Laboratory Manual

For

ADVANCED MICROPROCESSOR ARCHITECTURE

(IT 506)

B.Tech. (IT)

SEM V

July 2016
Faculty of Technology
Dharmsinh Desai University
Nadiad.
www.ddu.ac.in

Table of Contents

LABWORK BEYOND CURRICULA

1. **Implement different String Operations.**
2. **Study of graphics using the interfacing of C module with assembly module and calling C.**

Sample Experiment

1 AIM: Write a program to multiply two 16bits numbers.

2 TOOLS: Turbo Assembler

3 STANDARD PROCEDURES:

3.1 Analyzing the Problem:
Create a file with file_name.asm. Define the variables in data segment and then write a code in code segment.

3.2 Designing the Solution:

```
Define a data segment
Data_here

{Define the variables}

Data_here ends

Code_here
Assume  cs: Code_here  ,ds: Data_here
{Wrtite Instructions}

Code_here ends
```

3.3 Implementing the Solution
Create a new file in editor with name multiplication.asm. Define a data segment with three variables, one with multiplicand, multiplier and product. Define a code segment and write ASSUME statement to tell the assembler that logical segment with name **Code_here** is a code segment and logical segment with name **Data_here** is a data segment. Now, initialize data segment base register. Then write instructions to load multiplier and multiplicand. Multiply the inputs using MUL instruction and store the result to product variable

3.3.1 Writing Source Code

Multiplication.asm

```
Data_here  segment
Multiplicand  dw  1234h;
Multiplier      dw  5678h;
Product          dw  2 dup(0);
Data_here  Ends

Code_here  segment
 Assume  cs: Code_here  ,ds: Data_here
```

```
Start: mov ax,data;
       mov  ds,ax;
       mov ax, Multiplicand ;
       mul  Multiplier ;
       mov Product,ax;
       mov Product+2,dx;
       int 3;
Code_here  Ends
       End Start
```

