CS 216

Laboratories 8 & 9:

x86 Assembly Language

Weeks of 31 March and 7 April 2008

Objective:

These labs are meant to familiarize you with the process of writing, assembling, and linking assembly language code. The purposes of the in-lab and post-lab activities are to investigate how various C++ language features are implemented at the assembly level.

Background:

The Intel x86 assembly language is currently one of the most popular assembly languages and runs on many architectures from the x86 line through the Pentium 4. It is a CISC instruction set that has been extended multiple times (e.g. MMX) into a larger instruction set.

Reading(s):

- x86 assembly language handout (on Collab)

Lab Procedure

The lab procedures are the same for both labs. The differences are the particular programs that need to be written and the report deliverables; these differences are noted below.

Pre-lab

1. Complete the appropriate tutorial
    a. For lab 8, run through the C++/assembly tutorial found on Collab (called nasmexamples.html).
    b. The tutorial for lab 9 will be ready during the week of lab 9.
2. Read through the pre-lab pages on compiling C++ with assembly, as well as the vecsum program.
3. Follow the pre-lab instructions in this document. They requires you to write a program in x86 assembly (mathlib.s for lab 8 or threexplusone.s for lab 9).
4. Files to download:
    a. For lab 8: vecsum.s, main.cpp
    b. For lab 9: timer.cpp, timer.h
5. Files to submit:
    a. For lab 8: mathlib.s, mathfun.cpp
    b. For lab 9: threexplusone.s, threexinput.cpp

In-lab

1. Choose a lab partner, if you would like. It needs to be somebody *different* than who you worked with before. Note that you cannot work with the same partner for labs 8 and 9.
2. Follow the in-lab instructions in this document.
    a. For lab 8, address at least one of the topics in **list 1** of the in-lab section. Be sure to address all the issues in each topic! You will have to complete both of these topics for the post-lab report.
    b. For lab 9, address at least one of the topics in **list 2** of the in-lab section. Be sure to address all the issues in each topic! You will have to complete three (total) of these topics for the post-lab report.
3. We are looking for a brief write-up indicating that you addressed at least one of the topics, and the results that you found. You do not need to make it a full fledged report yet (that's the post-lab).
4. Indicate who your partner is in your report. You only need submit one report for you and your partner.
5. Files to download: none (other than the results of your pre-lab)
6. Files to submit: inlab8.doc or inlab9.doc

Post-lab

1. You may continue working with your partner on the post-lab. Like the in-lab, the post-labs are still a group submission. You can also work alone on the post-lab, if you choose.
2. Indicate who you and your partner are in your report.
3. Finish addressing the topics listed in the in-lab section (both of list 1 for lab 8, three of list 2 for lab 9). We are looking for a quality write-up here, as detailed below.
4. Files to download: none (other than the results of your pre-lab and in-lab)
5. Files to submit: postlab8.doc or postlab9.doc

Pre-lab

Compiling Assembly With C++

To compile a program written partly in x86 assembly and partly in C++, we have to build the program in parts. We build the C++ file as we have in the past:

```
g++ -c -o main.o main.cpp
```

Note that we used the -c flag, which tells the compiler to compile but not link the program. Linking it will create the final executable – but as there is not a vecsum() function defined (yet), the compiler will report an error stating that it does now know the vecsum() function. The '-o main.o' part tells g++ to put the compilation output into the file named main.o. Note that the -o flag wasn't really necessary here (as g++ will use main.o when compiling a main.cpp), but we wanted to include it, as we are going to use it below.

Next, we need to compile the assembly file. To do this, we enter the following:

```
nasm -f win32 -o vecsum.o vecsum.s
```

This invokes nasm, which is the assembler that we are using for this course. We'll get to the '-f win32' part in a moment. The '-o vecsum.o' option is the same as with g++ – it is telling the assembler to put the output into a file named vecsum.o. If you do not specify a filename with the -o flag, it will default to vecsum.obj, NOT vecsum.o – this is why we are using the –o flag. The assembly file name is specified by the 'vecsum.s' at the end of the command line.

The new flag here is the '-f win32'. This tells the assembler the output format for the final executable. Operating systems can typically execute a number of different formats – Windows can execute both .exe and .com files, for example (they are different formats), although one rarely sees .com files these days. As we are running under Cygwin, we specify the win32 format (this format will works for any 32-bit Windows OS).

