AC 2008-1776: INTEGRATION OF C INTO AN INTRODUCTORY COURSE IN MACHINE ORGANIZATION

Eric Freudenthal, University of Texas at El Paso

Eric Freudenthal is a member of the Computer Science faculty at the University of Texas at El Paso. Dr. Freudenthal's research interests include self-organizing distributed systems, computer security, and the effective teaching of foundational concepts in computation and science.

Brian Carter, University of Texas at El Paso

Brian Carter is an undergraduate studying Computer Science at the University of Texas at El Paso.

Frederick Kautz, University of Texas at El Paso

Frederick Kautz is an undergraduate studying Computer Science at the University of Texas at El Paso.

Alexandria Ogrey, University of Texas at El Paso

Alexandria Ogrey is an undergraduate studying Computer Science at the University of Texas at El Paso.

Robert Preston, University of Texas at El Paso

Robert Preston is an undergraduate studying Computer Science at the University of Texas at El Paso.

Arthur Walton, University of Texas at El Paso

Arthur Walton is an undergraduate studying Computer Science at the University of Texas at El Paso.

Integration of C into an Introductory Course in Machine Organization

Abstract

We describe the reform of a fourth-semester course in computer organization in the Computer Science BS curriculum at the University of Texas at El Paso (UTEP), an urban minority-serving institution, where Java and integrated development environments (IDEs) have been adopted as the language and development environment used in the first three semesters of major coursework. This project was motivated by faculty observations at UTEP and elsewhere[1] and industry feedback indicating that upper-division students and graduates were achieving reduced mastery of imperative languages with explicit memory management (most notably C), scriptable command line interfaces, and the functions of compilers, assemblers, and linkers.

The pre-reform computer organization course[2] focused on foundational concepts such as machine instructions, registers, the random-access memory model, and the generalized fetch-execute cycle. Projects included assembly-language programming of a Motorola M68HC11 processor installed in a two-wheeled robot. The reformed curriculum, which uses the same embedded target, integrates the study of C and thus also able to focus on the implementation of high-level language features and linkage between C and assembly language routines. Student labs use traditional command-line tools including bash, gcc, gas, ld, and make.

Lectures include collaborative learning components in which student groups are tasked with the development and refinement of first C, and then assembly language implementations of program fragments. Lab assignments utilize both languages and introduce students to command interpreters, scripting, collaborative development tools, and subroutine linkage of procedural languages. Assignments are distributed, "handed in," and grades distributed using the subversion source code repository.

The reformed course's outcomes are a superset of the original, with extensions including (1) understanding of C and its runtime environment, (2) parse trees, and (3) implementation of dynamic memory management.

Context

Object-oriented design is accepted as a primary programming model[2] and many computer science departments have adopted Java as their principal teaching language in many lower-division courses. Furthermore, Java programs are commonly developed, compiled, and executed within seamless IDEs. As a result, students who have attended a third-semester course in data structures may neither be exposed to the relationship between memory addressing and variable

allocation nor the process of compilation and linkage prior to attending a course in computer organization.

We describe a reform to an upper-division course in computer organization whose previous curriculum was chosen when a non-garbage-collected procedural language was used in introductory courses. The prerequisite skills list for the pre-reform course listed mastery of "pointers and dynamically allocated memory" at the synthesis level.

After the adoption of Java as the principal teaching language at UTEP, procedural languages with explicit memory management were principally relegated to a language survey course that compare abstractions provided by various languages. C permits explicit pointer arithmetic and thus has semantics reflecting the behavior of the underlying memory system that appears arcane and inordinately complex when viewed through the lens of formal language abstractions. Despite Java's syntactic similarity to C, faculty teaching upper-division systems-oriented courses and potential employers of our graduates observed that students primarily trained to program in Java have increased difficulty understanding and composing programs in C. Faculty at other institutions have made similar observations.[1]

Java's wide adoption by industry was facilitated by its close syntactic similarity to C. Our course takes the opposite approach and leverages students' familiarity with Java's syntax to teach C and, in turn, uses this knowledge to bootstrap an understanding of the concepts underlying assembly-language programming. C has semantics similar to byte-addressable storage and provides a syntactically clearer expression of variable manipulation and pointer manipulation in assembly-level programs. C's inter-procedural linkage and memory model are sufficiently simple to permit exploration of implementation of the high-level language features such as dynamic memory management, composite types, and recursive functions.