3.3.2 Compilation /Running and Debugging the Solution

Step: 1

Compile File Using Tasm

C:/tasm>tasm Multiplication.asm

Step: 2

Link File Using Linker

C:/tasm>tlink Multiplication.obj

Step: 3

Generate Executable file using Tasm

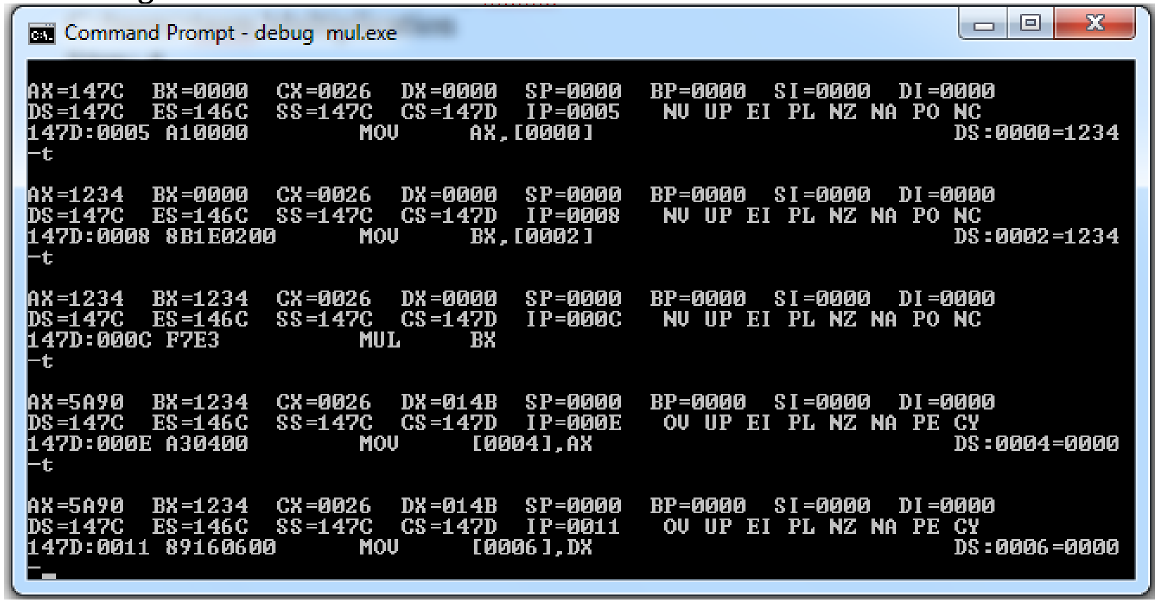
C:/tasm>tasm Multiplication

Step: 4

Execute File Using Debug

```
C:/tasm>debug  Multiplication.exe
-u
-g=address
```

3.4 Testing the Solution



```
AX=147C  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=147C  ES=146C  SS=147C  CS=147D  IP=0005   NV UP EI PL NZ NA PO NC
147D:0005 A10000        MOV     AX,[0000]                          DS:0000=1234
-t

AX=1234  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=147C  ES=146C  SS=147C  CS=147D  IP=0008   NV UP EI PL NZ NA PO NC
147D:0008 8B1E0200      MOV     BX,[0002]                          DS:0002=1234
-t

AX=1234  BX=1234  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=147C  ES=146C  SS=147C  CS=147D  IP=000C   NV UP EI PL NZ NA PO NC
147D:000C F7E3          MUL     BX
-t

AX=5A90  BX=1234  CX=0026  DX=014B  SP=0000  BP=0000  SI=0000  DI=0000
DS=147C  ES=146C  SS=147C  CS=147D  IP=000E   OV UP EI PL NZ NA PE CY
147D:000E A30400        MOV     [0004],AX                          DS:0004=0000
-t

AX=5A90  BX=1234  CX=0026  DX=014B  SP=0000  BP=0000  SI=0000  DI=0000
DS=147C  ES=146C  SS=147C  CS=147D  IP=0011   OV UP EI PL NZ NA PE CY
147D:0011 89160600      MOV     [0006],DX                          DS:0006=0000
-_
```

4 Conclusions

We can write a code for multiply two 16bits numbers using turbo assembler.

EXPERIMENT-1

Aim: Study of DOS Debug Commands

Tools / Apparatus: Turbo Assembler

Procedure:

The DOS "Debug" program is an example of a simple debugger that comes with MS-DOS. Below are summarized the basic DOS-Debugger commands.

COMMAND	SYNTAX
Assemble	A[address]
Compare	C [range] [address]
Dump	D[range]
Enter	E address [list]
Fill	F address range [list]
Go	G[=address] [addresses]
Hex Arithmetic	H [value1][value2]
Input	I port
Load	L[address][drive][firstsector[number]
Move	[range] [address]
Name	N[pathname][arglist]
Proceed	P[=address][number]
Quit	Q
Trace	T[=address][value]
Register	R[register]
Search	[address range] [list]
Unassemble	U[range]

<u>EXPERIMENT-2</u>

Aim: Study of Turbo Assembler

Tools / Apparatus: Turbo Assembler

Procedure:
In this practical first we discuss about segment directives which are used to write program of 8086

Segment directives:
8086 uses the directive SEGMENT(segment) to identify the beginning of a segment. The name of the segment should precede the keyword SEGMENT (segment). A segment thus declared will contains the code or the data depending on whether the declared segment is code segment or data segment. The words code or data or any other user defined names can be used to declare a particular segment. Its definition however will be Clear in the code segment when you assign a segment name to the segment register. The name of the segment should be a unique without any blank spaces and may be upto 31 Characters long. Keywords cannot be used as name of the segment.

ENDS directive:
8086 make use of ENDS (ends) directive to mean the end of a segment. The keyword ENDS(ends) must be followed with the name of the segment to end.

END directive:
The end END(end) directive tell the assembler to stop reading and assembling the program after the end directive. The statements after the end directive will be ignored by the assembler. It should be used as the last statement in a program.