Finally, we have to link the two files into the final executable. We do this as before:

```
g++ -o vecsum main.o vecsum.o
```

This tells g++ to link both of the .o files created above into a executable called vecsum. Note that there isn't any compiling done at this stage (the compilation was done before) – this just inks the two object files into the final executable.

Linux instructions: If we are running under Linux, we would use the 'elf' format (i.e. 'nasm -f elf -o vecsum.o vecsum.s'). Also, depending on the versions of the assembler and g++ that you are using, you may have to name the function 'vecsum' instead of '_vecsum' (note the lack of underscore). In the final linking step (below), if you get a message such as, "main.cpp:(.text+0x12): undefined reference to `vecsum'", then you should change the name of the function. **However**, you must make sure it compiles in Cygwin before it is submitted!

Vecsum

This part of the pre-lab only needs to be done for lab 8 – no need to repeat it for lab 9.

Complete the tutorial on how to create a C++/Assembly project. See the nasmexamples.html file in the lab08 directory on Collab. The original is found at http://cs.lmu.edu/~ray/notes/nasmexamples/. A local copy is in the Collab folder for this lab (called 'nasmexamples.html' – you'll also need the .css file for pretty formatting). You can skip a few of the sections (feel free to look at them if interested, but they are not needed): Floating Point Instructions, SIMD Parallelism, Saturated Arithmetic, and Graphics. Note that these sections have been removed from the version on Collab.

Examine the vecsum subroutine (in vecsum.s) posted on the Collab site. Use the "Tiny Guide to Programming in 32-bit x86 Assembly Language" document to help understand what is happening in vecsum.s. Make sure you understand the prologue and epilogue implementation, as well as the instructions used in the subroutine.

Compile and run the vecsum program.

- Use the tutorial as a guide, but see the instructions on the previous page.
- If you forget the gdb commands described below, see the Wiki section of the Collab site, which has a summary of all of these commands.
- You can find the assembly and C++ source code on the Collab site. For the C++ code compilation (i.e. main.cpp) and the final link, use the '-ggdb' flag – it's just like the '–g' flag, but it works a bit better with gdb.
- Use the debugger to step through the assembly code, view the register contents, and view the computer's memory.
    - Set a breakpoint at the line in the main.cpp where the vecsum() function is called (probably line 38).
    - Normally, you would use the 'step' function to step into the next instruction. However, since no debugging information was included with the assembler (a shortcoming of nasm running on a Windows platform), we can't use 'step' – it will just move to the next C++ instruction (the cout). Instead, we will use 'stepi', which will step exactly one *assembly instruction*, which is what we want.
    - To display the assembly code that is currently being executed, enter 'disassemble'. This is just like 'list', but it displays the assembly code instead of the C++ code.
    - Note that this prints things in a different assembly format. To set the format to the style we are used to (and the style we are programming in with nasm), enter 'set disassembly-flavor intel'. Now enter 'disassemble' again – the format should look more familiar. You only have to enter that command once (unless you exit and re-enter gdb).
    - To see the vecsum function, enter 'disassemble vecsum'. Note that this only lists the first third (or so) of the routine – up until the 'vecsum_loop' label. To see the rest of the code, enter 'disassemble vecsum_loop', 'disassemble vecsum_done', etc.
    - To show the contents of the registers, use the 'info registers' command.

Pre-lab program for lab 8

You will need to write two routines in assembly, one that computes the product of two numbers, and one that computes the power of two numbers.

The first subroutine will compute the product of the two integer parameters passed in. The restrictions are that it **can only use addition**, and thus cannot use a multiplication operation. We will assume that the second parameter is a positive integer. It must compute this **iteratively**, not recursively. The resulting product is then returned to the calling routine. This subroutine should be called `product`. We will assume that values will not be provided to the subroutine that will cause an overflow.

The second subroutine will compute the power of the two integer parameters passed in. We will assume that the first parameter is the base, and the second parameter is the exponent. Again, both are integers. The restrictions on this routine are that it **can only use the multiplication routine** described above – it cannot call any exponentiation routine. Furthermore, it must be defined **recursively**, not iteratively. This routine should be called `power`.

Both of these routines should be in a file called mathlib.s, and must use the proper C-style calling convention. We are providing you with a mathfun.cpp file, which calls both of your subroutines.

