

Responding to Java-Centric CS Curricula: Integration of C into a Course in Computer Organization

Eric A. Freudenthal, Brian A. Carter, Rafael Escalante
University of Texas at El Paso, efreudenthal@utep.edu, {bacarter, rescalante}@miners.utep.edu

Abstract – This paper describes the reform of a sophomore-level course in computer organization for the Computer Science BS curriculum at The University of Texas at El Paso, where Java and integrated IDEs have been adopted as the first and primary language and development environments. This effort 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 frames. An online lab manual has been developed for this course that is freely available for extension or use by other institutions.

INTRODUCTION

We report on the evaluation of a reformed sophomore-level course in computer organization at an ABET-accredited Computer Science Department. The department has adopted an object-first curriculum as defined by the ACM Computing Curriculum 2001 Report [0] where Java is used as the principal teaching language in most major coursework. As we reported previously [1,2] after adoption of this object- and Java-centric pedagogical approach, faculty teaching upper-division courses and potential employers detected a dramatic reduction in upper-division students' ability to understand or design programs written in strictly imperative languages that reflect the semantics of the underlying memory model, such as C. Schonberg and Dewar report similar observations of students graduating from other programs that adopted Java-centric curricula[3]. While these deficits are not common at schools with architecture-first curricula [4-5], object-centric curricula are asserted to provide complementary advantages. Rather than taking a position on whether architecture-first curricula are strictly superior to object-first, we implemented compensatory reforms that appear to be successful, as observed by upper division systems faculty and employers who report that recent graduates have attained a dramatically improved ability to program in C.

Like Schonberg and Dewar, we conjectured that these problems are likely due to the large semantic gap between the formal semantics of Java as the principal language used by students to solve problems in coursework and the low-level languages such as C whose semantics are much more similar to the underlying instruction-set-architectures. As reported previously[1,2] our primary intervention has been the reform of a sophomore-level course in computer organization titled "Architecture I." Students attending this course must develop mastery of a large number of concepts including the representation and encoding of an instruction set, the mechanics of separate compilation and linkage, signed and unsigned arithmetic, control flow, the various roles of registers and random-access memory, and the function of a range of addressing modes.

The pre-reform computer organization course focused on foundational concepts such as machine instructions, registers, the random-access memory model, and a generalized fetch-execute cycle [6]. Projects included assembly-language programming of a Motorola M68HC11 processor embedded within a robot. The reformed course [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, (3) implementation of simple dynamic memory management, and (4) usage of traditional command-line development tools. While no formal evaluation has yet been conducted, examinations and projects indicate that most students develop competence in the both the previous and extended outcomes, and students have expressed enthusiasm for learning C due to its high perceived commercial relevance.

The reformed course exploits the syntactic similarity between Java and C: students attending Architecture I are already familiar with Java, so they are already familiar with much of C's syntax. Further, we exploit the semantic similarity between C and assembly language by interleaving their instruction in a manner where high-level C constructs and their low-level assembly-language "implementations" are simultaneously presented. This direct examination of

C's implementation renders C's otherwise opaque semantics intuitively apparent. Furthermore, this presentation exposes students to a range of compilation techniques that motivates further study; students who have completed the reformed course attended a well-subscribed course in compilation techniques which was previously canceled multiple times due to insufficient enrollment.

Ironically, this work was inspired by the recent work of Yale Patt who developed a architecture-first (a.k.a. "breadth-first") curricula[4] that introduces students to underlying architecture and machine language concepts prior to high-level language programming in C [4], [5]. The approach described here is 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.

In lecture, C syntax and semantics are seamlessly interleaved with implementation. Static variables are introduced simultaneously with labels and mnemonics to reserve memory. Integer representations (signed and unsigned) and C's arithmetic and bitwise logical operators are similarly taught with the instructions that implement them. For students to gain proficiency with these concepts and constructs, lab projects include integer-to-ASCII conversion functions in C for multiple radices.

The curriculum reforms described in this paper are intended to provide a understanding of the layering of high level language constructs upon assembly language abstractions, and an underlying instruction-set-architecture. We have found it helpful to initially introduce students to simplified (but correct) models that enable them to start programming early, and to introduce additional features as pragmatically motivated solutions to complications that arise when programming increasingly complex problems.

