A Simple but Realistic Assembly Language for a Course in Computer Organization

Eric Larson, Moon Ok Kim
Seattle University, elarson@seattleu.edu, kimm@seattleu.edu

Abstract **- Computer science curriculums, constantly evolving to include new material and methodologies, have reduced the amount of time spent on low-level computer hardware and organization. Our institution recently combined a course on computer organization and a course on assembly language programming into one course covering both topics. The choice of assembly language is a critical decision that contributes to the success of the course. ANNA (A New Noncomplex Architecture) is a new 16-bit instruction set architecture that is similar to MIPS but has fewer instructions. The instruction set, while small, is sufficient in illustrating how high-level languages are translated into assembly, how to design a CPU datapath, and how to implement pipelining. Real-world assembly languages such as IA-32 or MIPS have many subtleties that complicate the learning experience for students and consume valuable class time. This paper describes the ANNA assembly language and the assembler and simulator tools that can be used in the classroom. In addition, the paper describes how ANNA can be used effectively in a combined course on assembly language programming and computer organization. ANNA was used in three courses with very positive results based on our observations and feedback from students.**

Index Terms - assembly language, computer architecture, computer organization, computer science education.

INTRODUCTION

Curriculums are constantly evolving in computer science departments worldwide. With new and upcoming areas such as bioinformatics and computer security, along with the increased use of high-level languages and specifications during software development, computer science curriculums have expanded coverage on these important topics. Unfortunately, this often means that less time is spent on low-level topics – digital design, computer organization, and assembly language programming.

Our institution recently went from requiring two hardware courses to one. In the two course sequence, the first course covered digital design and computer organization. It included the following topics: data representation, combinational and sequential logic, computer arithmetic, CPU organization/design, memory, and I/O. The course also had a significant design component. The second course was a traditional assembly language course. In addition to programming using an assembly language, students learned instruction set design principles and exposed to linkers and loaders. Both courses were five-credit quarter long courses (or equivalently three-credit semester courses).

The new single required course essentially combines these two courses into one five-credit quarter long course. The basic idea is that assembly language programming is added to the computer organization course. The assembly language programming portion comprises about one-third of the course. In this compressed format, the choice of assembly language is very important. It need not be overly complicated so students can quickly learn and program in the language. However, the language needs to be sufficiently adequate and realistic. This permits students to understand how high-level programming constructs (such as loops and function calls) are converted to assembly language and how to design a basic CPU (control and datapath).

Real-world assembly languages such as IA-32 or MIPS are large and complex and are not practical where assembly language programming is only allotted one third of the course. These languages have subtleties that make it difficult for the student to learn the underlying principles. A few specific examples include the IA-32 variable length instruction set, a plethora of addressing modes, and pseudoinstructions that map to multiple machine language instructions.

This paper describes a simple but realistic assembly language that was used the past three times the course was offered. The language and underlying instruction set architecture called ANNA (A New Noncomplex Architecture) is similar to MIPS but is 16 bits. ANNA has 16 instructions and has 16 general purpose registers. It is a RISC instruction set and a load-store architecture meaning that only the load and store instructions access memory. Base plus offset addressing is used for memory accesses and PC-relative addressing is used for the two conditional branch instructions.

In keeping with the simple design of ANNA, memory is word-addressable with a word size of 16 bits. The only two data types are integers and addresses (both 16 bits). Characters and floating point data values are not supported. This avoids confusing the student with assorted type size and conversion issues. Simplified input and output instructions merely cause the simulator prompt the user for an integer

and display the contents of a register onto the screen respectively.

An assembler and simulator are provided for ANNA including versions for both Windows and Linux. The source code is available allowing instructors to tailor the instruction set and its architecture to fit their needs.

