Education in Computer Science and Computer Engineering Starts with Computer Architecture

Yale N. Patt
Univeristy of Michigan
Ann Arbor, MI

Abstract

At The University of Michigan, the first required course in both the Computer Science curriculum and the Computer Engineering curriculum starts with the basics of computer architecture. This change in the curriculum started in Fall semester, 1995, after years of frustration with the results of the first course, which conventional wisdom dictates ought to be a high level language programming course. This paper describes the contents of this first course, The paper also includes some observations, having taught the course twice (Fall, 1995, Winter, 1996).

1 Extended Abstract

1.1 Some non-technical information about the course

The course, EECS 100, is the first course for undergraduates wishing to major in Computer Science or Computer Engineering. It is a prerequisite for EECS 270, Digital Logic Design, and for EECS 280, Programming and Data Structures. [The University of Michigan numbering system uses the first digit of a course number to indicate the year in which a course is expected to be taken – 1:freshman, 2: sophomore, etc.] The course was offered for the first time this past academic year to 130 students (Fall term) and 170 students (Winter term). [Winter term at the University of Michigan is equivalent to what most universities call their Spring semester.]

The course consists of 41 50-minute lectures, meeting three times a week for lecture and once a week for a discussion session with a graduate student TA. The course was team taught this first year by Professor Kevin Compton and me. We are both regular members of the EECS faculty. Professor Compton's specialty is algorithms and compiler technology; my work is in computer architecture and high performance implementation.

Students were required to write six programs (one in LC-2 machine language, two in LC-2 Assembler, and three in C) and complete several problems sets. There were two hour exams and a final exam in the course.

1.2 Technical Description of the material taught

The course covers three major topics: computer organization and architecture, programming in C, and algorithm analysis. The major thrust of the course is to first give the student a fundamental understanding of how a computer works, and then tie high level constructs (control structures, functions, arrays and structures, pointers, etc.) to that fundamental understanding. In point of fact, each time a C construct is introduced, the student sees what a compiler would generate in terms of machine code to carry out this task. Thus the student sees how activation records are generated at run time, as well as how physical I/O is handled through device registers by the operating system.

The first lecture introduces computing from a number of perspectives: from the standpoint of what is computable (Turing machines, etc.), as well as from the standpoint of multiple levels of transformation that are required to convert a problem from a natural language description to the electronic devices that actually carry out the task, From there, we go on to explain that information is carried along wires in the form of the absence or presence of voltages (0 or 1), and introduce various codes for representing information. We explain the difference between ASCII codes for representing characters and 2's complement integers to represent signed numbers. We show both logical and arithmetic operations on this information. Next, we describe both P-type and N-type CMOS transistors, and construct inverters, logic gates, latches, muxes, decoders, and finally a small unit of random access memory consisting of four locations of three bits each. Our observation is that this introduction to memory clears the mystery of "what is memory," resulting in far fewer student problems when pointers are introduced later in the course.

From there, we introduce the Von Neumann execution model, followed quickly by an introduction to the LC-2 [for Little Computer 2], the next generation of the LC-1 which I have successfully used in junior level computer organization courses for about 15 years. The LC-2 is a 16 bit machine; its ISA is Load/Store, with 8 general purpose registers. Opcodes are 4 bits wide, and include LD/ST using direct, indirect, and base-offset addressing, operates (ADD, AND, and NOT), cedure call/return, and conditional branches on N, Z, and not-Z condition codes. I/O is handled by the TRAP instruction which vectors the processor to an operating system routine which does physical I/O from a keyboard data register or to a monitor data register. Keyboard and monitor status registers support the I/O activity. Matt Postiff, one of our undergraduates has written a simulator that takes as input a machine language version or an LC-2 Assembler version of a program, and executes it. The simulator contains mechanisms for single stepping (if desired), setting breakpoints, depositing and examining values in registers and memory, etc.

Once the student is solid on the intracies of the LC-2, we introduce a high level language. This first year, we used C, but are considering including some C++ constructs in the fall. [The reader's opinions on this matter are hereby solicited.] In our treatment of C, we cover the usual constructs in a serious introductory course in high level language programming, as well as some of the stuff that beginning students never get to see, like activation records and the run-time stack. The main topics we cover in this part of the course include: variables and activation records, control, functions, basic input/output, the run time stack, recursion, arrays, structures, pointers, and basic file i/o.

We have found that the pace can be accelerated and additional insights can be provided specifically because the student has a foundation in the underlying structure of the computer. Explanations of C constructs in terms of these underlying structures solidifies the knowledge faster than we have found to be the case when the student does not have this foundation.

The final block of material of the course teaches some elementary analysis of algorithms. The student is introduced to big-O notation, and is shown how various algorithms grow. For example, since the student has been exposed to recursion, he/she can understand the differences in growth between generating the Fibonacci sequence recursively and iteratively.

1.3 Some observations

Almost all the feedback we have received has been very supportive. Students enjoy the course enormously, – we think because they are given a comprehensive understanding of what is going on, rather than having to learn rules that make no sense to them.

Constructs that have proven difficult for students to master in the past do not seem to offer the same challenge in this course. Pointer variables are a piece of cake after analyzing the design of a four by 3-bit memory early in the course. Recursion is no longer magic after understanding activation records, and in particular the linkage between a calling function and the function it calls. Once the student sees how function A calls function B, it is a very small step to function A calling itself.

I think the bottom line is that this approach works because it does what education in science is good at, building knowledge from the bottom up. Unlike the standard introductory programming courses offered in response to the professional societies' CS1 and CS2 guidelines, when the Michigan student finally gets to high level language programming (about 40% into the course), he/she has a foundation on which to pin the concepts. I believe that goes a long way in the apparent mastery of the material that we have seen.

1.4 Acknowledgements

This course has been in the works at Michigan for more than three years. Several people have contributed a lot to its development. First, of course, is my co-instructor, Kevin Compton, who shares the responsibility for the course. He has been the Chairman of the Curriculum Committee at U-M since the day I first brought up at a curriculum committee meeting the conceptual framework of this course. He has championed it from the start as our curriuculum chairman, and he has provided important insights into its technical material as its co-instructor. David Kieras and Ann Ford, two other members of the U-M faculty, have also provided their insights into the development of this course. The entire Computer Science and Engineering faculty should be acknowledged for strongly endorsing this paradigm shift in computer science and engineering education. We have been fortunate in attracting an excellent slate of students to work with us. Sanjay Patel, Paul Racunas, Dave Cybulski, Vinodha Ramasamy and Peter Kim have served enthusiastically as teaching assistants. Matt Postiff has put a lot of work into developing an LC-2 simulator that students have found to be user-friendly, and writing up the LC-2 documentation for students' use as a reference manual. Sanjay Patel, in particular, deserves special cknowledgement. He returned to U-M Fall term, 1995 to do research in computer architecture toward the PhD degree. I told him I would support him as a research assistant, but offered to include him in this teaching opportunity, if he wished. He couldn't say no, and as I expected, provided enormous positive energy to the course. None of us were surprised when the local ASEE chapter chose him as one of their Graduate Teaching Instructors of the Year for 1995-96.