The ANT Architecture--
An Architecture for CS1

The ANT Architecture — An Architecture for CS1

Daniel J. Ellard, Penelope A. Ellard, James M. Megquier, J. Bradley Chen, Margo I. Seltzer
Harvard University

A central goal of high-level programming languages, such as those we use to teach introductory computer science courses, is to provide an abstraction that hides the complexity and idiosyncrasies of computer hardware. Although programming languages are effective at achieving this goal, certain properties of computer hardware cannot be hidden, or are useful for students to know about. As a consequence, many of the greatest conceptual challenges for beginning programmers arise from a lack of understanding of the basic properties of the hardware upon which computer programs execute.

To address this problem, we have developed a simple virtual machine called ANT for use in our introductory computer science (CS1) curriculum. ANT is designed to be simple enough that a CS1 student can quickly understand it, while at the same time providing an accurate model of many important properties of computer hardware. After two years of experience with ANT in our CS1 course, we believe it is a valuable tool for helping young students understand how programs and data are represented in a computer system.

1 ANT Overview

The guiding philosophy of the ANT architecture is simplicity, but not at the cost of functionality. The result is an architecture that is simple in every important aspect: easy for students to learn, easy to implement, and whose assmbly language is easy to assemble. Despite this simplicity, the ANT architecture is rich enough to support many interesting applications and to accurately model how computers execute programs. The ANT instruction set follows the RISC design philosophy. It uses fixed-width instructions with opcodes, registers, and constants in fixed positions so that the instructions are very easy to decode. The only ANT instructions that directly access memory are the load and store instructions. The ANT register set consists of 14 general-purpose registers and two special-purpose registers. There are no status registers or condition codes, and the program counter is not directly accessible to the programmer.

ANT uses 8-bit two's-complement integers, 8-bit addresses, and 16-bit instruction words. In its current implementation, it uses separate address spaces for instructions and data, making it possible to have as many as 256 16-bit instructions and 256 8-bit words of data simultaneously.

The ANT architecture also includes a single `sys` instruction that can perform a variety of tasks, depending on its parameters: it can be used to halt the processor, dump the contents of registers and memory (for debugging), and to read and write characters and integers.

There are currently only twelve instructions in the ANT architecture, including the `sys` instruction. Although this seems like a small number of instructions, we have found this set to be adequate for our use and have even considered *removing* some infrequently-used instructions

Despite its small number of instructions and tiny address space, the ANT architecture is full-featured enough to support a wide variety of interesting programs of a level of complexity similar to the first several programming assignments in a CS1 course. Some examples include:

- `hello.asm` – The "Hello World" program, written in ANT assembly language (3 instructions).
- `echo.asm` – Copy stdin to stdout one character at a time (7 instructions).
- `reverse.asm` – Reads lines from the user, and prints them out in reversed order (7 instructions).
- `sort.asm` – Bubble-sorts numbers read from user (40 instructions).
- `rotate.asm` – Prints ``rotated'' versions of a string (52 instructions).
- `hi-q.asm` – Plays the game of Hi-Q (253 instructions).

In fact, it is possible to implement a slightly simplified ANT virtual machine in ANT and execute simple ANT programs on this virtual machine.

The limited size of the ANT address space makes it easy to understand and debug ANT programs—in fact, the ANT debugger can display the contents of all of the registers and the *entire* contents of data memory in a single 24-by-80 text window. This simplicity is a crucial feature that distinguishes ANT from a "real'' architecture. The smallness of the machine means that programs never get too big or complex. As a result, there are relatively few bugs that our students cannot find and fix themselves, which increases their self-confidence and reduces the time required from teaching staff.

Because the complete machine specification for ANT fits on fewer than seven pages, students can be expected to understand every detail of the specification. We consider this to be essential, and we believe that this would be impossible for any real architecture.

2 ANT Programming

The ANT programming environment consists of an assembler, interpreter, and debugger. The assembler converts an assembly language source file into a simplified format that the interpreter can directly read and execute.

The debugger is an extended version of the interpreter. It allows the user to set and remove breakpoints, generate a program trace, single step through a program, or reinitialize the processor, as well as examine the contents of registers and data memory and disassemble the instructions.

It would be easy to combine the assembler, interpreter, and debugger into one program. We did not choose this approach, however, because it would make some of our assignments more difficult—in 1997, we had students write both an ANT interpreter and an ANT assembler, and we believe that keeping these programs separate helped the students by giving them a concrete example to emulate.

3 How Do We Use ANT in CS1?

We introduce the ANT architecture in the sixth week of CS1, after the students have mastered the basics of C programming, including loops, conditional execution, and arrays, but before they are exposed to pointers, structures, or dynamic memory allocation.