Pre-lab program for lab 9

The 3x+1 conjecture (also called the Collatz conjecture) is an open problem in mathematics, meaning that it has not yet been proven to be true. The conjecture states that if you take any positive integer, you can repeatedly apply the following function to it:

$$f(x) = \begin{cases} x/2 & \text{if } x \text{ is even} \\ 3x+1 & \text{if } x \text{ is odd} \end{cases}$$

The conjecture is that eventually, the result will reach 1. For example, consider $x = 13$:

$f(13) = 3*13+1 = 40$
$f(40) = 40/2 = 20$
$f(40) = 20/2 = 10$
$f(10) = 10/2 = 5$
$f(5) = 3*5+1 = 16$
$f(16) = 16/2 = 8$
$f(8) = 8/2 = 4$
$f(4) = 4/2 = 2$
$f(2) = 2/2 = 1$

Note that this took 9 steps to reach the value 1. And it also shows that this conjecture is true for a number of other values (2, 4, 5, 8, 10, 16, 20, and 40).

This conjecture has been proven for all integers up to $5.6 * 10^{13}$, but has not yet been proven for all (positive) integers. It is widely believed to be true, however. If you are interested, more information on this conjecture can be found at http://en.wikipedia.org/wiki/3x%2B1.

As the conjecture has been proven for numbers up to $5.6 * 10^{13}$, and our 32-bit machines can only count as high as $4.3 * 10^{9}$ (that's 2^{32}), we can safely assume that it is true for all of the input values that we will use.

Your task is to write a routine, called threexplusone, that will return the number of steps required to reach 1. An input of 13 takes 9 steps, as shown above. The Wikipedia page shows a few other input sizes and the number of steps: an input of 6 takes 8 steps; an input of 14 takes 11 steps; an input of 27 takes 111 steps.

This routine MUST call itself recursively using the proper C-style calling convention. The assembly code should be in a threexplusone.s file. You may want to write up a C++ file that calls the subroutine and prints out the result.

Once the subroutine is done, you will need to optimize it as much as possible. With the exceptions listed below, any optimization is valid, as long as it computes the correct result. The grade on this pre-lab will be based both on the correctness of the subroutine and the optimizations included. The only exceptions to the optimizations are that it must still be a recursive subroutine, and must still follow the proper C style calling conventions.

Note that we, too, can write the function in C++ and compile it with –O2 –S –masm=intel. And we will be looking at that assembly code when we grade the pre-lab. If you write your program this way, it constitutes an honor violation, so please hand-code the assembly yourself.

In an effort to time how fast your assembly routine runs, we have included the timer code from the hash table lab (timer.cpp and timer.h). You will need to include a C++ file that performs the following steps:

1. Asks the user for an input value, *x*, which is the parameter to pass to the subroutine.
2. Asks the user for an input value, *n*, which is the number of times to call the subroutine.
3. Runs the subroutine *n* times with the parameter *x* as the input.

You can assume that both *x* and *n* are positive integers. See the hash lab (lab 6) for details as to how to use the timer code.

In-lab

Come to lab with a functioning version of the pre-lab, and be prepared to demonstrate that you understand how to build and run the pre-lab programs. If you cannot, work through the tutorial during lab (that part is individual, not group work). If you are unsure about any part of the pre-lab, talk to a TA. The in-lab will ask you to write C++ code and examine the generated assembly language for a variety of topics.

You should be able to explain and write recursive functions for the final exam, so make sure that you understand how to implement the pre-lab program. Speak to a TA if you have any questions.

The general activity of this in-lab will be to write small snippets of C++ code, compile them so that you can look at the generated assembly code, then make modifications and recompile as needed in order to deduce the representation of a number of C++ constructs (listed below).

For lab 8, you and your partner must work on the two items in list 1 together. For lab 9, you and your (different) partner will need to tackle three of the more complex items from list 2. Keep working on more items as time permits, as you will have time to finish addressing the problems in your final post-lab report. Both partners should be prepared to explain both items from list 1 (for lab 8) or list 2 (for lab 9) to the TA.

The deliverable for the in-lab is a Word document named inlab8.doc. In it, each partner should explain something from the items from the appropriate list in the in-lab report. Note that this is a group submission.

Recall that using the '-S' flag with g++ will generate the assembly code. You will also want to use the –masm=intel flag.

