byte to A and jump if not equal	3	2

Appendix B – 8051 Special Function Registers



SYMBOL	DESCRIPTION	DIRECT ADDRESS	BIT ADDRESS, SYMBOL, OR ALTERNATIVE PORT FUNCTION MSB							LSB	RESET VALUE
ACC*	Accumulator	E0H	E7	E6	E5	E4	E3	E2	E1	E0	00H
AUXR#	Auxiliary	8EH	–	–	–	–	–	–	–	AO	xxxxxxx0B
AUXR1#	Auxiliary 1	A2H	–	–	–	LPEP[2]	WUPD	0	–	DPS	xxx000x0B
B*	B register	F0H	F7	F6	F5	F4	F3	F2	F1	F0	00H
DPTR:	Data Pointer (2 bytes)										
DPH	Data Pointer High	83H									00H
DPL	Data Pointer Low	82H									00H
			AF	AE	AD	AC	AB	AA	A9	A8	
IE*	Interrupt Enable	A8H	EA	–	ET2	ES	ET1	EX1	ET0	EX0	0x000000B
			BF	BE	BD	BC	BB	BA	B9	B8	
IP*	Interrupt Priority	B8H	–	–	PT2	PS	PT1	PX1	PT0	PX0	xx000000B
			B7	B6	B5	B4	B3	B2	B1	B0	
IPH#	Interrupt Priority High	B7H	–	–	PT2H	PSH	PT1H	PX1H	PT0H	PX0H	xx000000B
			87	86	85	84	83	82	81	80	
P0*	Port 0	80H	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	FFH
			97	96	95	94	93	92	91	90	
P1*	Port 1	90H	–	–	–	–	–	–	T2EX	T2	FFH
			A7	A6	A5	A4	A3	A2	A1	A0	
P2*	Port 2	A0H	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8	FFH
			B7	B6	B5	B4	B3	B2	B1	B0	
P3*	Port 3	B0H	$\overline{RD}$	$\overline{WR}$	T1	T0	$\overline{INT1}$	$\overline{INT0}$	TxD	RxD	FFH
PCON#[1]	Power Control	87H	SMOD1	SMOD0	–	POF	GF1	GF0	PD	IDL	00xx0000B
			D7	D6	D5	D4	D3	D2	D1	D0	
PSW*	Program Status Word	D0H	CY	AC	F0	RS1	RS0	OV	–	P	000000x0B
RACAP2H#	Timer 2 Capture High	CBH									00H
RACAP2L#	Timer 2 Capture Low	CAH									00H
SADDR#	Slave Address	A9H									00H
SADEN#	Slave Address Mask	B9H									00H
SBUF	Serial Data Buffer	99H									xxxxxxxxB
			9F	9E	9D	9C	9B	9A	99	98	
SCON*	Serial Control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	00H
SP	Stack Pointer	81H									07H
			8F	8E	8D	8C	8B	8A	89	88	
TCON*	Timer Control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00H
			CF	CE	CD	CC	CB	CA	C9	C8	
T2CON*	Timer 2 Control	C8H	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$	00H
T2MOD#	Timer 2 Mode Control	C9H	–	–	–	–	–	–	T2OE	DCEN	xxxxxx00B
TH0	Timer High 0	8CH									00H
TH1	Timer High 1	8DH									00H
TH2#	Timer High 2	CDH									00H
TL0	Timer Low 0	8AH									00H
TL1	Timer Low 1	8BH									00H
TL2#	Timer Low 2	CCH									00H
TMOD	Timer Mode	89H	GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0	00H

* SFRs are bit addressable.

# SFRs are modified from or added to the 80C51 SFRs.

– Reserved bits.

1. Reset value depends on reset source.
2. LPEP – Low Power EPROM operation (OTP/EPROM only)

Appendix C - An Introduction to WinLV

Introduction:

The WinLV is an interface to the functionality of the MicroMaster LV48 produced by ICE Technology. It has a user-friendly GUI (Graphical User Interface) that is optimized to enable quick and easy access to all functionalities. The software contains many features and a very good help system. We will cover here the minimum functionality that is required for this experiment; however the **Help** system offers many tutorials that we found very helpful.

