Name: _____ Date: _____ Lab Section: _____

Lab partner's name: _____ Lab PC Number: _____

Objectives: Understanding video memory and character mapping of CGA characters in ROM BIOS, using the DOS **debug** command. Writing simple assembly-language programs using DOS **debug**.

Equipment: Bootable PC, monitor, keyboard, mouse, cables. MS DOS or Win 98 with DEBUG command.

You need a DOS command-line window for this lab. Either boot MSDOS, or restart Win98 in MSDOS mode (command prompt only), or start a MSDOS command window inside Windows 98 (e.g. Run | command).

Ensure that the **debug** command is installed by typing its name at the DOS command prompt, and then use the **Q** (Quit) command to exit debug. Use the **Return** or **Enter** key at the end of every command line you type:

```
C:\>debug
-?                  <-- typing a question mark displays a help screen
-Q                  <-- typing Q quits the debug program and returns you to the DOS prompt
C:\>
```

Intel Architecture *Segment:Offset* Memory Addressing

To use **debug** to address memory in an Intel x86 architecture, you need to understand an Intel *segment* and *offset* address format. Because the original Intel x86 CPU chips only had 16-bit registers, a register could only address 2^{16} memory locations. Intel devised a way to combine and overlap two 16-bit registers to allow 20 bits of address, by shifting the contents of the first register left by four bits and adding it to the second register to generate a 20-bit memory address. The shifted register is called the *segment register* and the added register is called the *offset register*. An address is written in two 16-bit parts as **segment:offset** with a colon separating the two parts, e.g. **B800:0123**. The effective 20-bit address is calculated as: **B8000 + 0123 = B8123**

Below are some examples of **segment** and **offset**, along with their calculated effective **20-bit memory addresses.** The 16-bit segment is always shifted left four bits before adding. Pay special attention to the last entries in the table that all have *different* segments and offsets but all result in the *same* effective address:

16-bit Segment	16-bit Offset	Shift Segment Left 4 bits and Add Offset	Effective 20-bit Address
B800	0123	B8000 + 0123 =	B8123
B800	1234	B8000 + 1234 =	B9234
B800	FFFF	B8000 + FFFF =	C7FFF
B801	0011	B8010 + 0011 =	B8021
B802	0001	B8020 + 0001 =	B8021
B000	8021	B0000 + 8021 =	B8021
A900	F021	A9000 + F021 =	B8021

Note that many, many *different* **segment:offset** pairs can generate exactly the *same* 20-bit effective memory address. (How many?) In every case, the effective address is computed by shifting the segment address left by four bits (one hex digit) and adding it to the offset register to generate a 20-bit memory address (five hex digits).

The **debug** utility requires that all addresses be entered in **segment:offset** format; so, if you are told to use the 20-bit address **B8123** you must first turn **B8123** into an equivalent **segment:offset** address, e.g. something like **B000:8123** or **B812:0003** or **B800:0123** (or any other equivalent **segment:offset** combination that adds up to the given 20-bit address **B8123**). The **debug** utility can not use 20-bit addresses; convert them first.

1. Calculate whether **B800:0123** and **A813:FFF3** are equivalent 20-bit addresses (yes/no): _____
2. Calculate whether **0000:FFF0** and **0FFF:0000** are equivalent 20-bit addresses (yes/no): _____
3. Calculate whether **0101:0101** and **0000:1111** are equivalent 20-bit addresses (yes/no):_____

The DOS DEBUG Utility: Command List

The DOS **debug** utility allows you to view and modify the contents of DOS memory. The commands we will be using in **debug** are given below. (Search the Internet for a DOS **debug** manual if you want the full details on each command.) In the table below, an *address* is a single **segment:offset** value. (You cannot use a 20-bit address in **debug**.) A *range* is a start *address* followed by an optional ending *offset* or by the letter **L** (for length) and a number of bytes. Use spaces to separate arguments. A *list* is a list of one or more bytes (in hex). Any item in [square] brackets is optional:

?
Help - list commands

D [range]
Dump (display on screen) memory contents. A missing end *offset* means dump 128 bytes (often displayed as eight rows of 16 bytes per row).

E address [list]
Enter (overwrite) memory with the *list* of bytes (in hex), starting at this *address*. A missing *list* will result in a prompt for a byte to enter.

U [range]
Un-assemble (diassemble) memory to 8086 assembly language. A missing end *offset* means disassemble approximately 32 bytes.

