To print higher-resolution math symbols, click the
Hi-Res Fonts for Printing button on the jsMath control panel.

Laboratory 8: x86 Assembly Language, part 1

Week of Monday, November 4, 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)

1. A Tiny Guide to Programming in 32-bit x86 Assembly Language

Procedure

Pre-lab

1. You should be familiar with the document, A Tiny Guide to Programming in 32-bit x86 Assembly Language, which is available on the Collab workspace. This details the x86 material that this lab requires.
2. Complete the C++/assembly tutorial found on Collab. Note that this tutorial is in C, not C++, so you will notice a few minor differences:
    1. Using ‘clang’ to compile versus ‘clang++’
    2. Using printf() as the output, not cout
    3. You cannot use ‘//’ for comments – instead you must use ‘/* … */’
3. Read through the pre-lab pages on compiling C++ with assembly, as well as the vecsum program.
4. This pre-lab must be submitted in **32-bit Linux** form for the x86 files.
5. Follow the pre-lab instructions in this document. They require you to write a program in x86 assembly called mathlib.s. To see other examples of nasm code, you should look at the vecsum.s program, as well as the code in the nasm tutorial.
6. Make sure your mathfun.cpp takes in only the input described in the pre-lab section! Input is to be provided via standard input (i.e., cin), not through command-line parameters.
7. Your code must compile with ‘make’!
    1. And does your code work on a 64-bit Linux machine? It will need to in order to receive credit.
8. Files to download vecsum.s, main.cpp, Makefile
9. Files to submit mathlib.s, mathfun.cpp, Makefile

In-lab

1. Address at least one of the topics shown in the in-lab section. Be sure to address all the issues in that topic! You will have to complete both 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: inlab8.pdf

Post-lab

1. Finish addressing the topics listed in the in-lab section. We are looking for a quality write-up here, as detailed in the post-lab section. Be sure to address all the issues in each topic!
2. Files to download: none (other than the results of your pre-lab and in-lab)
3. Files to submit: postlab8.pdf

Pre-lab

Different Architectures

There are four different platforms that students are developing their code on:

1. 32-bit Linux (what is in the 001 lab and what is on the VirtualBox image)
2. 64-bit Linux (what the submission server is running)
3. 32-bit Mac OS X (although we doubt anybody actually has this anymore)
4. 64-bit Mac OS X

Your code must compile and run on the submission server, which is a 64-bit Linux machine!

There are three changes that will have to be made to compile your program (and this to the Makefile) depending on your platform:

- You may have to name the function 'vecsum' instead of '_vecsum' (note the lack of underscore) in vecsum.s (this file is described more below). In the final linking step, if you get a message such as, "main.cpp:(.text+0x12): undefined reference to `vecsum'", then you should change the name of the function.
- Some systems will have to supply a command-line parameter to clang++; this can be put on the CXX or CXXFLAGS macro(s) line in your Makefile
- All systems will have a specific nasm file format option ('-f') that will need to be specified.

The first bullet point highlights a compatibility problem between Linux and Mac OS X. When calling a subroutine, which in C++ would be called foo(), there are two standards as to how to name the assembly routine: you can name it either _foo (adding an underscore is added before the name), or name it foo (with no underscore). Unfortunately, Linux uses a different standard than Mac OS X, so we have to make (minor) code modifications in order to compile the code on the other system: in Mac OS X, the vecsum.s file should have the subroutine be called _vecsum, and under Linux, it should be called vecsum (this is twice, on lines 9 and 21).

In an effort to make sure all the files submitted conform to one standard or the other, **all assembly and C/C++ code must be submitted in Linux form** (i.e. will be called ‘foo’ and not `_foo'). Note that in many programs, such as the vecsum.s that we provided you, you have to change the name in TWO places: on the ‘global’ line (line 9 of vecsum.s) and on the label line (line 21 of vecsum.s). **If your code does not compile on the submission system, you will receive a zero!**

Also note that your code must compile with ‘make’. We provide a sample Makefile that will compile vecsum, so you can just modify this Makefile to compile your pre-lab program. **Please note that you should NOT specify a –o flag to clang++ (not even ‘-o a’)**, as we want it to be named the default (a.out). This allows easy porting between the two operating systems.

