Memory Layout & Access Lab Manual, Chapter Four

To write the shortest and fastest possible assembly language programs you need to understand how the CPU references data in memory. The Intel 80x86 processor family provides a wide variety of memory addressing modes that allow efficient access to memory. Unless you master these addressing modes, you will not be able to write the most efficient programs.

The 80386 and later processors support an extended set of memory addressing modes. For those working on ‘386 (or greater) processors, the task is both simpler and more complex. On the one hand, the greater variety of addressing modes opens even more opportunity for optimization. On the other hand, the additional modes complicate the task of selecting the most appropriate mode.

In this laboratory you will experiment with the PC’s memory and study the various 80x86 addressing modes. You will also explore various high level language data types and their implementation in assembly language. To support these experiments, you will learn how to use the Microsoft CodeView™debugger, the 80x86 version of SIM886. Finally, you will begin writing real assembly language programs, assembling and linking them with the MASM 6.x assembler.

4.1 Debuggers and CodeView™

The SIM886 program is an example of a very simple debugging program. It should come as no surprise that there are several debugger programs available for the 80x86 as well. In this chapter you will learn the basic operation of the CodeView debugger. CodeView is a professional product with many different options and features. This short chapter cannot begin to describe all the possible ways to use the CodeView debugger. However, you will learn how to use some of the more common CodeView commands and debugging techniques.

One major drawback to describing a system like CodeView is that Microsoft constantly updates the CodeView product. These updates create subtle changes in the appearance of several screen images and the operation of various commands. One of the main reasons for putting the discussion of CodeView in this lab manual is to allow the timely update of this material on the CodeView debugger. As new versions of CodeView appear, updates to this chapter can reflect those changes. However, it’s quite possible that you’re using an older version of CodeView than the one described in this chapter, or this chapter describes an older version of CodeView than the one you’re using. Well, don’t let this concern you. The basic principles are the same and you should have no problem adjusting for version differences.

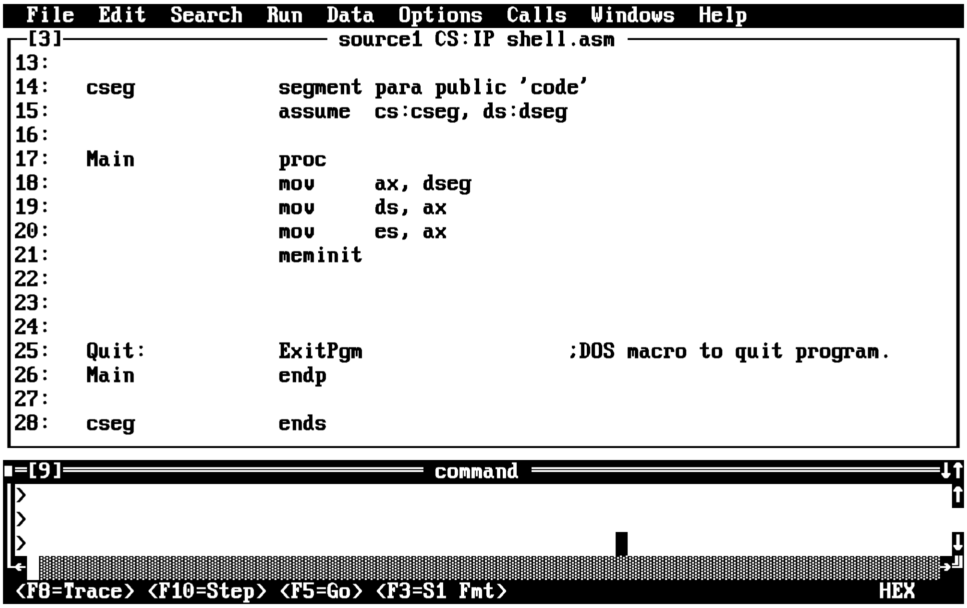
4.1.1 A Quick Look at CodeView

To run CodeView, simply type the following command at the DOS command line prompt:

```
c:> CV program.exe enter
```

Program.exe represents the name of the program you wish to debug (the “.exe” suffix is optional). CodeView requires that you specify a “.EXE” or “.COM” program name. If you do not supply an executable filename, CodeView will ask you to pick a file when you run it.

CodeView requires an executable program name as the command line parameter. Since you probably haven’t written an executable assembly language program yet, you haven’t got a program to supply to CodeView. To alleviate this problem, use the SHELL.EXE program found on the diskette accompanying this lab manual. To run CodeView using SHELL.EXE just use the command “CV SHELL.EXE”. This will bring up a screen which looks something like the following:

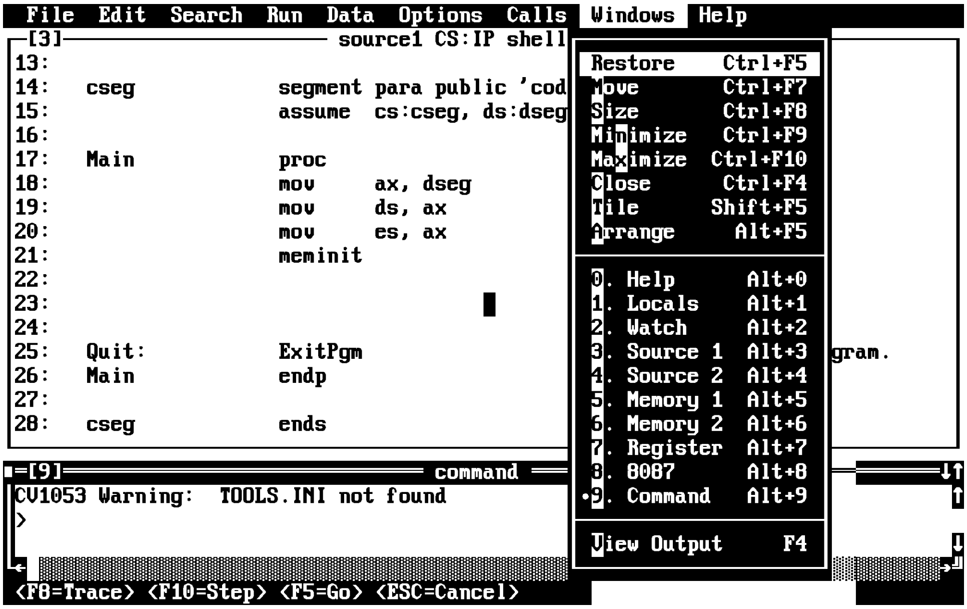

File Edit Search Run Data Options Calls Windows Help
[3] source1 CS:IP shell.asm
[9] command
<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt>
HEX

```

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit



Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends
```

```
>
>
>
```

There are four sections to the screen above: the *menu bar* on the first line, the *source1* window, the *command* window, and the help/ status line. Note that CodeView has many windows other than the two above. CodeView remembers which windows were open the last time it was run, so it might come up displaying different windows than those above. At first, the Command window is the active window. However, you can easily switch between windows by pressing the F6 key on the keyboard.

The windows are totally configurable. The Windows menu lets you select which windows appear on the screen. As with most Microsoft windowing products, you select items on the menu bar by holding down the alt key and pressing the first letter of the menu you wish to open. For example, pressing alt W opens up the Windows menu:



```
CV1053 Warning:  TOOLS.INI not found
>
```

4.1 What keystrokes would you use to bring up the “File” menu?

____

4.2 How do you switch between the windows in CodeView?

____

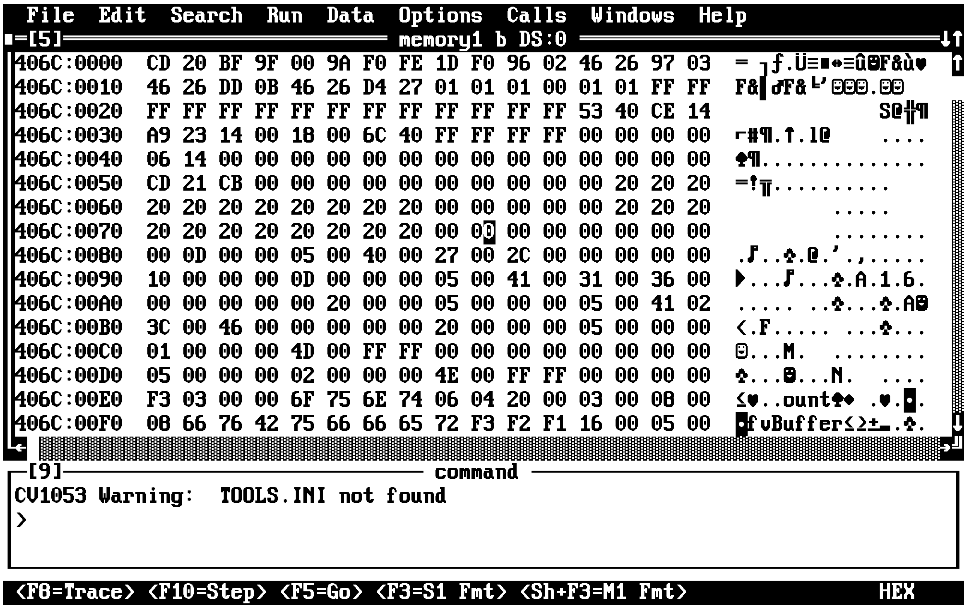
4.1.1.1 The Source Window

The Source1 and Source2 items let you open additional source windows. This lets you view, simultaneously, several different sections of the current program you're debugging. Source windows are useful for source level debugging. This chapter does not cover source level debugging. We will return to this feature in a later chapter.

4.1.1.2 The Memory Window

The Memory item lets you open a memory window. The memory windows lets you display and modify values in memory. By default, this window displays the variables in your data segment, though you can easily display any values in memory by typing their address.

