Laboratory 9: x86 Assembly Language, part 2

Week of November 18, 2013

Objective

This lab is one of two labs 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

Pre-lab

1. If you need to, read through the pre-lab pages from the previous lab on compiling C++ with assembly, as well as the vecsum program.
2. In particular, you should be familiar with the differences between the various platforms, and the required submission format (it's Linux), as described in the pre-lab section.
3. Follow the pre-lab instructions in this document. They require you to write a program in x86 assembly called threexplusone.s.
4. List, as comments in that assembly file, the optimizations that you used
5. Read Tutorial 9: C. You will need to implement the linkedlist.c program for the post-lab, not for the pre-lab.
6. Make sure your Makefile compiles your code, and that it compiles it in Linux format! This means elf as the nasm format, and no underscore for the subroutine name (both on the 'global' line and on the line label itself).
7. Also note that your timer.h file you should have an #include <string.h>.
8. Files to download: timer.cpp, timer.h (from the lab 6 directory)
9. Files to submit: threexplusone.s, threexinput.cpp, timer.cpp, timer.h, Makefile

In-lab

1. Address at least one of the topics in the list of the in-lab section. Be sure to address all the issues in each topic! You will have to complete two (total) of these topics for the post-lab report.
2. 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).
3. Files to download: none (other than the results of your pre-lab)
4. Files to submit: inlab9.pdf

Post-lab

1. Finish addressing two of the topics listed in the in-lab section. We are looking for a quality write-up here, as detailed below.
2. While this seems like a long post-lab, these list items should have been worked on during the pre-lab and the in-lab, which makes the post-lab much shorter.
3. Implement the linkedlist.c program from the Tutorial 9: C
4. Make sure your Makefile compiles your code!

5. Files to download: none (other than the results of your pre-lab and in-lab)
6. Files to submit: postlab9.pdf, linkedlist.c, Makefile

Pre-lab

You may want to reference the “Compiling Assembly With C++” and “Vecsum” sections from the previous x86 lab.

Pre-lab program: threexplusone.s

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 (20) = 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).

An image (from Wikipedia) shows how paths of most integers less than 50 converge to 1.

This conjecture has been proven for all integers up to at least 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/Collatz_conjecture.

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 17 steps; an input of 27 takes 111 steps. If the input is 1, the output should be zero. Your program need not consider values of zero or -1 (we will never test those values, as the conjecture does not apply in those cases).

This routine MUST call itself recursively using the proper C-style calling convention. The assembly code should be in a threexplusone.s file. You will need to write a C++ file that (called threexinput.cpp) that calls the subroutine and prints out the result. **If you write your function so that it is an iterative solution, you will not receive credit for this pre-lab.**

The routine MUST take only ONE parameter – the number that you are inputting into the conjecture (13, in the example above), and must return the count of the number of steps taken, as per the C calling convention.

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.

What optimizations do you use? First, try to figure out how you can write the same routine using fewer x86 instructions. Some optimizations, such as using the lea instruction (which can do addition and multiplication in one instruction) to quickly add or multiply numbers, are specific to the x86 architecture. Also take a look at http://en.wikipedia.org/wiki/Category:Compiler_optimizations for various optimizations. You will need to include at least one optimization beyond just figuring out how to write your subroutine with fewer instructions. You should put the optimizations used as a comment in the beginning of your assembly file.

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, you should download the timer code from the hash table lab (lab 6: timer.cpp and timer.h). In your threexinput.cpp file, you will need to perform 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. Your program should print out the number of steps for the given input, and the average time taken per function call. You should use an appropriate precision number to make sure you don't report zero when you divide the total time by the number of runs. Your timer code should only include the loop of n times that calls the routine with x as the parameter. Nothing else (including the print statement) should be inside the timer code.

You may find the cdq instruction useful – do a Google search for 'cdq x86' or 'cdq intel'.

