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.

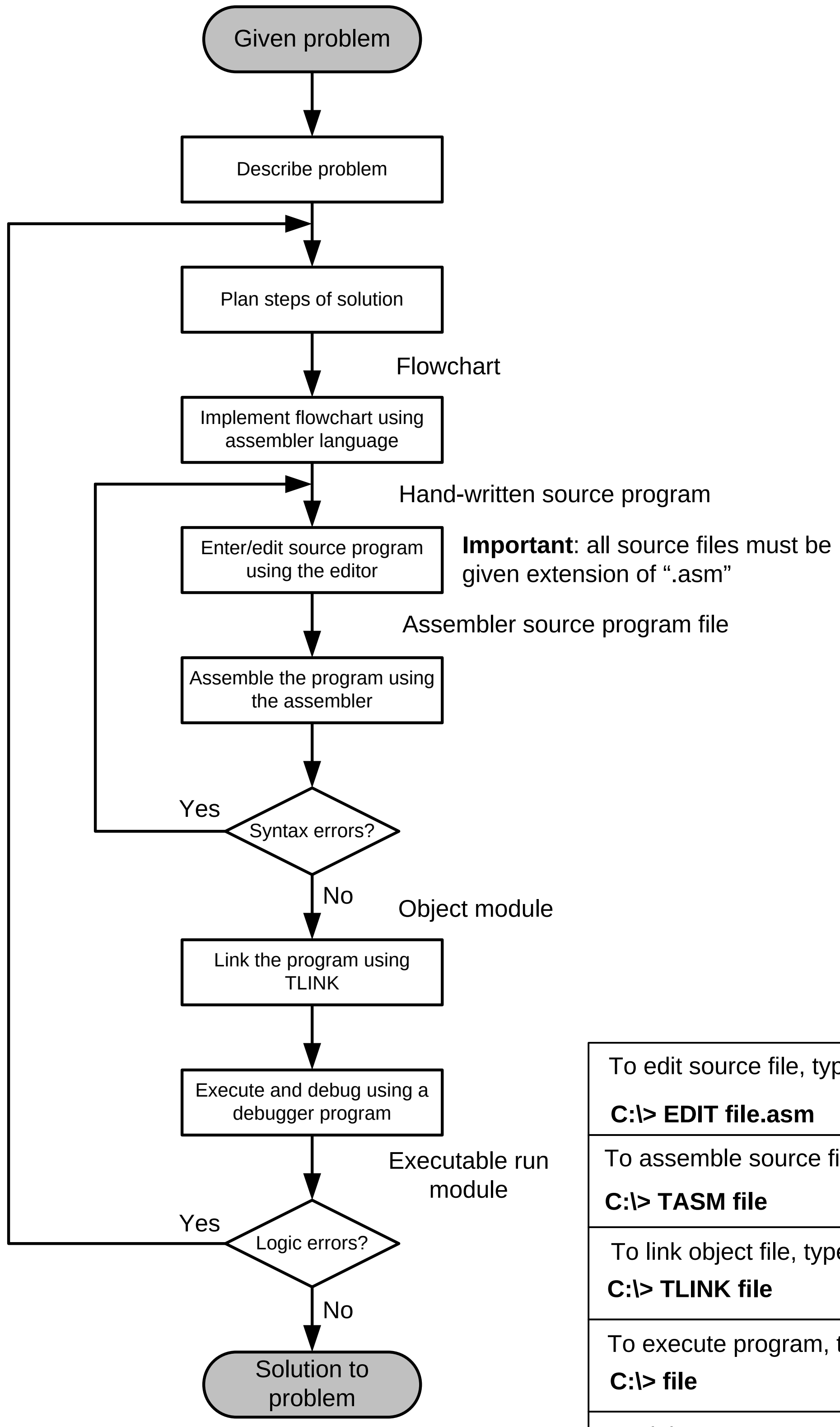
Program Template

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

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’.