


LAB 2

THE MEMORY DISPLAY & BASIC CONTROL FLOW USING JECXZ

Directions. This builds on Lab 1. *If you need a copy of Lab 1 for reference (e.g., how to copy the sample project, or how to read/write unsigned integers), a PDF is posted in Canvas.* As before, questions to answer are marked with a  symbol. You should be able to complete pages 1–4 today; turn them in at the end of class. Submit your solution to the programming portion on page 5 in Canvas.

1. EXPLORING DATA IN THE MEMORY WINDOW

```
INCLUDE Irvine32.inc

.DATA
firstMsg    BYTE "The memory address of this string is:", 0dh, 0ah, 0
secondMsg   BYTE "h", 0
mysteryMsg  BYTE 0dh, 0ah, "B", 121, 65h, 00000000b
moreBytes   BYTE "The End", 0

.CODE
main PROC
    ; Write "The memory..." followed by CRLF
    mov edx, offset firstMsg
    call WriteString

    ; Write the memory address of firstMsg in hexadecimal
    mov eax, offset firstMsg
    call WriteHex    ; Displays the 32-bit value in EAX in hexadecimal

    ; Write "h" (no trailing CRLF)
    mov edx, offset secondMsg
    call WriteString

    ; Write the mystery message
    mov edx, offset mysteryMsg
    call WriteString
    exit
main ENDP
END main
```

- ☐ **Enter the above program in Visual Studio.** Start the same way you started Lab 1: make a copy of the Project_sample project and rename it. Then, open the project in Visual Studio, open the *main.asm* file, delete everything in that file, and type the code above instead. (You can omit the comments when you type it.)
- ☐ **Set a breakpoint on the line that reads “mov edx, offset mysteryMsg”.**
- ☐ **Start debugging.** The debugger will stop at the breakpoint.

- ☐ **Switch to your running program, at look at the output.** At this point, the output should be something like the following:

```
The memory address of this string is:  
00405000h
```

- ☐ **❓ Write the number that your program printed here:** _____
- ☐ **Switch back to Visual Studio, and open the Memory window.**
Click Debug > Windows > Memory > Memory 1.
- ☐ **The Memory window contains a text box labeled Address. In that text box, type `&firstMsg` and then press Enter.**
- ☐ **Notice that the Address box now displays the same hexadecimal number that your program displayed two steps ago.** This is a 32-bit number describing the location in memory where the first byte of the string “firstMsg” is stored.

Now, the Memory window displays the bytes in your computer’s memory (RAM), starting with the first character in *firstMsg*. On the left side of the Memory window, byte values are displayed in hexadecimal. On the right side, the ANSI characters for the same bytes are displayed, for byte values that correspond to printable characters in the ANSI character set (recall from lecture that ANSI characters 0–127 are the same as ASCII).

- ☐ **Notice that the byte values in the Memory window exactly match the bytes defined in the .data section of your code.** The first character in *firstMsg* is a capital *T*, which has ASCII code 84 = 54h, so the first value displayed in the Memory window is “54”. The next character is a lowercase *h*, which is ASCII 104 = 68h, so the second value displayed is “68”, and so forth.
- ☐ Recall from lecture that strings are NUL-terminated, i.e., a string consists of all of the characters up to the first byte with value 0. **In the Address box in the Memory window, type `&mysteryMsg` to display the bytes starting from the first character of the mystery message.** Now, can you guess what will be displayed when the mystery message is printed?