A [address]
Assemble 8086 assembly language and load the assembled machine language starting at *address*

N pathname
Remember this file name for use by **L** or **W**

L
Load from the file previously named by **N**

W [address]
Write memory into file named by **N** starting at *address*; length to write is in the **BX** and **CX** registers (high and low bytes).

Q
Quit debug

All numbers in **debug** are in hexadecimal. Remember to always turn 20-bit addresses into their **segment:offset** form for use in **debug** addresses. You cannot use 20-bit addresses directly in **debug**; convert them first.

Example: Dump (display) 20 bytes at address **B8123** — `D B800:0123 L 20`
Example: Disassemble bytes from **B8123** to **B812F** — `U B000:8123 812F`
Example: Overwrite memory starting at **B8123** — `E B812:0003 1A FF 3B C7 33 CB 25`

Dumping (Displaying) Memory

To dump (display) 128 bytes of memory to the screen in hexadecimal and ASCII, starting from the 20-bit memory location **C0000**, start **debug** and type the dump command given below (remembering to convert the 20-bit address **C0000** to **segment:offset** form first). Sample output is shown; your actual output may differ:

```
C:\>debug
-D C000:0000          (remember to push Return or Enter at the end of each debug command)
C000:0000  55 AA 40 E9 54 01 BD 6F-A6 00 00 00 00 00 00 00   U.@.T..o........
C000:0010  64 03 27 01 00 00 00 00-00 01 18 01 00 00 49 42   d.'...........IB
C000:0020  4D 20 43 4F 4D 50 41 54-49 42 4C 45 0A 50 68 6F   M COMPATIBLE.Pho
C000:0030  65 6E 69 78 56 69 65 77-28 74 6D 29 20 56 47 41   enixView(tm) VGA
C000:0040  2D 43 6F 6D 70 61 74 69-62 6C 65 20 42 49 4F 53   -Compatible BIOS
C000:0050  20 56 65 72 73 69 6F 6E-20 00 0D 0A 43 6F 70 79    Version ...Copy
C000:0060  72 69 67 68 74 20 28 43-29 20 31 39 38 34 2D 31   right (C) 1984-1
C000:0070  39 39 32 20 50 68 6F 65-6E 69 78 20 54 65 63 68   992 Phoenix Tech
```

- On the far left are the hexadecimal memory addresses (in **segment:offset** form) of the bytes that start each row of output. Each starting memory address is 16 bytes larger than its predecessor, since each row contains 16 bytes. The first row starts at offset **0000** and ends at offset **000F** for a count of 16 bytes.
- To the right of each address are the 16 bytes of the memory that are found starting at that address. Each row contains 16 eight-bit bytes, each byte shown as two hexadecimal digits. A dash separates the middle two bytes of each row for readability (between byte 7 and byte 8 of bytes 0 through 15).
- To the right of the 16 hexadecimal bytes is a row of 16 ASCII characters, which are the ASCII characters that correspond to the 16 bytes shown in the hexadecimal byte dump for that row. If a byte cannot be turned into a valid printable ASCII character (e.g. it might be an unprintable control character or it might be a non-ASCII character), a period (dot) is used in the ASCII dump output instead.

If you now enter a single '**D**' in **debug** (followed by *Return*), **debug** will show the *next* 128 bytes of memory.

Look at this row of dump output below (this single row is taken from the above printed sample dump):

```
C000:0020  4D 20 43 4F 4D 50 41 54-49 42 4C 45 0A 50 68 6F   M COMPATIBLE.Pho
```

The starting address of the first byte in the above row is **C000:0020** or **C0020**. The last byte in the row has address **C000:002F** or **C002F**. The ASCII letter '**C**' is located two bytes from the beginning of the row, at address **C000:0022** or **C0022**. We can see in the above row dump that ASCII '**C**' has hexadecimal value **43**. The value of the byte at address **C000:002C** (or **C002C**) is **0A**, which is not a printable ASCII character and shows up as a period '**.**' in the ASCII dump section. The **0A** is an ASCII "**LF**" [linefeed or newline] character, written as '**\n**' in C language. Another common unprintable character found in DOS/Windows text files is the "**CR**" [carriage return] character with hex value **0D**. The **CR** byte is followed by an **LF** byte in DOS/Windows text files and the pair of bytes **0D 0A** is often written in documentation as **CR+LF**, **CR/LF**, or **CRLF**.