Our previous reports describe the tools used in the course, the pedagogical advantages of incrementally introducing addressing modes as solutions to pragmatic problems, and the simplification of code generation though the generation of intermediate representations of control-flow (if-then-else), composite types, and expressions to assembly language. [3]

This report includes a review of the pedagogical techniques developed for the reformed course reported in previous papers and an enumeration of learning outcomes including an assessment of the course's effectiveness of achieving them.

NON-INTEGRATED DEVELOPMENT TOOLS

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 native and cross-compilation and linkage tools from the Gnu herd (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 express excitement when they observe that their finished product is a binary program that, like commercially acquired programs, could be executed on its own, without the assistance of an IDE.

DECONSTRUCTING HIGH-LEVEL PROGRAMMING CONSTRUCTS INTO MACHINE-LEVEL IMPLEMENTATIONS

Assembly language programming is taught through examination of strategies for translating high level C-language constructs into assembly language. This is accomplished through the discussion and manual application of techniques that generally are reserved for courses on compiler construction. As illustrated in Figure 1, manually generated parse trees are introduced as a technique for identifying sub-expressions, determining evaluation order, and managing temporary variables in a manner that is easily understood and applied by students.

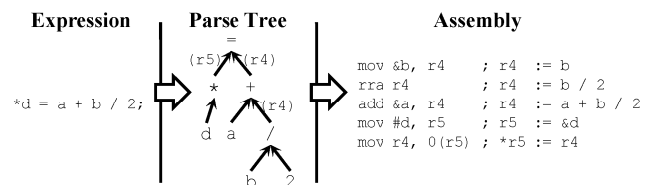


FIGURE 1. ILLUSTRATION OF A PARSE TREE USED AS AN AID TO CONVERT A C-STYLE ARITHMETIC EXPRESSION TO ASSEMBLY-LANGUAGE CODE

EXPRESSIONS-FIRST APPROACH

The course begins with a brief review of Boolean signed and unsigned arithmetic, which is followed by an introduction to variable declarations and expression syntax in C, focusing on similarities with Java. Our course utilizes Texas Instruments' MSP430 processor[ti msp430], which has an "absolute" direct addressing mode that can be used for both source and destination operands, and permits, for the purposes of these early exercises, all variables (and even constants) to be statically stored at fixed addresses in memory.

TABLE 1. LIMITED FORM OF TWO-OPERAND INSTRUCTIONS INITIALLY PRESENTED IN CLASS.

1 st word (instruction)				2 nd word	3 rd word
Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0	Src addr	Dest addr
Op	2	9	2		

A small set of instructions required for simple algebraic operations and the reduced two-operand instruction format

of Table 1 are presented along with pseudo-ops to reserve memory for storing variables (and constants). Labels are presented at the same time and opportunities to practice are provided through in-class exercises. After just a few lectures, students are competently translating C code snippets into assembly and machine language. Typical projects, which are first practiced in groups and then individually, are illustrated in the first two examples in Table 2.

After students develop competence using direct addressing mode and linearization of arithmetic expressions (as measured by daily in-class quizzes), we present the role of arithmetic condition codes in implementing relational operators (equal to, less than, etc.), conditional branching, and multi-word arithmetic (e.g., add with carry) as illustrated by the last example in Table 2 without the added complication of addressing mode selection.

A preliminary version of this course took the complementary approach of initially presenting register addressing modes. However, the need to include constants in expressions inconveniently necessitated the introduction of immediate addressing modes. The course begins with a brief overview of Boolean signed and unsigned arithmetic, which is followed by an introduction to variable declarations and expression syntax in C, focusing on similarities with Java. The MSP430 processor has an “absolute” direct addressing mode that can be used for both source and destination operands, and permits, for the purposes of these early exercises, all variables (and even constants) to be statically stored at fixed addresses in memory. A small set of instructions required for simple algebraic operations are presented along with pseudo-ops to reserve memory for storing variables (and constants) and labels are presented immediately and opportunities to practice are provided through in-class exercises. After just a few lectures, students are competently translating C code snippets into assembly and machine language such as the example illustrated in Table 2.

INTRODUCTION OF REGISTERS AND INDEXED ADDRESSING MODE

We delay discussion of other addressing modes until students have demonstrated mastery of condition codes and the encoding of branches (which include a relative word offset field) on exercises and quizzes. Registers are initially introduced as a convenient storage for temporary values. Later they are exploited by indexed addressing modes to provide a mechanism to implement pointers.