_____ (make a guess... it’s OK if you’re wrong here... I’d really like you to guess...)

- ☐ **Allow the program to terminate. What is the mystery message that it prints?**



2. EXPLORING INSTRUCTIONS IN THE MEMORY WINDOW

```
INCLUDE Irvine32.inc

.code
main PROC
    nop
    mov eax, 12345678h
    mov al, 0ABh
    add al, 1
    nop
    exit
main ENDP
END main
```

- ☐ **Refer to Activity 1b, and translate the following assembly language instructions to machine language, writing the values in hexadecimal.**

<u>Assembly Language</u>	<u>Machine Language</u>
nop	_____
mov eax, 0	(you don't know how to translate this yet)
mov al, 0ABh	_____
add al, 1	_____
nop	_____

- ☐ **Enter the above program in Visual Studio.** You can overwrite the program from the previous part of this lab, or you can make a new copy of Project_sample and modify it, as you did last time.
- ☐ **Set a breakpoint on the first “nop” line—the very first instruction.**
- ☐ **Start debugging.** The debugger will stop at the breakpoint.
- ☐ **Open the Registers window, if it is not already open.** If you forgot how, review Lab 1.
- ☐ **🔍 What value does the EIP register contain (in hex)?** _____

You will need this value later, so here's a star to remind you where it's at. ★

Remember: EIP is the instruction pointer. It contains the memory address of the next instruction that will execute. Since the debugger stopped at the breakpoint on the first instruction (nop), that instruction has not executed yet. So, EIP contains the memory address where the first instruction is stored in memory.

- ☐ **Use the debugger's Step Over command to execute the first `nop` instruction. The debugger will stop at next instruction (the second `nop`).** (If you forgot how to Step Over, refer to Lab 1.)

When you click Step Over, the processor executed one iteration of the fetch-decode-execute cycle for your program. Afterwards, EIP contains the memory address of the *next* instruction to execute—`mov eax, 0`.

- ☐ **❓ What value does the EIP register contain (in hex)?** _____
This is the address where the `mov eax, 0` instruction is stored.

- ☐ **Step Over again.**

- ☐ **❓ What value does the EIP register contain (in hex)?** _____
This is the address where the next instruction (`mov al, 0ABh`) is stored.

- ☐ Compare your last two answers—the last two values of EIP you wrote down.

- ☐ **❓ After the processor fetched the `mov eax, 0` instruction, how much did it increase the value in EIP by?** _____

You'll need to remember this answer, so here's an umbrella to remind you: ☂

- ☐ **Open the Memory window, if it is not already open.** If you forgot how, look at the previous page.
- ☐ **In the Address box in the Memory window, enter `0xffffffff` but replace `ffffffff` with the hex value you wrote down from the starred question ★ above.** For example, if you wrote 1234ABCD above, you'd enter 0x1234ABCD.

Remember: the value you wrote for that question was the memory address where the *first* instruction in your assembly language program was stored in memory.

- ☐ **Compare the byte values displayed in the Memory window to the machine language codes that you wrote at the beginning of this exercise.** The first byte should be 90h (the machine language translation of `nop`). After that is the translation of `mov eax, 12345678h` (which you didn't know how to translate above). After that is the translation of `mov al, 0ABh`. And so forth. Now, you know the translations of all the instructions except for one, but you can figure it out from the contents of the Memory view. So...

- ☐ **❓ How is `mov eax, 12345678h` translated into machine language?**

(To verify that you're correct: The number of bytes needed to encode this instruction should be the same as your answer to ☂ above. Do you know why?)

3. EXERCISE: COUNTING

In this exercise, you will need to use the `jmp` and `je` instructions. Refer to Friday's slides if you forgot how they work. You will also need to read and write unsigned integers; refer to Lab 1 if you forgot how to do that.

❓ Write an assembly language program that does the following. Submit your `.asm` file in Canvas.

1. Read a nonnegative integer from the keyboard. (We'll call this integer n .)
2. The user is not allowed to enter zero. If n is 0, go back to step 1.
3. Display the first n positive integers, where n is the input from the user.

For example, if the user inputs 5, the program should display "12345".

Important: Don't ever try to write an entire program at once! Write a few lines, then run your program (or step through it in the debugger, watching the Registers window) to make sure it does, in fact, do what you think it does. Then add a few more lines, and run it again. Add a few lines, and run it again. If something breaks, but you only changed 3 lines of code since it was working last, it's easy to figure out where the problem is.