Installation:

The software is spanned on 3 floppy disks. The installation file is located on the first disk. Installation of the WinLV program is as easy as any standard windows installation. Check www.icetech.com for any updates.

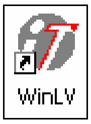
PC connection:

MicroMaster LV48 box hooks to the parallel port of a PC using a standard parallel port cable.

Software Start-up:

ICE TECHNOLOGY
WinLV © ICE Technology 1999
WinLV
Version 1.0.0194
This software is protected as described in Help About

Start Screen

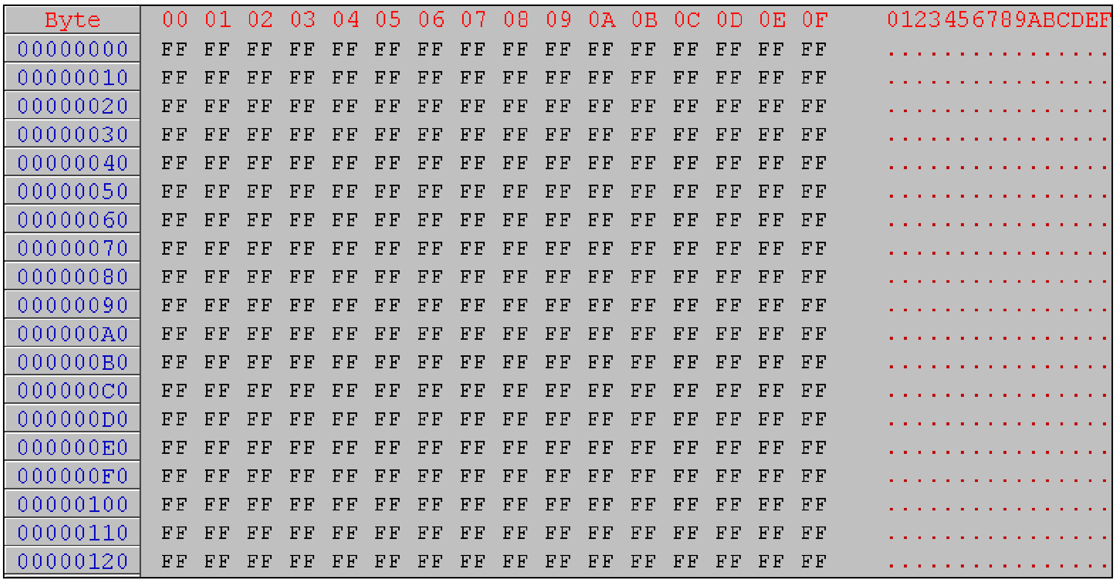


WinLV icon

Before you start the software make sure that the programmer is connected to the PC and its power supply is plugged. The software will recognize the type of the programmer when run, there are other programmer types produced by ICE that uses the same software.

Basic Screen Components:
The basic on-screen components of the WinLV software are as follows-:

1) Programming Buffer Window



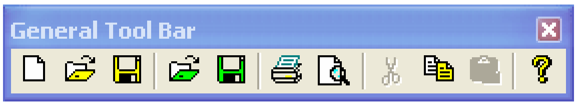
```
  Byte     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
00000000   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000010   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000020   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000030   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000040   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000050   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000060   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000070   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000080   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000090   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000000A0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000000B0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000000C0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000000D0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000000E0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
000000F0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000100   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000110   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
00000120   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
```

Most part of the window is used for displaying the programming buffer. The buffer can be viewed differently by selecting any of the Buffer View options (see Display Modes below). The buffer can also be edited so that data can be changed before or after it has been programmed into a device.

2) WinLV Toolbars & Display Panels

The toolbars & display panels allow you to view current WinLV information and settings as well as select specific functions as an alternative to using the WinLV menus. The Toolbars and Panels are-:

General Toolbar:



The options from left to right are:

1. Opens a new Buffer or Fuse map depending on which device is currently selected.
2. Opens a previously saved file.
3. Saves the currently opened file.
4. Opens a previously saved project.