If you plan to develop it in Mac OS X, we suggest that you develop it normally (putting in the ‘_’ before the subroutine name). Then, once you have verified everything works, remove the underscores from both lines, test it out on a 32-bit Linux machine, such as the VirtualBox image, before submitting it.

Platform Specifics

Each of these different platforms has different compilation lines to allow it to compile. Some of them require changing the assembly files as well. You only need to read the line(s) pertaining to the platform(s) you are developing on.

The provided code works under both 32-bit and 64-bit Linux (although 64-bit Linux usage should read below).

32-bit Linux: The provided code and Makefile works on 32-bit Linux. All assembly sub-routine names must **NOT** have a leading underscore (i.e. they should be 'vecsum' and not '_vecsum'). nasm is invoked with the '-f elf' option. We have the '-m32' flag on the CXX macro line so that it will work fine on a 64-bit Linux machine (including our submission server), but it is technically not necessary (it doesn't hurt, it just doesn't do anything).

64-bit Linux: You have to explicitly tell clang++ to compile in 32-bit mode by passing in the '-m32' parameter (note that this parameter is fine to pass in on 32-bit systems). You need to install the g++-multilib package - we realize that we are not using the g++ compiler, but this installs the correct library in the correct place (if this differs with your version of Linux, then let us know!).

32-bit Mac OS X: to run the code, will need to rename all the assembly function names with the leading underscore (i.e. '_vecsum' not 'vecsum'). You will also have to use the '-f macho' format for nasm, and tell clang++ to generate the correct architecture code; the

latter **WAS** done by providing the '-arch i386' parameter to the compiler we used previously; we are not sure if this is necessary with clang++. In the provided Makefile, try the 'CXX' macro line to be 'clang++ -arch i386' (instead of the default 'clang++'), and change the ASFLAGS macro line to '-f macho' (instead of the default '-f elf'). You will probably want to remove the '-m32' flag on the CXX macro line, but be sure to put that back in before you resubmit. Note that you MUST change all of this back in order for it to compile via the submission system! Note that Mac OS X may not support the format of assembly that we use in this course, which means that you may be stuck reading the assembly in the other format we discussed in class.

64-bit Mac OS X: As far as we can tell, this is the same as the 32-bit Mac OS X. If you run into problems with this (or get it working!), please let us know. We don't have a lot of access to 64-bit Mac OS X machines. Note that Mac OS X may not support the format of assembly that we use in this course, which means that you may be stuck reading the assembly in the other format we discussed in class.

Below is a table summarizing the changes

Platform	nasm flag	x86 subroutine name	clang++ flags	Notes
32-bit Linux	-f elf	vecsum	-m32	This is the platform on the 001 machines and in the VirtualBox image; -m32 is optional, but keep it in there for compatibility with 64-bit Linux
64-bit Linux	-f elf	vecsum	-m32	This is what our submission server is running, **and what your code must work on.** If you have it on your computer, you must install a few packages as well - see above
32 bit Mac OS X	-f macho	_vecsum	-arch i386	We are unsure about the -arch i386 flag's necessity. May not be able to print the assembly in the format discussed in class.
64 bit Mac OS X	-f macho	_vecsum	-arch i386	We think this is the same as 32-bit Mac OS X platform. We are unsure about the -arch i386 flag's necessity. May not be able to print the assembly in the format discussed in class.

Compiling Assembly With C++

For this part, you will need to download three files: vecsum.s, main.cpp, and Makefile.