List 1: (You must do ALL of these for lab 8)

Parameter passing

Show and explain assembly code generated by the compiler for a simple function and function call that passes parameters by a variety of means. Be sure to show what is happening both in the caller (the function which makes a call to another function) and in the callee (the function which is called by another function, possibly a recursive call). You do not need to describe parts of the C calling convention we described in class (e.g. saving registers, saving the base pointer, how the call instruction works). The focus here is on examining in detail what happens when parameters are passed.

1. You should explain how ints, chars, pointers, floats, and objects that contain more than one data member such as user-defined classes are passed by value and by reference.

2. In addition, show how arrays (you may pick any type) are passed in C++. Be sure to show both how these values are passed into a function and how the callee accesses the parameters inside of the function. Recall that you can use gdb to pause execution in the assembly code, and then see actual memory addresses – see the pre-lab for details. This question asks about exactly where the data values are placed, so you will need to determine at least a register-relative address, just saying the parameter is accessed as [var] is not enough. Be sure to ask if you do not understand.

Objects

1. Explain how data layout, data member access, and method invocation works in C++ objects. For data layout, how are they kept in memory? How does C++ keep different fields of an object "together"? For data access, how does the assembly know which data member to access? We know how local variables and parameters are accessed (offsets from the base pointer) – describe how this is done for data fields. For method invocation, we know how functions are invoked. But what about methods – how does the assembly know which object it is being called out of? Remember that assembly is not object oriented.

2. Describe where data is laid out for a sample C++ class. You should include at least five data members in your class. Be sure to include data members of different types (ints, chars, and other user-defined classes) and different access levels (public and private) in your class. To demonstrate data layout in objects be sure to show screenshots of the data window while your program is running. We are looking for something more than the data window diagram in ddd here – that diagram is a good start, but not complete. For example, if you define a char and an int (total of 5 bytes needed), how is it laid out in memory?

3. Next, demonstrate how data members are accessed both from inside a member function and from outside. In other words, describe what the assembly code does to access member functions in both of these situations.

4. Finally, show how public member functions are accessed for your sample class. How is the "this" pointer implemented? Where is it stored? When is it accessed? How is it passed to member functions? When (if ever) is it updated?

List 2: (You must do THREE of these for lab 9)

1. Inheritance (data layout, construction, and destruction): Create an instance of an object that inherits data members from another class, and also includes data members of its own. Show in memory where data members are laid out in that object. Then explain how construction and destruction happens in this class hierarchy. Explain what happens when a user-defined object is instantiated and what happens when it goes out of scope. What if anything is "destroyed" by the destructor? Show this process happening in the assembly code using a simple class hierarchy. Point out in the assembly code exactly where the destructors and constructors are getting called.

2. Dynamic dispatch: Describe how dynamic dispatch is implemented. Note that dynamic dispatch is NOT the same thing as dynamic memory! Show this using a simple class hierarchy that includes virtual functions. Use more than one virtual function per class.

3. Other architectures: Compare code generated by g++ for the x86 architecture to code generated by g++ for a different architecture, and describe the differences you find. If you have access to a blue.unix account, that will qualify as a different architecture (it's a Solaris box), as will a Mac (if not an x86 chip) or an AMD chip. Any 64 bit chip that you compile using 64 bit code will also qualify. A cross-compiler, if you have access to one, also works. Compiling the code on a native x86 Linux box and comparing it to a Cygwin compilation does not count – we are looking for code that will run on *different* processors. Identify as many similarities to x86 instructions as you can. Compare things such as: labels, op codes, number of operands, addressing modes. Describe at least four (non-trivial) differences you see between the two listings.

4. Optimized code: Compare code generated normally to optimized code. To create optimized code, you will need to use the '–O2' compiler flag. Can you make any guesses as to why the optimized code looks as it does? What is being optimized? Be sure to show your original sample code as well as the optimized version. Try loops and function calls to see what "optimizing" does. Be aware that if instructions are "not necessary" to the final output of the program then they may be optimized away completely! This does not lead to very interesting comparisons. Describe at least four (non-trivial) differences you see between 'normal' code and optimized code.

5. Templates: What does the code look like for the instantiation of a simple templated class you wrote? You may use Weiss templated code if you wish, but may need to simplify it to understand what is going on. What if you instantiate the class for different data types, what code is generated then? Is it the same or different? If the same, why? If different, why? Compare code for a user-defined templated class or function to a templated class from the STL (e.g. classes such as vectors or functions such as sort).

