Lab: 6
Familiarization with DOS DEBUG Program and Assembly Language Programming

Familiarization with DOS DEBUG program

A Debugger is a program which displays the contents of memory quickly and easily, showing registers and variables as they change. You can step through a program one statement at a time, making it easier to find logical errors. Debugger is used to test new programming ideas in assembly language programming. It takes supreme self-confidence to write an assembly language program and run it directly from DOS the first time! If one should forget to match Pushes & Pops, for example, a return from a subroutine will branch to an unexpected location. Any call or jump to an allocation outside your program will almost surely cause DOS to crash. For this reason, you would be wise to run newly, written programs in debug first, watch the stack pointer very closely as you step through the program, and note any unusual changes to CS and IP registers. This experiment is designed to cover the DEBUG.COM utility program of DOS that helps you to test and debug assembly language programs.

Debugging Functions:

Some of the most rudimentary functions that a 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.
- 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.

DEBUG is called an assembly level debugger because it displays only assembly mnemonics and machine instructions. Even if you use it to debug a compiled C program, you will simply, see a disassembly of the programs machine instructions. You may load and run it by typing the command DEBUG from the DOS prompt.

In order to trace or execute a machine language program, you must type the name of the program being debugged on the same command line. For example to debug program TEST.EXE type DEBUG TEST.EXE

DEBUG command summary:

DEBUG commands can be classified into four categories as program creation and debugging, memory manipulation, miscellaneous, and input-output. When DEBUG is first loaded, the following defaults are in effect:

- All segment registers are set to the bottom of free memory, just above the debug program.
- IP is set to 0100H.
- DEBUG reserves 256 bytes of stack space at the end of the current segment.
- All of available memory is allocated (reserved).
- BX:CX arc set to the length of the current program or file.
- The flags are set to the following values:

NV (No Overflow)	OV (Over flow)
UP (Direction UP)	DN (Direction Down)
El (Interrupts Enabled)	DI (Interrupts Disabled)
PL (Sign Positive)	NG (Sign Negative)
NZ (No Zero)	ZR (Zero)
NA (No Auxiliary carry)	AC (Auxiliary Carry)
PO (Odd Parity)	PE (Parity Even)
NC (No Carry)	CY (Carry)

The details of individual commands are described as under.

1. **A (Assemble):** Assemble a program into machine language. The syntax is

```
A [address]
```

Address is assumed to be an offset from the address in CS, unless another segment value is given

Example	Comment
A 100	Assemble at CS:100H
A	Assemble from the current location.
A DS:2000	Assemble at DS:2000H

2. **D (Dump):** The D command display's memory on the screen as single bytes in both hexadecimal and ASCII. The syntax formats are

```
D [address]
D [range]
```

If no address is specified, the location begins where, the last D command left off, or at location DS:0 if the command is being types for the first time.

Example	Comment
D F000:0	segment - offset
D ES:100	segment register - offset
D 100	offset only

3. **E (Enter):** The E command places individual bytes in memory. You must supply a starting memory location where the values will be stored. The syntax formats are

```
E address
E address [list]
```

Example	Comment
E 100	Enter hex or character data at DS:100
E CS:100 "Test"	Enter string 'test' into memory location S:100

4. **F (Fill):** The F command fills a range of memory with a single value or a list of values. The syntax is .

```
F range list
```

The range must be specified as two offset addresses or .segment-offset addresses.

Example	Comment
F 100 500	Fill with spaces
F CS:300 1000 FF	Fill with hex 0FF
F 100 L 20 'A'	Fill 20 hex bytes with letter 'A' starting at location 100

5. **G (Go):** Execute the program in memory. You can specify a starting address and break point causing thee program to stop at a given address. The syntax is

```
G [=start address] [bkpt address] |bkpt address .....]
```

If no break point is specified, the program runs until it stops by itself. Up to 10 break points can be defined.

Example	Comment
G	Execute from IP to the end of the program
G 50	Execute from the IP to CS:50H and stop
G=10 50	Begin execution at offset 10H and stop before offset 50H

6. **H (Hex arithmetic):** The H command performs addition and subtraction on two hex numbers entered in the following formats:

```
H   value1 value2
```

For example H 1A 10 command add and subtract to hex numbers 1A and 10 and display the results as:

```
002A  000A
```

7. **M (Move):** The M command copies a block of data from one memory location to another. The syntax is

```
M range address
```

The range consists of the starting and ending locations of the bytes to be copied. Address is the target location, to which the data will be copied.

