Work in Progress - Combined Introduction of C and Assembly with a Focus on Reduction of High-level Language Constructs

Eric A. Freudenthal, Brian A. Carter, Frederick F. Kautz, and Alexandria N. Ogrey
University of Texas at El Paso, efreudenthal@utep.edu, {bacarter,ffkautz,anogrey}@miners.utep.edu

Abstract - We describe the reform of a sophomore-level course in computer organization for the Computer Science BS curriculum at the University of Texas at El Paso, an urban minority-serving institution, where Java and integrated IDEs have been adopted as the only language and development environments used in the first three semesters of study. This project was motivated by faculty observations and industry feedback indicating that upper-division students and graduates were failing to achieve mastery of non-garbage-collected, strictly imperative languages, such as C. The similarity of C variable semantics to the underlying machine model enables simultaneous mastery of both C and assembly-language programming and exposes implementation details that are difficult to teach independently, such as subroutine linkage and management of stack frame. An online lab manual has been developed for this course that is freely available for extension or use by other institutions. In this paper, we report on pedagogical techniques for facilitating student understanding of the relationships between high-level language constructs, such as algebraic expression syntax, block-structured control-flow structures, and composite data types, and their implementations in machine code.

INDEX TERMS - computer architecture, CSE, assembly language, compilation

INTRODUCTION

Our project was motivated by faculty observations at the University of Texas at El Paso and elsewhere [1] and industry feedback indicating that upper-division students and graduates were not achieving proficiency in programming in 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 focused on foundational concepts such as machine instructions, registers, the random-access memory model, and the generalized fetch-execute cycle [2]. Projects included assembly-language programming of a Motorola M68HC11 processor embedded within a robot. The reformed curriculum [3], which uses a different embedded target, integrates the study of C and is thus also able to focus on the implementation of high-level language features and the linkage between C and assembly-language routines. Student labs use traditional command-line tools including bash, gcc, as, ld, and make.

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.

Ironically, our work was inspired by the recent work of Yale Patt in developing and promoting architecture-first (a.k.a. “breadth-first”) curricula that ground students in underlying architecture and machine language concepts prior to introducing high-level language programming in C [4], [5]. While we do not take a position in the high-level vs. architecture-first argument, we view our approach as complementary to Patt’s since it exploits understanding gained from the prior study of high-level (and even object-oriented) languages to facilitate the combined understanding of C and computer organization.

ADOPTION OF THE TI MSP430 MICROCONTROLLER

In the spring of 2008, we adopted the TI MSP430 embedded controller for this course. Like the HC11, the MPS430 is supported by the free Gnu development tools. Advantages of this controller over the previously used M68HC11 include a smaller orthogonal (and thus easier to learn) instruction set and inexpensive (~$20) development kits.

DECONSTRUCTING HIGH-LEVEL PROGRAMMING CONSTRUCTS INTO MACHINE-LEVEL IMPLEMENTATIONS

A key objective of our course is to present strategies that enable students to easily decompose the challenge of constructing assembly language programs into smaller, more manageable pieces. This is accomplished through the exploration of techniques that generally are reserved for courses on compiler construction. For example, manually generated parse trees are introduced as a technique for mechanically detecting sub-expressions, determining evaluation order, and managing temporary variables.

Early cooperative class exercises include the design of program fragments in assembly language that implement program snippets that students first express in C. Later projects explore inter-procedural linkage. The remainder of this paper presents an introduction to techniques we use to introduce students to the relationship between high level

October 22 – 25, 2008, Saratoga Springs, NY
38th ASEE/IEEE Frontiers in Education Conference
T1A-1
CONTROL FLOW

Students who are only familiar with block-structured programming can easily be confused by the translation of block-structured control-flow constructs to equivalent branching assembly-language routines. Branching instructions have similar semantics to the C (and FORTRAN) goto statement. Our course exploits C’s support for both (conditional) goto and block-structured primitives to enable the students to explore the reduction of block-structured programs to “goto C.” Thus students are able to first restructure their programs from block-structured C code to goto C independently of the challenge of translating basic blocks to a machine’s instruction set. As is illustrated by Figure 1, the subsequent translation of goto C to (MSP430) assembly language is straightforward.

![FIGURE 1
REDUCTION OF BLOCK-STRUCTURED C TO BRANCHING CODE](image)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Parse Tree</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x=3) goto x_not_3; y = 4; goto end_not3; x_not_3; y = 5; end_not3;</td>
<td></td>
<td>cmp #3, ix jmp end_not3 mov #4, iy jsp end_not3</td>
</tr>
</tbody>
</table>

The process of manually translating complex mathematical expressions to assembly language is complex and error-prone since it involves the identification of sub-expressions, determination of valid evaluation ordering, and the implicit allocation of temporary variables to hold their results. We introduce expression parse trees as a simplifying intermediate step in this process. Students readily understand the generation and evaluation of this representation which, as illustrated in Figure 2, directly exposes sub-expression dependencies and exposes the need for temporary variables. Each sub-expression (represented by an internal node) typically translates to a small sequence of assembly-language instructions. Students are thus provided a general-purpose technique for decomposing and then efficiently generating code to evaluate arbitrary arithmetic expressions.

COMPOSITE DATA TYPES

In this course, we discuss composite data types declared using C’s struct primitive. Lectures and exercises examine the memory representation of ADTs whose members are defined as structs including the the implementation of programs that access their members. A linked list example is illustrated in Figure 3. Lab projects include programs that manipulate abstract data types (such as linked lists) that are accessed by mainline code in C and by I/O handlers written in assembly language. This figure also illustrates a technique used by two-pass assemblers to determine the address of instructions prior to opcode generation.

REFERENCES


AUTHOR INFORMATION

Eric A. Freudenthal is a member of the Computer Science Faculty at the University of Texas at El Paso.

Brian A. Carter, Frederick F. Kautz, and Alexandria N. Ogrey are undergraduate students studying Computer Science at the University of Texas at El Paso.