The Course

The lecture course, which includes a closed lab section, begins with an introduction to the key components and concepts undergirding computer architecture, including byte-addressable memory, registers, the ALU, opcodes, the program counter, and the fetch-execute model. Arithmetic machine instructions are introduced simultaneously in C (nominally using Java syntax) and assembly language in a manner that illustrates the role of a compiler in managing storage and translating operations.

Prior to attending this course, most students have only developed programs using an IDE. Early labs introduce these students to a POSIX shell (bash), a set of command-line tools such as subversion (which is used to disseminate and collect assignments), a keyboard-centric editor (Emacs), and explicit compilation and linkage (gcc, ld, etc.). Early programming assignments are in C. These early assignments exclusively manipulate scalar variables and exploit language

features that will be familiar to a Java programmer and expose students to modular program designs that exploit global symbols and separate compilation. A compilation management tool (make) is introduced as a solution to the problem of managing compilation and linkage of multiple source files. Makefiles are required for lab submissions. The students were visibly excited when they observed that their finished product was a binary program that, like commercially acquired programs, could be executed on its own, without the assistance of an IDE.

Integer representations (signed and unsigned) and C's bitwise logical operators (&, |, <<, >>) are introduced early in the lecture course. For students to gain proficiency with these concepts and constructs, lab projects include integer-to-ASCII conversion functions in C for multiple radixes. Arithmetic machine instructions and related mnemonics (as a programmer convenience) are introduced in the lecture course.

Direct addressing, labels, and pseudo-ops that reserve memory are presented as a solution to the problem of managing multiple scalar variables. Cooperative class exercises include the design of program fragments in assembly language that implement arithmetic functions that students first express using C's algebraic syntax. Manually generated parse trees are introduced as a technique for mechanically detecting sub-expressions, determining evaluation order, and managing temporary variables.

We employ pointer arithmetic and arrays in C first to illustrate the use of C pointers and then to motivate the role of indexed addressing modes. Students translate code snippets that implement vector operations in C to assembly in cooperative class exercises. These cooperative groups frequently generate solutions that illustrate important peephole and reduction-of-strength optimizations which are identified and discussed by the instructor.

Prior to attending this course, students have only been exposed to stacks as an abstraction useful for traversing graphs introduced while studying common data structures and algorithms. In this course, stacks are introduced as a solution to the problem of storing variables whose lifetime is equal to the activation of a subroutine, such as return addresses, parameters, and local variables which are accessed using indexed addressing modes. Cooperative class exercises include the design of recursive subroutines in assembly language and lab assignments include C programs that call assembly language subroutines.

Allocation of memory within composite types (structs in C) is examined and compared to Java classes. Both cooperative class exercises and subsequent lab projects manipulate linked lists using programs written in both C and assembly language. Dynamic memory management is discussed and a lab exercise includes the construction of a slab memory allocator and functions that manipulate linked lists. The course utilizes an embedded controller with memory-mapped I/O. To build familiarity with the cross-development environment, students initially cross compile a C program that illuminates a memory-mapped LED. One subsequent lab consists of a

timing loop to control the LED's brightness using pulse-width modulation and thus provides an opportunity to explore factors contributing to execution time.

Interrupts are introduced as a mechanism to manage asynchrony and provide notification of the passage of time. A later lab uses clock interrupts to drive state machines that first pulse-width modulate an LED and later drive motors on a small robot. In this case, the interrupt handler, written in assembly language, serves as a trampoline to a C service routine.