Examine the full 128-byte sample dump printed above (*not* the one on your screen!) and answer these questions:

4. Give the starting memory location (20-bit address) of the sequence **CR+LF**: _____
5. What printable ASCII character is at memory location **C000:007A**: _____
6. What is the hexadecimal value of the byte at location **C003:004B**: _____

Entering (Changing) Memory

You can enter (change) the values in memory using the **debug** Enter command; but, be careful not to change things that might have serious consequences for your computer or data. (It is highly unlikely but possible to trigger an accidental reformat of your hard drive by changing some particular values in memory!)

Let's start by entering a list of bytes into memory and then dumping what we changed. Starting at **88000**, keep dumping memory (using the Dump command) until you find a memory area that contains only zero bytes. Remember the start address of this block of zeroes. Suppose your zero block starts address was **89000**. We will dump a length of 9 bytes starting at that address, then use the Enter command to change some values starting at that address, then dump that address again. Here are the bytes you should enter:

```
-D 8900:0000 L 9
8900:0000  00 00 00 00 00 00 00 00-00                    .........
-E 8900:0000 4C 69 6E 75 78 20 52 4F 58
-D 8900:0000 L 9
8900:0000  4c 69 6E 75 78 20 52 4F-58                    ?????????
```

Note the list of 9 changed bytes that we entered into memory using the Enter command.

7. Give the ASCII text corresponding to the bytes entered: _____

Changing Video Memory using DEBUG

The DOS (not **debug**) **cls** command ("clear screen") will clear your DOS command window. For the next part of the lab, you may have to periodically exit **debug** and issue the **cls** command through DOS to clear the screen, then re-enter **debug**. Do this now - exit **debug**, clear your screen, then re-enter **debug**.

A) Clear the screen before you start this section. Use **debug** to dump the memory starting at address **B8000**. Verify that the memory does not contain any **FF** bytes. If it does, dump starting at address **B0000** instead. Use whichever address shows you a series of '**20 07**' bytes repeated. Call over your instructor if you do not see this. We will assume that **B8000** is the correct address in this section.

B) Use **debug** to enter the bytes **41 12 42 75 43 34 44 25 45 57** starting at address **B80A0**. Five characters near the top of your DOS window should change to be the coloured upper-case ASCII letters **ABCDE** on various colour backgrounds. Call over your instructor if you do not see this.

Your DOS screen represents every character using two bytes. The first byte is the ASCII character value (e.g. **41='A'**) and the second byte is an *attribute* byte that determines how that character should be displayed (e.g. **12**). The upper nybble (half-byte) of the attribute byte controls the Background colour of the character; the lower nybble controls the Foreground colour (the letter itself) using this mapping:

	BACKGROUND			***FOREGROUND***			
Bit number:	3	2	1	0	3	2	1	0
Attribute:	F	R	G	B	H	R	G	B

Above is a bit-map of the eight bits of the *attribute* byte.

H: this bit means to use high intensity
R: this bit means turn on red
G: this bit means turn on green
B: this bit means turn on blue
F: this bit means to flash the foreground on/off

For example, to have a blue background with a green foreground letter, use attribute byte value **12h** which has bit pattern **0001 0010**. For a green background and red letter, set the attribute byte to **24h (0010 0100)**.

8. (S-1) Display a high-intensity white **A** on a black background followed by a black **B** on a white background followed by a green **C** on a red background. Call your instructor for a sign-off: _____

CGA Character Tables in ROM

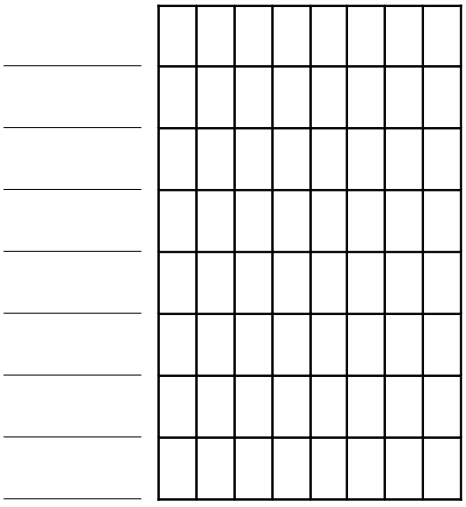
To display the letter shapes of characters on the DOS screen, the BIOS routines must know what pixels to turn on and off. The pixels are controlled by bit patterns. These bit patterns are read from a table in upper DOS read-only memory (ROM). Since a DOS character occupies seven bits, with 2^7 possible values, there are 128 possible characters, and the BIOS character table in ROM has 128 bit patterns in it.

