Name: ______________________________

LAB 1

GETTING STARTED WITH X86 ASSEMBLY LANGUAGE AND MICROSOFT VISUAL STUDIO 2010/MASM

Directions. Actions you must take are marked with a box. A ? symbol indicates questions you are to answer; answer them in the space provided. If you complete this activity today, turn in this paper at the end of class. Otherwise, complete it on your own time, and turn it in at the start of the next class.

1. COPYING THE SAMPLE PROJECT

☐ **Open your local documents folder (C:\Users\*yourid*\Documents).** Press Windows-R to open the Run dialog. Type Documents in the text field, and click OK; this will open the Documents folder in an Explorer window. Click in the address bar at the top of the window, and make sure it reads C:\Users\yourid\Documents. *It is important that you store your files in this directory: you will encounter problems if you attempt to assemble programs stored on a network share. (On the lab machines, your Desktop is stored on a network share, so don't try to store your files there.)*

☐ **Copy the C:\Irvine\Examples\Project_sample folder to your local documents folder.** One way to do this is as follows. Keep your Documents window open. Press Windows-R to open the Run dialog. Type C:\Irvine\Examples in the dialog, and click OK; this will open the Examples folder in an Explorer window. Right-click the Project_sample folder, and click Copy. Switch to the Documents window you opened in the previous step, right-click in the empty list area, and click Paste. Then, close the C:\Irvine\Examples window.

☐ **Rename the Project_sample folder to HelloWorld.** In your Documents window, select the Project_sample folder, and press the F2 key to rename it.

☐ **Open the HelloWorld folder** by double-clicking it. Note that it contains several files, including *main.asm* (an assembly language source file) and several files named Project.

☐ **Open the solution in Visual Studio 2010.** In the HelloWorld folder, locate the Project file whose type is listed as "Microsoft Visual Studio Solution." Its icon has a small "10" in the upper-right corner.

Important: If an "Open With" dialog appears, you selected the wrong Project file.

The examples and descriptions in the remainder of activity are intended to function as a "cookbook," so that you can start writing simple assembly language programs through copy-and-paste, without fully understanding how they work. The descriptions here are oversimplified, imprecise, and incomplete. Do not trust this too deeply, particularly beyond the first few weeks of the semester.

2. Exploring "Hello World" in the Debugger

- ☐ **After Visual Studio has opened, open the main.asm file in an editor.** In the Solution Explorer on the right side of the Visual Studio window, expand Project, then double-click on main.asm.

In this activity, every assembly language program you write will be created by modifying the "Hello World" project you just opened.

- Note that assembly language source files have a **.asm** filename extension.
- The **TITLE** line is essentially a comment; it is simply a brief name or description of the program.
- All lines starting with semicolons (`;`) are **comments**. Comments can be added almost anywhere in a program; they may appear on their own line or at the end of another line. Comments start with a semicolon (`;`) and continue to the end of the line. You can also add **blank lines** virtually anywhere.
- The **INCLUDE** line tells the assembler that you will be using the Irvine32 link library, which was written by the author of your textbook. This is *not* a standard library – if you get a job writing assembly code later in life, you will not use this library – but it will simplify things for now.
- The remainder of the file is divided into two sections, **.data** and **.code**. These directives, like most keywords in MASM, are case-insensitive.
- The **myMessage BYTE …** line defines a string.
- The **PROC** line starts the definition of the *main* procedure, and the ENDP line denotes the end of this procedure. This is similar to the definition of a *main* method in Java: it is the entrypoint for the program.
- Currently, the *main* procedure clears the screen (or rather, the output window), displays "MASM program example," moves the cursor to the next line, and then terminates the program.

Now, we will step through the "Hello World" program in the debugger.

- ☐ **Start debugging the Project.** Click Debug > Start Debugging. You may be prompted that "This project is out of date. Would you like to build it?" Click Yes. Notice that build output is displayed in the Output window at the bottom of the main Visual Studio window.

  A black window will briefly appear then disappear. That was your Hello World program running, then terminating, before you could see anything useful.

- ☐ **Set a breakpoint on the "exit" line (line 18).** Move the editor caret to line 18, then click Debug > Toggle Breakpoint (or press the F9 key). A large red dot will appear next to that line in the left margin of the editor.

- ☐ **Start debugging the project again.** Click Debug > Start Debugging (or press the F5 key). The project will not be rebuilt unless you have changed *main.asm*.

  A yellow arrow will appear on top of the breakpoint marker. This indicates that the debugger paused the program's execution at that line. Notice that, in the task bar, you can switch to your program and see the output "MASM program example."

  In the Visual Studio window, click Debug > Continue (or press F5) to allow the program to terminate normally.

- ☐ **Remove the breakpoint on line 18, and set a breakpoint on the "mov edx, offset myMessage" line instead.** Move the editor caret to line 18, then click Debug > Toggle Breakpoint (or press F9). The breakpoint marker will disappear from the editor margin. Move the caret to the "mov edx, offset myMessage" line (line 15), and set a breakpoint there.

- ☐ **Start debugging the project again.** The debugger will stop at the breakpoint. Notice that, if you switch to your running program, it has not yet displayed any output.
- ☐ **Open the Registers window.** Click Debug > Windows > Registers.
- ☐ **Step to the next line.** Click Debug > Step Over (or press the F10 key). In the Registers window, notice that the value of the EDX register is displayed in red: that is because it was changed by the instruction that just executed.

  ? What value does the EDX register contain at this point? ________

- ☐ **Step to the next line.** Notice that, after stepping past the "call WriteString" line, your running program produces output (as one might expect).
- ☐ **Step past the exit line, or click Debug > Continue** and allow your program to terminate.

