AC 2009-2175: A GENTLE INTRODUCTION TO ADDRESSING MODES IN A FIRST COURSE IN COMPUTER ORGANIZATION

Eric Freudenthal, University of Texas, El Paso

Eric Freudenthal is an Assistant Professor of Computer Science at the University of Texas at El Paso.

Brian Carter, University of Texas, El Paso

Brian Carter earned a B.S. in Computer Science at the University of Texas at El Paso. He is now a software engineer at Lockheed Martin.

A Gentle Introduction to Addressing Modes in a First Course in Computer Organization

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.

Our previous papers reported 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, along with their implementations in machine code. While this integrated approach to introducing control-flow structures has been successful, many students have been confused by the large number of different addressing modes. The present paper describes further extensions of this integrated C-and-assembly language pedagogical approach in which addressing modes are introduced incrementally as solutions to pragmatically motivated problems. Initial results, as measured by quizzes and in-class exercises, are highly encouraging.

Introduction

We report on reforms to a 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[4] 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.[5] While these deficits are not common at schools with architecture-first curricula,[3,4,5] 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.

We exploit 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.

Our previous reports describe the tools used in the course[1] and the exploitation of transformations used by compilers to translate simple control-flow (if-then-else), composite types, and expressions to assembly language.[2] We observe that students have the most trouble mastering addressing modes and exploitation of arithmetic condition codes. After these reforms were implemented, we observed that students had sustained confusion regarding the semantics and roles of the available addressing modes which was frustrated by their need to simultaneously reason about numeric representations and arithmetic condition codes. Subsequent reforms addressed this confusion through the incremental presentation of addressing modes in a manner that permits students to master these concepts sequentially rather than concurrently.

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, 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, as presented in class.

Instruction			
High nibble	Nibble$_2$	Nibble$_2$	Low nibble
Operation	2	9	2

Extension Word 1
Address of source operand

Extension Word 2
Address of dest. operand

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.

Table 2. Example of early arithmetic code snippet translation projects that use only absolute addressing mode. On this processor, operation codes for two-operand instructions are specified by the most significant nibble and 0x292 specifies that both operands are "absolute" direct-mode addresses stored in extension words.

C Source Code	Assembly Language	Machine Code
`short a, b,`	`      .data`	
`one=1;`	`a:    .word 0`	`1000: 0000`
	`b:    .word 0`	`1002: 0000`
`a = b + 1;`	`one:  .word 1`	`1004: 0001`
	`      .text`	
	`      mov &b, &a`	`2000: 4292 1002 1000`
	`      add &one, &a`	`2006: 5292 1004 1000`
`long a;`	`      .data`	
	`a:    .word 0;low word`	`1000: 0000`
`a += 0xdeadbeef;`	`      .word 0;high word`	`1002: 0000`
	`onel:.word 0xbeef`	`1004: beef`
	`      .word 0xdead`	`1006: dead`
	`      .text`	
	`      add &onel, &a`	`2000: 5292 1004 1000`
	`      addc &onel+2, &a+2`	`2006: 6292 1006 1002`
`unsigned short a;`	`      .data`	
	`a:    .word 0`	`1000: 0000`
`if (a >= 1)`	`one:  .word 1`	`1002: 0001`
`  a++;`	`      .text`	
	`      cmp &one, &a`	`2000: 9292 1002 1000`
	`      jc isNeg`	`2006: 2d03 (+3 words)`
	`      add &one, &a`	`2008: 5292 1002 1000`
	`isNeg:`	`200c:`

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.

Introduction of Registers and Indexed Addressing Mode

As illustrated in Table 3, which is 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.

Table 3. Encoding of two-operand (binary) instructions and addressing modes.

High nibble (3)	Nibble$_2$	Nibble$_1$			Low nibble (0)
Operation	source register	A_d	Byte	A_s	destination register

A_a	Register	Syntax	Mode	Description
0	N	**Rn**	**Register**	Operand is the contents of Rn
1	N	**x(Rn)**	**Indexed**	Operand in memory at Rn+x (x within extension word)
2	N	**@Rn**	**Indirect**	Operand in memory at address held in Rn.
3	N	**@Rn+**	**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
			A_d: Only addressing modes 0 and 1 are available for the destination operand.	

Thus, absolute addressing mode used in early examples of two-operand instructions (see Table 1) is a variant of indexed addressing mode in which R2 provides a zero offset.

Register addressing mode is specified by using zero for either A_d and A_s and is therefore relatively straightforward for students to comprehend. Registers are initially introduced as convenient storage for temporary values. Later they are exploited by indexed addressing modes to provide a mechanism to implement pointers.

Thus, indirect modes are introduced together after students have mastered the use of arithmetic operations and branching including use of arithmetic flags. While indirect mode provides the most efficient mechanism to dereference pointers (it is only available for source operands), we delay teaching it until after students are comfortable with pointer dereferencing using indexed mode as illustrated by example exercises in Table 4. Thus, by initially only presenting two-operand arithmetic instructions with operands in absolute, register, and indexed modes, students are able to write program fragments that translate arithmetic expressions, including pointers and control-flow statements, using only three addressing modes.

Table 4. Dereferencing pointers using indexed addressing mode.

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

Remaining Addressing Modes

The MSP-430's remaining addressing modes are subsequently introduced as optimizations, such as:

- Indirect autoincrement addressing mode implements a two-cycle "pop" operation that, when used in conjunction with the program counter (R0), also implements the MSP-430's immediate addressing mode.
- The small set of frequently used constants that can be efficiently specified using arcane combinations of A_s and R2 and R3.

While students are expected to be familiar with the existence and usage of these modes, their instructional value is primarily limited to demonstrating the value of peephole optimizations and motivating discussion of issues in instruction set design.

Initial Evaluation

Grades from in-class exercises provide strong evidence that this approach is more effective than a traditional enumeration of addressing modes. Additionally, far fewer students are confused about the addressing mode semantics than in previous years.

Dividing the material being taught to students by addressing mode has an additional benefit: the assembly language instructions associated with each addressing mode are often logically related to each other. For example, a global variable's lifetime is the same as the entire program's lifetime; thus, it makes most sense to store global variables at fixed addresses in memory. This is an excellent opportunity for the introduction of direct addressing mode, which enables the necessary direct access to a fixed memory location. However, direct memory access is slow: students would next be introduced to registers and register addressing modes. Thus, each introduction of a new addressing mode serves to address an existing problem that the students already understand.

Introducing material incrementally by addressing mode in a manner that solves problems allows students to completely understand the usefulness of all aspects of the addressing mode: students begin to hypothesize that other instructions would be useful with each addressing mode. In this way, students fully understand the significance of instructions and addressing modes, often even before they are formally introduced to the class.

References

[1]Eric Freudenthal, Brian Carter, Frederick Kautz, Alexandria Ogrey, Robert Preston and Arthur Walton, Integration of C into an Introductory Course in Machine Organization, Proc. ASEE Annual Conference, June 2008.

[2]Eric A. Freudenthal, Brian A. Carter, Frederick F. Kautz, and Alexandria N. Ogrey, Work in Progress - Combined Introduction of C and Assembly with a Focus on Reduction of High-level Language Constructs. Proc. Frontiers of Education, 2008.

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

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

[5]Association for Computing Machinery, Computing Curricula 2001 Computer Science, ACM, http://www.sigcse.org/cc2001/cc2001.pdf

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

[7]Steve Underwood, MSPGCC project documentation. http://mspgcc.sourceforge.net/