

EXPERIMENT 1

FAMILIARITY WITH DEBUG, x86 REGISTERS and MACHINE INSTRUCTIONS

Pre-lab:

This lab introduces you to a software tool known as DEBUG. Before the lab session, read the first two sections of chapter 4 in the Uffenbeck textbook for an introduction to the X86 registers and a discussion of DEBUG. Appendix B in the Irvine textbook has a tutorial on DEBUG with a description of all of the commands. If you are unfamiliar with the DOS command line window, read Appendix A in this lab before proceeding. Appendix A also describes how to capture DOS command line windows for inclusion in your lab report.

Post-lab:

Answer the 4 questions in this lab and submit them with your lab report.

Objective:

Become familiar with the internal structure of the **x86** microprocessor register set and the Microsoft **DEBUG** utility. **DEBUG** is a utility program shipped with DOS/Windows that can be used to debug programs by allowing the user to examine memory and register contents, assembling and disassembling programs, single-stepping, and a variety of other functions.

Experiment:

To invoke the **DEBUG** program, a user opens a **DOS** window and enters the following: (NOTE: you can also click on the start button, choose run, and enter debug from windows 95, 98 or NT).

```
C:\> debug <enter>
```

DEBUG responds with the hyphen, -, prompt. The hyphen prompt indicates that **DEBUG** is ready to accept commands in an interactive environment. One of the most important commands is ? which gives a summary of all valid commands. A list of all valid **DEBUG** commands is given on page 155 of the class textbook.

Invoke **DEBUG** now and list all commands:

```
C:\> debug <enter>
-? <enter>
```

You should get a list of all commands displayed to the screen. They may all be invoked with single letters, some requiring additional arguments.

Arithmetic Using DEBUG:

H is the **DEBUG** command that returns both the sum and difference of two arguments. **DEBUG** computes in Hexadecimal only!

From a **DEBUG** prompt, enter the following:

```
-H 3 2 <enter>
```

You should see the following appear on your screen:

```
0005 0001
```

This is the sum and difference respectively, $3+2=5$ and $3-2=1$. Now try:

```
-H 2 3 <enter>
```

In this case, the negative difference, $2-3$, is represented in 2's complement form. A negative number can always be identified by checking if the leading bit (MSB) is set to 1. However, a positive sum can also have MSB=1. It is *UP TO THE PROGRAMMER* to interpret the result correctly. As an example, enter the following:

```
-H E1F6 1E09 <enter>
```

The positive sum is also FFFF! Now try:

```
-H 5C3F0 4BC6 <enter>
```

The error occurs because **DEBUG** operates on *WORDS* only. For the **x86** architecture, a word is 16 bits in length.

Registers and Memory:

The **x86** (at least the **8086** version) contains 14 basic registers. To examine the content of the registers, enter the following:

```
-r <enter>
```

Notice the 2-letter abbreviations on the far right side of the second line. These indicate the values of the bits in the **FLAGS** register. To examine and change the content of a single register, the **r** command can be used with a register name as the argument. As an example, enter the following:

```
-r ax <enter>
ax      0000
:3A7 <enter>
```

Now examine the registers again using the **r** command. You should see that register **ax** now contains **03A7**.

Load the value, **92A** into the **bx** register using the **r** command. Verify that **bx** and **ax** contain the values **3A7** and **92A** respectively.

Instead of adding **3A7+92A** using **DEBUG**, we will let the **x86** microprocessor add them by placing *machine instructions* in its' memory that "tell" it to perform addition for us. The command for examining and changing memory is **e**. Enter the following:

```
-e 100                                <enter>
3756:0100  E4.01                        <enter>
-e 101                                <enter>
3765:0101  85.D8                        <enter>
```

You will probably have different values displayed from the **3756**, **E4** and **85** shown above. This is OK since these values depend on what was present in memory prior to invoking **DEBUG**. You should enter the **01** and **D8** values at offsets **0100** and **0101**. By typing **e 100**, you are telling **DEBUG** to load a byte of

data at address offset 0100 in the current code segment. In the example above, the current byte, E4, present at segmented address 3756:0100 is being replaced (overwritten) by the user-entered byte, 01. In the example above, the code segment is 3756, yours will probably be different, but it doesn't matter here. Multiplying the code segment by hexadecimal 10 and adding the offset will produce the physical address. x86 systems generally specify memory addresses in segmented form.