Assessment

Anecdotal reports from undergraduate peer leaders indicate that students attending the reformed course are more highly motivated by their increased understanding of how "real systems" work and have expressed dramatically increased interest in a course on compilation. Results from midterm examinations, final examinations, and lab projects indicate a strong understanding of both the traditional and extended course outcomes. A teaching and lab manual has been developed with the assistance of student volunteers who attended intermediate versions of the reformed course; the manual is freely available for extension.[4]

Related Work

Recent trends in computer systems organization have included advances in development and debugging environments for students and architecture-first (a.k.a. "breadth-first") curricula that introduce computer systems organization in a first-semester course.[8;9]

A rich variety of simulation environments suitable for teaching assembly language have been developed.[3, 4, 5, 6] These educational simulators graphically portray the execution of instructions and the contents of memory and registers in a manner that facilitates the understanding of the execution of assembly-language programs. In contrast, our course examines the implementation of high-level language features. Thus, it is useful to have an appropriate debugging environment that is suitable for understanding programs composed of modules written both in machine language and C in a uniform manner. Furthermore, our course exposes students to both native and cross-development contexts and we need tools suitable for both. To that end, we employ gdb, the GNU Project Debugger,[7] which supports both source- and machine-level debugging of a variety of local and remote targets that can be traced or simulated. gdb provides this uniform interface in a mode that appears to be well-suited for our students' preparation and needs: students are initially introduced to gdb while debugging simple programs written exclusively in C and thus become competent using its command-line interface which provides familiar debugger functionality for debugging familiar high-level language constructs. gdb's uniform interface for examining memory and controlling execution by symbol name or address using C-like syntax provides students with a simultaneous (and, in this context, intuitive) view of the

execution of a program translated from a high-level language as a sequence of understandable machine instructions.

Ironically, our work was inspired by the recent work of Yale Patt in developing and promoting architecture-first (a.k.a. "breadth-first") curricula[8, 9] that ground students in underlying architecture and machine language concepts prior to introducing high-level language programming in C. This approach has the advantage of providing students with an intuitive and continuous understanding of hardware and software constructs that are obscured by the now-common imperative- or object-first curricula.[2, 9] We view our approach as complementary since it exploits understanding gained from prior study of high-level (and even object-oriented) languages to facilitate the understanding of C and its runtime environment.

Bibliography

[1] **Dewar, Robert and Sconberg, Edmond.** Computer Science Education: Where are the Software Engineers of Tomorrow. *STSC Crosstalk.* January 2008.

[2] *Computing Curricula 2001 Report.* **The Joint Task Force on Computing Curricula of the IEEE Computer Society and of the ACM.** ACM, 2001. http://www.computer.org/education/cc2001/report.

[3] *Teaching Computer Architecture with a Computer-Aided Learning Environment: State-of-the-Art Simulators.* **Yehezkel, Cecile, Yurcik, William, and Pearson, Murray.** Society for Computer Simulation (SCS) Press, 2001. Proc. International Conference on Simulation and Multimedia in Engineering Education (ICSEE).

[4] *Combining learning strategies and tools in a first course in computer architecture.* **Teller, Patricia, Nieto, Manuel, Roach, Steve.** : IEEE, 2003. Proc. 2003 workshop on Computer architecture education: Held in conjunction with the 30th International Symposium on Computer Architecture.

[5] *Teaching computer organization/architecture with limited resources using simulators.* **Wolffe, Gregory, Yurcik, William, Osborne, Hughe, and Holliday Mark.** 2002. Proc. 33rd SIGCSE technical symposium on Computer Science.

[6] **Cassel (Boots), Lillian, Holliday, Mark, Kumar, Deeepak, Impagliazzo, John, Bolding, Kevin, Pearson, Murray, Davies, Jim, Wolffe, Gregory, Yurcik, William.** Distributed expertise for teaching computer organization & architecture. *ACM SIGCSE.* 2001, Vol. 33, 2.

[7] **Free Software Foundation.** *GNU Project Debugger.* http://sourceware.org/gdb.

[8] *Education in Computer Science and Computer Engineering Starts with Computer Architecture.* **Patt, Yale.** ACM, 1996. proc. 1996 Workshop on Computer Architecture Education. ACM.

[9] **Patt, Yale and Patel, Sanjay.** *Introduction to Computing Systems.* McGraw Hill, 2004. ISBN 0-07-121503-4.