2. More Work with Debug

Background

This lab will continue the exploration of the DEBUG program introduced in Lab #1. DEBUG, supplied by MS-DOS, is a program that can be used to debug x86 programs. Using DEBUG, you can easily enter an x86 program into memory and save it as an executable MS-DOS file (in .COM format). Other features of DEBUG include examining/changing memory/register contents, single-stepping programs, and inserting breakpoints into programs. DEBUG cannot be used to debug programs that contain 32-bit instructions (i.e., new instructions added to later versions of the 8086 such as the Pentium). This lab reviews some of the DEBUG features discussed in the previous lab and covers some additional concepts within DEBUG and the x86 instruction set.

Objectives: Learn to

A. Use DEBUG to examine the contents of memory and internal registers.

B. Modify contents of memory, registers, and address space.

C. Write short assembly language sequences using DEBUG.

D. Single step through a program using the P (trace) function.

E. Assemble and Unassemble code using DEBUG.

F. Learn what the flags register is used for.

Pre-Lab

1. Define the difference between a logical address and a physical address. Show how a physical address is generated from a logical address.
2. Define what each of the following registers are used for: DS, CS, IP, AX
3. Define the function each of the following flag bits in the flag register: Overflow, Carry, Sign, and Zero.

Lab

In the first portion of the lab, we will work with the DEBUG program. The answers to any questions marked as 'Lab Questions' must be included in your lab report.

A. Working with DEBUG

At a MS-DOS prompt, type ***debug***. DEBUG's prompt is a dash (-). Type a question mark (?) to get a list of available commands.

You may return to the DOS prompt by entering *Q* and carriage return (<CR>).

While in the DEBUG program, type **D** to initiate the dump command. The dump command displays the contents of memory.

Lab Question 1: A). What occurs for this command? How many bytes are displayed?

B) What is starting SEGMENT value, Offset value (on left hand side, displayed as XXXX:XXXX)

C) What is the relationship between what is displayed on the right hand side and the HEX codes that are displayed?.

Lab Question 2. The command also accepts a logical address (segment:offset)

A) What happens if you type **D1000:0100**?

B) What is different about this memory range and the previous memory range?

On the x86 there are two types of addresses: the physical address and the logical address. The physical address is the address that is actually put on the address pins of the microprocessor and decoded by the memory interface. For the 8086/8088 CPUs, the address range is 00000H to FFFFFH which is equal to 1 megabyte of memory (2^{20} memory locations, 20 address pins for the 8088/8086) . The logical address consists two parts, each 16 bits in length: the *segment* and the *offset*. The segment and offset values are combined to form the physical address. A logical address of 2500:0010 (segment:offset) has a physical address of 25010H. To convert a logic address to a physical address, shift the segment value to the left by four bits (shift left 2500H by 4 = 25000H) and add the offset value (25000H+0010H = 25010H) to produce the 20 bit physical address.

How would you display memory locations 10000H thru 1000EH? Remember the physical address locations are displayed as a combination of the segment address and the offset.

Use the ENTER command to load 12H, 13H, 14H into location 10000H (**E1000:0000**). The E command displays the contents of the memory location and waits for new data or a space character (hit the spacebar) to advance to the next location. The number before the period is the contents of the memory before the change. If no change is necessary, enter a space to go to the next location.

Lab Question 3: Use the Dump command to view the contents at location 10000H.

A) Are the values you entered in memory display at that location?

B) How is the physical address 10000H calculated from the logical address 1000:0000? Show this calculation in your lab report.

We can also use the mini-assembler to enter data. The A command accesses the mini-assembler. An assembler is a program that translates x86 instructions into machine code. Enter the following data (<CR> stands for carriage return - the enter key. The bold type is what you enter, the italicized is what DEBUG displays):

```
               A 1000:0000 <CR>
1000:0000    DB 15,64,FE <CR>
1000:0003    <CR>.
```

Use the dump command to display the contents of the segment address 1000:0000,0002 and verify that the data you entered is actually in memory. The 'DB' command above stands for "DEFINE BYTE" and is used to tell the assembler to enter these 8-bit values into memory (the 8-bit values are in HEX).

