On the Design of a New CPU Architecture for Pedagogical Purposes

Daniel Ellard, David Holland, Nicholas Murphy, Margo Seltzer
{ellard,dholland,nmurphy,margo}@eecs.harvard.edu

Abstract

Ant-32 is a new processor architecture designed specifically to address the pedagogical needs of teaching many subjects, including assembly language programming, machine architecture, compilers, operating systems, and VLSI design. This paper discusses our motivation for creating Ant-32 and the philosophy we used to guide our design decisions and gives a high-level description of the resulting design.

1 Introduction

The Ant-32 architecture is a 32-bit RISC architecture designed specifically for pedagogical purposes. It is intended to be useful for teaching a broad variety of topics, including machine architecture, assembly language programming, compiler code generation, operating systems, and VLSI circuit design and implementation.

This paper gives our motivation for creating Ant-32, lists our design goals and how these goals influenced our design decisions, discusses some of the more important details of the resulting architecture, and describes our future plans for continuing development of the architecture and integrating it into existing curricula.

2 The Motivation for Ant-32

Before describing the process by which we created Ant-32, it is important to say *why* we felt it was useful to create Ant-32 at all. The courses at our university have frequently used several different architectures to illustrate different points, and often each course used a different architecture.

A negative result of using a multitude of architectures was that each course had to spend time and energy teaching the particular details of the architectures used by that course. This forced the professor to make an unpleasant choice between removing other material from the course, or adding to the workload of the course (which is already a problem at our institution, where Computer Science has an unfortunate reputation as one of the most arduous majors).

In order to minimize this problem in our introductory-level courses, several years ago we designed a simple eight-bit architecture named Ant-8, which is now used in both of our introductory programming courses as well as the introductory machine architecture course. This architecture has been successful and is now in use at several other institutions. Its utter simplicity and tiny size make it easy to learn, while providing a realistic illustration of a machine architecture, capable of running interesting applications.

Unfortunately, Ant-8 is too small and simple to be used for higher-level courses, such as compilers, operating systems, and advanced machine architecture. Therefore, we decided to create a 32-bit architecture, using the lessons we learned from our eight-bit processor, but with the goal of creating a single processor that can be used across a much wider range of courses.

We felt that it was worth the effort to create a new architecture, rather than using one of the myriad existing architectures, because we could not find any that were truly suitable. The "real" architectures (such as x86, alpha, and MIPS) are, in our opinion, too complicated and require mastery of too many arcane details in order to accomplish anything interesting. The many architectures created for purely pedagogical purposes offer more hope, but the systems of which we are aware are too finely tuned for illustrating or experimenting with a small number of concepts, and were never meant to be used as a general framework.

3 Goals and Requirements

The core philosophy of the Ant-32 architecture is that it must be clean, elegant, and easy to understand, while at the same time it must support all of the important functionality of a real processor. In short, it must maximize the number of concepts it can be used to teach, while minimizing the complexity and number of unrelated details the students must struggle through in order to absorb those concepts.

The functional requirements of the Ant-32 architecture can be described in terms of the different curricula that Ant-32 is designed to augment: simple assembly language programming, compiler code generation, operating system implementation, and VLSI design and implementation.

Addressing all of these different needs required a number of trade-offs and difficult design decisions, which are described in the remainder of this section.

3.1 Assembly Language and Machine Architecture

In an introductory assembly language programming unit, we believe that it is desirable to use an architecture that has a small number of instructions and simple memory and exception architectures. We also believe that it is important that the architecture be based on RISC design principles, because we believe that RISC principles will be the dominant influences on future processor designs. In addition, we have found that RISC architectures are generally easier for students to understand and implement.

In an earlier project, several members of the Ant-32 team were involved in the development of Ant-8, an eight-bit RISC architecture designed for introductory programming and introductory machine architecture courses. This architecture is extremely small, simple and easy to learn. We have had positive feedback from professors and students who have used it, both at our institution and elsewhere.

The first draft of Ant-32 was a direct extension of Ant-8 to thirty-two bits. It contained approximately twenty instructions, and was designed with the intention that all of our second-year students (who were familiar with the eight-bit architecture from their introductory classes) would feel familiar with the architecture and be able to read and write Ant-32 assembly language programs almost immediately. Like Ant-8, there was no support for virtual memory or any form of protection, and the exception architecture consisted of having the machine halt and dump core whenever any error occurs.

3.2 Code Generation

There are two aspects of the orignal Ant-32 design that made it unsatisfactory as the target of a code generator: the absence of relative jumps and branches and an overly simplified instruction set.

Our original Ant-8 architecture used absolute jumps and branches, because our students found absolute addressing more intuitive and easier to debug than relative addressing. However, automated code generators see the world in a different way than their human counterparts, and in many contexts relative addresses are easier to generate. The ability to use relative addresses also greatly simplifies separate compilation and linking (which has never been an issue for Ant-8, but which we expect will be important for Ant-32).

The original Ant-32 architecture also did not include any immediate arithmetic instructions. As a result, simple and commonplace operations such as incrementing the value in a register required at least two instructions. Adding a rich set of immediate arithmetic instructions make it possible to investigate a number of useful code optimizations.

In addition, we found it useful to extend the original Ant-32 programming model by adding basic register usage conventions, in order to provide a common framework for function calling and linkage conventions. These conventions are *not* part of the architectural specification, however, and there is nothing implicit in the architecture that limits how the processor is programmed. For example, there is no register dedicated to be the stack pointer in the Ant-32 architecture, although programmers can choose to adopt a register usage convention that creates that impression. Programmers are free to choose or experiment with different conventions.

3.3 Operating Systems

Operating systems courses require a more complex view of the processor, including an exception and virtual memory architecture, mechanisms to access memory and processor state, and an interface to an external bus to support devices separate from the CPU.