Addition Using the x86:

Examine the register content:

```
-r <enter>
```

The display should resemble:

```
AX=03A7 BX=092A CX=0000 DX=0000 SP=#### BP=#### SI=#### DI=####
DS=#### ES=#### SS=#### CS=#### IP=0100 NV UP DI PL NZ NA PO NC
####:0100 01D8 ADD AX, BX
```

Where # represents a hexadecimal digit. The last line of the display indicates that 01D8 is stored in memory. This, in fact, is the machine language representation for the ADD AX, BX instruction which causes the contents of the two registers to be added and the sum placed in the AX register. Since the x86 is a byte-addressable architecture, the data is organized as follows:

```
3756:0100 01
3756:0101 D8
```

Although the instruction is present in memory, before it can be executed the x86 must be "told" where the instruction is located. The method used by the x86 is to use two special registers, the CS and IP which contain the segmented address of the "next" instruction to be executed and hence effectively 'point' to the instruction in memory by containing the code segment (in the CS register) and the offset (in the IP register). If necessary, use the r command to load IP with the correct offset, 0100.

DEBUG has a trace instruction, t, which causes a single machine language instruction to be executed. Instruct DEBUG to execute the trace command as follows:

```
-t <enter>
```

The AX register should now contain 0CD1 which is the sum of 03A7 and 092A.. Notice that the original content of AX has been overwritten with the sum. Also note that the IP now contains 0102. The x86 automatically increments IP in anticipation of executing the next instruction in memory.

Now use the x86 to add 092A (in BX) to the value 0CD1 (in AX). Use the r instruction in DEBUG to set IP to the proper value.

To perform subtraction, it is necessary to store the machine language command for subtracting into the code segment. Do this now by using the e command to store 29D8 beginning at offset 0100 in the current code segment. Use r to verify that you have indeed stored the SUB AX, BX instruction in memory, then execute using the DEBUG trace command.

Using DOS Functions from Assembler language Programs:

The INT instruction allows the x86 to access special programs that are present in memory when the computer is booted and DOS has copied them from the harddisk into main memory. For now, you can think of INT as a 'function call'.

Using **DEBUG**, load **AX** with **0200** and **DX** with **0041**. The machine instruction for **INT 21** is **CD21**. Put **CD** in memory in the code segment at offset **0102** followed by **21** in the next location. Set **IP** to **0102** and verify the registers and memory contain the proper values by using the **r** command. Next, execute the instruction by entering the following:

```
-g 104 <enter>
```

g is **DEBUG**'s "go until" instruction. You are telling the **x86** to execute all instructions from the current location in the code segment (specified by the **CS:IP** content) to **CS:0104**. If the **t** command were used, all instructions in the **INT 21** routine would need a **t** to execute, that is, you would have to "single-step" through the entire **INT 21** routine which could be lengthy. Using **g** allows execution to proceed until **CS:0104** is reached.

Note that an **A** was printed on your monitor. The particular **INT** routine actually executed was type number **21**. This is the **INT** used to access **DOS** functions. The **02** in register **AH** (**AH** is most significant byte of **AX**) told **DOS** to print a character to the monitor. The actual character printed was specified by the **41** that was loaded into **DL**. **41** is the ASCII character code for the letter **A**.

Exiting Programs:

When an assembly (or machine language) program finished execution, the **x86** must be allowed to be controlled by the operating system. One way to accomplish this is to call **INT 20**. When you create programs in a high-level language, such as **C**, the compiler automatically inserts this command at the end of the program. At this lower level of programming, it is up to you, the assembler language programmer to include **INT 20** at the end of your programs.