Debug > Continue instructs the debugger to continue executing your program at normal speed until it hits another breakpoint or execution terminates. This is useful when you only want to single-step through a small portion of your program.

You can terminate your program before it finishes executing by clicking Debug > Stop Debugging (or pressing Shift-F5).

At this point, you should be comfortable with the process of (1) starting your program in the debugger, (2) setting breakpoints, (3) single-stepping, (4) restarting execution (using Debug > Continue), and (5) stopping debugging. If you have questions, ask them now.

3. Writing and Reading Strings

To **display a string**,

1. Add the string to the .DATA section with the following form.

   identifier `BYTE "`*Text to display*`", 0`

   To end the string with a carriage return/newline, use

   identifier `BYTE "`*Text to display*`", 0dh, 0ah, 0`

   In both cases, don't forget "`, 0`" at the end. If you look at your ASCII chart (Activity 2), this is the NUL character; it indicates the end of the string.

2. Add the following two instructions to the .CODE section:

   ```
   mov edx, offset identifier
   call WriteString
   ```

To **read a string** from the input (i.e., from the keyboard),

1. Add a definition to the .DATA section with the following form.

   identifier `BYTE` *maximum_length_plus_one* `DUP(0)`

   In the program below, `name`'s definition is `128 DUP(0)`, so it can hold a string that is at most 127 characters long. (Yes, the declaration says `128`, but it will only accept a 127-character string. It needs one extra byte for the NUL character, which indicates the end of the string.)

2. Add the following instructions to the .CODE section:

   ```
   mov edx, offset identifier
   mov ecx, sizeof identifier
   call ReadString
   ```

As an example, the following program reads a string from the keyboard (up to 127 characters long), then echoes it back to the user.

```
TITLE Example Program - Reading/Writing Strings
INCLUDE Irvine32.inc

.DATA
input   BYTE 128 DUP(0)
message BYTE "You entered: ", 0

.CODE
main PROC
    ; Read a string from the input (up to 127 characters)
    mov edx, offset input
    mov ecx, sizeof input
    call ReadString

    ; Display "You entered: "
    mov edx, offset message
    call WriteString

    ; Display the string that the user entered
    mov edx, offset input
    call WriteString

    exit
main ENDP
END main
```

A NOTE ON COMMENTS AND FORMATTING: In assembly language, it often takes several instructions to perform one (simple) task. This can make assembly code difficult to read. Can you tell what tbe following program does? (Spend a minute on it, then continue reading…)

```
INCLUDE Irvine32.inc
.DATA
input   BYTE 128 DUP(0)
message BYTE "You entered: ", 0
.CODE
main PROC
mov edx, offset input
mov ecx, sizeof input
call ReadString
mov edx, offset message
call WriteString
mov edx, offset input
call WriteString
exit
main ENDP
END main
```

Answer: This is exactly the same program as the one at the top of this page, but with all of the comments and formatting removed. Compare the two. In the first program, you can *skim* the code and get the gist of what it does. You can see "the big picture" without having to read and think about every single instruction.

Don't go overboard—you don't need a comment on every line—but a few judiciously chosen comments can greatly enhance the readability of your code, especially when they're combined with reasonable spacing and indentation.

4. Registers, Addition, & Writing Integers

The following general-purpose **registers** can store a nonnegative integer between 0 and 4,294,967,295. You can use them more or less arbitrarily to store values in your program. (There are more registers, but for now, use these four.)

```
eax         ebx         ecx         edx
```

The **`mov`** instruction stores a value into a register. For example, to place the value 1234 into the EAX register, use the following instruction.

```
mov eax, 1234
```

Values can also be specified in hexadecimal or binary, as we discussed in lecture.

```
mov eax, 0C3Bh
```

You can also copy values from one register to another.

```
mov ebx, 1234
mov eax, ebx      ; After this, eax also contains 1234
```

The **`add`** instruction takes the value in a particular register, adds a constant value to it, and replaces the contents of the register with the sum.

```
mov eax, 1234
add eax, 5  ; After this, eax contains 1239
```

Two values in registers can also be added together.

```
mov eax, 1234
mov ebx, 5
add eax, ebx      ; After this, eax contains 1239
```

The **`sub`** instruction similarly performs subtraction.

To **display a nonnegative integer value**, put it in EAX, then `call WriteDec`.

To **read a nonnegative integer value** from the keyboard (input), `call ReadDec`.
The value will be stored in the EAX register.

```
TITLE Example Program – Reading/Writing Integers
INCLUDE Irvine32.inc

.CODE
main PROC
    call ReadDec   ; Read a value from the keyboard into eax
    sub eax, 10    ; Now eax contains the user’s value - 10
    call WriteDec  ; Display this value
    exit
main ENDP
END main
```

5. Exercise: Name/Age

Ⓡ Fill in the assembly language program on this and the next page so that it does the following. (Develop your program in Visual Studio, then write it below after you are confident that it works.)

1. Displays "`Please enter your name: `"
2. Reads the user's name from the input (a string of up to 127 characters).
3. Displays "`What year were you born? `"
4. Reads a nonnegative integer (a year) from the input.
5. Displays "`Hello,` *name*`.  At the end of this year, you'll be` *age* `years old.`" followed by a carriage return-newline, where *name* is the name entered by the user and *age* is 2014 minus the year that the user entered.

Hint: In the .DATA section, do not call a declaration "name." MASM will produce *error A2008: syntax error : name* since "name" is a reserved word in MASM. Call it "userName" or something else instead.

```
TITLE Name/Age Exercise (Lab 1)
INCLUDE Irvine32.inc

.DATA








.CODE
main PROC
























main ENDP
END main
```