University of Virginia Computer Science
CS216: Program and Data Representation, Spring 2006
http://www.cs.virginia.edu/cs216/ps/ps7/
10 April 2006

[PDF version for printing]

Problem Set 7
Eighty-Sixing Compilers

Out: 10 April
Due: Monday (beginning of class), 17 April

Collaboration Policy - Read Carefully

For this assignment, you may work on your own or with any one other person of your choice. If you work with a partner, you should turn in one assignment with both of your names on it. You may (and probably will need to) consult any outside resources including books, papers, web sites and people you wish except for answers or comments on CS216 labs from previous semesters. You are also encouraged to discuss these problems with students in the class. You must acknowledge any people and outside resources you work with on your assignment. If you discuss the assignment with people other than your partner, you may not take any written materials out of your discussion. It is fine to bounce ideas off other people, but the answers you turn in must be your own.

As always, you are **strongly encouraged** to take advantage of the staffed lab hours posted on the CS216 web site.

Purpose

- Explore the x86 Instruction Set
- Learn to read and write x86 Assembly Programs
- Examine how the C compiler compiles some code fragments

Reading: Before beginning this assignment, we recommend reading the *x86 Assembly Guide*. While working on the assignment, you should also go through the *Guide to Using Assembly in Visual Studio*.

Assembly Instructions

These questions examine the x86 assembly language. For each question below, we provide a short snippet of assembly code. Your answer should show a shorter snippet of assembly code with exactly the same meaning. You may use any instructions you want, but there are possible answers to each question that only use instructions described in the *x86 Assembly Guide*.

(Note: Since assembly instructions are variable length, a sequence of 4 instructions may actually be "shorter" than a sequence of 3 instructions in terms of the number of bytes needed to encode it. For these questions, we interpret "shorter" to mean fewer assembly instructions. Note that a shorter sequence, according to this interpretation, is not necessarily smaller or faster.)

1. Find a shorter instruction sequence with the same behavior as:

```
mov BYTE PTR [eax], 200
add BYTE PTR [eax], 16
```

2. Find a shorter instruction sequence with the same behavior as:

```
   xor ebx, ebx
   sub ecx, ecx
   neg ecx
   and edx, 0
   mov eax, ebx
   add eax, ebx
   shl eax, cl
```

3. Find a shorter instruction sequence with the same behavior as:

```
label1:
   inc eax
   cmp eax, ebx
   jl label2
   jmp label1
label2:
   cmp ebx, eax
   jle label1
   imul eax, [var]
```

Calling Conventions

The *Guide to x86 Assembly* describes the C calling convention. A calling convention involves many design decisions that impact the efficiency, size, and complexity of generated code.

4. An alternative calling convention would use registers to pass some parameters. Suppose we used the EBX, ECX, and EDX registers to pass the first three parameters instead of the stack.

a. Describe other changes that would need to be made to the calling convention.
b. Discuss the advantages and disadvantages of such a change. Would it improve the running time of typical programs?

Exploring Compiled C

For the next group of questions, your goal is to understand the assembly code produced by the C compiler. You can use any C compiler you want for this, but if you use something other than the Visual C++ compiler in the ITC labs, make sure to document the compiler you used.

For each C language feature, you should:

a. Explain how the compiler implements the feature
b. Describe the research you did to determine this. Your answers might include examples of test cases (C code and the assembly it produces), dumps of memory contents at a particular execution point, references to language specifications or other documents, and any other resources you used to answer the question. You should be careful not to jump to conclusions based on a single simple example. Try enough experiments to be more confident in your answers.
c. A discussion of why you think the implementation is done this way. Speculate on why this implementation is used and what alternatives there are.

You can use any approach you want to do this.