The following is an example of a memory display:


File Edit Search Run Data Options Calls Windows Help
[5]
memory1 b DS:0
[9]
command
<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt> <Sh+F3=M1 Fmt>
HEX

```
406C:0000  CD 20 BF 9F 00 9A F0 FE 1D F0 96 02 46 26 97 03   ═ ┐ƒ.Ü≡■↔≡û☻F&ù♥
406C:0010  46 26 DD 0B 46 26 D4 27 01 01 01 00 01 01 FF FF   F&▌♂F&╘'☺☺☺.☺☺
406C:0020  FF FF FF FF FF FF FF FF FF FF FF FF 53 40 CE 14               S@╬¶
406C:0030  A9 23 14 00 18 00 6C 40 FF FF FF FF 00 00 00 00   ⌐#¶.↑.l@    ....
406C:0040  06 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ♠¶..............
406C:0050  CD 21 CB 00 00 00 00 00 00 00 00 00 00 20 20 20   ═!╦..........
406C:0060  20 20 20 20 20 20 20 20 00 00 00 00 00 20 20 20           .....
406C:0070  20 20 20 20 20 20 20 20 00 00 00 00 00 00 00 00           ........
406C:0080  00 0D 00 00 05 00 40 00 27 00 2C 00 00 00 00 00   .♪..♣.@.'.,.....
406C:0090  10 00 00 00 0D 00 00 00 05 00 41 00 31 00 36 00   ►...♪...♣.A.1.6.
406C:00A0  00 00 00 00 00 20 00 00 05 00 00 00 05 00 41 02   ..... ..♣...♣.A☻
406C:00B0  3C 00 46 00 00 00 00 00 20 00 00 00 05 00 00 00   <.F..... ...♣...
406C:00C0  01 00 00 00 4D 00 FF FF 00 00 00 00 00 00 00 00   ☺...M.  ........
406C:00D0  05 00 00 00 02 00 00 00 4E 00 FF FF 00 00 00 00   ♣...☻...N.  ....
406C:00E0  F3 03 00 00 6F 75 6E 74 06 04 20 00 03 00 08 00   ≤♥..ount♠♦ .♥.◘.
406C:00F0  08 66 76 42 75 66 66 65 72 F3 F2 F1 16 00 05 00   ◘fvBuffer≤≥±▬.♣.
```

```
CV1053 Warning:  TOOLS.INI not found
>
```

The values on the left side of the screen are the segmented memory addresses. The columns of hexadecimal values in the middle of the screen represent the values for 16 bytes starting at the specified address. Finally, the characters on the right hand side of the screen represent the ASCII characters for each of the 16 bytes at the specified addresses. Note that CodeView displays a period for those byte values that are not printable ASCII characters.

When you first bring up the memory window, it typically begins displaying data at offset zero in your data segment. There are a coup.le of ways to display different memory locations. First, you can use the PgUp and PgDn keys to scroll through memory[1]. Another option is to move the cursor over a segment or offset portion of an address and type in a new value. As you type each digit, CodeView automatically displays the data at the new address.

If you want to modify values in memory, simply move the cursor over the top of the desired byte's value and type a new hexadecimal value. CodeView automatically updates the corresponding byte in memory. This is quite a bit easier than SIM886's *Enter* command!

CodeView lets you open multiple Memory windows at one time. Each time you select Memory from the View memory, CodeView will open up another Memory window. With multiple

[1] Mouse users can also move the thumb control on the scroll bar to achieve this same result.

memory windows open you can compare the values at several non-contiguous memory locations on the screen at one time. Remember, if you want to switch between the memory windows, press the F6 key.

4.3 How would you tell CodeView to display the memory values starting at location 1000:0 in the memory window?

____

____

____

4.4 How would you modify the byte at location 1000:0 to store a zero there?

____

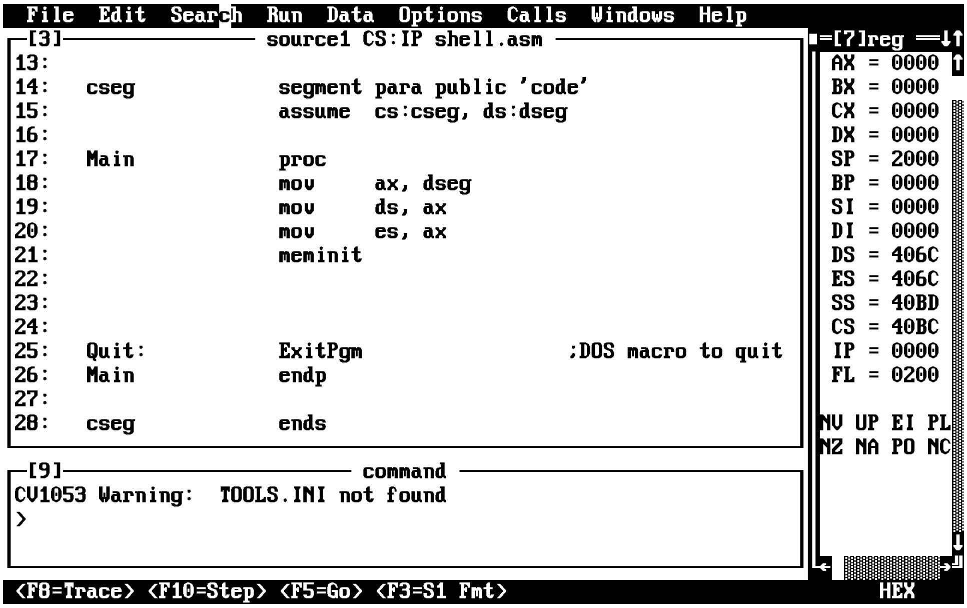
____

____

Pressing shift F3 toggles the data display mode between displaying hexadecimal bytes, ASCII characters, words, double words, integers (signed), floating point values, and other data types. This is useful when you need to view memory using different data types. You only have the option of displaying the contents of the entire window as a single data type; however, you can open multiple memory windows and display a different data type in each one.

4.1.1.3 The Register Window

The Register item in the Windows menu displays or hides the 80x86 registers window. This windows displays the current 80x86 register values:


File Edit Search Run Data Options Calls Windows Help
[3] source1 CS:IP shell.asm
[7]reg
[9] command
<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt>
HEX

```
cseg          segment para public 'code'
              assume  cs:cseg, ds:dseg

Main          proc
              mov     ax, dseg
              mov     ds, ax
              mov     es, ax
              meminit



Quit:         ExitPgm                  ;DOS macro to quit
Main          endp

cseg          ends
```

```
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = 2000
BP = 0000
SI = 0000
DI = 0000
DS = 406C
ES = 406C
SS = 40BD
CS = 40BC
IP = 0000
FL = 0200

NV UP EI PL
NZ NA PO NC
```

```
CV1053 Warning:  TOOLS.INI not found
>
```

To change the value of a register, activate the register window (using F6, if it is not already selected) and move the cursor over the value you wish to change. Type a new value over the desired register's existing value. Note that FL stands for *flags*. You can change the values of the flags in the flags register by entering a new value after the FL= entry. Another way to change the flags is to move the cursor over one of the flag entries at the bottom of the register window and press an alphabetic key (e.g., "A") on the keyboard. This will toggle the specified flag. The flag values are (0/1): overflow=(OV/NV), direction=(DN/UP), interrupt=(DI/EI), sign=(PL/NG), zero=(NZ/ZR), auxiliary carry=(NA/AC), parity=(PO/PE), carry=(NC/CY).

Note that pressing the F2 key toggles the display of the registers window. This feature is quite useful when debugging programs. The registers window eats up about 20% of the display and tends to obscure other windows. However, you can quickly recall the registers window, or make it disappear, by simply pressing F2.

4.1 alt f

4.2 Press F6

4.5 Describe how to modify a register value.

4.1.1.4 The Command Window

The **Command** window lets you type SIM886 style commands into CodeView. Although almost every command available in the command window is available elsewhere, many operations are easier done in the command window. Furthermore, you can generally execute a sequence of completely different commands in the command window faster than switching between the various other windows in CodeView. The operation of the command window will be the subject of the next section in this chapter.

4.1.1.5 The Output Menu Item

Selecting **View Output** from the **Windows** menu (or pressing the F4 key) toggles the display between the CodeView display and the current program output. While your program is actually running, CodeView normally displays the program's output. Once the program turns control over to CodeView, however, the debugging windows appear obscuring your output. If you need to take a quick peek at the program's output while in CodeView, the F4 key will do the job.

4.1.2 The CodeView Command Window

CodeView is actually two debuggers in one. On the one hand, it is a modern window-based debugging system with a nice mouse-based user interface. On the other hand, it can behave like a traditional command-line based debugger (e.g., SIM886). The command window provides the key to this split personality. If you activate the command window, you can enter debugger commands from the keyboard. The following are some of the more common CodeView commands you will use:

A *address*	Assemble
BC *bp_number*	Breakpoint Clear
BD *bp_number*	Breakpoint Disable
BE *bp_number*	Breakpoint Enable
BL	Breakpoint List
BP *address*	Breakpoint Set
D *range*	Dump Memory
E	Animate execution
Ex *Address*	Enter Commands (x= " ", b, w, d, etc.)
G {*address*}	Go (address is optional)
H *command*	Help
I *port*	Input data from I/O port
L	Restart program from beginning

`MC` *range address*	Compare two blocks of memory
`MF` *range data_value(s)*	Fill Memory with specified value(s)
`MM` *range address*	Copy a block of memory
`MS` *range data_value(s)*	Search memory range for set of values
`N` *Value*$_{10}$	Set the default radix
`O` *port value*	Output value to an output port
`P`	Program Step
`Q`	Quit
`R`	Register
`R`*xx value*	Set register *xx* to *value*
`T`	Trace
`U` *address*	Unassemble statements at *address*

In this chapter we will mainly consider those commands that manipulate memory. Execution commands like the breakpoint , trace, and go commands appear in a later chapter. Of course, it wouldn't hurt for you to learn some of the other commands, you may find some of them to be useful.