Post-lab

You may work alone or with your in-lab partner for finishing the post-lab. If you do work with your partner, you will submit ONLY ONE postlab report.

Explore, investigate, and understand both items from List 1 (for lab 8) or 3 of the items from list 2 (for lab 9). Be able to answer "how" and possibly "why" for each item. Use test cases and the debugger as resources. Additionally use resources other than yourself (e.g. books, Web, people). Be sure to credit these sources.

Prepare a report that explains your findings. Follow the guidelines in the Post-lab Report Guideline section, below. Address the following: How the compiler implements the construct at the machine and assembly levels. What leads you to this conclusion. You must show evidence of this behavior in the form of assembly code, C++, screenshots, memory dumps, manual quotations, output, etc. Where you found the information that lead to your conclusion. (i.e. your sources)

Your report should be in MS Word file called postlab8.doc or postlab9.doc.

Tips for Getting Started on the Post-lab

Think about how best to investigate the issues you choose. A good starting point is to write a small C++ program that illustrates one of the issues. This program should be as simple as possible.

Look at the assembly code associated with your C++ code. To examine the disassembled code you have two main options: you can step through the code in the debugger using the disassembly view, or you can have the C++ code output to an assembly file (using the '-S' and '-masm=intel' flags), which you can then browse or edit.

Generating assembly listings: to generate an assembly listing in g++, use the flags described above (and see the wiki page for details). Probably the most useful listing will include source, and assembly code. For some issues it will be of interest to see the machine code as well.

A couple of things you will notice almost immediately about these assembly files is that a) they can be surprisingly long, and b) they contain a bunch of labels, directives, and instructions that at first glance appear to have little to do with your original source program. Don't despair, with a little perseverance you will be able to make heads and tails of a good bit of this.

Note that printing out these disassembled files is probably not your most useful option. You will most likely find that it is significantly easier to view the files in a browser of your choice, such as emacs. In this way you can navigate through the file, searching for particular labels or C++ code. Besides, you may want to make a slight modification to your C++ code and recompile often anyway.

Still stuck? Some of these issues are non-trivial to figure out. Remember that you can use basically any resource whatsoever to figure these things out. There will almost certainly still remain some things in the disassembled code that you do not fully understand. Don't let this paralyze you. Focus on devising experiments that will help you learn more about the particular issues in lists 1 and 2. By tracing though some parts of the code and by modifying your C++ code and comparing the generated assembly code for the two different versions, you should be able to come up with some reasonably good hypotheses about what is happening. Seek out books, manuals, and web pages that explain the issue. Keep in mind that you are required to list your sources in your post-lab report.

Post-lab Report Guidelines

Each group should submit a nicely formatted report that explains his, her, or their findings. The report should be a MS Word file called postlab8.doc or postlab9.doc. At a minimum your post-lab report should address the items in List 1 or List 2. In your report, label the items according to what list they came from (1 or 2) and their item number within that list (e.g. List 2, Item 2. Dynamic dispatch).

Remember that this is supposed to be a polished project. Code snippets should be embedded into the document, not just printed out on a page by themselves and added in at the end. Similarly, screen shots (if any) should be embedded in the document. Highlighting portions of code or drawing arrows between things may help make your explanations clearer. I would expect the explanation of each item to be at least a couple of pages long, including embedded code snippets and screenshots. Keep in mind that you are only submitting one file for this report: postlab8.doc or postlab9.doc. Thus, everything must be included in that one file.

You are encouraged to work with your classmates outside of class, but the work you submit must be complete your own (or work by both you and your partner). If you find yourself looking at somebody else's report or source code, you are in violation of the honor policy.

Other than your own experiments, feel free to use machine manuals (Intel's x86 documents are on Collab), C++ books, your classmates or other experts, resources you may find on the Internet or elsewhere. **Discussing these issues is encouraged, however, remember that your code and final report must be your own work and that you must credit ANY resources used.**

Extra Credit: Exceptional reports or reports that address more than the required number of items may receive an appropriate amount of extra credit. The instructor will determine the amount of extra credit given. This could conceivably be sizable, but it is EXTRA and will not affect other students grades. Basically I want to encourage folks who are interested to really get into this assignment, and I will in turn try to give you some credit for your effort.