ASSUME directive:
Since there can be 64 total segments in the .EXE format and of which at least four segments one of each type can be active at any time, the ASSUME (assume) directive tells the assembler which segments are active. The format of the ASSUME (assume) directive is :
ASSUME CS:cs_name, DS: ds_name, SS: ss_name, ES: es_name

OFFSET directive:
The offset directive will be used where the offset from starting of the segment is required in the program. The example of OFFSET is:
MOV SI, OFFSET string1; copies the offset of string1 in SI register

PTR directive:
It is called pointer directive and very useful while referring to the operands stored in memory.
MOV AL, BYTE PTR[SI]

Data declaration directives

DB - Defined Byte. DB declares a variable of type byte and reserves one location in memory for the variable of type byte.
Example
num1 DB 15h, Char1 db 'A'
numbers db 100 dup(0); Reserve an array of 50 words of memory and initialize All bytes with 00. Array is named as numbers.

DD- Defined double Word, DW declares a variable of type word and reserves two locations in memory for the variable of type word.
Examples:
num1 DW 1234h
ARR1 DW 1A3Bh, 3A4Bh, 5A6CH ; this declares an array of 3 words and initialized with Specified values.
ARR2 DW 50 DUP('0'); Reserve an array of 50 words of memory and initialize All words with 0000. Array is named as ARR2.
DQ-Define Quadword
This directive is used to define a variable of type quadword or to reserve storage location of type quadword in memory. For example
Quad_word DQ 1234123412341234H
DT-Define Ten Bytes
This directive is used to define a variable which is 10 bytes in length or to reserve 10 bytes of storage in the memory. For example
Ten_bytes DT 11223344556677889900

DUP directive: DUP directive is used to duplicate the basic data definition 'n' number of times; or saying it the other way is that DUP is used to declare array of Size n.
ARRY DB 10 dup (0), declares array of 10 byte.

Now we will see how to write program for multiplication of two 16-bits numbers using turbo assembler

Step: 1 Create a file with name multiplication.asm
Step: 2 Define a logical data segment
Step: 3 Define a logical code segment
Step: 4 Compile File Using Tasm
C:/tasm>tasm multiplication.asm
Step: 5 Link File Using Linker
C:/tasm>tlink multiplication.obj
Step: 6 Generate Executable file using Tasm
C:/tasm>tasm multiplication
Step: 7 Execute File Using Debug
C:/tasm>debug multiplication.exe
-u
-g=address

EXPERIMENT-3

Aim: Study of string related instructions

Tools / Apparatus: Turbo Assembler

Procedure:
To move a string from one location to another location using string related instructions and REP prefix byte. In this practical we will study about string instructions MOVS/MOVSB/MOVSW

Program Steps:

```
initialize source pointer, si
    initialize destination pointer, di
    initialze counter, cx
    repeat
        copy byte from source to
        destination
        increment source pointer
        increment destination pointer
        decrement counter
    until counter = 0
```

Do the following Exercise.

Exercise1: Write a program to move a string in same segment.
Exercise2: Write a program to move a string from one segment to another segment.

EXPERIMENT-4

Aim: To study multi module program within divide 32bits number by 16bits numbers.

Tools / Apparatus: Turbo Assembler

Procedure:

On the 8086 CPU the DIV instruction takes as input a 32-bit numerator in DX::AX and divides it by a denominator given as the operand of the instruction (a register or a memory address). After the division, AX holds the quotient, while DX contains the remainder of the division. More formal:

```
DIV   Unsigned divide
Syntax:  DIV    op8 (8-bit register or memory)
         DIV       op16 (16-bit register or memory)
Action:  If operand is op8, unsigned AL = AX / op8 and AH = AX % op8
         If operand is op16, unsigned AX = DX::AX / op16 and DX = DX::AX % op16
```

Program Steps:

1. Create two files one with name main.asm and second with name divide_proc.asm
2. Initialize the data segment in both the files.
3. Write code to call a divide procedure in main.asm.
4. Write a procedure divide in divide_proc.asm.
5. Load the higher 32-bit number to be divided in AX.
6. Load the 16-bit number in DX register.
7. Take the division in BX register.
8. Perform the unsigned division.
9. Store the result in variables.
10. End.