4.1.2.1 The Radix Command (N)

The first command window command you must learn is the RADIX (base selection) command. By default, CodeView works in decimal (base 10). This is very inconvenient for assembly language programmers so you should always execute the radix command upon entering CodeView and set the base to hexadecimal. To do this, use the command

```
N 16
```

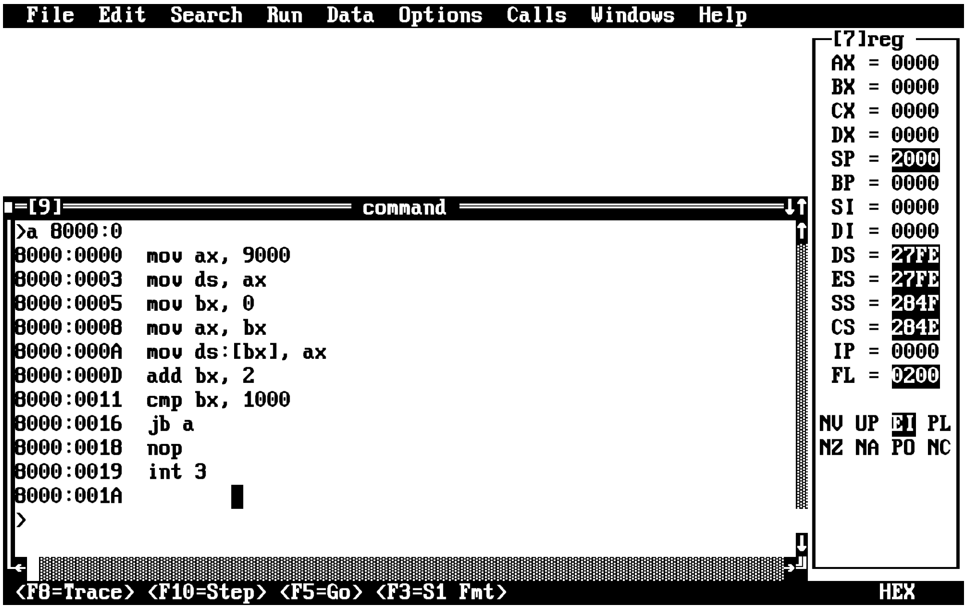
4.1.2.2 The Assemble Command

The CodeView command window **Assemble** command works in a fashion not unlike the SIM886 assemble command. The command uses the syntax:

`A` *address* enter

Address is the starting address of the machine instructions. This is either a full segmented address (*ssss:oooo*, *ssss* is the segment, *oooo* is the offset) or a simple offset value of the form *oooo*. If you supply only an offset, CodeView uses CS' current value as the segment address.

After you press enter, CodeView will prompt you to enter a sequence of machine instructions. Pressing enter by itself terminates the entry of assembly language instructions. The following is an example from inside CodeView:


File Edit Search Run Data Options Calls Windows Help
[9] command
[7]reg
<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt>
HEX

```
>a 8000:0
8000:0000  mov ax, 9000
8000:0003  mov ds, ax
8000:0005  mov bx, 0
8000:0008  mov ax, bx
8000:000A  mov ds:[bx], ax
8000:000D  add bx, 2
8000:0011  cmp bx, 1000
8000:0016  jb a
8000:0018  nop
8000:0019  int 3
8000:001A
>
```

```
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = 2000
BP = 0000
SI = 0000
DI = 0000
DS = 27FE
ES = 27FE
SS = 284F
CS = 284E
IP = 0000
FL = 0200

NV UP EI PL
NZ NA PO NC
```

(This code uses a few 80x86 instructions we've haven't discussed yet. The next chapter will discuss them.)

The Assemble command is one of the few commands available *only* in the command window. Apparently, Microsoft does not expect programmers to enter assembly language code into memory using CodeView. This is not an unreasonable assumption since CodeView is a a high level language source level debugger.

In general, the CodeView Assemble command is useful for quick *patches* to a program, but it is no substitute for MASM 6.x. Any changes you make to your program with the assemble command will not appear in your source file. It's very easy to correct a bug in CodeView and forget to make the change to your original source file and then wonder why the bug is still in your code.

4.6 Suppose you've just entered the code above and you want to see the hexadecimal opcodes generated by the assembler for these instructions. Describe how you could do this:

____

____

4.1.2.3 The Compare Memory Command

The Memory Compare command will compare the bytes in one block of memory against the bytes in a second block of memory. It will report any differences between the two ranges of bytes. This is useful, for example, to see if a program has initialized two arrays in an identical fashion or to compare two long strings. The compare command takes the following forms:

```
MC start_address end_address second_block_address enter
MC start_address L length_of_block second_block_address enter
```

The first form compares the bytes from memory locations *start_address* through *end_address* with the data starting at location *second_block_address*. The second form lets you specify the size of the blocks rather than specify the ending address of the first block. If CodeView detects any differences in the two ranges of bytes, it displays those differences and their addresses. The following are all legal compare commands:

```
MC 8000:0 8000:100 9000:80 enter
MC 8000:100 L 20 9000:0 enter
MC 0 100 200 enter
```

The first command above compares the block of bytes from locations 8000:0 through 8000:100 against a similarly sized block starting at address 9000:80 (i.e., 9000:80..180).

The second command above demonstrates the use of the "L" option which specifies a length rather than an ending address. In this example, CodeView will compare the values in the range 8000:0..8000:1F (20h/32 bytes) against the data starting at address 9000:0.

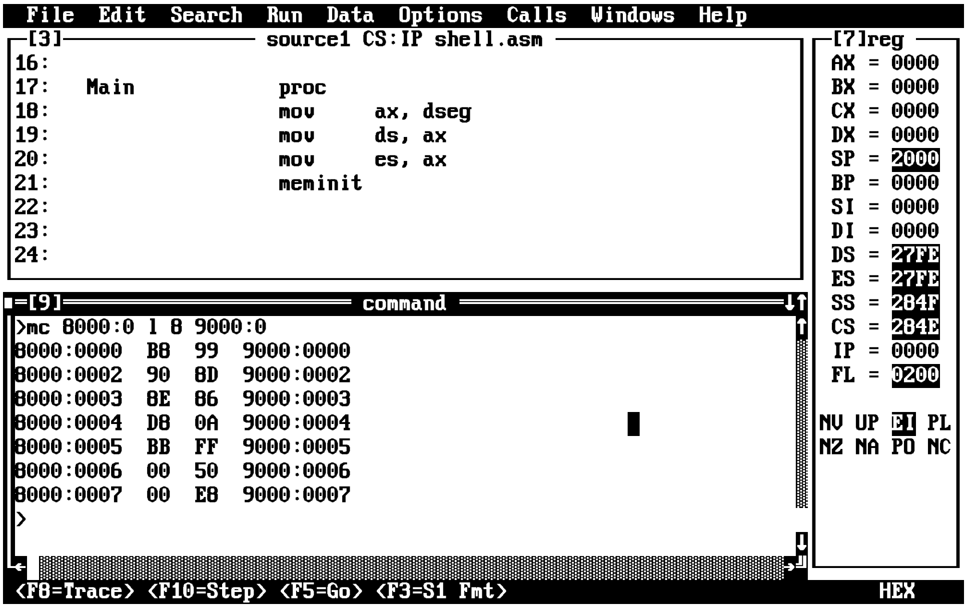
The third example above shows that you needn't supply a full segmented address for the *starting_address* and *second_block_address* values. By default, CodeView uses the data segment (DS:) if you do not supply a segment portion of the address. Note, however, that if you supply a starting and ending address, they must both have the same segment value; you must supply the same segment address to both or you must let both addresses default to DS' value.

If the two blocks are equal, CodeView immediately prompts you for another command without printing anything to the command window. If there are differences between the two blocks of bytes, however, CodeView lists those differences (and their addresses) in the command window:

4.3 Type "1000" over the segment portion of an address in the memory window, then type "0" over the offset portion of an address in the memory window. CodeView will now display the contents of memory starting at address 1000:0.

4.4 Move the cursor over the data byte after the 1000:0 address and type the new value over the old one.

4.5 Move the cursor over the register value you want to change and type the new value.


File Edit Search Run Data Options Calls Windows Help
[3] source1 CS:IP shell.asm
[7]reg
[9] command
<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt>
HEX

```
Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit
```

```
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = 2000
BP = 0000
SI = 0000
DI = 0000
DS = 27FE
ES = 27FE
SS = 284F
CS = 284E
IP = 0000
FL = 0200

NV UP EI PL
NZ NA PO NC
```

```
>mc 8000:0 l 8 9000:0
8000:0000  B8  99  9000:0000
8000:0002  90  8D  9000:0002
8000:0003  8E  86  9000:0003
8000:0004  D8  0A  9000:0004
8000:0005  BB  FF  9000:0005
8000:0006  00  50  9000:0006
8000:0007  00  E8  9000:0007
>
```

In the example above, memory locations 8000:0 through 8000:200 were first initialized to zero. Then locations 8000:10 through 8000:1E were set to 1, 2, 3, ..., 0Fh. Finally, the **Memory Compare** command compared the bytes in the range 8000:0…8000:FF with the block of bytes starting at address 8000:100. Since locations 8000:10…8000:1E were different from the bytes at locations 8000:110…8000:11E, CodeView printed their addresses and differences.

4.7 Give the complete command to compare the 1000h bytes beginning at address F00:100 against the block of 1000h bytes at address 2000:0

____

4.8 Give the command to compare the bytes at address DS:100h through DS:1FFh with the bytes at address DS:200h

____

4.1.2.4 The Dump Memory Command

The Dump command lets you display the values of selected memory cells. The Memory window in CodeView also lets you view (and modify) memory. However, the Dump command is sometimes more convenient, especially when looking at small blocks of memory.

The Dump command takes several forms, depending on the type of data you want to display on the screen. This command typically takes one of the forms:

D *starting_address ending_address* enter

D *starting_address* L *length* enter

By default, the dump command displays 16 hexadecimal and ASCII byte values per line (just like the Memory window).

