The ANT-Architecture--An Architecture for CS1

The ANT Architecture – An Architecture For CS1

Dan Ellard, Penelope Ellard, James Megquier, J. Bradley Chen

A central goal in 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 very effective at achieving this goal, certain properties of computer hardware cannot be hidden, or are useful 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 actually represented in a computer system.

This paper gives a short introduction to ANT. We start with a brief description of the architecture, and then describe how we use ANT in our CS1 course. We include specific examples that demonstrate how ANT can give students intuition about pointers, the representation of data in memory, and other key concepts.

1 A Description of ANT

ANT is a simple and extremely small architecture. 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 and to implement as a virtual machine, and with a machine language that is easy to assemble. Despite this simplicity, the ANT architecture is rich enough to support many simple but interesting applications, and to accurately model how computers actually execute their 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 first four bits of every instruction are the opcode. The second four bits always represent a register number. The final eight bits represent either a pair of register numbers, a register number and a 4-bit constant, or an 8-bit constant. Figure 1 describes the ANT instruction formats. Note that the use of four-bit fields means that an ANT instruction can be trivially written or read in hexadecimal.

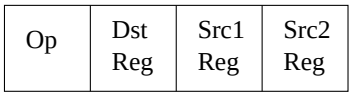
Also following the RISC philosophy, the only ANT instructions that directly access memory are the load and store instructions.

The ANT register set consists of 14 general-purpose registers and two special-purpose registers. The first special-purpose register, `r0`, al-

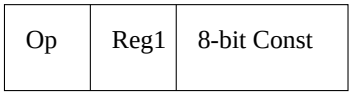
Figure 1: The ANT Instruction Formats

All fields are four bits wide, unless otherwise noted.

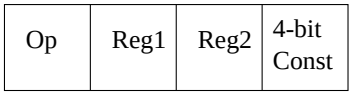
- The instruction format for register-to-register instructions.



- The `lc` (load constant), `inc` (increment), `jmp` (jump) and `sys` (system call) instructions use a variant of the standard format that allows for an 8-bit constant.



- The `ld` (load) and `st` (store) instructions use a four-bit constant in place of the third register specifier.



ways contains the constant zero, and the second, `r1`, is used to hold results related to the previous instruction (such as overflow or underflow from arithmetic operations). 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. These address spaces could be combined, but this would crowd the tiny address space even further, and introduce the possibility of new kinds of programming bugs, such as programs that accidently overwrite their own instructions.

The ANT architecture also includes a single `sys` instruction, which 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), or to read or write characters or integers from or to standard I/O.

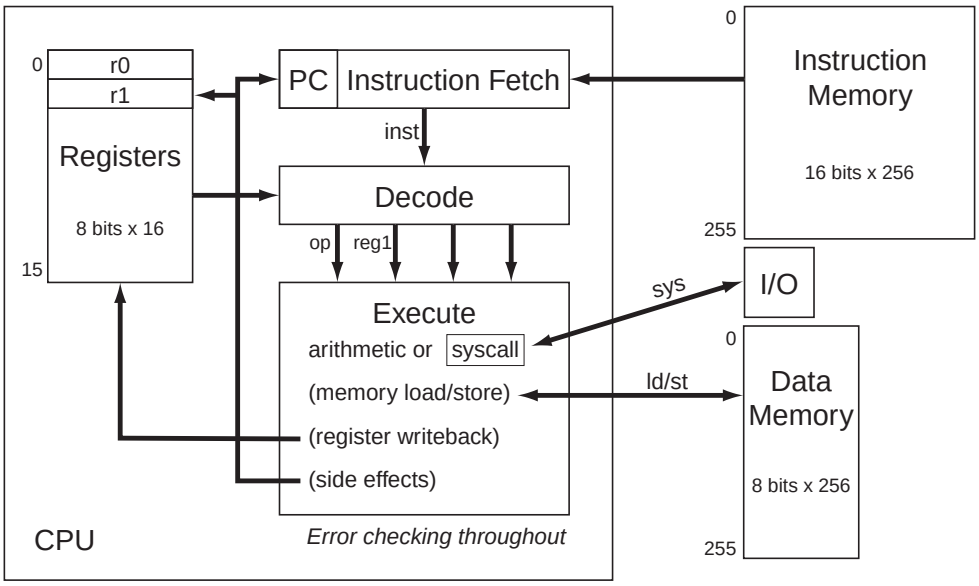
A block diagram of the ANT architecture is shown in Figure 2.