TABLE 2.
EXAMPLE OF EARLY ARITHMETIC CODE SNIPPET
TRANSLATION PROJECTS USING
ABSOLUTE ADDRESSING MODE.

C SOURCE CODE	ASSEMBLY LANGUAGE	MACHINE CODE
short a, b, one=1;	.data	
	a: .word 0	1000:0000
a = b+1;	b: .word 0	1002:0000

	one: .word 1 .text mov &b, &a add &one, &a	1004: 0001 2000: 4292 1002 1000 2006: 5292 1004 1000
long a; a += 0xdeadbeef;	.data a: .word ;low .word ;hi t1: .word 0xbeef .word 0xdead .text add &t1, &a addc &t1+2, &a+2	1000:0000 1002:0000 1004: beef 1006: dead 2000: 9292 1002 1000 2006: 2d03 (+3)
unsigned short a; if (a >= 1) a++;	.data a: .word one: .word 1 .text cmp &one, &a jc isNeg add &one, &a isNeg:	1000: 1002: 0001 2000:9292 1002 1000 2006:2d03 (+3) 2008:5292 1002 1000 200c:

In order to limit cognitive load upon our students, we introduce addressing modes incrementally with priority given to more general concepts. For example, on our target architecture, while register indirect mode provides an efficient 1-word encoding for instructions whose source operand’s address is specified by a register, this mode is not available for destination operands. In order to permit students to focus on pointer dereferencing and field offsets, we prefer to defer presentation of indirect mode, favoring instead indexed (register+offset16) mode, which is available for both source and destination operands. This permits students to construct program fragments including the referencing of pointers using only three addressing modes (absolute, indexed, and register).

TABLE 3.
ENCODING OF TWO-OPERAND INSTRUCTIONS.

Bits 15-12	Bits 11-8	Bits 7-4		Bits 3-0
Operation	source register	A _d	Byte	A _s dest register

As illustrated in Tables 3 and 4, which are adapted from the MSP-GCC project’s [7] documentation, the encoding of binary operations is complex and is specified as a sixteen-bit integer that references two registers and contains three control fields: a destination addressing mode (A_d), an operand size selector (Byte), and a source addressing mode (A_s). Addressing modes 0 and 1 are available for both source and destination operands, and two additional addressing modes, 2 and 3, are available for source operands. Registers four through fifteen are general-purpose, R0 is the program counter, R1 is the stack pointer, and R2 and R3 principally serve as constant generators with varying value and interpretation, depending upon the addressing mode being used.

By initially restricting addressing modes that are introduced to students, they are spared from subtle details that distracted previous cohorts of students from learning basic control-flow and algebraic idioms. For example, on our target architecture, the absolute addressing mode used in previous examples is implemented as a variant of Indexed addressing mode (1) where register R2, a constant generator,

provides a zero offset to a constant address provided within an extension word.

TABLE 4.
FULL LISTING OF ADDRESSING MODES

A _n	Register	Syntax	Mode	Description
0	N	R _n	Register	Operand is the contents of R _n
1	N	x(R _n)	Indexed	Operand in memory at R _n +x (x within extension word)
2	N	@R _n	Indirect	Operand in memory at address held in R _n .
3	N	@R _n +	Indirect Autoincrement	As above; then the register is incremented by 1 or 2.
Addressing modes using R0 (PC)				
1	0 (PC)	LABEL	PC-relative	Operand is in memory at address PC+x (x(PC).
3	0 (PC)	#x	Immediate	Operand in extension word. (@PC+)
Addressing modes using R2 and R3, special-case decoding				
1	2 (SR)	&LABEL	Absolute	Operand in memory at address specified by extension word.
2	2 (SR)	#4	Constant	Operand is the constant 4.
3	2 (SR)	#8	Constant	Operand is the constant 8.
0	3 (CG)	#0	Constant	Operand is the constant 0.
1	3 (CG)	#1	Constant	Operand is the constant 1.
2	3 (CG)	#2	Constant	Operand is the constant 2.
3	3 (CG)	#-1	Constant	Operand is the constant -1
<i>A_d: Only addressing modes 0 and 1 are available for the destination operand.</i>				

Students are introduced to register addressing only after mastering absolute addressing. Its form is simple and is specified by addressing mode zero for both A_d and A_s.