Our purpose for teaching machine architecture and assembly language programming in CS1 is to focus on basic issues of data representation and machine architecture. We use ANT extensively to demonstrate these issues. However, ANT has proven flexible enough to be tied in to a number of other important concepts, as described below.

3.1 Data Representation and Machine Architecture

Our CS1 begins to introduce elements of data representation such as binary and hexadecimal notation, two's-complement arithmetic and ASCII codes early in the semester, followed immediately by the revelation that computer programs themselves are stored in "machine language'' using a very similar representation, and are executed in an entirely mechanical manner.

We use several small ANT programs to illustrate these points, using the ANT debugger to demonstrate how the state of the ANT machine changes as it executes each instruction. Finally, to reinforce these ideas, we have students write small amounts of ANT assembly language code themselves.

3.2 Pointers

Our CS1 is taught in C, and pointers in C are a subject that confuses many CS1 students. Over the past several years, we have experimented with different methods of reducing the initial problems with pointers. Teaching a small segment on assembly language using ANT before teaching pointers seems to help students over the initial hurdles of understanding pointers—once students are familiar with the relatively simple semantics of `load` and `store`, the concepts behind C pointers are much easier to grasp. Similarly, students who have written ANT code to stride through arrays generally have an easier time understanding address arithmetic and the convenient and crucial relationship of arrays and pointers in C.

3.3 Design and Style

It is nearly impossible to write nontrivial programs in assembly language without careful planning and design. Similarly, it is very difficult to understand, modify, or debug improperly organized or uncommented assembly language.

This is even more true for ANT assembly language programs, because the current assembler does not support macros or similar constructs that would make the resulting code more readable. Therefore, the ANT assembly programming exercises force students to think ahead before beginning to write their ANT programs, allocating registers and choosing label names with some care. They must organize their code in a readable manner and carefully document it. We believe that this is a pivotal experience in the course for many of our students—particularly the students who come in to CS1 with some prior programming experience, and therefore manage to hack through the first few assignments without a clear design or documentation. The ANT programming assignments are generally two short problems, each of which can be completed in approximately one page of well-commented code—not dauntingly difficult, but challenging enough so that the benefit of thinking ahead is clear.

3.4 Virtual Machines

ANT is currently implemented only as a virtual machine, although there has been some interest in creating a hardware implementation for use in one of the introductory hardware architecture courses. Since ANT is a virtual machine, we can conveniently introduce the topic of virtual machines in the same context as machine architecture.

To reinforce these concepts, we have students implement nearly all of an ANT virtual machine themselves. We supply the code to load ANT programs from file, and code to implement most of the `sys` instruction, because at this point in the semester our students have not seen file I/O yet. Our students have not yet seen structures, pointers, or dynamic memory allocation, but the assignment is designed so that it does not require these constructs. A well-designed solution to this assignment requires approximately 300 lines of commented C code.

For this assignment, we provide the students with an automated test suite that they could use to test their ANT implementations for conformance to the assignment specification. This introduces them to the idea of rigorous, automated testing. We also encourage students to write their own tests, to augment our test suite.

3.5 Compilers and Assemblers

Even after being exposed to the ideas of machine language and assembly language, it is not clear to many students how these ideas are related to higher-level languages. To demystify this relationship, we “hand compile’’ some simple C programs into ANT, leaving the students with an intuition of how some parts of compilation can be performed.

In an assignment near the conclusion of the semester, students write a substantial fraction of an assembler for ANT itself, allowing them to gain deeper insight into how translation from one language to another can be accomplished. The students are required to implement functions to translate assembly language statements into machine code, check for illegal or ill-formed instructions, maintain a symbol table of labels, and perform backpatching. The solution typically required 100 lines of code for the symbol table (not counting reuse of code from libraries developed in earlier assignments) and approximately 500–700 lines of code for the rest of the assembler.

With this assignment complete, students could write their own programs in ANT assembly language, assemble them with their own ANT assembler, and execute them on their own ANT virtual machine.

4 Conclusion

We have found ANT to be effective in our CS1 course. It allows introductory-level students to develop good intuition about concepts such as pointers and compilation, while avoiding the complexity of real architectures. It also provides an effective way to discuss topics such as virtual machines and automated testing, while the corresponding programming assignments give ample opportunity for students to strengthen their programming and design skills.

For more details on ANT, including tutorial documentation for students, example programming exercises, and the ANT specification, visit the ANT home page[1]. Please contact Dan Ellard (`ellard@eecs.harvard.edu`) for more information.

1. `http://www.eecs.harvard.edu/~ellard/ANT`