RELATED WORK

Other educators have created simplified instruction sets for educational purposes. The Ant project from Harvard [1] has two variants. The 8-bit variant is designed to introduce assembly language concepts in CS1 courses. The language is similar to ANNA but does not have a jump and link register instruction that is needed for function calls. The 32-bit version is a full featured assembly language. Its underlying architecture contains support for exceptions, virtual memory, and supervisor mode. It makes a good choice for courses that combine computer architecture and operating systems. Hatfield et. al. [2] proposes using a simple MIPS-like instruction set for use in a computer organization and architecture. Their course emphasizes simulation and implementation rather than assembly language programming.

Recent textbooks have catered to this trend of combining courses in digital design, assembly language programming, and computer organization. The 3rd edition of Computer Organization and Design by Patterson and Hennessy [3] is largely a computer organization book but includes appendices on assembly language programming and digital design. They also use MIPS which is a rather large instruction set and has peculiarities could hinder students from learning the primary topics. Harris and Harris's book [4], Digital Design and Computer Architecture covers both topics fairly extensively. The book includes a chapter that is devoted to assembly language programming (MIPS). A comment in the preface of the book indicates that other universities have expressed a need of a combined digital design and computer architecture book to the authors.

Since most modern assembly languages employ general-purpose registers, we did not consider accumulator-based architectures. Some examples of simple accumulator-based architecture include MARIE by Null and Lobur [5] and SIC/XE by Beck [6].

DESCRIPTION OF ANNA

This section describes the architecture of the 16-bit ANNA (A New Noncomplex Architecture) processor. It contains 16 user-visible registers and an instruction set containing 16 instructions.

Some of the key features of ANNA:

- Memory is word-addressable where a word in memory is 16 bits or 2 bytes. An address is 16 bits corresponding to the 64 K words of memory available.
- Memory is shared by instructions and data. A 16 bit word either corresponds to an instruction, an integer, or an address. No other data types are allowed.
- ANNA is a load/store architecture; the only instructions that can access memory are the load and store instructions.
- The ANNA processor has 16 registers that can be accessed directly by the programmer, named `r0` through `r15`. Register `r0` always has the value zero.
- When the program is loaded into memory, the first instruction is placed in address zero. All registers, including the program counter, are set to zero.

I. ANNA Instruction Set

The 16 instructions of ANNA are presented in Table I.

TABLE I
ANNA INSTRUCTION SET

Opcode	*Operands*	*Description*
add	*Rd* Rs_1 Rs_2	Addition
sub	*Rd* Rs_1 Rs_2	Subtraction
lli	*Rd Imm8*	Set lower bits using immediate
lui	*Rd Imm8*	Set upper bits using immediate
and	*Rd* Rs_1 Rs_2	Bitwise and
or	*Rd* Rs_1 Rs_2	Bitwise or
not	*Rd* Rs_1	Bitwise not
shf	*Rd* Rs_1 Rs_2	Bit shift
lw	*Rd* Rs_1 *Imm4*	Load word from memory
sw	*Rd* Rs_1 *Imm4*	Store word from memory
in	*Rd*	Get a word from user input
out	*Rd*	Send a word to output
bez	*Rd Imm8*	Branch if equal to zero
bgz	*Rd Imm8*	Branch if greater than zero
jalr	*Rd* Rs_1	Jump and link register
halt		Halt the program

All of the logical and arithmetic operands (except for unary `not`) take three operands corresponding to a destination register and two source registers. The memory operations `lw` and `sw` both use base plus displacement addressing. This is the most popular addressing mode in instruction set design. It is used in the course to illustrate how local variables can be accessed as an offset from a stack pointer. It is also used to access arrays (assuming a constant index) and data members of structs.

Control instructions include two conditional branch instructions. Both of these instructions are PC-relative, just like their counterparts in MIPS. For function calls, the `jalr` (jump and link register) allows a programmer to jump to the callee function and save the return address. The `jalr` also serves as a jump register instruction by setting the link register to `r0`.