While register indirect (A_s=2) provides the most efficient mechanism to dereference pointers, it is only available for source operands. Once again, to initially limit cognitive load we instead initially introduce indexed addressing mode (1), which is available for both source and destination operands. Thus, as illustrated by Table 5, fragments dereferencing pointers using only three addressing modes (register, indexed, and absolute), are actually encoded using only modes 0 and 1..

TABLE 5.
DEREFERENCING POINTERS WITH INDEXED ADDRESSING MODE

C source code	Assembly language	Machine Code
short a=0, *b=&a,	.data a: .word 0	1000: 0000
	b: .word a	1002: 1000
a = *b + 1;	one: .word 1 .text mov &b, r4 mov 0(r4), &a add &one, &a	1004: 0001 2000: 4214 1002 2004: 4492 0000 1000 200a: 4292 1004 1000

The MSP-430’s “indirect auto increment” implements a convenient “pop” operation that, when used in conjunction with the program counter (R0) also implements the MSP-430’s immediate addressing mode. While we do require students to understand this addressing mode, we only introduce it after students demonstrate understanding of indexed addressing mode in the context of pointers.

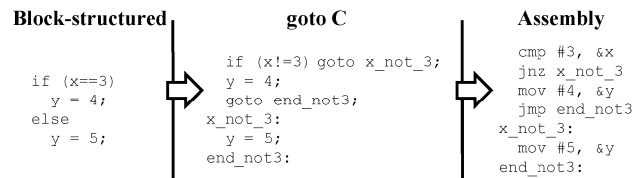


FIGURE 2. REDUCTION OF BLOCK-STRUCTURED C TO BRANCHING CODE THROUGH GOTO-C

CONTROL FLOW

Students who are familiar only with block-structured programming can easily be confused by the translation of block-structured control-flow constructs (e.g if-then-else clauses, switch statements and for or while-loops) to equivalent branching assembly-language code. Branching instructions have similar semantics to the C (and FORTRAN) goto statement. Rather than introducing translation templates, as fixed idioms to be memorized, the course exploits C’s support for both goto and block-structured primitives to enable the students to examine the logical reduction of block-structured programs to “goto C” prior to translation to assembly language. As is illustrated by Figure 2, the subsequent translation of goto C to (MSP430) assembly language is straightforward.

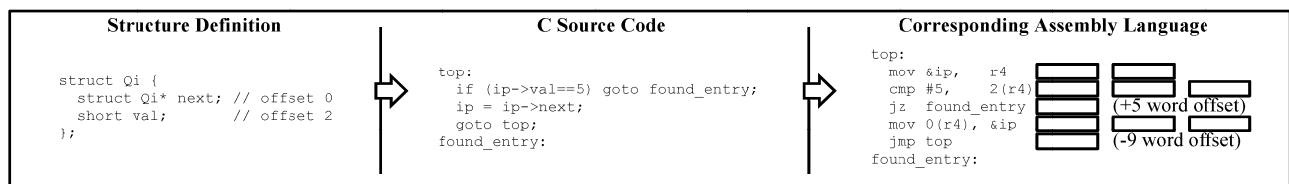


FIGURE 3.
COMPOSITE DATA TYPE EXAMPLE INCLUDING ILLUSTRATION OF MEMORY ALLOCATION FOR INSTRUCTION WORDS TO ENABLE COMPUTATION OF BRANCH TARGETS

COMPOSITE DATA TYPES

The course also includes labs in which students manipulate composite data types declared using C's struct primitive. Lectures and exercises examine the memory representation of abstract data types such as trees and linked-lists whose members are defined as structs. A linked list example is illustrated in Figure 3. Lab projects include programs that manipulate abstract data types (such as linked lists) 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.

UNANTICIPATED ELICITATION OF INTEREST IN COMPILATION

There was insufficient interest in an elective course on compilation for several years prior to the introduction of reforms described in this paper that the computer science department was unable to offer it. However, a sufficient fraction of the students attending the reformed course were engaged by the compilation techniques examined for a course on compilation to be successfully offered in Fall 2008.

ASSESSMENT

There is both qualitative and quantitative evidence that the reforms are successful.