EXPERIMENT-5

Aim: To Study of the response of Type-0 interrupt.

Tools / Apparatus: Turbo Assembler

Procedure: Write an assembly language program of dividing two numbers. If the result of the division is too large to fit in the quotient register then the 8086 will do a type 0 interrupt immediately after the divide instruction finishes.

Here we write two programs one is main line program which contain div instruction and second program is interrupt service routine which handle the type 0 interrupt.

Steps for program:
Mainline program:

```
Initialization List
   Repeat
   Get Input Value
    Divide by scale factor
    If result valid Then
        Store result as scaled value
   Else store zero
Until all values scaled
```

Interrupt Service Procedure:

```
Save Registers
Set Error Flag
Restore registers
Return to mainline
```

EXPERIMENT-6

Aim: To study the interfacing of C module with assembly module and calling C library functions from assembly module.

Tools / Apparatus: Turbo Assembler

Procedure: Write a C program to convert Celsius to Fahrenheit where the functions "C2F" and "Show" are assembly language functions.

It is often necessary to include short assembly routines in a compiled high-level program to take advantage of the speed of machine language.
All high-level languages have specific calling conventions which allow one language to communicate to the other; i.e., to send variables, values, etc. The assembly-language program that is written in conjunction with the high-level language must also reflect these conventions if the two are to be successfully integrated. Usually high-level languages pass parameters to subroutines by utilizing the stack. This is also the case for C.

Steps for program:

1. Create the C module using the editor. Do not forget any required extern directives.
2. Compile the module and repeat the edit-compile cycle until the compile is successful.
3. Create the assembly language module with editor. Do not forget to include any required public and extrn directives. Save the module in a file with a .asm extension.
4. Press the Alt key and the spacebar to get to the menu containing turbo assembler. Move the highlighted box to the turbo assembler line and press the enter key.
5. Repeat the edit-assemble loop until the assemble is successful.
6. Go to the project menu and select open project. When the dialog box appears, type in some appropriate name for your project and give it a .prj extension.
7. Use the add item line the project menu to add the name(s) of the C source(.C) files and the names of your assembly language object (.obj) files to the project file. Press the Esc key to get back to the project window.
8. Go to the options menu and select linker. In this menu go to the case sensitive link and press the enter key to turn it off. TASM produces uppercase for all names, and this toggle will prevent link errors caused by uppercase/lowercase disagreements.
9. Go the compile menu, select build all, and press the enter key. This tells the IDE tools to do a "make" on the files specified in the project list. Make checks the times and dates on the .obj files and the associated source files. If the times are different, the source modules are automatically recompiled. The resulting object files are linked with the object files from .asm modules and object modules from libraries to produce the final .exe file.
10. Go to run menu and press the R key to run the program.
11. Load the .exe file in debugger find your program code.

EXPERIMENT-7

Aim: Study of DOS and BIOS function calls

Tools / Apparatus: Turbo Assembler

Procedure: In this experiment will study about DOS functions call and write following programs using these functions.
Program1: Write a program to take one character from key board and echo on screen.

Program2: Write a program to take one character from key board and convert into lowercase.

Program3: Write a program to convert uppercase string to lowercase.

Program4: Write a program to check password given by user with original password stored in data segment.

Using following DOS function call we can write above programs.

(1) AH = 01h/INT 21h -read character from standard input, with echo.
Return: AL = character read.

(2) AH = 02h/INT 21h -write character to standard output.
Entry: DL = character to write
Return: AL = last character output

(3) AH = 09h/INT 21h -write string to standard output
Entry: DS:DX -> - The string is terminated with '$'
Return: AL = 24h

(4) AH = 0Ah /INT 21h -buffered input
Entry: DS:DX -> buffer (reads from standard input)
Return: buffer filled with user input

EXPERIMENT-8

Aim: Study of implementation of Recursion in assembly language.