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

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. Uses

Figure 2: A block diagram of the ANT architecture.



three instructions. This program is shown in Figure 6.

- `echo.asm` - copy stdin to stdout one character at a time. Uses seven instructions.
- `reverse.asm` - reads lines from the user, and prints them out in reversed order. Uses 27 instructions. This program is shown in Figure 7.
- `sort_example.asm` - bubble-sort numbers read from user. Uses 40 instructions.
- `rotate.asm` - prints "rotated" versions of a string. Uses 52 instructions.
- `hi-q.asm` - plays the game of Hi-Q. Uses 253 instructions. The students write a Hi-Q game in C as part of an earlier assignment, so they can easily understand this program.

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

We typically assign short assembly language programs, such as `reverse.asm`, as a part of student assignments, but we do not assign longer assembly language programs. We do not expect our students to become seasoned or even particularly proficient assembly language programmers; our goal of giving them some intuition about architecture is achieved well before that point. Nonetheless, ANT programs such as `rotate` and `hi-q` provide students with examples of assembly language programming, and demonstrate that interesting programs can be implemented for the ANT virtual machine. This gives motivated students an opportunity to exercise their creativity and explore the design space provided by a simple virtual machine.

The limited size of the ANT address space makes it very 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.

Although ANT is fully adequate for achieving our educational goals, there are many things that were purposefully left out of the architecture, in order to keep it small and simple. ANT contains no support for interrupts, memory protection, any kind of MMU, or any of the other features that would be required to implement a full-fledged operating system or run more than one program at a time. ANT also does not include a number of architectural features that are common in modern processors– it does not include any sort of cache, nor does it have an execution pipeline. It does not include support for floating point arithmetic, and supports only a limited set of I/O operations. Although it has some instructions that can be used to implement a stack, there is no hardware support for a stack pointer. (We do not ask students to write ANT programs that would require a stack.) These features would be helpful for writing more complex ANT programs, or to provide a more realistic example of a computer architecture– but detailed realism is not our goal. Even so, many of these features are mentioned in discussions of how we could make the architecture more powerful and realistic.

An extension we have considered is increasing the data word size to 16 bits. In considering such features we weigh the advantages of enabling new applications against the risk of degrading the simplicity and elegance of the architecture, and generally give priority to the latter. Because the complete machine specification for ANT is less 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, with debugging functionality added. 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– this year, we had students write both an ANT interpreter and an ANT assembler, and we believe that keeping each of these programs separate helped the students by giving them a concrete example to emulate.

2.1 ANT Assembly Language

This section briefly describes the instructions and system calls available in ANT assembly language. For a complete specification of the assembly language, please refer to the documentation mentioned in the conclusion of this paper. The ANT instructions and their mnemonics are listed in Figure 3, and the system calls are described in Figure 4. The notation used in Figures 3 and 4 is documented in Figure 5.

Note that for all instructions except `sys`, register `r1` is always updated *after* all register reads

for the instruction are complete, so that it is always safe to use `r1` as a source register for these instructions. (The current behavior of `sys` with respect to `r1` is a relic of an early design decision, and will be removed in the next revision of the ANT architecture.)