Use the mini-assembler to enter the following data:

```
                    A 1000:0000 <CR>
1000:0000     DW 1234, 88AB <CR>
1000:0004     <CR>.
```

The 'DW' command above stands for "DEFINE WORD" and is used to tell the assembler to enter 16 bit values in memory (16 bits = 2 bytes). Dump the contents of memory using "D 1000:000".

Lab Question 4: A) What bytes are in locations 1000:0000, 1000:0001, 1000:0002, 1000:0003? Explain this byte arrangement (look in the notes and see what 'little endian' means in terms of byte ordering).

Example 1.1 is a short program that uses a DOS software interrupt to display a character string (you can think of a DOS software interrupt as a 'function call' to DOS). For the DOS software 21H interrupt, we use the 'AH' register value to select a particular operation. In this case, AH=9 selects the function that prints out a character string. The starting address of the character string must be passed in the DS:DX registers. The character string must end with a $. The "INT 3" instruction is a breakpoint that stops program execution and causes the registers to be displayed.

Example 1.1

```
-A  0100 <CR>
                MOV DX,0108
????:0103 MOV AH,9
????:0105 INT 21
????: 0107  INT 3
????:0108  DB 'This is my first assembly program$'
????:012A
```

The ???? above represents the code segment value. This value may differ depending on your computer configuration. Follow the example above and store the program in memory. Use the U 0100,0105 command to unassemble the program and to display it on the monitor using DEBUG to verify that your program got entered correctly. The above command does not specify a SEGMENT value for the address "0100" so the current value of the CS register (Code Segment register) is used for the segment value. To display the register values, use the "r" command.

Lab Question 5: A)How does the displayed program differ from Example 1.1?

B) What is the logical address (segment:offset) of the character string?

C) Type **D 108 L21,** how does the ascii text appear? This command says to display 21 bytes starting at location 0108.

D) What is the starting address of the program as a logical address (segment:offset)? as a physical address? (single 20-bit address XXXXX)

Once we are sure that the program is stored in memory, we can execute it in three different ways. The first way is to use the G (Go) command. For example, the G=1000:0000 command informs DEBUG to execute the program beginning at location 1000:0000. The = sign must precede the address. If a breakpoint is desired, it is entered as a second parameter.

Before executing the program, make sure that the DS register (data segment has the same value as the CS register. Use the 'r' command to display the registers. If the DS register is different from the CS register, then use the command "r ds" to modify the DS register value -- when the ":" prompt appears, enter the new DS value. Use the 'r' command to verify any change to the DS register that you made was actually made.

Lab Question 6: A) Use the G command to execute the example program and stop it at offset address 0005. (G=0100,0105). What are the contents of the AX, DX registers when the program stops? Why?

B) What is the value of the IP register? Why?

To continue execution of the program, use the "P" (proceed) command. This will execute the current instruction that is currently pointed to by the CS:IP address, print the registers, and then halt execution. This is a good way of single stepping through your program.

Lab Question 7: A) Use the P command to execute the next instruction (should be 'INT 21H'). What happens?

The 'T' (trace) command can also be used to execute one instruction at a time. The trace command is a true single step command in that if the instruction is a function call like an "INT 21H", the trace command will step into the DOS function and halt there. The 'T' (trace) command is said to 'step-into' function calls. The 'P' command will not trace into a function call, but rather execute it completely, and then stop at the next instruction. The 'P' (proceed) command is said to 'step-over' function calls. You never want to use the 'T' command when tracing a DOS function call because the DOS function call may execute hundreds of instructions. You will probably find the 'P' command more useful than the 'T' command most of the time, but you need to understand the operation of both.

If you use the 'G' (go) command without ANY parameters, then it will begin execution at the current CS:IP address, and it will continue execution until it reaches a breakpoint command (INT 3). You need to make sure that end of your program contains a breakpoint. If we left off the "INT 3" at the end of Example 1.1 this would cause a problem - the hex codes that represent the ascii string "This is my first assembly program" would be executed as if they were instructions!!! Be sure to either include a stopping location when using the 'G' command, a breakpoint (INT 3), or use the 'P' command to step through your program.