5. Saves the currently opened project.

6. Print the contents of the buffer/fuse map.

7. Preview the buffer contents to see how it will look when printed.

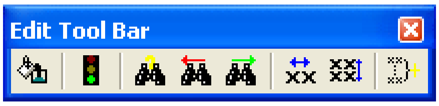
8. Cut the highlighted portion of the buffer (move to the clipboard).

9. Copy the highlighted portion (move to the clipboard).

10. Paste the current contents of the clipboard.

11. WinLV Help

Edit Toolbar



The options from left to right are:

1. Fill buffer with user-defined value.

2. Go to specific address/value in the buffer.

3. Search the currently open buffer to find a specific value at an address.

4. Search the currently open buffer to find the previous occurrence of specific find value.

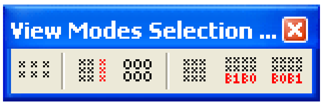
5. Search the currently open buffer to find the next occurrence of specific find value.

6. Swap bytes in the currently open buffer.

7. Swap words in the currently open buffer.

8. Open the checksum calculator.

Display Mode Panel

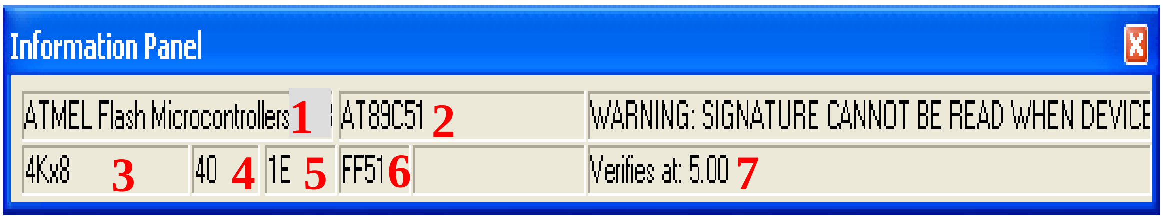


It displays modes operative when viewing programming address buffer.

The options from left to right are:

1. View the currently open buffer in Nibble Mode.
2. View the currently open buffer in Byte Mode.
3. View the currently open buffer in Octal Mode.
4. View the currently open buffer in 12 Bit Mode.
5. View the currently open buffer in Word Mode (byte 1:byte 0).
6. View the currently open buffer in Word Mode (byte 0:byte 1).

Information Panel

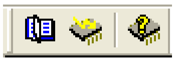


ATMEL Flash Microcontrollers | AT89C51 | WARNING: SIGNATURE CANNOT BE READ WHEN DEVICE

4Kx8 | 40 | 1E | FF51 | Verifies at: 5.00

The displayed information is:

1. Device manufacturers name
2. Device part number
3. Device memory size
4. Number of pins
5. Manufacturers ID code
6. Device ID code
7. Verification information

The Programmer toolbar 

The options from left to right are:

1. Select Device

2. Device Operations

3. Match 74/4000 device

3) WinLV Menus

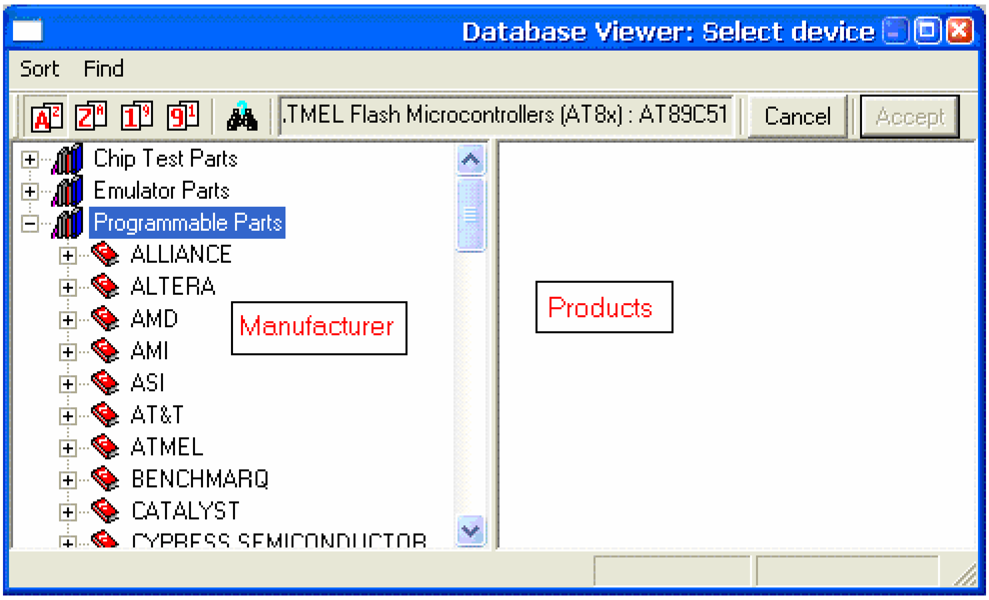
The menu selection bar along the top of the screen allows you to view the WinLV menus and then select specific menu items. Use the mouse or keyboard to select the menus and menu commands.

