LAB 5

VISUAL C++/ASSEMBLY INTEROPERABILITY

In Lecture 30, we discussed one way to compute the minimum of two integers using only bitwise and arithmetic operations, to avoid using a conditional. In this lab, you will write an assembly language procedure implementing this formula and call your assembly language procedure from a C++ program. Next, you will implement a variation on this formula using *inline assembly,* where you embed assembly language instructions directly into C++ code.

If you are not familiar with C++, you should be able to get by with "copy and paste" (and your knowledge of Java), but please do ask for help if you're confused about anything.

1. CREATE A C++ APP & CALL A LINKED PROCEDURE[1]

First, you will create a Visual C++ application where the main function is written in C++, but it calls a procedure written in assembly language. (These steps are very similar to Lab 4, Part II.)

- ☐ In Visual Studio, click **File > New > Project** to open the New Project dialog.
- ☐ In the list on the left, under Installed Templates, select **Visual C++ > Win32**.
- ☐ In the center panel, select **Win32 Console Application**.
- ☐ In the **Name** text box, enter `Lab5`
- ☐ In the **Location** text box, select a location on a non-network drive (such as your local documents folder – C:\Users*yourid*\Documents\ – or a folder on a USB drive).
- ☐ Click **OK**. The Win32 Application Wizard dialog will open.
- ☐ Click **Next** to proceed to the next page of the wizard.
- ☐ Under **Additional options**, check **Empty Project**.
- ☐ Click **Finish** to close the wizard.
- ☐ In the Solution Explorer (on the right side of the Visual Studio window), right-click on the Lab5 project, and from the context menu, select **Build Customizations**. This will open the Visual C++ Build Customization Files dialog.
- ☐ Check the box labeled **masm(.targets, .props)**.

[1] The instructions here are partially based on
http://scriptbucket.wordpress.com/2011/10/19/setting-up-visual-studio-10-for-masm32-programming/
A StackOverflow discussion of compiling assembly language code in Visual Studio 2010 is at
http://stackoverflow.com/questions/4548763/compiling-assembly-in-visual-studio
which in turn references instructions for configuring build customizations at
http://connect.microsoft.com/VisualStudio/feedback/details/538379/adding-macro-assembler-files-to-a-c-project

- ☐ Click **OK** to close the dialog.
- ☐ In the Solution Explorer, right-click on **Source Files,** and select **Add > New Item**. This will open the Add New Item dialog.
- ☐ In the list on the left, select **Visual C++**. In the center panel, choose **C++ File (.cpp)**.
- ☐ In the **Name** text box, enter lab5.cpp
- ☐ Click **Add** to close the dialog.
- ☐ Type the following into the *lab5.cpp* file, and then save it.

```
#include <cstdint>
#include <iostream>

/* The subtract procedure is defined in util.asm */
extern "C" int32_t __cdecl subtract(int32_t a, int32_t b);

int main() {
    int32_t difference = subtract(3, 5);
    std::cout << "3 - 5 = " << difference << std::endl;
    return 0;
}
```

(The type `int32_t` is defined in the <cstdint> header and denotes 32-bit signed integers. The `__cdecl` specifier indicates that the function uses the C calling convention.)

- ☐ In the Solution Explorer, right-click on **Source Files,** and select **Add > New Item**. This will open the Add New Item dialog.
- ☐ In the list on the left, select **Visual C++**. In the center panel, choose **Text File (.txt)**.
- ☐ In the **Name** text box, enter `util.asm`
- ☐ Click **Add** to close the dialog.
- ☐ Type the following into the util.asm file, and then save it.

```
.model flat, c
.code
; Subtracts 32-bit integers, a - b (C calling convention)
; Receives: [ebp+8]  Minuend (a)
;           [ebp+12] Subtrahend (b)
; Returns:  EAX      Difference (a - b)
subtract PROC
    enter 0, 0

    mov eax, [ebp+8]
    mov ebx, [ebp+12]
    sub eax, ebx

    leave
    ret
subtract ENDP
END
```