Input and output is accomplished using the `in` and `out` instructions. These instructions allow the user to input a value or display a value in the simulator. While this implementation is greatly simplified over actual I/O hardware, it allows students a means of testing their programs with different input combinations without being overly burdensome.

```
# Multiplication program

# Register usage:
# r1: constant one
# r2, r3: two numbers to multiply
# r4: final answer
# r5: stores sign of final answer
# (0-positive, nonzero - negative)

       # store constant one into r1
       lli r1 1
       lui r1 1

       # get two input numbers
       in  r2
       in  r3

       # check for negative numbers
       bgz r2 &posA
       not r5 r5           # flip sign
       sub r2 r0 r2        # negate
posA:  bgz r3 &loop
       not r5 r5           # flip sign
       sub r3 r0 r3        # negate

       # main loop: while r3 > 0
loop:  bez r3 &done
       add r4 r4 r2    # add another r2 to r4
       sub r3 r3 r1    # decrement r3
       bez r0 &loop

       # output answer with proper sign
done:  bez r5 &disp
       sub r4 r0 r4        # negate
disp:  out r4
       halt
```

FIGURE 1
SAMPLE ANNA ASSEMBLY PROGRAM

II. ANNA Assembler

The ANNA assembler reads an ANNA assembly language program and either generates a machine code file or a list of errors in the program.

The assembler has support for comments and labels. Labels can be used as addresses for instructions with 8-bit immediates (`lui`, `lli`, `bez`, `bgz`). Using `lui` and `lli` allows the address of an instruction or data value to be loaded into a register. For branches, labels can be used to specify the target address. The assembler automatically computes the proper PC-relative immediate.

There is one assembler directive: `.fill`. This directive uses a 16-bit immediate and directs the assembler to fill the next word of memory with the 16-bit value. This directive is intended to be used to declare global variables in memory. In keeping with the straightforward approach, no other assembler directives or pseudoinstructions are supported by the ANNA assembler.

Programs in ANNA can be divided into three sections. The first section always contains code (since the simulator starts executing at address zero). The second section is a data section where global variables can be declared using .fill directives. Finally, a third section is used for the stack. It simply contains one `.fill` directive preceded by a label indicating the start of the stack. The stack grows to higher addresses during the program. Less complex assembly programming exercises typically do not need data and/or stack sections. The notion of sections are merely a convention and is not enforced by the assembler.

A sample program written in the ANNA assembly languages if shown in Figure 1. This program multiplies two numbers entered by the user and displays the product. The program initially determines the sign of the product by checking the sign of the each input number. If a number entered by the user is negative, it is converted into a positive number. The main loop successively uses addition to arrive at the final answer. Before printing the answer, the number is converted into a negative number if the product is negative. This simple program could be improved by using a faster multiplication algorithm using shifts and by checking for overflow.

III. ANNA Simulator

The ANNA simulator starts by reading in a machine code file generated by the assembler and loading it into memory. Registers, including the program counter, are initialized to zero. Then, the simulator waits for user input. A screen shot of the simulator is shown in Figure 1.

There are six panes for the simulator:

- The Code pane shows the assembly instructions, the address for each instruction, and an arrow to the current instruction. It also shows which instructions currently have enabled break points set.
- The Registers pane shows the contents of each register including the program counter (PC). Any change to a register is automatically reflected in the display.
- The Memory pane allows the user to show the contents of memory of up to five addresses at a time. The user can enter an address to see its contents. As with registers, any changes to the memory location are automatically updated as the program runs.
- The Break Points pane allows the user to set break points at various points in the program. The program will stop executing if the PC matches one of the four break point addresses (if the break point is enabled).
- The Controls pane allows the user to control execution. The LOAD button displays a dialog box where a user can select a machine code file to load. The RESET button resets the current program to its initial state. The CONTINUE button causes the simulator to execute the program until a breakpoint is encountered, the program halts, or the user hits the STOP button. The NEXT button directs the simulator to execute one instruction.
- The Output pane displays output values from out instructions and other information. If an input