There are several additional forms of the Dump command that let you specify the display format for the data. However, the exact format seems to change with every version of CodeView. For example, in CodeView 4.10, you would use commands like the following:

```
DA address_range         Dump ASCII characters
DB address_range         Dump hex bytes/ASCII (default)
DI address_range         Dump integer words
DIU address_range        Dump unsigned integer words
DIX address_range        Dump 16-bit values in hex
```

`DL` *`address_range`*	`Dump 32-bit integers`
`DLU` *`address_range`*	`Dump 32-bit unsigned integers`
`DLX` *`address_range`*	`Dump 32-bit values in hex`
`DR` *`address_range`*	`Dump 32-bit real values`
`DRL` *`address_range`*	`Dump 64-bit real values`
`DRT` *`address_range`*	`Dump 80-bit real values`

4.6 Open a memory window and set the display address to the same address as your code.

You should probably check the help associated with your version of CodeView to verify the exact format of the memory dump commands. Note that some versions of CodeView allow you to use **MDxx** for the memory dump command.

4.9 Which command above displays REAL4 variables?

____

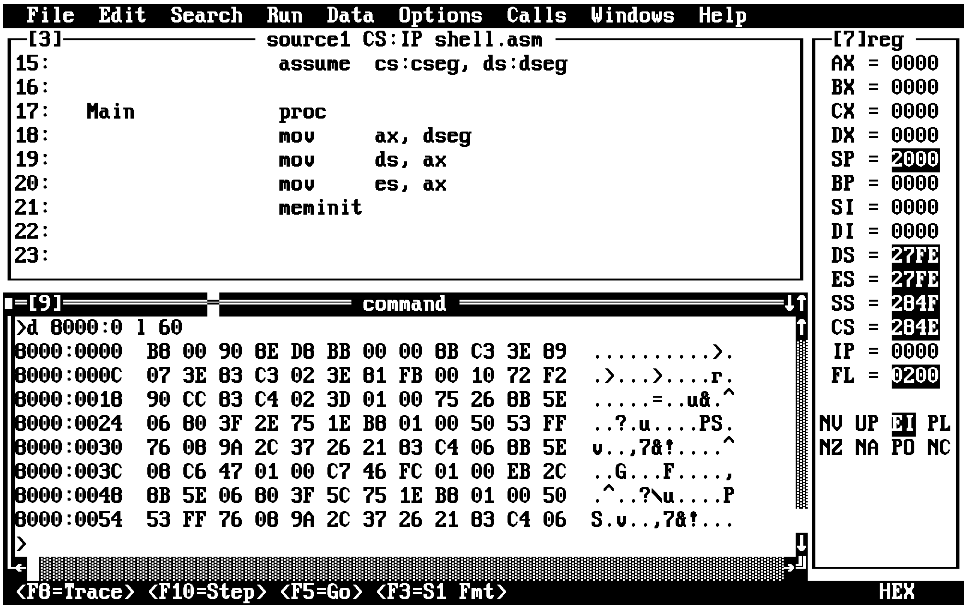
4.10 Which command above displays REAL10 variables?

____

4.11 Which command above displays a string?

____

Once you execute one of the above commands, the "D" command name displays the data in the new format. The "DB" command reverts back to byte/ASCII display. The following figure provides an example of these commands:


File Edit Search Run Data Options Calls Windows Help
[3] source1 CS:IP shell.asm
[7]reg
[9] command
<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt>
HEX

```
                    assume  cs:cseg, ds:dseg

        Main        proc
                    mov     ax, dseg
                    mov     ds, ax
                    mov     es, ax
                    meminit
```

```
>d 8000:0 l 60
8000:0000  B8 00 90 8E D8 BB 00 00 8B C3 3E 89  ..........>.
8000:000C  07 3E 83 C3 02 3E 81 FB 00 10 72 F2  .>...>....r.
8000:0018  90 CC 83 C4 02 3D 01 00 75 26 8B 5E  .....=..u&.^
8000:0024  06 80 3F 2E 75 1E B8 01 00 50 53 FF  ..?.u....PS.
8000:0030  76 08 9A 2C 37 26 21 83 C4 06 8B 5E  v..,7&!....^
8000:003C  08 C6 47 01 00 C7 46 FC 01 00 EB 2C  ..G...F....,
8000:0048  8B 5E 06 80 3F 5C 75 1E B8 01 00 50  .^..?\u....P
8000:0054  53 FF 76 08 9A 2C 37 26 21 83 C4 06  S.v..,7&!...
>
```

```
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = 2000
BP = 0000
SI = 0000
DI = 0000
DS = 27FE
ES = 27FE
SS = 284F
CS = 284E
IP = 0000
FL = 0200

NV UP EI PL
NZ NA PO NC
```

If you enter a dump command without an address, CodeView will display the data immediately following the last dump command. This is sometimes useful when viewing memory.

4.1.2.5 The Enter Command

The CodeView Memory windows lets you easily display and modify the contents of memory. From the command window it takes two different commands to accomplish these tasks: Dump to display memory data and Enter to modify memory data. For most memory modification tasks, you'll find the memory windows easier to use. However, the CodeView **Enter** command handles a few tasks easier than the Memory window.

Like the Dump command, the **Enter** command lets you enter data in several different formats. The commands to accomplish this are

```
EA-       Enter data in ASCII format
EB-       Enter byte data in hexadecimal format
ED-       Enter double word data in hexadecimal format
EI-       Enter 16-bit integer data in (signed) decimal format
EIU-      Enter 16-bit integer data in (unsigned) decimal format.
EIX-      Enter 16-bit integer data in hexadecimal format.
EL-       Enter 32-bit integer data in (signed) decimal format
ELU-      Enter 32-bit integer data in (unsigned) decimal format.
ELX-      Enter 32-bit integer data in hexadecimal format.
ER-       Enter 32-bit floating point data
ERL-      Enter 64-bit floating pont data
ERT-      Enter 80-bit floating point data
```

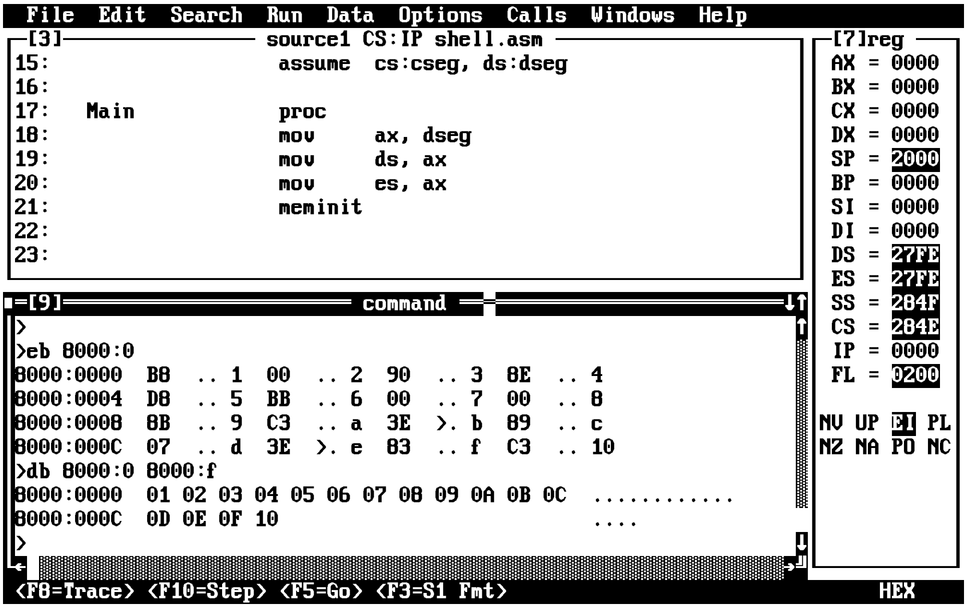
Like the Dump command, the syntax for this command changes regularly with different versions of CodeView. Be sure to use CodeView's help facility if these commands don't seem to work. *MExx* is a synonym for *Exx* in CodeView.

Enter commands take two possible forms:

```
Ex starting_address enter
Ex starting_address list_of_values enter
```

The first form above is the *interactive* ***Enter*** *command*. Upon pressing the key, Codeview will display the starting address and the data at that address, then prompt you to enter a new value for that location. Type the new value followed by space and CodeView will prompt you for the value for the next location; typing space by itself skips over the current location; typing enter or a value terminated with enter terminates the interactive Enter mode. Note that the EA command does not let you enter ASCII values in the interactive mode. It behaves exactly like the EB command during data entry.

The second form of the Enter command lets you enter a sequence of values into memory a single entry. With this form of the Enter command, you simply follow the starting address with the list of values you want to store at that address. CodeView will automatically store each value into successive memory locations beginning at the starting address. You can enter ASCII data using this form of **Enter** by enclosing the characters in quotes. The following figure demonstrates the Enter command:


File Edit Search Run Data Options Calls Windows Help
[3] source1 CS:IP shell.asm
[9] command
[7]reg
<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt>
HEX

```
                    assume  cs:cseg, ds:dseg

     Main           proc
                    mov     ax, dseg
                    mov     ds, ax
                    mov     es, ax
                    meminit
```

```
>
>eb 8000:0
8000:0000  B8  .. 1  00  .. 2  90  .. 3  8E  .. 4
8000:0004  D8  .. 5  BB  .. 6  00  .. 7  00  .. 8
8000:0008  8B  .. 9  C3  .. a  3E  >. b  89  .. c
8000:000C  07  .. d  3E  >. e  83  .. f  C3  .. 10
>db 8000:0 8000:f
8000:0000  01 02 03 04 05 06 07 08 09 0A 0B 0C  ............
8000:000C  0D 0E 0F 10                          ....
>
```

```
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = 2000
BP = 0000
SI = 0000
DI = 0000
DS = 27FE
ES = 27FE
SS = 284F
CS = 284E
IP = 0000
FL = 0200

NV UP EI PL
NZ NA PO NC
```

4.12 Besides using the Enter command, describe another method you can use to modify the contents of memory

There are a couple of points concerning the Enter command of which you should be aware. First of all, you cannot use "E" as a command by itself . Unlike the Dump command, this does not mean "begin entering data after the last address." Instead, this is a totally separate command (Animate). The other thing to note is that the current display mode (ASCII, byte, word, double word, etc.) and the current entry mode are not independent. Changing the default display mode to word also changes the entry mode to word, and vice versa.