Example	Comment
M 100 105 110	Move bytes in the range DS:100-105 to location DS:110
M CS:100 105 CS:110	Same as above except that all offsets arc relative to CS

8. **P (Ptrace):** Execute the next instruction and stop; if the instruction calls a procedure, execute the procedure and then stop. The LOOP instruction and string primitive instructions (SCAS, LODS etc.) are executed completely up to the next instruction.
9. **R (Register):** The R command may be used in one of three ways:
   - Display the contents of one of the register, allowing it to be changed.
   - Display registers, flags, and the next instruction about to be executed.
   - Display the eight flag settings, allowing any or all of them to be changed.

Example	Comment
R	Display the contents of all registers.
R IP	Display the contents of IP and allow it to be modified.
R CX	Same for the CX register.
R F	Display all flags and change them if desired.

10. **S (Search):** Search a range of addresses for a list of bytes or a string. Syntax:

```
S range list
```

Example	Comment
`S 100 1000 0D`	Search `DS:100` to `DS:1000` for the value `0DH`
`S 100 1000 CD,20`	Search for the sequence `CD 20`
`S 100 9FFF COPY`	Search for the word `COPY`.

11. **T (Trace):** Execute one or more instructions from the current `CS:IP` location or an optional address if specified. The contents of the registers are shown after each instruction is executed. Syntax

```
T [=addrcss], [.value]
```

Example	Comment
`T`	Trace one instruction from the current location
`T 5`	Trace 5 instructions
`T =5.10`	Start tracing at `CS:5` and trace the next 16 steps

This command traces individual loop iterations, so you may want to use it to debug statements within a loop. Also, the T 'command traces into a procedure call.

12. **U (Unassemble):** The `U` command translates memory into assembly language mnemonics. This is called unassembling or disassembling memory. Syntax

```
U [address) [range]
```

Example	Comment
`U`	Unassemble the next 32 bytes from the current location
`U 0`	Unassemble 32 bytes from local ion 0
`U 100,108`	Unassemble all bytes from offset `100H` – `108H`

13. **Q (Quit):** Quit from `DEBUG` program and return to `DOS` prompt.

Sample program

Start the debug program from `DOS` prompt and write following lines sequentially

```
- A 100
- MOV AX,5
- MOV BX,10
- ADD AX,BX
- INT 20                ; To terminate the program
```

Now run the program in different ways and trace the different registers and segments using different commands as mentioned above.

Assignments: (Use DOS DEBUG)

1. Write an assemble language program in `DEBUG` to add four 16-bit numbers and test it.
2. Write a program in `DEBUG` to add two 8-bit numbers stored in the memory location `DS:300` and `DS:302` and store the result at `DS:304`.
3. Convert the problem no 2 for 16-bit numbers.
4. Write a program to add four 8-bit numbers stored in the memory location starting at 200 and store the result at the end of the table.
5. Write a program to subtract an 8-bit number stored at 204 from the number stored at 202 and store the result at location 206.
6. Solve problem 5 for subtraction of 16-bit numbers.
7. Convert the problem no 4 for the 16-bit numbers.
8. Write the program of your own and test it using `DEBUG`.

Introduction to assembly language program

The symbolic instructions that we code in assembly language are known as known as the *source program*. The assembler is a program that translates the source program into machine code known as *object program*. Then we use a linker program to complete the machine addressing for the object program, generating an *executable module*.

Several assemblers are available that run under the disk operating system, such as the Microsoft Macro assembler called MASM and Borland's Turbo assembler. We use MASM Ver. 5 in our lab.

Inside every computer there is a microprocessor called central processing unit (CPU) which is heart of a computer system. IBM PC used the Intel 8088 processor and the optional Intel 8087 math-coprocessor designed for high speed floating point math operations. Later on the IBM PCs has started to use Intel 80286, 80386 processors as CPU. The computers in our lab are using Intel Pentium processor as CPU.

CPU Registers:
The processor's registers are used to control instructions being executed, to handle addressing of memory, and to provide arithmetic capability.