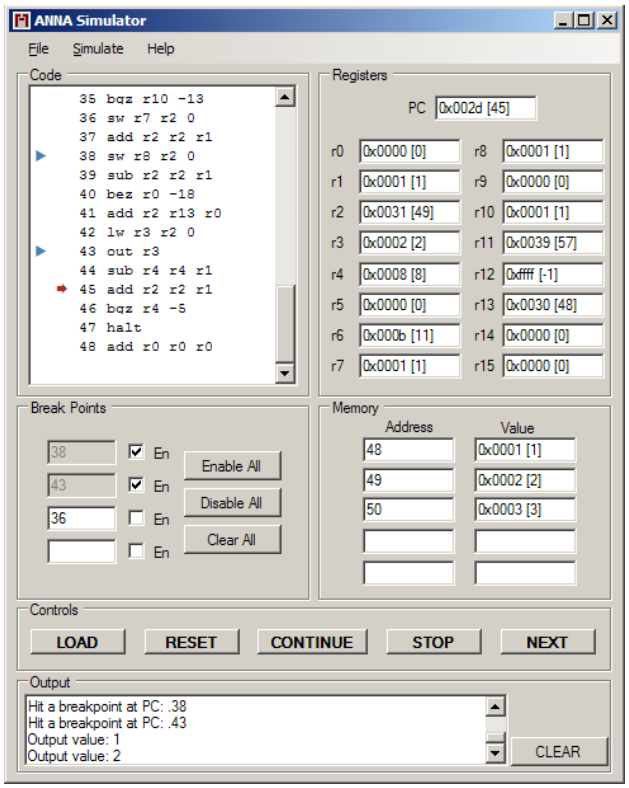

FIGURE 2
ANNA SIMULATOR

instruction is executed, a dialog box is displayed prompting the user for an input value.

All output numbers are displayed in both hexadecimal and in decimal as a two complement's signed number. Addresses can be entered in either decimal or hexadecimal. The addresses in the code pane are displayed in decimal. Since ANNA is word addressable and does not have any alignment restrictions, we choose to use decimal numbers (opposed to hexadecimal numbers) to represent addresses.

The Linux version provides similar functionality except that is completely text based like the GNU debugger `gdb`. The commands are shown in Table II.

TABLE II
ANNA SIMULATOR COMMAND REFERENCE (LINUX)

Command	*Description*
`c`	Continue program
`n`	Execute next instruction
`R`	Reset program
`r`	Print registers
`m` *addr*	Display memory contents
`t`	Toggle trace mode
`b` *addr*	Set breakpoint
`x` *addr*	Remove breakpoint
`X`	Remove all breakpoints
`h`	Help
`q`	Quit

IV. Documentation and Source Code

One concern when using a new tool created by the instructor is documentation as students do not have a book to fall back on when they get stuck. Students receive a guide that includes a description of the ANNA architecture and of each instruction, a section on how to write assembly files and use the assembler, a simulator reference, and a style guide that includes a sample program. They also receive a quick reference card that has the list of instructions and other useful information.

The source code for the assembler and simulator is available and is straightforward to modify. You may decide to alter the instruction set, reduce the number of registers, add new assembler directives, or enhance the simulator. The source code can also be used in conjunction with assignments. A sample assignment may ask the student to implement a new instruction, alter the addressing mode of certain instructions, or to add a new feature.

COURSE OVERVIEW

This section describes the computer organization course and how the ANNA instruction set and tools are used. The course was taught twice using ANNA. The course met for fifty hours during the quarter and utilized a traditional lecture format with some time devoted to in-class exercises.

It is a sophomore level course and requires the first two C++ programming courses as prerequisites. The course is required by all computer science majors. It is not required, and typically not taken, by students in other majors (including computer engineering). One of the two primary goals of the course is for students to obtain an understanding of computer hardware. The second primary goal is for students to learn how C++ code is translated into assembly and subsequently executed in hardware at the logic gate level. In addition, there are key fundamental computer science concepts that are required to be covered in this course: floating point representation (approximation and errors), finite state machines, and caching. A breakdown of class content is shown in Table III.