4.7 MC F00:100 L 1000h 2000:0

The command window Enter command is particularly useful for entering small lists of values or ASCII strings, especially at different points throughout memory. When dealing with values in a small area of memory, working in the Memory window is probably easier.

4.13 Suppose the string starting at address 1000:108 was misspelled "Attacked" rather than "Attached". Describe how to fix this with the Enter command

4.8 MC DS:100 DS:1FF DS:200

4.14 How could you enter the REAL8 value "1.25" into memory?

4.15 If you use an "EW" command to enter a two-byte value, what data type would the "D" command display if used afterwards?

4.1.2.6 The Fill Memory Command

4.9 DR

The Enter command and the Memory window let you easily change the value of individual memory locations, or set a range of memory locations to several different values. If you want to clear an array or otherwise initialize a block of memory locations so that they all contain the same values, the Memory Fill command provides a better alternative.

The Memory Fill command uses the following syntax:

```
MF starting_address ending_address values enter
MF starting_address L block_length values enter
```

The Memory Fill command fills memory locations *starting_address* through *ending_address* with the byte values specified in the *values* list. The second form above lets you specify the block length rather than the ending address.

4.10 DRT

The *values* list can be a single value or a list of values. If *values* is a single byte value, then the Memory Fill command initializes all the bytes of the memory block with that value. If *values* is a list of bytes, the Fill command repeats that sequence of bytes over and over again in memory. For example, the following command stores 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5... to the 256 bytes starting at location 8000:0

```
F 8000:0 L 100 1 2 3 4 5
```

Unfortunately, the Fill command works only with byte (or ASCII string) data. However, you can simulate word, doubleword, etc., memory fills breaking up those other values into their component bytes. Don't forget, though, that the L.O. byte always comes first.

4.11 DA (or DB)

4.16 Suppose you have an array of 256 double words starting at address 1000:200. How could you initialize each element of this array to zero using the Fill command?

4.17 What does the (legal) Fill command do?

F 1000:0 L 100 “Hi There”

____

4.1.2.7 The Move Memory Command

This Command window operation copies data from one block of memory to another. This lets you copy the data from one array to another, move code around in memory, reinitialize a group of variables from a saved memory block, and so on. The syntax for the Memory Move command is as follows:

```
MM starting_address ending_address destination_address enter
MM starting_address L block_length destination_address enter
```

If the source and destination blocks overlap, CodeView detects this and handles the memory move operation correctly.

4.18 Suppose you have a short 132h byte routine sitting at location FF0:124 that you would like to move to location 8000:0. What command will do this?

____

4.19 What command could you use to verify that the 132h bytes you copied from location FF0:124 properly appear at address 8000:0?

____

4.1.2.8 The Input Command

The Input command lets you read data from one of the 80x86’s 65,536 different input ports. The syntax for this command is

```
I port_address enter
```

where *port_address* is a 16-bit value denoting the I/O port address to read. The input command reads the byte at that port and displays its value.

Note that it is not a wise idea to use this command with an arbitrary address. Certain devices activate some functions whenever you read one of their I/O ports. By reading a port you may cause the device to lose data or otherwise disturb that device.

Note that this command only reads a single byte from the specified port. If you want to read a word or double-word from a given input port you will need to execute two successive Input operations at the desired port address and the next port address.

This command appears to be broken in certain versions of CodeView (e.g., 4.01).

4.20 The LPT1: status input port is at I/O address 3F9h. What is the command to display the byte at this port?

____

4.1.2.9 The Output Command

The Output command is complementary to the Input command. This command lets you output a data value to a port. It uses the syntax:

```
O port_address output_value enter
```

4.12 Use the memory window

Output_value is a single byte value that CodeView will write to the output port given by *port_address.*

Note that CodeView also uses the “O” command to set options. If it does not recognize a valid port address as the first operand it will think this is an Option command. If the Output command doesn’t seem to be working properly, you’ve probably switched out of the assembly language mode (CodeView supports BASIC, Pascal, C, and FORTRAN in addition to assembly language) and the port address you’re entering isn’t a valid numeric value in the new mode. Be sure to use the **N 16** command to set the default radix to hexadecimal before using this command!

4.21 Output port 3F8 is where you store data being output to a printer. What is the command to store the character “A” (ASCII code 41h) to this port?

4.13 EA 1000:10D “h”

____

4.22 Reading this port returns the last value sent to the printer. What command would you use to read the LPT1: data port?

____

4.1.2.10 The Quit Command

Pressing Q (for Quit) terminates the current debugging session and returns control to MS-DOS. You can also quit CodeView by selecting the Exit item from the File menu.

4.14 ERL *address* 1.25

4.1.2.11 The Register Command

The CodeView Register command lets you view and change the values of the registers. To view the current values of the 80x86 registers you would use the following command:

```
R enter
```

This command displays the registers and disassembles the instruction at address CS:IP.

You can also change the value of a specific register using a command of the form:

```
Rxx enter
  -or-
Rxx = value
```

4.15 It will display word values.

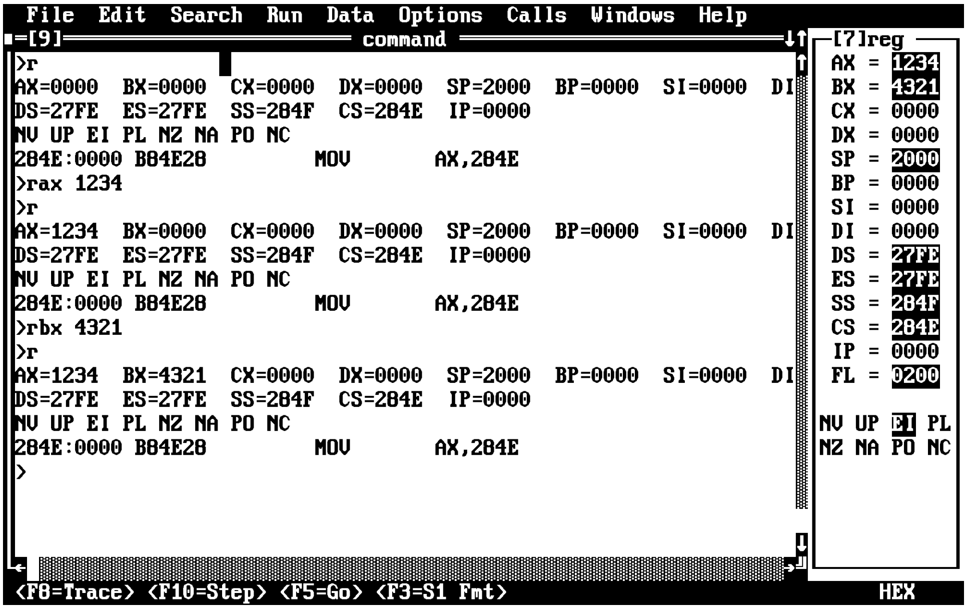
where *xx* represents one of the 80x86’s register names: AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, ES, SS, IP, or FL. The first version (“R*xx* enter”) displays the specified register and then prompts you to enter a new value. The second form of this command above immediately sets the specified register to the given value.

4.23 There is another way to view the 80x86’s registers. What is it?

____

4.16
MF 1000:200 L 400 0
or
MF 1000:200 1000:5ff 0

The following figure demonstrates the operation of these register commands:


File Edit Search Run Data Options Calls Windows Help
[9] command
[7]reg
<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt>
HEX

```
>r
AX=0000  BX=0000  CX=0000  DX=0000  SP=2000  BP=0000  SI=0000  DI
DS=27FE  ES=27FE  SS=284F  CS=284E  IP=0000
NV UP EI PL NZ NA PO NC
284E:0000 B84E28        MOV      AX,284E
>rax 1234
>r
AX=1234  BX=0000  CX=0000  DX=0000  SP=2000  BP=0000  SI=0000  DI
DS=27FE  ES=27FE  SS=284F  CS=284E  IP=0000
NV UP EI PL NZ NA PO NC
284E:0000 B84E28        MOV      AX,284E
>rbx 4321
>r
AX=1234  BX=4321  CX=0000  DX=0000  SP=2000  BP=0000  SI=0000  DI
DS=27FE  ES=27FE  SS=284F  CS=284E  IP=0000
NV UP EI PL NZ NA PO NC
284E:0000 B84E28        MOV      AX,284E
>
```

```
AX = 1234
BX = 4321
CX = 0000
DX = 0000
SP = 2000
BP = 0000
SI = 0000
DI = 0000
DS = 27FE
ES = 27FE
SS = 284F
CS = 284E
IP = 0000
FL = 0200

NV UP EI PL
NZ NA PO NC
```

4.1.2.12 The Unassemble Command

The Command window Unassemble command works just like it's SIM886 counterpart. It will disassemble a sequence of instructions at an address you specify, converting the binary machine codes into (barely) readable machine instructions. The basic command uses the following syntax:

```
U address
```

Note that you must have a source window open for this instruction to operate properly!

In general, the Unassemble command is of little use because the Source window lets you view your program at the source level (rather than at the disassembled machine language level). However, the Unassemble command is great for disassembling BIOS, DOS, TSRs, and other code in memory.

4.1.3 CodeView Function Keys

CodeView uses the function keys on the PC's keyboard for often-executed operations. The following table gives a brief description of the use of each function key.

Table 10: Function Key Usage in CodeView

Function Key	Alone	shift	Ctrl	alt
F1	Help	Help contents	Next Help	Prev Help
F2	Register Window			
F3	Source Window Mode	Memory Window Mode		
F4	Output Screen		Close Window	

Table 10: Function Key Usage in CodeView

Function Key	Alone	shift	Ctrl	alt
F5	Run			
F6	Switch Window	Prev Window		
F7	Execute to cursor			
F8	Trace	Prev History	Size window	
F9	Breakpoint			
F10	Step instrs, run calls.	Next History	Maximize Window	

The F3 function key deserves special mention. Pressing this key toggles the source mode between *machine language* (actually, disassembled machine language), *mixed*, and *source*. In source mode (assuming you've assembled your code with the proper options) the source window displays your actual source code. In mixed mode, CodeView displays a line of source code followed by the machine code generated for that line of source code. This mode is primarily for high level language users, but it does have some utility for assembly language users as you'll see when you study macros. In *machine mode*, CodeView ignores your source code and simply disassembles the binary opcodes in memory. This mode is useful if you suspect a bug in MASM (they do exist) and you're not sure than MASM is assembling your code properly.