2.2 ANT Assembler Directives

- **Comments** - A comment begins with a `#` and continues until the following newline. The only exception to this is when the `#` character appears as part of an ASCII character constant.
- **Constants** - Several ANT assembly instructions contain 8-bit or 4-bit constants. The 8-bit constants can be specified in a variety of ways: as labels, as decimal, octal, or hexadecimal numbers, or as ASCII codes, using the same conventions as C. The value of a label is the index of the subsequent instruction in instruction memory for labels that appear in the code, or the index of the subsequent `.data` item for labels that appear in the data. The 4-bit constants must be specified as unsigned numbers, using decimal, octal, or hexadecimal notation. ASCII constants and labels cannot be used as 4-bit constants, even if the value represented fits into 4 bits.
- **The `_data_` Label** - The special label `_data_` is used to mark the boundary between the instructions of the program, which must appear before the `_data_` label, and the data of the program, which must appear afterward.
- **The `.data` Pseudo-Op** - The `.data` directive is a pseudo-op that assembles the given bytes (8-bit integers) into the next available locations in the data segment. As many as 8 bytes can be specified on the same line. The byte values may be specified as hex, binary, decimal or C character constants (as described above).

  An example use of the `.data` directive is shown in Figure 6, where `.data` is used to assemble a string in data memory.

3 How Do We Use ANT in CS1?

This year, we introduced 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.

Figure 3: The ANT Instruction Set

Op	Operands	Description
`add`	*dst, src1, src2*	Register *dst* gets *src1* + *src2*. `r1` gets any overflow/underflow from this addition.
`sub`	*dst, src1, src2*	Register *dst* gets *src1* - *src2*. `r1` gets any overflow/underflow from this subtraction.
`mul`	*dst, src1, src2*	Compute *src1* × *src2*, leaving the low-order byte in register *dst* and the high-order byte in register `r1`.
`div`	*dst, src1, src2*	Compute *src1* / *src2*, leaving the quotient in register *dst* and the remainder in register `r1`.
`beq`	*reg, src1, src2*	Branch to *reg* if *src1* == *src2*. `r1` is set to the address of the instruction following the `beq`.
`bgt`	*reg, src1, src2*	Branch to *reg* if *src1* > *src2*. `r1` is set to the address of the instruction following the `bgt`.
`ld`	*dst, src1, uconst4*	Load the byte at *src1* + *uconst4* into *dst*. `r1` is unchanged.
`st`	*reg, src1, uconst4*	Store the contents of register *reg* to *src1* + *uconst4*. `r1` is unchanged.
`lc`	*dst, const8*	Load the constant *const8* into *dst*. `r1` is unchanged.
`jmp`	*reg, uconst8*	Branch unconditionally to the specified constant. *reg* is ignored.
`inc`	*dst, const8*	Add *const8* to register *dst*.
`sys`	*reg, code*	Makes a system call. See Figure 4 for a list of the ANT system calls.

Figure 4: The ANT System Calls. Note that all syscalls set `r1` to 0 if successful, and set `r1` to non-zero values to indicate failure.

Service	Code	Description
`halt`	0	Halt the processor. *reg* is ignored.
`dump`	1	Dump core to file `ant.core`. *reg* is ignored.
`put_int`	2	Print the contents of *reg* as a decimal number.
`put_char`	3	Print the contents of *reg* as an ASCII character.
`put_string`	4	Print the zero-terminated ASCII string that starts at *reg*.
`get_int`	5	Read an integer into *reg*. *reg* must not be `r0` or `r1`. If EOF, `r1` is set to 1. *Does not check for illegal input.*
`get_char`	6	Read a character into *reg*. *reg* must not be `r0` or `r1`. If EOF, `r1` is set to 1.

Figure 5: Notations used in Figures 3 and 4.

dst	Must always be a register, but never `r0` or `r1`.
reg, src1, src2	Must always be a register.
const8	Must be an 8-bit constant (-128 .. 127): an integer (signed), char, or label.
uconst8	Must be an 8-bit constant (0 .. 255): an integer (unsigned) or label.
uconst4	Must be a 4-bit constant integer (0 .. 15).