- ☐ In the Solution Explorer, right-click the Lab5 project, and select **Properties** from the context menu.
- ☐ In the tree on the left, navigate to **Configuration Properties > Linker > General**.
- ☐ In the panel on the right, set **Enable Incremental Linking** to **No (/INCREMENTAL:NO)**.
- ☐ In the tree on the left, navigate to **Configuration Properties > C/C++ > General**.
- ☐ In the panel on the right, set **Debug Information Format** to **Program Database (/Zi)**.
- ☐ Click **OK** to close the Lab5 Property Pages dialog.
- ☐ Set a breakpoint on the first line of `main`, where the `subtract` procedure is called. (Move the editor caret to that line, and click Debug > Toggle Breakpoint).
- ☐ Debug the application, stepping **into** the `subtract` function. As you step through the program, Visual Studio should move from the .cpp file to the .asm file and back. Make sure you get the correct output at the end: 3 – 5 = –2.

2. WRITE A C++ FUNCTION WITH INLINE ASSEMBLY

- ☐ In *lab5.cpp*, add the following function *above* the `main` function:

```
int32_t divide(int32_t a, int32_t b) {
    /* Visual C++ generates the prologue and epilogue */
    __asm {
        mov eax, a     ; You could also have written: mov eax, [ebp+8]
        mov ebx, b     ; You could also have written: mov ebx, [ebp+12]
        cdq
        idiv ebx
    }
    /* If there is no return statement, Visual C++ assumes the value in EAX is returned */
}
```

- ☐ In the *main* function, call the *divide* function to test it. For example:

```
std::cout << "11 / 4 = " << divide(11, 4) << std::endl;
```

- ☐ Compile and run the program. Make sure you get the result you expect.

3. WHAT JUST HAPPENED?

If you write an operating system, or if you work with embedded systems, you are likely to encounter situations where you need to write *some* assembly language code, even though most of your system could be written in a higher-level language like C or C++.

Just because *some* parts of a system need to be written in assembly language doesn't mean the *entire* system has to be written in assembly language. In the preceding example, you saw two ways that C++ and assembly language code can interoperate:

1. Some functions can be written in assembly language, while others are written in C++. Functions written in one language can call functions written in another language.

2. C++ functions can include *inline assembly language*, where assembly language instructions are inserted directly into the C++ function.

NOTES ON LINKED PROCEDURES

For the first option, where an entire procedure is written in assembly language:

- The code from Part 1 uses the C calling convention. This is the default for C++, but we made it explicit in the C++ function declaration by adding the `__cdecl` keyword.

- Your assembly language procedure must save and restore the **ESI, EDI, EBX, and EBP** registers if it modifies them. (Recall that EBP is pushed/popped when creating/destroying a stack frame. ESI, EDI, and EBX must be pushed/popped separately.)

- You do **not** need to push/pop the EAX, ECX, and EDX registers. Visual C++ assumes that any function call may destroy the values in these registers. If your function returns a value, it should be returned in EAX, as usual.

NOTES ON INLINE ASSEMBLY

Inline assembly language is especially interesting because you can insert assembly language instructions into your C++ code, just like any other C++ statement. For example, you could use inline assembly inside a *for* loop:

```
int32_t example() {
    int32_t nums[] = { 1000, 200, 30 };
    int32_t sum = 0;
    for (int i = 0; i < 3; i++) {
        /* This is an assembly language equivalent of sum += nums[i] */
        __asm {
            lea ebx, nums            ; Put the memory address of nums into EBX
            mov ecx, i               ; Load the value of i from memory into EBX
            mov eax, [ebx + 4*ecx]   ; Load nums[i] into EAX
            add sum, eax
        }
    }
    return sum;
}
```