Tools / Apparatus: Turbo Assembler

Procedure: Implement a program to calculate factorial of a given 8-bit number using recursion.

Program Steps:

1. Start.
2. Initialize data segment.
3. Get the number N.
4. If N=1 then factorial=1 and return.
5. If N!=1 then decrement N Call facto.
6. Multiply (N-1)! * Previous N
7. Return.
8. End.

EXPERIMENT-9

Aim: Study of various methods of passing parameters to a procedure.

Tools / Apparatus: Turbo Assembler

Procedure: -

Often when we call a procedure, we want to make some data values or addresses available to the procedure. Likewise, we often want a procedure to make some processed data values or addresses available to the main program. There are four major ways of passing parameters to and from a procedure are:

a. Through registers
b. General memory
c. Using Pointers
d. Using Stack

Here we write a program of passing parameters to procedure by taking an example of BCD to Binary conversion.

(1)Through registers:
The parameters are passed to a procedure through processor registers.

(2) General memory:
The offset of parameters is directly access through variable name.

(3) Using Pointers:
The pointer approach is more versatile because you can pass the procedure pointers to data anywhere in memory. You can pass pointers to individual values or pointers to arrays or strings.

(4) Using Stack:
To pass parameters to a procedure using the stack, we push the parameters on the stack somewhere in the mainline program before we call the procedure. Instructions in the procedure then read the parameters from the stack as needed

EXPERIMENT-10

Aim: Study of implementation of TSR.

Tools / Apparatus: Turbo Assembler

Procedure: - Write a passive TSR program which will be activated whenever key ‘p’ is pressed and will display a message on screen “P is pressed”.

TSR: In computers, a terminate and stay resident program (commonly referred to by the initialism TSR) is a computer program that uses a system call in DOS operating systems to return control of the computer to the operating system, as though the program has quit, but stays resident in computer memory so it can be reactivated by a hardware or software interrupt.[1] This technique partially overcame DOS operating systems' limitation of executing only one program, or task, at a time. TSR is unique to DOS and not used in Windows.

Normally in DOS operating systems, only one program can run at any given time. To stop running, it gives control back to the DOS shell program, COMMAND.COM, using the system call INT 21h/4Ch.The memory and system resources that were used by the program are then marked as unused. This in effect makes it impossible to restart parts of it again without reloading it from scratch. However, if a program ends with the system call INT 27h or INT 21h/31h, the operating system does not reuse a certain specified part of the program's memory.

The original call, INT 27h, is called "terminate but stay resident", hence the name "TSR". Using this call, a program can make up to 64 KB of its memory resident. MS-DOS version 2.0 introduced an improved call, INT 21h/function 31h ('Keep Process'), which removed this limitation and let the program return an exit code. Before making this call, the program can install one or several interrupt handlers pointing into itself, so that it can be called again. Installing a hardware interrupt vector allows such a program to react to hardware events. Installing a software interrupt vector allows it to be called by the currently running program. Installing a timer interrupt handler allows a TSR to run periodically (see ISA and programmable interval timer, especially the section "IBM PC compatible").

The typical method of utilizing an interrupt vector involves reading its present value (the address), storing it within the memory space of the TSR, and installing a pointer to its own code. The stored address is called before or after the TSR has received the interrupt and has finished its processing, in effect forming a singly linked list of interrupt handlers, also called interrupt service routines, or ISRs. This procedure of installing ISRs is called chaining or hooking an interrupt or an interrupt vector.

By chaining the interrupt vectors TSR programs could take complete control of the computer. A TSR could have one of two behaviors:

Take complete control of an interrupt by not calling other TSRs that had previously altered the same interrupt vector. Cascade with other TSRs by calling the old interrupt vector. This could be done before or after they executed their actual code. This way TSRs could form a chain of programs where each one calls the next one.

The "terminate and stay resident" method was used by most DOS viruses which could either take control of the PC or stay in the background. Viruses would react to disk I/O or execution events by infecting executable (.EXE or .COM) files when they were run and data files when they were opened.