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:

```
clang++ -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 clang++ to put the compilation output into the file named main.o. Note that the -o flag wasn't really necessary here (as clang++ will use main.o by default when compiling 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 elf -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 elf' part in a moment. The '-o vecsum.o' option is the same as with clang++ – 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 elf'. This tells the assembler the output format for the final executable. Operating systems can typically execute a number of different formats. As we are running under Linux, we specify the elf format. Mac OS X uses ‘-f macho’ – see the above section for more details.

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

```
clang++ -o vecsum main.o vecsum.o
```

This tells clang++ 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 links the two object files into the final executable. Also note that for our submitted Makefiles, we will NOT have the '-o' flag present.

Vecsum

Complete the C++/assembly tutorial. The original is found at http://cs.lmu.edu/~ray/notes/nasmexamples/. You can skip a few of the sections of the original version (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 (named x86-doc.pdf) 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 above.
- 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 set 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: mathlib.s

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 both of the parameters are positive integers. 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.

You can assume that the numbers passed into the routine will both be positive, so you need not consider negative numbers or zero. Furthermore, as described above, no values will be used on your program that could cause an integer overflow.

Both of these routines should be in a file called mathlib.s, and must use the proper C-style calling convention. You must also provide a mathfun.cpp file, which calls both of your subroutines – see the main.cpp file provided as a template. The program should take in ONLY two integers (we'll call them *x* and *y*). It should then print out the output of calling product(*x*,*y*) and power(*x*,*y*). Thus, if the input is 3 and 4, it would print out 12 and 81.

Input is to be provided via standard input (i.e., cin), not through command-line parameters.

If you are going to have multiple routines in a single assembly file (as is needed for mathlib.s), you just have to have multiple 'global' lines for each subroutine that you plan on calling from your C/C++ 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 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 the two topics shown below – note that there will be a different topics to work through on the next lab.

The deliverable for this in-lab is a Word document named inlab8.pdf. It must be in PDF format! See How to convert a file to PDF for details.

In the report, you should explain all the items in one of the categories below (either objects or parameter passing). For the post-lab, you will need to have all items from both categories explained. We are looking for significant evidence that you were able to complete some work during the in-lab, and thus are not setting page length requirements.

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

NOTE: If you are getting errors by compiling with the '-masm=intel' flag, you should instead do

```
clang++ -S -mllvm --x86-asm-syntax=intel -m32 YOURFILE.cpp
```

In-lab 8 topics: you must do ALL of these for the post-lab, but only ONE of these for the in-lab

The questions posed below are example questions to answer. Some of them may overlap. Others may not be necessary. And there may be questions not listed that are worth answering. The purpose of posing those questions is to help you think about what topics and ideas you should address in your response - it's not meant to be a rigid structure that you absolutely must follow. We are going to look at whether you have addressed the general idea of each of the list topics.

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.
3. How does passing values by reference work in assembly? Is it different than passing values by pointer?

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. We are looking for something descriptive here – 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?

Post-lab

For the post-lab, you should explore, investigate, and understand all of the 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 post-lab!***

The idea is that you will take what you started in the in-lab, and flesh it out a bit more for the post-lab.

Prepare a report that explains your findings. Follow the guidelines in the Post-lab Report Guideline section, below. In particular, you should address how the compiler implements the construct at the machine and assembly levels, and what lead 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 did you find the information that lead to your conclusion (i.e. your sources)?

Your report should be in PDF file called postlab8.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

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 clang++, 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 they can be surprisingly long, and that 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

You should submit a nicely formatted report that explains your findings. The report should be a PDF file called postlab8.pdf. At a minimum your post-lab report should address the items in the in-lab list. In your report, label the items according to which list item they came from (parameter passing or objects), and their item number within that list.

Note that this is supposed to be a polished report. 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 (which are optional) 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 a page or two long at least, including embedded code snippets and screenshots. Keep in mind that you are only submitting one file for this report: postlab8.pdf. Thus, everything must be included in that one file.

We are being very careful to not specify the length of the report, only what we would guess is the estimated length. As long as you properly answer the questions, the length is up to you. Double versus single spacing is also up to you, but we prefer single spacing.

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

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!).

Untitled note

Find a notebook

Add tag

Add comments

Version 6.0.6: c128369/1.0.1.250

Web Clipper tutorial

Options

Saving clip...

To:

Share

Save

Selection

PDF