You must list, as comments in your assembly file, the optimizations that you used! Just a brief description of what optimizations you used.

Different Architecutres

See the last lab for details, but all code must be submitted to run under Linux, which is the platform that does the compilation on the submission system.

Complete the C tutorial

Read Tutorial 9: C. You will need to implement the linkedlist.c program for the post-lab, not for the pre-lab.

Compiling with make

Your code will be compiled with make. See the last lab for a sample Makefile that will compile assembly.

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 the in-lab, you will need to work on at least one of the items in the list below – note that this is a different list than the previous lab. You will need to tackle two of the more complex items from the list. Keep working on more items as time permits, as you will have time to finish addressing the problems in your final post-lab report. You should be prepared to explain the appropriate items from the list to the TA.

The deliverable for the in-lab is a PDF document named inlab9.pdf. It must be in PDF format! See How to convert a file to PDF for details.

In your report, you should explain something from at least one item in the list in the in-lab report. Note that for the post-lab, you will have to have two of the items fully explained, but you need only get through one for the in-lab. Your report would presumably include the code snippets (both C++ and assembly) that you generated during lab, images, screen shots, results, etc.

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

In-lab 9 list: (You must do TWO of these)

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. 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.
4. 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

Complete the C tutorial's exercise program

Read Tutorial 9: C. You will need to implement the linkedlist.c program.

Report!

Explore, investigate, and understand two of the four items from the in-lab list. 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. ***You must use (and cite!) additional resources for this!***

Prepare a report that explains your findings. Follow the guidelines in the Post-lab Report Guidelines section from the previous lab. 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. Also include where you found the information that lead to your conclusion. (i.e. your sources).

The deliverable for the in-lab is a Word document named postlab9.pdf. It must be in PDF format! See How to convert a file to PDF for details.

Tips for Getting Started on the Post-lab

See the section in the previous lab for these tips.

Post-lab Report Guidelines

See the section in the previous lab for these guidelines.

How much are we looking for?

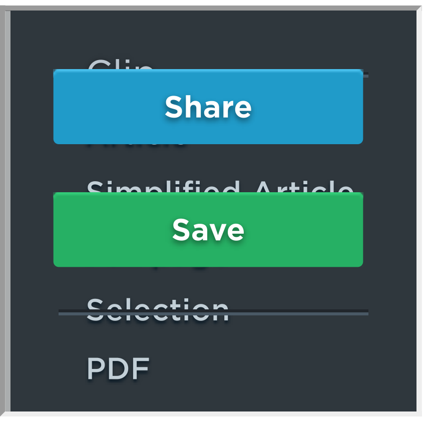
We want you to investigate the particular topic area from the given list, write code to discover the answers, and learn about this topic on your own. The questions that we pose are just meant to get you thinking about the possible ramifications of a given question. They aren’t meant to be specific questions that necessarily need answering one at a time.

As with the previous lab, I would expect the explanation of each item (you have to do two items) to be a page or two long, including embedded code snippets and screenshots (obviously, we want a reasonable amount of English text here – if you do a lot of screen shots, then your total length will be a bit longer). Did you investigate the topic? Did you write code to discover what you didn’t know? Was this written in a reasonably readable format? This is what we are looking for.

This is somewhat vague, and purposely so – research is often vague. If we told you exactly what to write, then there wouldn’t be much discovery of that on your part, which would defeat the whole point of this lab.

We are not looking for you to spend hours and hours and hours on this! A *page or two* per list item (and you have to do two of them) - which means your final report needs to be 2-4 pages long. Keep in mind if you have a lot of screenshots, that doesn't count much towards that page limit.

The grading will be based on a set of points that we would expect you to discover when investigating a given topic. Your grade will be based mostly on how well you hit those points. A small portion of your grade will be based on the overall report presentation and written ability (while we are a computer science class, we expect you to be able to write in English to some extent!).


Share
Simplified Article
Save
Selection
PDF