TABLE III
TOPIC COVERAGE IN COURSE

Command	*Lecture Hours*
Data representation	4
Digital design	10
CPU components and organization	3
Assembly programming	15
Datapath and pipelining	8
Memory and caching	5
Input / output	2
Review / exams	3

The assembly language programming unit starts by presenting the ANNA instruction set and its architecture. Significant time is spent on how to code loops, pointer operations, array references, and function calls using ANNA assembly language. In the latter part of the unit, focus shifts to instruction set design issues. Students are exposed to

accumulator and stack based instruction sets. We also discuss as a class, the pros and cons of variable length instruction sets and other design decisions. To illustrate the difference between ANNA and a real instruction set, students are given the Intel Pentium IV instruction set and is discussed in class. The unit concludes with a brief overview of the entire compilation process including linking and loading.

During the unit, students complete two assembly language programming assignments. The first assignment involves two to three smaller programming exercises. The second assignment consists of one larger program that requires the student to implement function calls. Grading is predominantly based on functionality but style is also considered (especially comments explaining how their registers were used). Exams include questions where students had to translate C++ code fragments to ANNA assembly and questions that ask students to convert between ANNA assembly and ANNA machine language.

The ANNA assembly language is also used in the datapath and pipelining unit. As a class, we design a single-cycle datapath for ANNA including a control ROM. This is followed by designs for multiple-cycle and pipelined implementations. Each of these designs use 13 of the 16 ANNA instructions (omitting `in`, `out`, and `halt`). Since the ANNA instruction set is small and word addressable, the datapath can presented in a clear and concise manner. Yet, the instruction set illustrates all of the key datapath elements in a CPU. For instance, tracing a `lw` instruction through the datapath requires the following: fetching the instruction, decoding the instruction, reading the address register, selecting the proper operands, computing the effective address, accessing memory, and writing the result to the register file. Homework is pencil and paper – we do not use any logical design software. A typical homework question asks the student to add instruction X (where X is a new assembly instruction that is magically added to ANNA) to the datapath.

Comparing the course content to the prior two quarter sequence, all topics are covered in less depth. Material that was omitted when going from two courses to one includes: floating point arithmetic (floating point representation is covered), carry-save addition, logic equation simplification (Karnaugh maps, Quine-McCluskey method), digital design process, microprogramming, and actual input/output in assembly . As you can see, most of the omitted material involve low-level hardware. In essence, logic design is only taught in enough depth so that students can understand the microarchitecture of a CPU.

FEEDBACK

The computer organization course was taught using the ANNA ISA three times: fall 2006, spring 2007, and spring 2008. In the first two quarters, the Linux assembler and simulator were used. At the end of the quarter, the students were given a questionnaire that contained questions pertaining to how well the course objectives were met and what they thought of the ANNA tools. This section discusses the results based on the questionnaires and experience from the instructor.

During the spring 2008 quarter, students were permitted to use either the Windows or Linux tools. In addition, the number of registers used in ANNA was reduced from 16 to 8. Having only seven registers (since register `r0` is always zero) forces the students to use the registers more judiciously. This is especially true for function calls where one register is the stack pointer, another stores the callee address, and a third saves the return address. This leaves only four registers for temporary values and parameter passing enforcing the need for a register calling convention. Since the spring quarter was not complete before the final submission deadline, surveys were not but we did gather informal feedback from the class.

I. ANNA Observations

With regard to assignments involving ANNA, all of the students felt they learned a lot when completing the assignments. All but one student felt that assembler and simulator were easy to use. All but one student (the same student as before) agreed that the documentation on ANNA was sufficient. Students had different opinions regarding the difficulty and time commitment with a majority thinking that the assignments were somewhat difficult and somewhat time consuming. Since we strive to have challenging assignments, these results are encouraging.

In the first quarter (fall 2006), a few students (including the student who didn't like ANNA above) felt that not enough class time was devoted to implementing function calls in assembly. As a result, more lecture time was spent on this topic in the spring. This came at the expense of some minor topics in other sections. Besides this change, there were no other significant differences between the two course offerings.