The bit pattern for each BIOS character is drawn using an eight-bit by eight-bit square, with each bit in the square representing a pixel on the screen. Each character uses 64 bits; see the 8x8 grids below. The square of 8x8 pixels is stored in memory as eight eight-bit bytes, with the eight bits in each byte being a row in the square. The first eight-bit byte is the top row of eight pixels of the character. Eight rows of eight-bit-bytes makes up the 64 bits of the character. The full BIOS character table is made up of 128 characters, which is 128 sets of eight bytes (64 pixels). The BIOS ROM character table starts at address **FFA6E**. Dump this address.

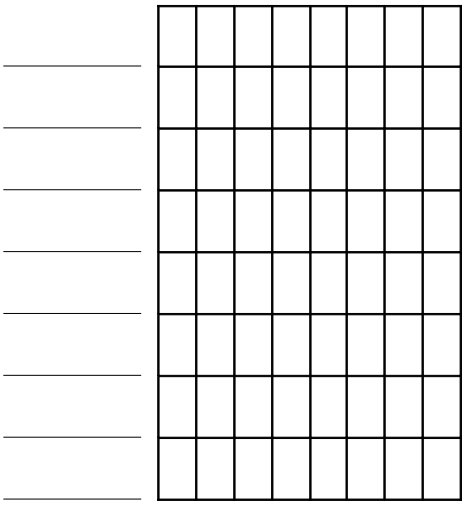
The first eight bytes of the dump should be zero, followed by some non-zero bytes. The first character in the ROM character table (BIOS character #0) is null (no bits on), which is eight bytes of **00**h. The next eight bytes are for BIOS character #1. The eight bytes for the ASCII character '**A**' (BIOS/ASCII character #41h) start **41h** table places after the start address, at **FFC76**. Dump this address. During an exam you could be asked to calculate the address of any letter in ROM, given its ASCII value and the start address of the table or the address of any other letter. Find a systematic way to calculate the 20-bit ROM memory address of any letter.

9. Write to the left of the leftmost 8x8 grid below the eight hex bytes that represent the bit pattern for the character '**A**', one byte per row, starting at the top row and working down to the bottom row. Shade or '**X**' the eight-bit patterns for each of the eight bytes in the grid to see how the letter '**A**' uses pixels.
10. Repeat the exercise for the letter '**K**', also giving its 20-bit ROM character table memory address:

Character: **A** Address: **FFC76**



Character: **K** Address: _____



11. What is the size in bytes of the BIOS character ROM table in decimal and hex? _____
12. What is the 20-bit address of the last byte in the BIOS character ROM table? _____
13. What character shape is stored right after the null at the start of the table? _____

Using DEBUG with Programs: Unassemble (Disassemble)

Debug will also allow us to view information about programs in memory. Programs start at offset **0100**h inside a segment because the first **100**h (256) bytes of memory are used as header information for the operating system to understand where the program will execute and what interrupts to call. Use **debug** to load into memory and dis-assemble (from machine code to assembly language) the start of the **command.com** program:

```
C:\>cd \windows
C:\windows>debug command.com
-U
0F96:0100 06            PUSH    ES
0F96:0101 17            POP     SS
0F96:0102 BE1B02        MOV     SI,021B
0F96:0105 BF1B01        MOV     DI,011B
0F96:0108 8BCE          MOV     CX,SI
0F96:010A F7D9          NEG     CX
0F96:010C FC            CLD
[... etc. ...]
```

You may not see the exact same code as displayed – it depends on what version of **command.com** is loaded into memory on your system.

The output above is the assembly language equivalent (an *assembly listing*) of the machine code at the start of the loaded **command.com** program. Memory addresses of the start of each instruction are on the far left in **segment:offset** form, followed by the machine-code byte contents of memory at those addresses, followed by the assembler mnemonics for the machine instructions. The bytes displayed to the left of the mnemonics are the bytes for the machine code of that instruction, e.g. the three bytes **BE1B02** starting at memory location **0F96:0102** are the machine code for the Intel instruction **MOV SI,021B**

Intel processors have variable-length instructions. Not all instructions use the same number of bytes. In the code above, note how the **PUSH**, **POP**, and **CLD** instructions take only a single byte each. The first two **MOV** instructions begin with bytes **BE** and **BF** and each instruction is three bytes long. Two of the three bytes are 16-bit addresses, which you can see stored in the memory dump, e.g. address **021B** is stored in memory as **1B02**. These 16-bit addresses appear to be stored backwards in memory!

