CS 237 Meeting 1 — 9/7/12

Introduction

1. Introduction
2. First time teaching 237 in over two decades, so...
   - There may be some rough edges, but...
   - I am very excited about the opportunity. The material this course covers is simultaneously practical, theoretically beautiful, and fun.
3. Attendance (i.e., more introductions)

Goals of CS 237

1. Two big ideas (from CS 134)
   - Digital is about representing information in a finite alphabet! Binary is just the smallest possible alphabet.
     - What is "digital" about "digital circuits" and "digital logic"? In digital logic circuits, the exact electrical properties of the circuit don't matter. Instead, one range of voltages are all considered 1 while a different range represents 0.
     - In 134 (and 136) we explore how to represent all sorts of data in digital form (text, images, lists, trees, etc.)
     - In 237 we focus on representing sequences of instructions (i.e., programs) digitally.
     - Our goal is to understand how it is possible to build a device that can read, understand, and in some sense obey such digital instructions.
   - Computer Systems have Layers
     - Those who took CS 134 with me probably recall the day I played the clip from Shrek where Shrek explains that ogres are like onions because they have layers.
     - Computer systems are also like onions. They have many, many layers. This layering is often described in terms of *virtual machines*.

* When you first learned to program, you programmed as if the computer actually understood the language you were learning to use (probably Java?).
* Do computers really understand Java (or Basic, or C, or Python, or . . .)?
* No! If you think about how you run a Java program (as I will demonstrate), you first "compile" it and then run it. The compilation process creates an alternate digital representation of the program you expressed in Java in a lower-level language called Java Byte Code.
  · Byte-code is what is stored in the .class files the compiler produces.
  · Byte-code is encoded in binary, but you can see a symbolic representation of some byte-code by using the "javap" command with option -c.
  · The byte-code does not look all that much like the original Java.
* So, the next obvious question is whether computers understand Java Byte-code?
* Again, the answer is no!
  · To run Java programs, a machine must have available another program that implements what is called the "Java Virtual Machine". This program examines the instructions encoded in byte-code within a Java .class file and performs the operations they describe. If the compiler translates from Java to byte-code correctly and the Java virtual machine program interprets byte-code instructions correctly, the combination gives the impression (illusion?) that the computer understands Java.
* Since the Java virtual machine interpreter is itself a program, it must be written in some programming language.
  · There are many such programs, but one popular example is called HotSpot.
  · HotSpot is written in C++. So...
* Do real computers understand C++?

* Of course, as you guessed, the answer is no!
* To run a C++ program, you must first translate it into a machine language program using a compiler like gcc.
  - As we will see, when you run gcc on a file of C or C++ code, it produces a object code file (.o file) containing the binary code for the source code in the compiled file(s).
  - Ultimately, gcc combines your object files with libraries of library code to form an executable file (like "java" or "gcc" or "a.out").
* The instructions in an executable file are encoded in what is commonly called "machine language". More accurately, they are encoded following the instruction level architecture (ILA) of the machine you are using. This suggests that your machine really does understand the instructions in such a file, but ...
* Can you take an executable file produced on a Linux system and run it on a Window system or on MacOS?
* Again, the answer is no!
* Each operating system defines a set of system calls that application programs can use to request services (creating a window, deleting a file, allocating more memory, ...). When a programmer/compiler writes/generates "machine code" these system calls serve as extensions to the basic instruction set of the computer. The operating system therefore effectively defines a virtual machine that is an extension of the machine defined by the ILA.
* Given that the Linux operating system is implementing a virtual machine, we can still ask whether the o.s. (which is after all just a program) is running on the "real" x86 machine.
* In case you had not noticed previously, all the demos I have done today using Linux have been done on my Mac which is running MacOS. The Linux system we have been using is running under a virtual machine monitor known as Parallels rather than on an actual machine.
* Parallels itself is a program. It almost certainly is not written in machine language! I am not sure which language was used by the Parallel developers, so for the sake of a bit of variety let us assume they used C. That is, they programmed for a C virtual machine.
* The illusion of the C virtual machine is again made possible by gcc, a compiler than handles both C and C++.
* The code that gcc produces when it compiles a program like Parallels consists of x86 code using system calls based on the interface provided by MacOS X. That is, the C compiler depends on a virtual machine generated by the code of MacOS X.
* MacOS X on my computer, actually appears to run on the x86 underlying hardware!
* That's quite an onion!
* Each layer of this onion is a virtual machine.
  - There are many ways to implement a virtual machine: interpretation, compilation, or as a library (i.e., OS interface).
  - It really does not matter to a piece of software or its programmer whether it runs on a real machine or a virtual machine.