Device Selection:

Before you can program any device you must select it and tell WinLV that you want to work with it. The process is easy:

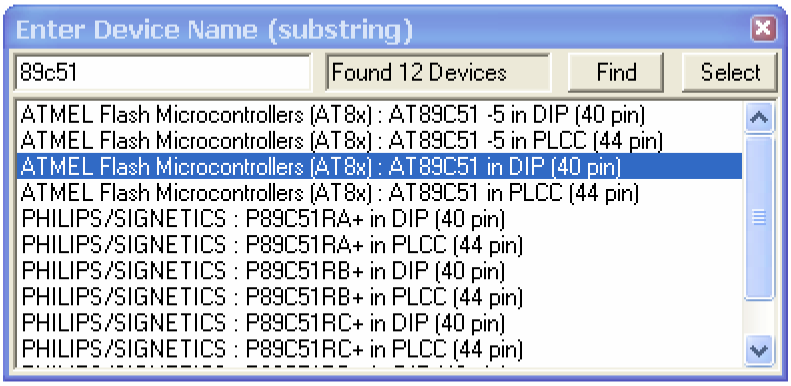
Go to the (Programmer → Select Device) option or press on the programmer toolbar.

The following window will appear:



TMEL Flash Microcontrollers (AT8x) : AT89C51

To select a device you can either use the left pane to find the manufacturer or use the searching tool 🔍 to find the device by part number as shown in the figure below:



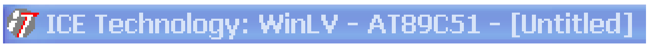
Enter Device Name (substring)

89c51

Found 12 Devices

ATMEL Flash Microcontrollers (AT8x) : AT89C51 -5 in DIP (40 pin)
ATMEL Flash Microcontrollers (AT8x) : AT89C51 -5 in PLCC (44 pin)
ATMEL Flash Microcontrollers (AT8x) : AT89C51 in DIP (40 pin)
ATMEL Flash Microcontrollers (AT8x) : AT89C51 in PLCC (44 pin)
PHILIPS/SIGNETICS : P89C51RA+ in DIP (40 pin)
PHILIPS/SIGNETICS : P89C51RA+ in PLCC (44 pin)
PHILIPS/SIGNETICS : P89C51RB+ in DIP (40 pin)
PHILIPS/SIGNETICS : P89C51RB+ in PLCC (44 pin)
PHILIPS/SIGNETICS : P89C51RC+ in DIP (40 pin)
PHILIPS/SIGNETICS : P89C51RC+ in PLCC (44 pin)

Once you find your device click "**Accept**" to go back to the previous "Database Viewer", then "**Accept**" again and you will go back to the main window. You will notice that your device name is displayed in the title of the window as shown below:

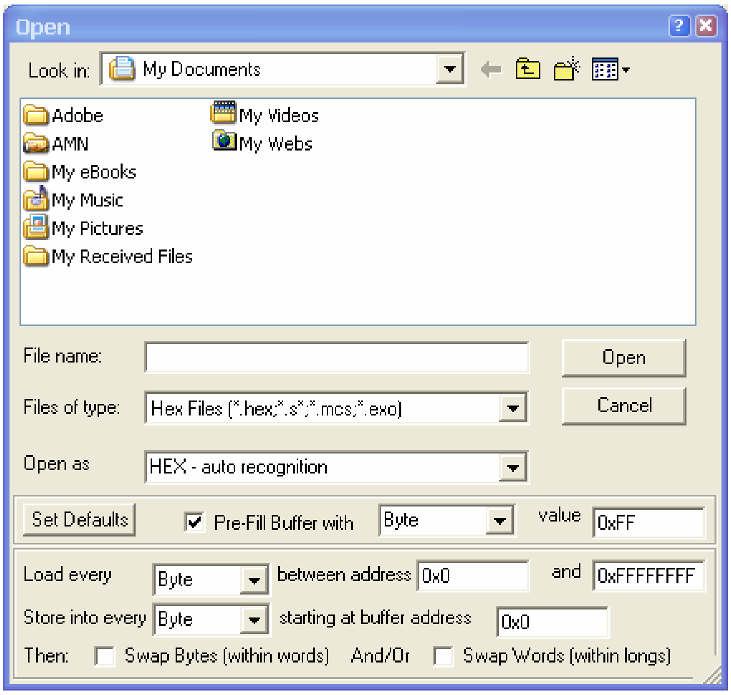


ICE Technology: WinLV - AT89C51 - [Untitled]

Loading Data:

Select the file you would like to program into the selected EPROM or Micro following these steps:

- Select the File menu and then Open… or press



Look in: My Documents

File name:

Files of type: Hex Files (*.hex;*.s*;*.mcs;*.exo)

Open as HEX - auto recognition

☑ Pre-Fill Buffer with Byte value 0xFF

Load every Byte between address 0x0 and 0xFFFFFFFF

Store into every Byte starting at buffer address 0x0

Then: □ Swap Bytes (within words) And/Or □ Swap Words (within longs)

- Select which file you want to load into the buffer. OR type the file name into the File Name box

- If any files that you expecting to see aren't there try using the Files of Type drop down menu to select which file type is displayed in the main window

- Use the Open as drop down menu to select which format the file will be opened as and loaded into the buffer.

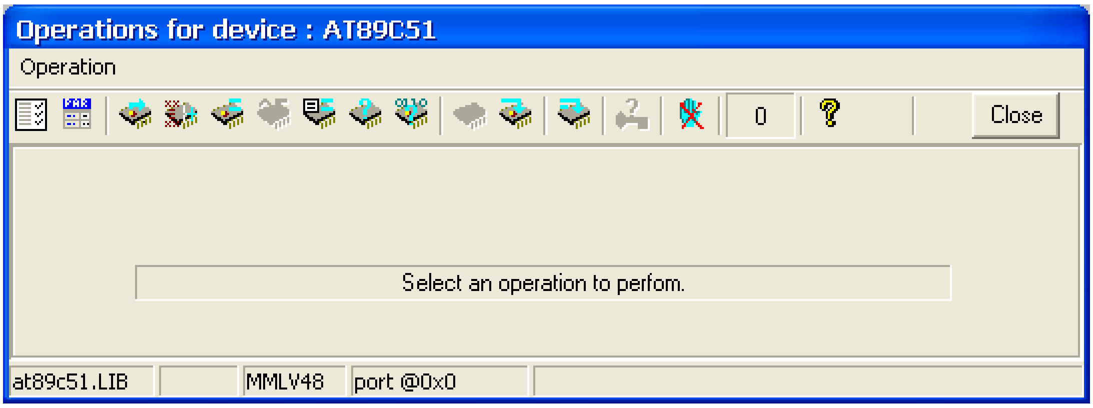
- Set the Defaults to pre-fill the buffer with 0xFF so that all empty address locations are filled with FF.

- Remember - If you want to specify more advanced file open settings this can be set in the lower half of the Open dialogue window.

- Remember - If the buffer size is not large enough for the file selected then it will truncate the file before loading. To increase the buffer size select New Buffer from the File menu and use the slider bar to increase the buffer size and then re-load the file.

Programming the Device

Once the data is in the buffer, you can go ahead and program it on the device. To do that go to (programmer→ Operations) or click . You will get the following window:



Select an operation to perfom.

You can do a lot of operations here, they include:

Read the existing data in the device.

Device checksum

Verify

Blank check

Erase, Program, Verify

The latter option is most commonly used. After clicking this option the contents of the buffer will be programmed on the device and it is ready to be connected to the circuit.