Here are some hints:

- 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.
- Next you need to take a look at the assembly code associated with your C code. To examine the disassembled code you have two main options: 1) You can step through the code in the debugger using the disassembly view, or 2) You can have the assembly code output to a separate file, which you can then browse or edit.
- To generate an assembly listing in Visual Studio, on project settings, go to C/C++, select the category listing files, then select the desired listing file type. 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. (Use the Visual Studio help to figure out what the resulting generated file will be called or how to rename the output file.)
- A couple of things you will notice almost immediately about these assembly files is that a) they can be surprisingly long, 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 or tails of a good bit of this. To see addresses of variables you can right click in the disassembly window and unselect "Show Symbol Names" which will also convert function names to raw addresses as opposed to their symbolic names.

  Note: 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 on the screen using the Visual Studio editor or your favorite text editor. In this way you can navigate through the file, searching for particular labels or C statements. 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. 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 your classmates to see if anyone else is working on the same issue. Seek out books, manuals, and web pages that explain the issue.

5. Determine how `int`s, `char`s, `float`s, and pointers are represented in memory. In addition, you should show how one and two-dimensional arrays of both `int` and `char` elements are represented. For each example you should show your C code, the relevant assembly code generated. When useful, also include a screen shot of the memory window while your program is running. Probably the easiest way to see this is to declare several items of these types and then to assign values into them. Then look in the watch window to find the address of variables (you can ask for this by typing `&var` into the watch window) and then locate that address in the memory window. Be sure to include a screenshot of the memory window and indicate where each item is located in that window, either by drawing/highlighting items in the memory window or giving the addresses.

6. Determine how the `if`, `while`, and `for` statments are implemented. Do the results tell you anything about whether it is more efficient to implement C code in particular ways?

7. Explain how the compile implements parameter passing and returns results. Be sure to examine what is happening both in the caller and in the callee. You should be able to identify how the C compiler follows the C calling convention described in the x86 Guide, but also fill in many unspecified details in the calling convention. You should at least answer these two questions: (1) how are arrays on the stack passed and returned? (2) how well does the compiler do in determining which registers must be saved and restored? In addition, answer at least one additional question of your choice where the calling convention is unclear.

8. Compare code generated using debug mode (the default) to optimized code. To do this you will need to change your project configuration from debug to release by selecting "Win32 Release" from the "Set Active Configuration" drop-down list on the Build menu. Then select: project-> settings->C++ and select an optimization level. Find some example programs where the assembly code generated with optimizations is substantially better than the code generated without optimizations. Speculate on what the optimizer is doing? Are there things it does surprisingly well on? Are there things it should to better on?

9. For this question, you can pick any language feature or compiler question you want, and answer it. This is open-ended: the more interesting things you can find out the better. For those of you who know (or are interested in learning C++), feel free to consider a C++ language feature for this.

10. (For x86 gurus and wannabe-gurus only, not required to reach green-star level) Devise a sequence of instructions that has two valid different interpretations depending on where the execution starts. One execution should start at the beginning of the first instruction; the other execution should start somewhere inside the first instruction (a number of bytes offset from the start location that is less than the length of the first instruction). An ideal solution would be something like this:

```
           start:
Your code:       ABCDEFGHIJKLMNOPQRSTUVWXYZ  (each letter is one byte)
First sequence:  AB CD EFGH IJ KL MNO PQ RST UV WX YZ
Second sequence: -BC DEF G HI JK LM NOP QR STU VW XY Z ...
```

Even more ideally, the `YZ` instruction (the last insturction in the first sequence) would jump to location start + 4 (instruction BC in the second sequence).

To do this you will need to examine memory more directly (as well as use your understanding from question 5) to know exactly what bytes are used to represent things. Note that many of the assembly instructions are actually different opcodes depending on the parameters you use. You will also find the Opcode Map (Appendix A of Intel's Instruction Set Reference manuals, starts at page 415) useful.

Include a description of how you found your sequence. Long (non-repeating) sequences are more impressive than short sequences, but the most impressive answers will have two sequences that perform meaningful and different behaviors. You also get bonus points if you can explain a reason why such an instruction sequence might be "useful"?

This assignment is heavily based on labs used in previous year's CS216 classes, especially Ruth Anderson's Fall 2004 lab.
It was revised for CS216 Spring 2006 by David Evans.