4.1.4 Some Comments on CodeView Addresses

The examples given for addresses in the previous sections are a little misleading. You could easily get the impression that you have to enter an address in hexadecimal form, i.e., *ssss:oooo* or *oooo*. Actually, you can specify memory addresses in many different ways. For example, if you have a variable in your assembly language program named *MyVar*, you could use a command like

```
D Myvar
```

to display the value of this variable[2]. You do not need to know the address, nor even the segment of that variable. Another way to specify an address is via the 80x86 register set. For example, if ES:BX points at the block of memory you want to display, you could use the following command to display your data:

```
D ES:BX
```

CodeView will use the current values in the ES and BX registers as the address of the block of memory to display. There is nothing magical about the use of the registers. You can use them just like any other address component. In the example above, ES held the segment value and BX held the offset— very typical for an 80x86 assembly language program. However, CodeView does not require you to use legal 80x86 combinations. For example, you could dump the bytes at address CX:AX using the dump command

```
D CX:AX
```

The use of 80x86 registers is not limited to specifying source addresses. You can specify destination addresses and even lengths using the registers:

```
D CX:AX L BX ES:DI
```

4.17 Fills 256 (100h) bytes with the characters "Hi There" repeated over and over starting at address 1000:0

4.18 MM FF0:124 L 132 8000:0

4.19 MC FF0:124 L 132 8000:0

4.20 I 3F9

4.21 O 3F8 "A"
or O 3F8 41

4.22 I 3F8

4.23 Use the CodeView Register window.

[2] This requires that you assemble your program in a very special way, but we're getting to that.

Of course, you can mix and match the use of registers and numeric addresses in the same command with no problem:

```
D CX:AX L 100 8000:0
```

You can also use complex arithmetic expressions to specify an address in memory. In particular, you can use the addition operator to compute the sum of various components of an address. This works out really neat when you need to simulate 80x86 addressing modes. For example, if you want to see which byte is at address 1000[bx], you could use the command:

```
D BX+1000 L 1
```

To simulate the [BX][SI] addressing mode and look at the word at that address you could use the command:

```
DIX BX+SI L 1
```

The examples presented in this section all use the Dump command, but you can use this technique with any of the CodeView commands. For more information concerning what constitutes valid CodeView address, as well as a full explanation of allowable expression forms, please consult the CodeView on-line help system.

4.24 What CodeView command will let you see the value that would be loaded into AX by the instruction "MOV AX, CS:4[bx][si]"?

____

4.25 What command would you use to simulate the instruction "MOV word ptr ES:2[DI], 0" from inside CodeView?

____

4.1.5 A Wrap on CodeView

We're not through discussing CodeView by any means. In particular, we've not discussed the execution, single stepping, and breakpoint commands which are crucial for debugging programs. We will return to these subjects in later chapters. Nonetheless, we've covered a considerable amount of material, certainly enough to deal with most of the experiments in this laboratory exercise. As we need those other commands, this manual will introduce them.

Of course, there are two additional sources of information on CodeView available to you— the section on CodeView in the "Microsoft Macro Assembler Programmer's Guide" and the on-line help available inside CodeView. In particular, the on-line help is quite useful for figuring out how a specific command works inside CodeView.

4.2 Segmented Addressing on the 80x86

The 80x86 processor family uses a powerful memory management scheme called *segmented addressing*. Segmented addresses consist of two components: a *segment* and an *offset*. Segmented addressing lets you organize your program into logically distinct components. Rather than placing all your variables and code into one spot, you can group related variables and code sections into different segments. This will make your programs easier to understand and maintain.

The only problem is that the 80x86 CPUs address memory as a linear array of bytes and they must convert segment:offset *logical* addresses into single-dimension *linear* addresses. When operating in real address mode, the 80x86 translates logical addresses into physical addresses by multiplying the segment value by 16 and then adding in the offset.

Some examples:

Table 11: Logical To Physical Address Translation

Logical Address	Computation	Physical Address
1000:100	1000***0***+100	10100
1008:80	1008***0***+80	10100
1010:0	1010***0***+0	10100
100C:40	100C***0***+40	10100
10:8000	10***0***+8000	8100
8000:10	8000***0***+10	80010

When using hexadecimal numbers the logical to physical address computation is almost trivial. All you need to do is stick a zero on the end of the segment value and add the result to the offset.

4.26 Convert 800:8000 to a physical address:

4.27 Convert 1234:5678 to a physical address:

4.28 Convert ABCD:1234 to a physical address:

An 80x86 program never really sees a physical address. Physical addresses are quantities appearing on the address bus. A program never manipulates them. Why then should you worry about physical addresses? To understand, take a look at the table again. Note that the first four addresses, though they are obviously different, all map to the same physical address. Whenever you store a data value into memory it winds up at some physical address. Two different variables, even if they have different logical addresses, are one and the same if they have the same physical address. For example, if variable I appears in memory at location 1000:10 and variable J appears in memory at location 1001:0, storing a value into I will wipe out J's value and vice versa since they are both located at the same physical address. Remember, it's the physical address where the CPU actually stores the data. That's why you need to know how to convert logical to physical addresses, so you can avoid overwriting the value of one variable with another simply because their logical addresses happen to map to the same physical address.

To see if one logical address is an *alias* of another (that is, they share the same memory location(s)), simply convert both logical addresses to physical addresses. If their physical addresses are the same, they are indeed aliases of one another.

4.29 Several of the following addresses are aliases of one another. Separate them into groups of aliases: 2000:2000 1000:8000 3800:0 2080:1800 2200:0 3000:8000 1FFF:2010 1800:a000 2F00:9000

4.3 Normalized Addresses on the 80x86

The possibility of aliases is only one reason why you have to worry about physical addresses. Comparing pointers is another big problem facing software designers on the 80x86. When comparing pointers it would be nice if they were equal if they pointed at the same object. When comparing segmented addresses it's quite possible to have two pointers which are not equal (that is, their bit patterns are not the same) yet they point at the same physical address. When the necessity to compare pointers arises, most programmers use a special canonical, or *normalized*, form for addresses. To normalize a logical address is very easy— just convert it to its physical address then stick a colon between the fourth and fifth digits of the 20-bit physical address:

Table 12: Normalizing an Address

Logical Address	Physical Address	Normalized Address
1000:100	10100	1010:0
1008:80	10100	1010:0
1010:0	10100	1010:0
100C:40	10100	1010:0
10:8000	8100	810:0
8000:10	80010	8001:0

As you can see, if your pointers contain normalized addresses they will contain the same bit patterns if they point at the same object.

4.30 Convert 8203:3583 to a normalized address:

____

4.4 Memory Addressing Modes on the 80x86

The 80x86 family supports 17 different memory addressing modes which can be easily broken down into about five different groups: displacement-only, register indirect, indexed, based indexed, and based indexed plus displacement.

The easiest addressing mode to understand is the displacement only addressing mode. Instructions using this addressing mode consist of an opcode (one or more bytes) followed by a 16-bit constant. This constant *is* the offset into the data segment for that instruction[3].

Examples:

Assume I is at offset 110h and J is at offset 112h in the data segment. Assume that K is at offset 100h in the extra segment.

```
mov ax, DS:I        ;Fetches AX from memory location DS:110h
mov DS:J, ax        ;Stores AX into memory location DS:112h
mov bx, ES:K        ;Loads BX from location ES:100h
```

4.31 Assume that DS contains 1000h and ES contains 2000h. What *physical addresses* do the three instructions above reference?

____

[3] Unless explicitly overridden, the displacement addressing mode always provides an offset into the Data Segment.

4.32 What would the normalized versions of those addresses be?

4.24 DW CS:BX+SI+4

The displacement addressing mode is very easy to understand; computing the *effective address* of the operand is trivial — the effective address is given to you. The only catch is that most of the time that you're using this addressing mode you're using symbolic names like "I" and "J" so you won't actually know the displacement (the assembler determines this value for you). Of course, you don't really *need* to know the displacement value in most cases, so it hardly matters.

4.25 EW ES:DI+2 0

Next up on the complexity scale is the register indirect addressing mode. The register indirect addressing modes include [BX], [BP], [SI], and [DI][4]. Instructions using these addressing modes do not encode an offset into the instruction. Instead, they consist of an opcode byte and a MOD-REG-R/M byte. These addressing modes compute their effective addresses by using the values in the specified register. By default, the [BX], [DI], and [SI] addressing modes use the data segment; the [BP] addressing mode uses the *stack* segment.

Examples:

4.26 10000

Assume BX=100h, SI=200h, DI=300h, BP=400h; DS=1000h, ES=2000h, and SS=3000h

```
mov ax, [bx]            ;Loads ax from location DS:100h
mov bx, [si]            ;Loads bx from location DS:200h
mov ss:[di], bx         ;Stores bx to location SS:300h
mov al, es:[bp]         ;Loads al from location ES:400h
```

4.33 What physical addresses do each of the above instructions access?

4.27 179B8

4.34 Assuming there were no segment override prefixes on the last two instructions above, which segment would each of the above instructions reference by default?

The indexed addressing mode is the next level of complexity. It is a combination of the displacement-only and register indirect addressing modes. The instruction encoding typically consists of an opcode byte, a MOD-REG-R/M byte, and a one or two byte constant displacement. The CPU computes the effective address by adding the (two's complement signed) displacement value with the value of the register. Possible addressing modes include disp[bx], disp[di], disp[si], and disp[bp]. MASM also allows you to enter these addressing modes as [bx+disp], [di+disp], [si+disp], and [bp+disp].