14. Why is the two-byte address **021B** stored in memory as **1B** followed by **02**? _____

15. What is the hexadecimal value in memory (above) at location **0F96:0103**? _____

16. What is the hexadecimal value in memory (above) at location **0FA67**? _____

Using DEBUG with Programs: Assemble

In this section below are the steps to follow to write a small Intel assembly language program, assemble it, save it to disk, load it from disk and finally run it. This program uses built-in BIOS "interrupt" system routines to do most of the input/output work. We simply set some register values and then call the BIOS routine to do the I/O for us. There are many system interrupt routines used by DOS to carry out its activities; see any DOS manual (or search the Internet) for details.

A) Start **debug** and enter the Intel assembly language program given in the table below using the **debug** Assemble command (see below). The **comment** column is there to explain what the code does – you don't type it in. Your code segment value may not be the same as the example below; i.e., you may not see the same segment **0B55**, but you will see the same offsets, starting at **0100**. After you enter each line debug will assemble it to machine code (object code) in memory. Note how the address offset increments as each instruction is stored in memory. Note that not all instructions are the same length.

-A	***this column below is for comments – do not type these in***
0B55:0100 **MOV AH,00**	*set up for BIOS function 00H ("set video mode") for use by INT 10H*
0B55:0102 **MOV AL,03**	*set up for 80 x 25 text*
0B55:0104 **INT 10**	*call INT 10H which uses the BIOS routine to do the work*
0B55:0106 **MOV AH,09**	*set up function 09H (which means "display character and attribute")*
0B55:0108 **MOV BH,00**	*set up to display page 0*
0B55:010A **MOV AL,41**	*specify the character ASCII code to display (41H for 'A')*
0B55:010C **MOV CX,50**	*specify to repeat the character 50H (80 base ten) times*
0B55:010F **MOV BL,14**	*specify the character attribute byte: red character on blue background*
0B55:0111 **INT 10**	*call INT 10H which uses the BIOS routine to do the work*
0B55:0113 **INT 3**	*call a debug break to stop the program*
0B55:0114	*enter a blank line to end the assembly*
-U 100	*disassemble your program starting at offset 100 and check your typing*

B) Make sure your disassembly listing matches the above code. Note that this program used 14h bytes of memory. We will save these 14h bytes to a file by naming the file **letters.com** and telling **debug** how many bytes to write, then using the Write command. The Register command prompts you for input:

```
-N letters.com
-R BX
BX 0000
:0
-R CX
CX 0000
:14                <-- this is the number of bytes to write to the file that you named with N
-W
Writing 14 bytes
```

C) Make sure you have written 14 bytes to the file **letters.com**. Now, we will re-load memory from the file, check that the program is correct using a disassembly, and then execute your program:

```
-N letters.com
-L                 <-- this loads into memory the file that you named with N
-U 100             <-- disassemble your program and check your typing before you run it
-G
```

The running program will put a coloured string of 'A's across the top of the screen.

D) Use the Enter command to modify the above program in memory. (Do *not* retype the whole program!) Unassemble the program and locate the character attribute byte in memory. (Read the program comments to locate this byte.) Use the Enter command to change just that one byte to be "blue on red". Locate the ASCII letter '**A**' in the memory dump and change it to be '**B**'. After you have made these two changes, reset the instruction pointer register (the Intel name for PC or "program counter") back to the beginning of the program at **0100**h and re-run the program:

```
-R IP
IP 0113
:100
-G
```

17. (S-2) Demonstrate to your instructor that your program writes 80 blue '**B**'s on red: _____

E) Name and save the modified program under the name **letters2.com**. (This modified program will also be 14h bytes long.)

F) Run a file compare command to demonstrate that the only differences between **letters.com** and **letters2.com** are the two bytes you changed with the Enter command:

```
C:\>fc letters.com letters2.com
Comparing files letters.com and letters2.com
```
[... the bytes that are different output here ...]

18. (S-3) Demonstrate to your instructor that the two files differ by only two bytes: _____

Fun With DOS and Assembly Language (optional)

Use debug to modify two bytes in your **letters.com** program so that it can output 20 (or 255) happy face characters instead of letters.

Writing to Intel I/O Ports (optional)

The Intel processors have special instructions to do Input/Output (I/O). I/O port **61**h is the speaker port on most Intel PCs. It is a special I/O port address allocated to the speaker. If you send a byte out through this port you will activate or deactivate the speaker. You can use the **debug** Output command to do this:

```
-O 61 7
```
will sound the speaker

```
-O 61 0
```
will stop the sound

A Bye Bye Program (optional)

Type the following, which sets some memory and register values, and then executes the code found at **FFFF0**:

```
C:\>debug
-E 0040:0072 34 12
-R CS
:FFFF
-R IP
:0
-G
```