Now, write a 2-line program:

```
INT 21  
INT 20
```

by doing the following steps:

1. load **ax** with **0200** and **dx** with **0043** using the **DEBUG r** command
2. load memory with the following:

```
CD  
21  
CD  
20
```
3. set **IP** and **CS** to contain the memory address of the **INT 21** instruction
4. use the **DEBUG g** command with a valid ending offset to execute the program

Congratulations! You have just written and executed your first machine language program. Demonstrate the running of this program to the lab instructor.

QUESTION 1: What was printed and why was it printed in the 2-line program above?

Assemble/Disassemble:

Most programs are longer than two lines. To view the mnemonics of the code (the human readable part), change the **IP** to the beginning of the 2-line program and enter the **DEBUG** unassemble command:

```
-u <enter>
```

You should see your 2-line program in mnemonic form followed by the hexadecimal machine language that is stored in memory. It is much easier for humans to enter the program into the computer using mnemonics rather than hexadecimal machine codes. **DEBUG** provides an assemble command, **a**, for this purpose. As an example, the two-line program can be entered at code segment offset **0100** by entering the following:

```
      -a 100      <enter>
int 21          <enter>
int 20          <enter>
              <enter>
```

The above commands allow you to enter the program in mnemonic form rather than hexadecimal machine codes.

DEBUG Scripts:

It is possible to create assembly language programs using **DEBUG** only (although there are some limitations that we will discuss in more detail later in the class). You can create a file containing **DEBUG** instructions and assembly code mnemonics and “pipe” it into **DEBUG** using the DOS redirection operator '<'. As an example, do the following exercise:

1. Create a file named **TEMP.DBG** on your PC. The file contents are given below. You may use any text editor or word processing program that you wish, but you must be sure to save the file as a **TEXT** or **ASCII** file!

```
a 100
mov ah, 02
mov dl, 2a
int 21
int 20

n temp.com
r bx
0
r cx
08
w
q
```

Be sure to insert the blank line in your file as shown above. This signals **DEBUG** that you are finished entering assembly language instructions.

2. Invoke from a **DOS** prompt in a **DOS** window as follows:

```
C:\> DEBUG < TEMP.DBG      <enter>
```

3. Give an explanation for each of the lines contained in the **TEMP.DBG** file. Also give a description of the other debug commands used in this lab (a, e, g, h, r, t).

You should now have an executable file named **TEMP.COM** on your harddisk.

QUESTION 2: Run **TEMP.COM**, what does it do?

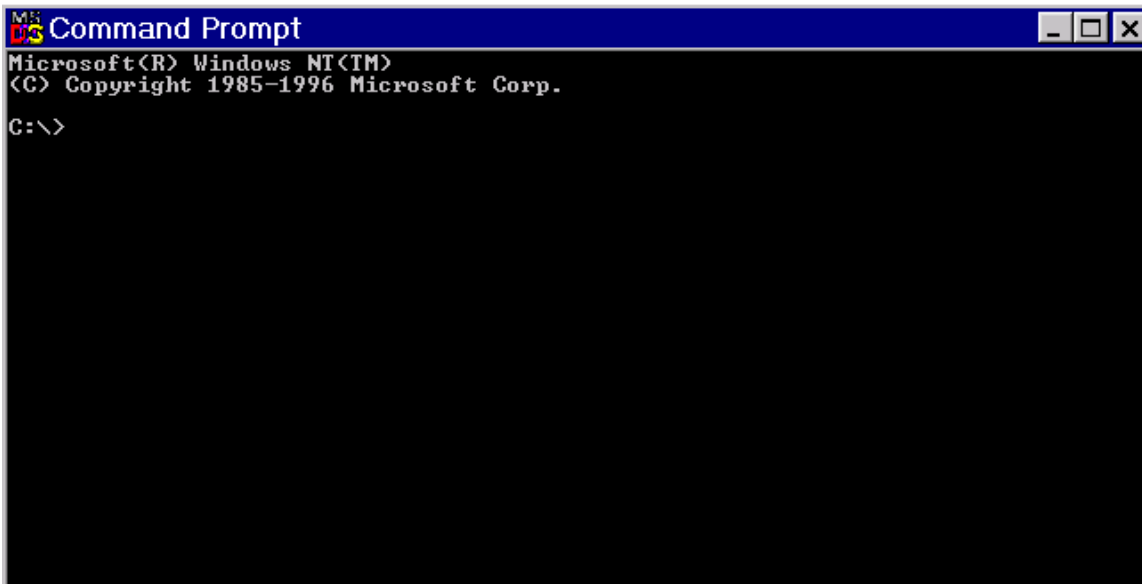
QUESTION 3: Give an explanation for each of the lines contained in the *TEMP.DBG* file. Also give a description of the other debug commands used in this lab (a, e, g, h, r, t).