4.28 ACF04

The disp[bp]/[bp+disp] addressing modes use the stack segment by default (if you don't supply a segment override); the other addressing modes use the data segment by default. The offset into the given segment is simply the sum of the register and the displacement. Displacements in the range -128…+127 require only one byte to encode in the instruction, displacements in the range ±32k require two bytes. Therefore, instructions using displacements in the range -128..+127 will be shorter (and therefore faster). The CPU sign extends these shorter displacements when computing the effective address.

4.29
(2000:2000, 2080:1800,
2200:0, 1FFF:2010,
1800:A000)
(3800:0, 3000:8000,
2F00:9000)
(1000:8000)

Examples: Assume DS=1000h, ES=2000h, SS=3000h, BX=10h, SI=-8, DI=100h, and BP=300h. Each of the following addressing modes generate the specified effective address (EA) and the corresponding displacement is one or two bytes long, as noted:

```
1[bx]              EA is 11h; one-byte disp.
10h[si]            EA is 8, one-byte displacement.
```

[4] Technically, "[BP]" turns out to be an indexed addressing mode, but we'll ignore that technicality for now.

```
[di-80h]          EA is 80h; one-byte displacement.
100h[bp]          EA is 400h; two-byte displacement.
```

4.35 What is the effective address of each of the following addresses?

1000h[bx]____________________ **80h[si]___________________________**
SS:1[di]______________________ **[bp-100h]_________________________**

4.36 What physical address will the 80x86 generate for each of the above (assume real mode)?

____

4.37 What is the size of the displacement operand in each of the above addressing modes?

____

____

The based indexed addressing modes compute their effective address as the sum of a base register (BX or BP) and an index register (DI or SI). As is usually the case, the [bx][di] and [bx][si] addressing modes provide offsets into the data segment, the [bp][di] and [bp][si] addressing modes compute offsets into the stack segment. The effective address is just the sum of the two registers.

The based indexed plus displacement addressing modes add a displacement along with two registers to obtain the effective address. The following examples use the register values from the previous examples:

```
1[bx][si]         EA is 19h; one-byte disp.
10h[bp][si]       EA is 308, one-byte displacement.
[bx][di-80h]      EA is 90h; one-byte displacement.
100h[bp][di]      EA is 500h; two-byte displacement.
```

4.38 What is the effective address of each of the following addresses?

1000h[bx][si]______________ **80h[bp][si]________________**
SS:1[bx][di]________________ **[bp-100h][di]______________**

To easily remember all the valid 80x86 addressing modes, you need only memorize the following chart:

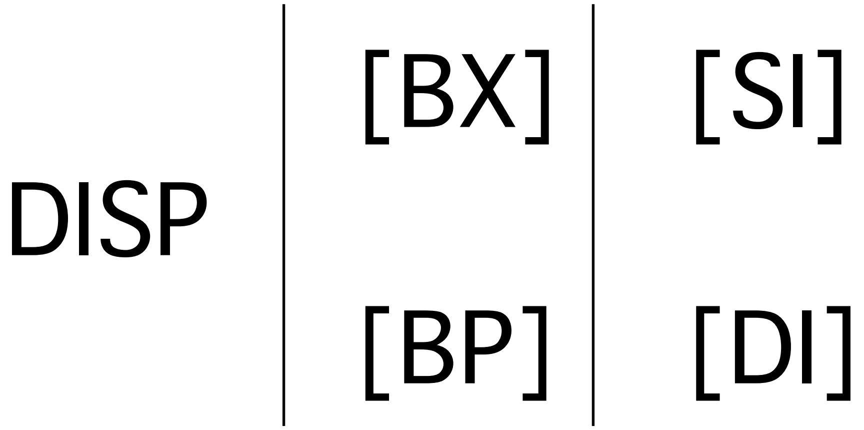


Table to Generate Valid 8086 Addressing Modes

If you choose zero or one items from each of the columns above and wind up with at least one item, you've got a valid 8086 addressing mode. Conversely, if you have an addressing mode which cannot be constructed by choosing zero or one items from each column above, then that addressing mode is illegal.

Note that all addressing modes involving [bp] on the 8086 use the stack segment by default. All other memory addressing modes use the data segment by default.

Examples of legal addressing modes:

1[bx] [bp] [si] [bx] 10h[bp][si] ds:[5] (displacement only) [bp+2] [di][bx] 10[bx][di]

Examples of illegal 8086 addressing modes:

[dx] [si][di] [bp+bx] [ax][bx] [bx][bx] [bx][si][di]

4.39 Which of the following addressing modes are *illegal*?

[bp][si+2], [si][bx] 2[di] [si][di] [bp] [bx][si] [si-4] [bp][bx]

____

____

4.40 For each of the legal memory addressing modes above, what is the default segment that the 8086 will use when accessing memory?

____

____

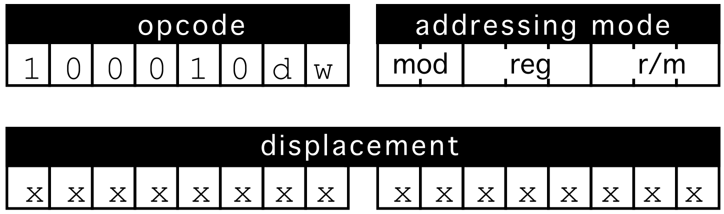
____

4.5 The 80x86 MOV Instruction

The 80x86 MOV instruction copies data from one location to another. The syntax for this instruction is

```
MOV    Dest, Source
```

There are several restrictions on the source and destination operands. The most common form of this instruction moves data between two registers or between a memory location and a register. That instruction uses the following encoding:



note: displacement may be zero, one, or two bytes long

Generic MOV Instruction

The fields of this instruction have the following meanings:

W: Selects whether this is an eight-bit (w=0) or 16/32-bit (w=1) MOV operation[5]

D: Selects the *direction* of the MOV operation.

4.30 855B:3

4.31
DS:I = 10110h
DS:J = 10112h
ES:K = 20100h

4.32
DS:I = 1011:0
DS:J = 1011:2
ES:K = 2010:0

4.33
DS:[bx] = 10100h
DS:[si] = 10200h
SS:[di] = 30300h
ES:[bp] = 20400h

4.34 DS, SS

[5] A different mechanism differentiates 16 and 32-bit moves.

If D=0, the MOV instruction copies data from the **reg** operand to the **mod-r/m** operand. If D=1 then the MOV instruction copies the data from the **mod-r/m** operand to the **reg** operand.

The **REG** field selects one of the 80x86's registers using the following bit patterns:

Table 13: REG Bit Encodings

reg	w=0	16-bit mode w=1	32-bit mode w=1
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

The **MOD** and **R/M** fields select an operand with the following encodings:

Table 14: MOD Encoding

MOD	Meaning
00	The r/m field denotes a register indirect memory addressing mode or a base/indexed addressing mode (see the encodings for r/m) *unless* the r/m field contains 110. If MOD=00 and r/m=110 the mod and r/m fields denote displacement-only (direct) addressing.
01	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is an eight-bit signed displacement following the mod/reg/rm byte.
10	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is a 16-bit signed displacement (in 16-bit mode) or a 32-bit signed displacement (in 32-bit mode) following the mod/reg/rm byte .
11	The r/m field denotes a register and uses the same encoding as the *reg* field

Table 15: R/M Field Encoding

R/M	Addressing mode (Assuming MOD=00, 01, or 10)
000	[BX+SI] or DISP[BX][SI] (depends on MOD)
001	[BX+DI] or DISP[BX+DI] (depends on MOD)
010	[BP+SI] or DISP[BP+SI] (depends on MOD)
011	[BP+DI] or DISP[BP+DI] (depends on MOD)
100	[SI] or DISP[SI] (depends on MOD)
101	[DI] or DISP[DI] (depends on MOD)
110	Displacement-only or DISP[BP] (depends on MOD)
111	[BX] or DISP[BX] (depends on MOD)

To see how to use these tables, consider the instruction:

```
MOV AX, BX
```

The first byte of the instruction is the opcode "10000001" or 81h[6]. The second (and final) byte of the instruction is the **mod-reg-r/m** byte. Since the D field of the opcode is zero, the **reg** field of the **mod-reg-r/m** byte is the source operand. In the instruction above, BX is the source operand. From the tables above you can see that the **reg** encoding for BX is 011. The destination operand is a register as well. Hence, the **mod** field must be 11 and the **r/m** field must be 000 denoting the AX register. This produces a **mod-reg-r/m** byte of 11011000 or D8h. Therefore, the first byte of this MOV instruction is 81h and the second byte is D8h[7]

4.41 There is a second, distinct, instruction which also copies BX in AX. This instruction sets the D field of the opcode to one to produce 83h. What would the corresponding mod-reg-r/m byte be?

____

4.42 What are the two encodings for the "`MOV BX, AX`" instruction?

____

This same basic opcode format lets the 80x86 processors move data between registers and memory as well. If the **mod** field is not "11" then the **mod** and **r/m** fields specify a memory location rather than a register. For example, the "`MOV AX, [BX]`" instruction is built as follows:

- The opcode is 83h since we're moving 16-bit data into a register (that is, a sixteen bit register is the **reg** operand).
- The **mod** field is 00 since there is a zero byte displacement associated with this instruction and it specifies a memory addressing mode ([BX] really means 0[BX] and this is a special addressing mode form which does not require a displacement byte on the 80x86).
- The **r/m** field is 111 which specifies the disp+BX addressing mode.
- The **reg** field is 000 which specifies the AX register as the destination.

4.35 1000h[bx]=1010h,
80h[si]=78h,
ss:1[di]=101h,
[bp-100h]=200h

4.36 1000h[bx]=11010h,
80h[si]=10078h,
ss:1[di]=30101h,
[bp-100h]=30200h

4.37 1000h[bx]=2 bytes,
80h[si]=1 byte,
ss:1[di]=1 byte,
[bp-100h]=2 bytes

4.38 1000h[bx]=1008h,
80h[si]=378h,
ss:1[di]=111h,
[bp-100h]=300h

4.39 [si][di] & [bp][bx]

4.40
[bp][si+2] = SS
2[di] = DS
[bp] = SS
[bx][si] = DS
[si-4] = DS
[bp][di] = SS

[6] You'll soon see that 83h would work as well.