Figure 6: The "Hello World" program, written in ANT assembly language.

```
# Note that ANT programs always begin execution at instruction 0.
# There is no "main".

        lc  r2, $hello_str      # load the address hello_str into r2
        sys r2, 4               # Print the string pointed to by r2.
        sys r0, 0               # Halt

# The data segment follows the _data_ label:
_data_:

hello_str:
        .data   'H', 'e', 'l', 'l', 'o', ' '
        .data   'W', 'o', 'r', 'l', 'd'
        .data   '\n', 0
```

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, and organizing the code in a readable manner, and it also encourages them to carefully document their code. 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 hard enough so that the benefit of thinking ahead is clear.

Figure 7: The reverse program.

```
# reverse.asm-- A program that reads lines of text from the user and prints out each line in
# reverse order.  If the line is longer than 80 characters, the extra chars are discarded.
# REGISTERS USED:
# r2 - a pointer to the beginning of the line buffer ($buffer).
# r3 - i, the counter.
# r4 - holds each char as it is read in and printed out.
# r5 - used as a scratch register for branches, or to compute the address of buffer [i].
# r6 - the constant '\n'.
# r7 - the constant 80, the line length limit.

                lc      r2, $buffer     # Initialize r2 to point to the buffer.
                lc      r6, '\n'        # Initialize r6 to '\n'.
                lc      r7, 80          # Initialize r7 to 80.

main_loop:      lc      r3, 0           # i = 0.

read_loop:      sys     r4, 6           # Read a character into r4.
                lc      r5, $exit
                bgt     r5, r1, r0      # Check for EOF; if we see it, exit.
                lc      r5, $end_read
                beq     r5, r4, r6      # if got a newline, exit read_loop.
                lc      r5, $read_loop
                beq     r5, r3, r7      # if already too many chars, ignore it,
                add     r5, r2, r3      # else compute the address of buffer[i],
                st      r4, r5, 0       # store character to buffer[i],
                inc     r3, 1           # and increment i.
                jmp     r0, $read_loop  # iterate.

end_read:       inc     r3, -1          # i--;

print_loop:     add     r5, r2, r3      # compute the address of buffer[i], and
                ld      r4, r5, 0       # load character at buffer[i] into r4,
                sys     r4, 3           # and print the character.
                inc     r3, -1          # i--
                lc      r5, $newline
                bgt     r5, r0, r3      # If 0 is greater than i, go to $newline,
                jmp     r0, $print_loop # else repeat the print_loop.

newline:        sys     r6, 3           # Print a newline.
                jmp     r0, $main_loop  # repeat the main_loop.

exit:           sys     r0, 0           # Halt.

_data_:
buffer:                                 # Everything past $buffer is available for the line.
```

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 provided the students with an automated test suite that they could use to test their ANT implementations for conformance to the assignment specification. This introduced them to the idea of rigorous testing, and that testing could be highly automated. We also encouraged students to write their own tests, to add to 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 insights into how translation from one language to another can be accomplished. The students were 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. We supply a tokenizer for the input and a function for handling constants in hexadecimal, octal, decimal, and ASCII because these tasks are very tedious, and the assignment was already very long. The solution typically required 100 lines of code for the symbol table (not counting reuse of code from libraries developed as part of earlier assignments) and approximately 500-700 lines of code for the rest of the assembler.

This is a substantial assignment for students at this level, and in retrospect it was probably too difficult. It did reinforce the value of design and modular code, however– students were urged to use a modular design, and those who did found the assignment much easier to implement and test than those who attempted an ad-hoc design.

This assignment also utilized an automated ANT assembler validation suite in a manner similar to that used in the ANT virtual machine assignment, which was provided to the students to help them test their assembler for conformance to the assignment specification, including proper handling of erroneous input.

With this assignment complete, students can 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 very effective in our CS1 course. It allows introductory-level students to develop very good intuitions about such concepts as pointers and compilation, while avoid-

ing 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]. The ANT virtual machine, assembler, debugger, documentation and sample ANT programs will be available in February 1998 as a binary or source code distribution. Please contact Dan Ellard (`ellard@eecs.harvard.edu`) for more information.

5 Acknowledgments

The ANT architecture would not exist without the encouragement of Brian Kernighan, Margo Seltzer, and the entire teaching staff of Harvard's CS50 for 1996 and 1997.

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