QUESTION 4: Describe what you learned from this experiment. Include both the good points and the areas that could use improvement.

APPENDIX A: The DOS Command Window

Use of a DOS command window may be unfamiliar to you. The DOS command window allows you to execute programs, change directories, create/delete directories/files, etc. via commands typed from the keyboard. This was the standard way of accomplishing these tasks before Microsoft created the Windows Graphical User Interface (GUI). Use of command line windows ('shells') are common in other operating systems such as Linux and UNIX-variants. Expert users normally find command line interfaces more efficient at common file system tasks than a GUI. We will use the DOS command line window for running certain programs in this class.

To open a DOS command line window, execute the program named *Command Prompt* (normally found under from the Start→Programs list under Windows). You should get a window that looks something like this:



The 'C:\' is called the command prompt and shows you the current directory ('folder') that you are in. You can change to another folder via the 'cd directoryname' command (cd stands for *Change Directory*).

```
C:\> cd Data
C:\Data>
```

The command prompt changes to show you the current directory that you are in. You can 'back up' one directory level by using '..' for the directory name:

```
C:\Data> cd ..
C:\
```

You can change drives by just typing the name of the drive letter followed by a colon ':' :

```
C:\Data> H:  
H:\
```

To see a list of files in the current directory, you use the 'dir' command (directory) command. To execute a program, just type the name of the program (myfile.exe or just 'myfile' -- you can leave off the .exe). The program either needs to be in the current directory or found in the list of directories specified in the 'PATH' environment variable. If DOS says that it cannot find the program, then type in the complete path to the program such as 'c:\data > C:\data\mydir\myfile.exe' .

To add directories to the PATH environment variable, you can do the following:

```
C:\ set path=%path%;c:\data\mydir
```

To see the list of the directories on the path variable just type 'path' after the prompt and the value of the path environment variable will be displayed.

You should not have to change the path variable on the PCs in the Micro I labs; all of the programs that we will use are in directories that are included on the path variable. However, you may have to change the path variable on your local PC.

We will not be using the DOS command prompt to create/delete directories -- you can do this via the normal Windows interface. However, if you want to access that directory from a DOS command window, you should keep your directory names to 8 characters or less and do not put space characters in the directory name.

If you want to find out more about the commands available from the DOS command line, type 'HELP'. You can also find many DOS tutorials on the WWW.

The Micro I lab PCs are set up such that when you log in, the 'H:' drive is your home directory that is available from your ECE Unix account. You will need to log into the Micro I PCs using your ECE username/password. You should always use the 'H:' drive for any files that you create in this class.

Occasionally, you may want to capture a DOS command window for inclusion in your lab report. One way to do this is hit ALT+PRINTSCREEN (hold down ALT and hit the print screen button) while you are in the DOS command window. This saves the current window to the clipboard as a picture object. You can now open the *Paint* program (under Start →Programs →Accessories) and use the Edit →Paste command to paste the window on the current image. Then use the 'File→Save As' command to save the image to file (use the 256 color bitmap format to reduce the file size of the image). When you are preparing your lab report, you can paste this image into your lab report.