[7] Treated as a 16-bit value, this would normally be written D881h. However, when dealing with instruction opcodes, most people write the first byte first followed by additional bytes separated by spaces; e.g., 81h D8h.

This yields the complete instruction 83h, 07h.

Note that there are actually *three* different instructions which will load the AX register with the word whose offset is given by the value in the BX register. Although the 80x86 provides a special "displacement+BX" addressing mode which doesn't require any displacement bytes, there is nothing to prevent you from encoding the instruction with one or two displacement bytes. You would, for example, specify a **mod** value of 01 and include a single byte of zero as the displacement . This produces the instruction bytes 83h, 43h, 00h which would have the same effect as the previous sequence. Of course, this instruction sequence is longer and might execute slower, so it's not a good choice in most cases.

4.43 What is the *third* way to construct this same instruction sequence? Provide the opcode bytes to accompany your explanation.

____

____

____

Please note that the 80x86's *displacement only* addressing mode really corresponds to the [BP] addressing mode. It turns out that programs rarely use the [BP] addressing mode, yet they use the displacement only addressing mode all the time. If you really need a [BP] addressing mode, simply use the 0[BP] addressing mode.

4.44 What is the encoding for the "`MOV BP, DS:[25h]`" instruction?

____

4.45 What is the encoding for the "`MOV BP, [BP]`" instruction?

____

The 80x86 processor uses special encodings for MOV instructions which copy immediate data (constants) into registers or memory locations, to copy data between a memory location or register into and a segment register, or to load the accumulator from the memory location specified by the displacement only addressing mode. There are also special forms of the MOV instruction on the 80386 and later processors which allow you to move data between 32-bit registers and memory locations. Please see the textbook for more details on the encodings of these instructions.

4.6 Memory Organization Laboratory Exercises

In this laboratory you will examine how the 80x86 family organizes values in memory. You will also create several data structures in memory and examine them with the CodeView debugger. Finally, you will also assemble and link some very simple assembly language programs and load them into memory with the CodeView debugger.

4.6.1 Before Coming to the Laboratory

Your pre-lab report should contain the following:

- A copy of this lab guide chapter with all the questions answered and corrected.
- A write-up on the CodeView debugger explaining, in your own words, how the following commands work in CodeView: A, D, E (Enter), F, G, I (input), M (Move), O (Output), Q, R, T, and U.
- A write-up explaining how the MOV instruction works.

See Chapter Two of this laboratory manual for an example pre-lab report.

Note: your Teaching Assistant or Lab Instructor may elect to give a quiz before the lab begins on the material covered in the laboratory. You will do quite well on that quiz if you've properly prepared for the lab and studied up on the stuff prior to attending the lab. If you simply copy the material from someone else you will do poorly on the quiz and you will probably not finish the lab. Do not take this pre-lab exercise lightly.

4.41 11000011 or C3

4.6.2 Questions for the Laboratory

4.46 **What are two different binary encodings for the "MOV AX, BX" instruction?**

____

4.47 **What are three different binary encodings for the "MOV AX, 0[BX]" instruction?**

____

4.42 (81h, C3h) & (83h, D8h)

4.6.3 Laboratory Exercises

In this laboratory you will perform the following activities:

- Demonstrate the use of the CodeView debugger and many of the commands in the debugger
- Demonstrate the operation of the 8086 MOV, LEA, LES, ADD, and MUL instructions and addressing modes.
- Enter several 8086 machine language programs into the CodeView and single step through the programs to execute them.
- Use the debugger to modify the operation of the programs.
- Examine memory locations using CodeView and explore the memory organization of the 8086
- Explore the various encodings of 8086 instructions.

□ Exercise 1: Demonstrate the use of A, D, E, MF, MM, R, and U CodeView command window debugging commands (obviously you should use the A command before the U command to create an arbitrary machine language program several instructions long). Capture screen output to create a log of your work (use the Print option in the File menu to capture screen output to a file or the printer).

For your lab report: Describe the operation of each of the above commands. Include the printout of each command demonstration in your lab report. Describe how to use the Go and Trace commands.

For additional credit: Figure out how to resize windows in CodeView. Include a description of this in your lab report.

□ Exercise 2: Set the DS register to 8000h and assemble (using the CodeView “A” command) the following short program that adds the values in locations DS:0 and DS:2 together and stores the result at DS:4. Assemble the code starting at location 7000:0. Single step through the program using the CodeView “T” command. Comment on the results.

```
mov     ax, ds:[0]
add     ax, ds:[2]
mov     ds:[4], ax
int     3               ;Terminates execution
```

For your lab report: Describe what happens after the execution of each of the above statements for different values of DS:0 and DS:2. Be sure to describe all the register and flag values.

For additional credit: Use the CodeView Go command from the command window to execute the code above. Include a screen dump in your lab report.

□ Exercise 3: Generate two different instruction byte sequences for the “MOV AX, BX” instruction. Enter these bytes into memory starting at location 7000:0 using the CodeView Enter command. Use the CodeView “U” command to disassemble the bytes you’ve entered into memory. Comment on the results. Generate *three* different encodings for the “MOV AX, 0[BX]” instruction (by varying the displacement size) and enter them into memory starting at location 7000:10. Unassemble these instructions to verify proper encoding. After entering the above, unassemble the code starting at location 7000:11. Explain the results.

For your lab report: Include the screen dump for the Enter, Dump, and Unassemble commands. Describe what they mean.

4.43 Have a MOD value of 10 and a two-byte displacement: 83h, 83h, 00h, 00h

4.44 8bh, 2eh, 25h, 00h

4.45 8bh, 6eh, 00h

4.46 81h, D8h and 83h, C3h

4.47 (81h, 03h),
(81h, 43h, 00h), and
(81h, 83h, 00h, 00h)

For additional credit: Find several other instructions that do the same thing but have different opcodes. Enter their opcodes into memory and verify that they disassemble to comparable instructions. Include screen dumps and a description in your lab report.

❑ Exercise 4: Using the CodeView Assemble command, enter the following sequence of instructions starting at location 7000:0 which load values into the AL register using each of the 8086's 17 different memory addressing modes. Set DS and SS to 8000h and load BP, DI, SI, and BX with two, four, six, and eight, respectively. Initialize memory locations 8000:0 through 8000:f with the values 0..F. Single step through the following 17 MOV instructions and explain the results.

```
mov     al, ds:[0]
mov     al, [bx]
mov     al, [si]
mov     al, [di]
mov     al, [bp]
mov     al, 2[bx]
mov     al, 2[si]
mov     al, 2[di]
mov     al, 2[bp]
mov     al, [bx][si]
mov     al, [bx][di]
mov     al, [bp][si]
mov     al, [bp][di]
mov     al, 2[bx][si]
mov     al, 2[bx][di]
mov     al, 2[bp][si]
mov     al, 2[bp][di]
int     3
```

For your lab report: Capture the trace of the above instructions to the printer or an output file. Include this trace in your lab report and hand comment exactly what happened on the execution of each instruction.

For additional credit: The addressing modes that use BP access the stack segment rather than the data segment. In the exercise above you've set SS and DS so they both point at the same segment. Modify SS so that it contains 9000 rather than 8000. Step through the above code again and see what happens.

❑ Exercise 5: Store a 16-bit value (using the CodeView "EW" command) at location 8000:20. Display the value at location 8002:0 using the "DW" command.. Enter a different value at location 8002:0 and then display the value at location 8000:20.

For your lab report: Explain the results, include screen dumps.

For additional credit: Choose some other segmented addresses whose physical address corresponds to the physical address of the above two values. Modify those addresses and capture the results. Include these screen captures in your lab report and explain the results.

❑ Exercise 11: Connect the circuit you built for labs two and three to the parallel printer port on your computer. Use the following output command to write a value to the LEDs on your circuit:

O *port   value*

Port is the base address of the parallel port to which you've connected your circuit. To determine the appropriate port address, dump memory locations 40:8 through 40:d. The first two locations (40:8 and 40:9) contain the base I/O address for LPT1:. The second two locations (40:a and 40:b) contain the base I/O address for LPT2:. The last pair of locations contain the base address for LPT3:. Typical addresses are 378h, 278h, and 3BCh. If a zero appears in one of these words, then the system did not recognize the associated device. Be sure to select the appropriate port value for your connection.

The *value* operand is the value to write to the printer port. For example, if you've connected your circuit to LPT1: and it is at I/O address 378h (i.e., the word at location 40:8 contains 378h), you can turn off all the LEDs with the command **O 378 0.** Likewise, you can turn on all the LEDs using the command **O 378 ff**. Try turning on each of the LEDS individually with a set of eight "O" commands.

For your lab report: Describe the use of this command to turn on and off various LEDs on your circuit.

4.7 Programming Projects

- ❏

4.8 Answers to Selected Exercises

1) 80x86 instruction typically use the value in the DS or SS register to supply the segment component of the address.

3) a) 11000 e) 0:0 (10000:0 wraps to 0:0) i)2110f

4) a) 1100:0 e) 0:0 i) 2110:f

6) [reg], disp[reg], disp[reg*n], [reg][reg], [reg][reg*n], disp[reg][reg], disp[reg][reg*n]
where reg is one of {eax, ebx, ecx, edx, esi, edi, ebp, esp] and n = 1, 2, 4, or 8.

8) The following are *some* possible examples. This list certainly does not list all the possibilities.
a) Accessing registers (e.g., local or temporary variables).
b) Global variables
c) Constants
d) Accessing data through pointers
e) Accessing elements of an array
f) Accessing a elements of an array when you have a pointer to the array
g) Accessing an element of an array of structures (and a field of that element)
h) Accessing elements of an array.

11) a) 89 d8 f) c6 07 02

15) Two good reasons are the fact that (in real mode) the mov ax, [ebx] instruction is longer and runs slower.

18) al, ah, bl, bh, cl, ch, dl, dh

22) EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP

26) FS and GS.