2. Topics covered in the course

- Digital Circuits
  - Where the buck finally stops (and the virtual machine becomes real).
  - We will talk a bit about transistors, but the beauty of digital logic is that it is about logic not about electronics.
- Instruction Level Architecture (Machine/Assembly language)
  - We will learn the ILA of a not-quite defunct machine (the MIPS processor) because:
    * It is what our text covers,

    - It is better designed than the "real" architecture of Intel processors
    - All ILAs share the same fundamentals
- C
  - No one should really set out to become a machine language programmer.
  - Instead, our goal in this course is to show you how to express code you might write in a high-level language at the machine language level.
  - Java is a bit too high-level for this.
  - C is just right.
  - Learning a second language is a good thing too!
- Microarchitecture
  - This refers to the way digital logic components are organized to design a system capable of interpreting programs written a the machine language of an ILA.
- Architectural Support for Programming Languages and Systems
  - CS 237 should really be a prerequisite that matters for other systems-oriented courses, particularly:
    - Compilers
    - Operating Systems
  - Particularly for OS, we will discuss traps, interrupts, input/output devices, and memory address translation mechanisms.

Course Organization (or lack thereof)

1. Exciting bureaucratic details:
   - Find the course web page at

     http://www.cs.williams.edu/ tom/courses/237
   - Staff:
     - TAs: Owen Barnett-Mulligan, Simon Chase, James Wilcox
   - Texts:

     Required Harris & Harris, Digital Design and Computer Architecture
     - Provides the best explanations I have seen for most of the topics we will be covering
     - Focuses a bit more on the digital design than we really want
     - Uses MIPS architecture rather than x86
     - Provides an appendix on C

     Recommended Kernighan & Richie, The C Programming Language, 2nd Edition
     - A classic! Every computer scientist should own a copy.
     - I have not ordered it for the bookstore. Look online.
   - Office Hours:
     - Will be set once CS 336 tutorial meeting schedule is finalized
     - Feel free to ask questions beyond office hours as long as you forgive me if I occasionally a) say "no" or b) miss/move my regular hours.
   - Work:
     - Labs:
       - Meeting at 1:00 or 2:30 each Tuesday
       - Most lab work will be completed outside scheduled meeting time
     - Homework
       - Most/many weeks written homework problems will be assigned beyond the lab work (be neat!)
     - Exams
       - Midterm + final. Still debating between scheduled and take-home.
   - Honor Code and Collaboration
     - Discussing course material including assigned labs and homeworks is an excellent way to learn the material. DO IT!

– Unless explicitly told to work in groups, all of the labs and homeworks you submit during the semester must be of your own work in the sense that even if you are inspired by insights gained while discussing the work with others, what you submit must be produced on your own (without active discussion or consulting discussion notes while completing the actual work). Roughly speaking, you should be able to reproduce anything you submit without assistance.

Introduction to C

1. Things that stay the same
   (a) Scalar Data types
      i. Many scalar types are the same
         - int (and long and short)
         - double (and float)
         - char
      ii. But a few are different
         - C distinguishes between signed and unsigned integer values
         - C's char type is roughly the same as Java' byte type (in Java a char is stored in 16 bits).
         - C has no boolean type. C uses int values where Java uses booleans treating 0 as false and everything else as true (though most people use 1 to mean true most of the time).
   (b) Scalar operations
      i. Many are the same (+, -, *, /, %,)
      ii. Relationals (<, >, <=, >=, ==, !=) look the same but they produce the int values 0 and 1 rather than booleans.
      iii. Logical operators (&&, !, and ||) look the same but work with int values.
   (c) Basic control structures
      i. In C, if and while constructs are just like Java.
      ii. The do-while loop is also the same (though you might not have seen it in Java).
      iii. The C for loop is like Java except that the loop variable must be declared before the loop rather than in the loop header.
      iv. The switch statement has the odd property that if you leave out a break one case falls through to the next one.