Our experience with ANNA is that we really felt that students were learning when doing the programming. While the Linux version seemed to work well, we felt a Windows version would aid in the understanding since students can see the changes to the registers and memory immediately after executing an instruction.

In the spring quarter, students could use either the Windows tools or the Linux tools. Most students, but not all, opted to use the Windows versions of the tools. Based on informal feedback, students felt that the tools were easy to use and aided in the learning of the material. Students also suggested some enhancements to the tools. Among the top suggestions were adding line numbers for the assembler editor so they could easily find errors and some mechanism to map labels to addresses in the simulator.

II. Course Outcomes

With regard to course outcomes, students are asked whether each of the main topics of the course on a scale from 1 (not covered) to 5 (extensively covered). Most students thought most of the topics were well covered (4 on the 5 point scale). No topic received a 2 or lower from any student but very few topics received a 5. This is not surprising considering the inherent lack of depth in the compressed course.

From our observations, we agree with the students on the course outcomes. The area most lacking in our opinion is in digital design. Without teaching a simplification technique such as Karnaugh maps, it is hard for students to design circuits and understanding the process of converting a finite state machine into a hardware circuit. Spending more time on this section would help students in the datapath section. To accommodate this change, we would likely eliminate or greatly reduce the coverage of the multiple cycle datapath. It is our opinion that the single-cycle datapath is sufficient for understanding the key concepts of how a CPU is organized. Pipelining would still be included as it is a good example of parallelism.

Based on results so far, the spring 2008 class showed no significant difference in terms of class performance on exams and assignments despite having access to the Windows tools. However, there was one notable area of improvement: students in the spring 2008 class had a better grasp of function calls. It is important to note that this improvement could be due to a variety of reasons: use of Windows tools, reducing the number of registers, different programming assignments, and variations in the lecture.

CONCLUSION AND FUTURE WORK

This paper describes ANNA, a simple MIPS-like assembly language. ANNA is a 16-bit architecture with 16 instructions. It was successfully used in two offerings of a course that combined assembly language programming and computer organization.

One change to the ANNA assembly language is to include an `addi` (add immediate) instruction. Incrementing or adding by a constant amount is a common operation for loops and stack pointer updates. Without this instruction, students often dedicated register `r1` for the constant one throughout the program, not a common practice in assembly language programming.

Students suggested possible enhancements to the Windows tools. Two suggested enhancements, mentioned earlier, are to add line numbers to the assembly editor and to incorporate labels into the simulator. The latter suggestion can be accomplished by storing the labels and the actual instructions in the machine code file and modifying the code listing window in the simulator.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their thoughtful reviews and advice. We would also like to thank the students taking CSSE 251 at Seattle University during the Fall 2006, Spring 2007, and Spring 2008 quarters for their valuable feedback on ANNA and the course itself.

REFERENCES

[1] D. Ellard, D. Holland, N. Murphy, M. Seltzer. On the Design of a New CPU Architecture for Pedagogical Purposes. Proceedings of the Workshop on Computer Architecture Education. May 2002.

[2] B. Hatfield, M. Rieker, L. Jin. Incorporating Simulation and Implementation into Teaching Computer Organization and Architecture. Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference. Oct. 2005.

[3] D. Patterson, J. Hennessy. Computer Organization and Design: The Hardware / Software Interface, 3rd Edition. Elsevier. 2005.

[4] D. M. Harris, S. L. Harris. Digital Design and Computer Architecture. Elsevier. 2007.

[5] L. Null, J. Lobur. The Essentials of Computer Organization and Architecture, 2nd Edition. Jones and Bartlett Publishers. 2006.

[6] L.L. Beck. System Software: An Introduction to Systems Programming, 3rd Edition. Addison Wesley Longman, Inc. 1997.

AUTHOR INFORMATION

Eric Larson Assistant Professor, Seattle University, elarson@seattleu.edu.

Moon Ok Kim, Undergraduate Student, Seattle University, kimm@seattleu.edu.