In the above code, remember that *procedure arguments and local variables are stored in memory.* Specifically, after the stack frame is created, arguments are at [EBP+8] and higher addresses, while local variables are at [EBP–4] and lower addresses. Since the Visual C++ compiler creates the stack frame, it also decides where, exactly, to place the local variables (`nums`, `sum`, and `i`) within the stack frame. So:

- In inline assembly, you can use the names of **function arguments and other variables as memory operands** for instructions.
- Visual C++ will generate a prologue and epilogue for the function, which will create and destroy the stack frame. So, the first function argument will be at [EBP+8], for example. However, it is conventional to reference the argument by name instead, as in the *divide* code.
- You can **modify EAX, EBX, ECX, and EDX** in your inline code. The Visual C++ compiler does not expect them to be preserved between statements.
- You must always **save and restore ESI, EDI, and EBP**, if you modify them.
- You can use the **PTR** operator in assembly code, e.g., `inc BYTE PTR [esi]`.
- You cannot use data definition directives or any other MASM operators other than PTR.
- You **cannot** use the **OFFSET** operator. Use the **LEA** instruction instead, as above.

For more information, C++ interoperability is described in your textbook in Chapter 13.

4. ASSIGNMENT

Review your notes from Lecture 30, where we discussed one way to compute the minimum of two integers using only bitwise and arithmetic operations, to avoid using a conditional. Then:

- ☐ Add a procedure to *util.asm*. Name your procedure `minimum`. It should compute the minimum of two signed, 32-bit integers using the formula given in Lecture 30 (i.e., using bitwise and arithmetic operations, without a conditional).
- ☐ In *lab5.cpp*, add code to the *main* function to test that your implementation of minimum is correct. (See below for some ideas.)
- ☐ Review the notes on *why* the minimum formula works. Then, figure out how to modify the formula to compute the *maximum* of two integers.
- ☐ Write a C++ function called `maximum` that uses *inline assembly language* to compute the maximum of two signed, 32-bit integers using your modified formula (i.e., without a conditional).
- ☐ In *lab5.cpp*, add code to the *main* function to test that your implementation of maximum is correct. (See below for some ideas.)
- ☐ If your code still contains the *subtract* and *divide* procedures from the first part of this lab, delete them and the calls to them. Now, *lab5.cpp* and *util.asm* should **only** contain your minimum and maximum functions, and the tests for them. Make sure your code still compiles and runs correctly!
- ☐ Submit *lab5.cpp* and *util.asm* in Canvas.

5. SOME THOUGHTS ON TESTING

In the written notes for Lecture 30, we gave a mathematical proof of correctness for the minimum function. However, the correctness of the *algorithm* does not necessarily imply that *your particular implementation* works correctly. This is why testing is important.

In *lab5.cpp*, you need to test your minimum and maximum functions. Here are some ideas.

- Printing a bunch of sample output values and visually inspecting the result is not a great way to test. A better option is to write an "obvious" minimum function, like this:

  ```
  int32_t obvious_min(int32_t a, int32_t b) {
    return a < b ? a : b;
  }
  ```

  Then, it's easy to check that that your minimum function gives the same result as the "obvious" minimum function.

  ```
  void check(int32_t a, int32_t b) {
    int32_t expected = obvious_min(a, b);
    int32_t actual = minimum(a, b);
    if (expected != actual) {
      std::cerr << "INCORRECT: " << a << ", " << b << std::endl;
    }
  }
  ```

- Do not try to check all 4 billion possible values of a and b. However, you should test enough "interesting" combinations of a and b values to guarantee that your formula will work even for the ones you didn't test.

- Check all three cases: $a < b$, $a = b$, and $a > b$.

- Check negative, positive, and zero values of both a and b.

- Check boundary cases. `INT32_MIN` and `INT32_MAX` are the smallest and largest numbers that can be stored in an `int32_t` variable. Sometimes 0 is also an interesting boundary case.