General-Purpose Registers: 16 Bit: AX (accumulator), BX (base), CX (counter), DX (data)
8 Bit: AH, AL, BH, BL, CH, CL, DH, DL

Segment Registers: A segment register provides for addressing an area of memory known as the current segment. The four segment registers are CS (code segment), DS (data segment), SS (stack segment) and ES (extra segment).

Index Registers: SI (source index), DI (destination index)

Special Registers: IP (instruction pointer), SP (stack pointer)

Flag Register: Overflow (NV/OV), Derection (UP/DN), Interrupt (EI/DI), Trap (1/0), Sign (NG/PL), Zero (NZ/ZR), Auxiliary Carry (AC/NA), Parity (PE/PO), Carry (CY/NC).

Program Structure:
A program is divided int separate segments. Each segment contains instructions or data whose addressed are relative to a segment register (CS, DS, ES or SS). MASM provides a simplified set of directives for declaring segments, called simplified segment directives.

A basic program structure to be used in most programs is shown below:

```
TITLE Sample Program No : 1
      .MODEL SMALL       ;SMALL MEMORY MODEL
      .STACK 32          ;SET STACK SIZE
      .DATA
VAL1 DW 4321H            ;DATA VARIABLES
VAL2 DB 10H
      .
      .

      .CODE              ;START OF CODE
MAIN PROC FAR
      MOV AX,@DATA
      MOV DS,AX          ;INITIALIZATION OF DATA SEGMENT REGISTER
      .
      .                  ;INSTRUCTIONS
      .
      .
      MOV AX,4C00H
      INT 21H
MAIN ENDP
      END MAIN
```

Structure of an assembly language program

Model directive: The model directive selects a standard memory model for your program. It determines the way segments are linked together, as well as the maximum size of each segment.

Model	Description
Tiny	Code & data together may not be greater than 64K
Small	Neither code not data may be greater than 64K
Medium	Only the code may be greater than 64K
Compact	Only the data may be greater than 64K
Large	Both code & data may be greater than 64K
Huge	All available memory may be used for code & data

Program Segments: Three segment directives are normally used, .stack, .code and .data. They may be placed in any order within a program. The .stack and .data directives define the size of stack and start of data segment respectively, and .code directive identifies the start of code (instructions) in a program.

Assembling and Linking: Before we assemble and link a program (`*.asm` file), the following programs should be present.

```
MASM.EXE
LINK.EXE
```

Assemble the program: Let a sample program has been created using a text editor and saved to disk as `SUM.ASM`. To assemble it we have to type

```
MASM SUM.ASM
```

Then MASM displays the number of errors if any. If there are no errors then it creates an intermediate file `SUM.OBJ`

Link the program: in this step, the linker reads the object file as input and creates an executable file `SUM.EXE`. The command is

```
LINK SUM.OBJ
```

Create a source program given below using a text editor and compile, link & execute the program

```
TITLE to add two numbers
        .MODEL SMALL
        .STACK 32
        .DATA
VAL1 DW 1234H
VAL2 DW 2345H
SUM DW ?
        .CODE
MAIN PROC FAR
        MOV AX,@DATA
        MOV DS,AX

        MOV AX,VAL1
        ADD AX,VAL2
        MOV SUM,AX

        MOV AX,4C00H
        INT 21H
MAIN ENDP
        END MAIN
```

Explain each step and purpose of the above program.

You can use the DOS DEBUG program to unassamble and check the program's execution. After compiling and linking the above program, load the SUM.EXE into the DOS DEBUG as

```
DEBUG SUM.EXE
```

Now you will see the DEBUG prompt as

```
-
```

Unassamble the above program with the U command as

```
-U
```

Explain what you see in the screen.
Use other DEBUG program to check the validity of the result.

Assignments:

1. Write an assembly language program to sum numbers from 0 to 255.
2. Write an assembly language program to add ten 16-bit numbers stored in memory and store the result.
3. There are two tables having ten 16-bit data in each. Write an assembly language program to generate the third table which contains the sum of corresponding element of first and second table.
4. Two tables of data are stored having ten 16-bit data each. Write an assembly language program to generate the third table which contains 1FFFH if the corresponding data in first table is less then that of second table, else store 0000.
5. Write a program to generate and store a multiplication table of a number stored as num1
6. Write a program to find the sum of the following series up to 20 terms and store the result
   2 x 3 + 4 x 5 + … to 20 terms

Assemble all these programs and check the validity of the result with the DOS DEBUG program.