Prior to the introduction of the reforms described above, employers that regularly hired UTEP computer science students prior to adoption of objects-first curricula indicated that graduates were no longer prepared for work that involved programming in C. These same employers indicate that students that attended the post-reform classes had dramatically stronger mastery of C. Faculty teaching upper division courses that also observed substantially improved ability of students to program in C.

Entries in the first column of Table 6 approximately correspond to entries in the official course outcomes document related to topics discussed earlier in this paper. The second column specifies a reform described in this paper:

- A: incremental introduction of addressing modes
- G: Goto-C
- 2: 2-pass assembly
- P: parse trees
- C: composite data types

Entries in the third and fourth columns indicate the percentage of students who demonstrated competence in two semesters of the 2008-2009 academic year. In this context we define competence as the ability to apply the skill appropriately in a familiar context. We attempt to isolate each skill and attempt to not penalize for obvious clerical errors. A single letter suffix indicates data source:

- M: Midterm exam
- F: Final exam

L: Lab exercises

T: Informal assessments from TAs

Of these four reforms, all reforms other than the incremental introduction of addressing modes (A) were introduced prior to or during the Fall 2008 semester. This (A) reform substantially reduced student confusion during early portions of the course that interfered with discussions of arithmetic operations and their implications on computations that affect control-flow throughout the course. As anticipated, students in the Spring 2009 section demonstrated greater aptitude in these skills (labeled by a parenthesized A) than the previous term.

We have concern that there is a large disparity between performance demonstrated in lab projects and examinations for pointer dereferencing. This was measured on the final exam by problems where students translated expressions from C to assembly. The score in lab assignments may be artificially elevated due to student ability to collaborate or artificially depressed on the exam due to the long delay between relevant assignments and the final examination.

TABLE 6. SKILL ASSESSMENT FROM AY 2008-2009

Skill	Reform	Fall 08	Spr 09
Arithmetic			
Serialization of expressions	P	70T	
Storage			
Static	A	73L 20F*	85F
Addressing			
Fixed (static)	A	40T	100F
Dereferencing pointer	A	80T	80L,50F
Array indexing	A	60T	75F
Struct index	A,C	50T	100L
Choosing correct mode	A	52F	80F
Control flow			
If/then to gotoC	G (A)	48F	90M
for/while to gotoC	G(A)	60T	90M
"Goto C" to assy	G(A)	92T	100M
Branch target offset	2	29F	60F
flags & subtraction order	G(A)	80T	95M
Machine code			
Branch target addr	2	75T	60F
Opcode & addressing mode	A		70F

CONCLUSION

The adoption of garbage-collected object-oriented languages as principal teaching languages have resulted in educational deficits observed at UTEP and elsewhere. The reforms described in this paper leverage the syntactical similarity of C and Java to simultaneously teach C and assembly language in a manner that compensates for this deficit and elicits future study of compilation techniques.

Future investigations will consider causes for differences between performance on examinations and lab exercises.

REFERENCES

- [0] Association for Computing Machinery, Computing Curricula 2001 Computer Science, ACM, <http://www.sigcse.org/cc2001/cc2001.pdf>
- [1] Freudenthal, E., Carter, B., Kautz, F., Ogrey, A., Preston, R., Walton, A., "Integration of C into an Introductory Course in Computer Organization," Proc. ASEE Annual Conference, 2009.
- [2] Freudenthal, E., Carter, B., *A Gentle Introduction to Addressing Modes in a First Course in Computer Organization*, Proc. ASEE Annual Conference, 2009
- [3] Dewar, Robert and Schonberg, Edmond, "Computer Science Education: Where are the Software Engineers of Tomorrow," *STSC Crosstalk*, January 2008.
- [4] Patt, Yale and Patel, Sanjay, *Introduction to Computing Systems*. McGraw Hill, 2004. ISBN 0-07-121503-4.
- [5] Patt, Yale, "Education in Computer Science and Computer Engineering Starts with Computer Architecture," *ACM 1996 proc. 1996 Workshop on Computer Architecture Education*.
- [6] Teller, Patricia; Nieto, Manuel; and Roach, Steve, "Combining learning strategies and tools in a first course in computer architecture," *IEEE proc. 2003 Workshop on Computer Architecture Education*.

AUTHOR INFORMATION

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

Brian A. Carter earned a B.S. at UTEP in computer science in 2008 and now works at Lockheed-Martin as a software engineer.

Rafael Escalante is a graduate student studying Computer Science at the University of Texas at El Paso.