Use the "r ip" command to change the IP register back to 0100. Use the 'G' command with no parameters and verify that your program executes until it hits the breakpoint.

B. Another Program -- Addition

Exit "DEBUG" by using the "q" (quit) command. Re-enter DEBUG and use the A command for DEBUG to enter the following program.

Example 1.2

```
MOV AL,4C
ADD AL,3E
INT 3
```

The first instruction moves the 8-bit value '4C' into register AL. The second instruction adds the value of '3E' to the current value in AL and stores the result in AL.

Lab Question 8: Use the 'P' command to single step through the 'MOV' and 'ADD' instructions. What is the final value of 'AL'? In the lab report, verify through binary addition or hex addition that this is the correct answer.

Lab Question 9: The above program does 8-bit addition since register AL is 8-bits wide. Modify the program to use register AX and use it to compute the sum of 7FE3 + 6DA0 = ______________ . Include this program in your report.

C. The *Flags* register

The Flags register is different from the other registers in the x86 because each bit in the flags register indicates a status of some kind. DEBUG prints out the value of the Flags register on the second line of the register listing and to the far right hand side. The table below shows how each flag is represented by DEBUG (a value of '1' for a flag means that the flag bit is SET, a value of '0' means the flag bit is CLEARED).

Flag Name	DEBUG Output	
	For Flag = 0 (CLEARED)	For Flag = 1 (SET)
Overflow	NV	OV
Direction	UP	DN
Interrupt	DI	EI
Sign	PL	NG
Zero	NZ	ZR
Auxiliary Carry	NA	AC
Parity	PO	PE
Carry	NC	CY

Debug displays the flags in the following order:

Overflow, Direction, Interrupt, Sign, Zero, Aux Carry, Parity, Carry

Section 3.1 in the Uffenbeck text and section 2.14 in the Irvine text describes what conditions these flags are set or cleared. Different instructions on the x86 affect different flags, some instructions do not affect the flags at all.

Appendex F in the Irvine textbook gives a description of each x86 instruction and the flags that they affect. We will only be concerned with the Overflow, Sign, Zero, and Carry flags in this lab.

The MOV instruction flag description in the Irvine textbook shows:

	O	D	I	S	Z	A	P	C
MOV								

The MOV instruction does not affect the status of any flags, so the boxes are blank. The ADD instruction is shown below. An '*' means that the final result of the ADD instruction will affect the flag:

	O	D	I	S	Z	A	P	C
ADD	*			*	*	*	*	*

Lab Question 10: Using the example program 1.2, determine two 8-bit numbers that will cause the following flag conditions to occur after the addition. Verify that your numbers cause the specified flag conditions by modifying your program with your new numbers, executing it, and recording the flag values. Use HEX numbers below:

A. Carry = 0, Overflow = 0, Zero = 0, Sign = 0: _________ + __________ = ___________

B. Carry = 0, Overflow = 0, Zero = 0, Sign = 1: _________ + __________ = ___________

C. Carry = 1, Overflow = 1, Zero = 0, Sign = 0: _________ + __________ = ___________

D. Carry = 0, Overflow = 1, Zero = 0, Sign = 1: _________ + __________ = ___________

E. Carry = 1, Overflow = 0, Zero = 0, Sign = 1 : _________ + __________ = ___________

F. Carry = 1, Overflow = 0, Zero = 0, Sign = 0 : _________ + __________ = ___________

G. Carry = 0, Overflow = 0, Zero = 1, Sign = 0 : _________ + __________ = ___________

Lab Question 11:

A. It is not possible to get a flag combination of Carry =1, Overflow = 1, Sign = 1 from an addition operation. Explain why.

B. It is not possible to get a flag combination of Zero = 1 and Sign = 1 from an addition operation. Explain why.

Lab Report

A. Describing What You Learned

Answer all of the "**Lab Questions**" in your lab report. There are NO DEMO requirements for this lab.