It was a challenge to add the functionality required to support a full-featured operating system without losing the ability to program Ant-32 *without* writing at least a bare-bones boot-strap OS. To achieve this goal, we designed the processor so that in its initial state, most of the higher-level functionality is disabled. This means that the programmer only needs to understand the parts of the architecture that they actually employ in their program.

3.4 Advanced VLSI Implementation

Considering the architecture from the perspective of an actual VLSI implementation was an extremely important influence on the design. It was often quite tempting to add powerful but unrealistic features to the architecture, in order to add "convenience" instructions, such as instructions to simplify the assembly language glue required for exception handlers, context switching, and related routines. Considering whether or not it would be realistic to actually implement these instructions in hardware was an essential sanity check to make sure that we were creating a plausible and realistic architecture.

3.5 Omitted Features

It is worth mentioning that there are a number of features present in many architectures that we felt comfortable omitting entirely from Ant-32, because we felt that they added unnecessary complexity. If necessary, the specification can be augmented to include these features. We have made an effort to make our design flexible, and in fact several features (such as support for floating point) were actually present in our design until late in the review process, when we decided to omit them.

- Ant-32 does not contain any floating point instructions: for our intended audience we believe that these instructions are rarely necessary, and they lengthen the specification of the architecture (and increase the complexity of implementing the architecture) to such an extent that we decided to drop them entirely.

- The Ant-32 architecture does not include a specification for an external bus; the only requirements are the ability to read and write memory external to the CPU. The bus can cause an interrupt to occur via a single IRQ channel.

  The separation of bus and processor architectures, as well as the simplicity of the interface to the bus, allows Ant-32 to integrate easily with many bus architectures. In our current implementation, we use a simple (but full-featured) bus architecture that was originally designed for use with the MIPS processor architecture, which allows us to use simulators for devices already written for this bus.

- The Ant-32 memory interface is extremely simple and does not include a specification of a cache. However, it does not preclude the presence of a cache, and is designed to allow the easy incorporation of nearly any caching architecture. In fact, our reference simulator for the architecture is designed to allow easy experimentation with different caching strategies.

- Ant-32 has a simple instruction execution model. Our main focus has been on the instruction-set architecture of Ant-32, and not on the actual implementation details. We have tried to avoid making any design decisions that would prevent the implementation of an Ant-32 processor with such contemporary features as pipelining, super-scalar execution, etc. The specification is written in such a way as to allow extension in this area. It is our belief that the Ant-32 instruction set architecture can be implemented in a number of interesting ways.

4 A Description of the Ant-32 Architecture

The core of our architecture is a straight-forward three-address RISC design, influenced heavily by the MIPS design philosophy and architecture. Since RISC architectures (and variants of MIPS) are ubiquitous, we will not describe the general characteristics of the architecture in detail, but will focus on where our architecture differs.

In a nutshell, Ant-32 is a 32-bit processor, supporting 32-bit words and addresses and 8-bit bytes. All instructions are one word wide and must be aligned on word boundaries. For all instructions, the high-order 8 bits of an instruction represent the opcode. There are a total of 62 instructions, including four optional instructions. There are 64 general-purpose registers. All register fields in the instructions are 8 bits wide, however, allowing for future expansion. Virtual memory is made possible via a TLB-based MMU, which is discussed in section 4.1. The processor has supervisor and user modes, and there are instructions and registers that can only be used when the processor is in supervisor mode.

The architecture also defines 8 special-purpose registers that are used for exception handling. These are described in section 4.2.

A somewhat unusual addition to the architecture is 8 cycle and event counters. These include a cumulative CPU cycle counter, a CPU cycle counter for supervisor mode only, and counters for TLB misses, IRQs, exceptions, memory loads and stores. We believe that these will be useful for instrumenting and measuring the performance of software written for the processor.

4.1 The Virtual Memory Architecture

The VM architecture was the focus of far more philosophical debate (and contention) than any other area of the architecture. Perhaps because of the energy and passion we put into airing our divergent views, and the fact that we eventually converged on a design that satisfied everyone, we feel that the resulting architecture is perhaps the most important contribution of the overall Ant-32 architecture.

The main focus of the debate was how much high-level support for virtual memory we should provide in hardware. In real applications, TLB operations (such as TLB miss exceptions, TLB invalidation during context switching, etc) are expensive and it is more than worthwhile to provide architectural support for them. For the purpose of pedagogy, however, providing this support makes the design and specification of the architecture considerably more complex. We feel that the architecture must be clear and elegant in order for the students to understand it well, and we are more concerned with how quickly students can implement their operating systems than how quickly their operating systems run. At the same time, however, we were still guided by the principle that our architecture must be realistic and full-featured.

Ant-32 is a paged architecture, with a fixed 4K page size. A software-managed translation look-aside buffer (TLB) maps virtual addresses to physical addresses. The TLB contains at least 16 entries, and may contain more. There are only three instructions that interact directly with the TLB: `tlbpi`, which probes the TLB to find whether a virtual address has a valid mapping, `tlble`, which loads a specific TLB entry into a register pair, and `tlbse`, which stores a register pair into a specific TLB entry.

In addition to the virtual to physical page mappings, each TLB entry contains information about the mapping, including access control (to limit access to any subset of read, write, and fetch), and whether the TLB entry is valid.

Ant-32 has a one gigabyte physical address space. Physical memory begins at address 0, but need not be contiguous. Memory-mapped devices are typically located at the highest physical addresses, and the last page is typically used for a bootstrap ROM, but the implementor is free to organize RAM, ROM, and devices in virtually any way they deem appropriate. The only constraint placed on the arrangement of memory is that the last word of the physical address space must exist; this location is used to store the address of the power-up or reset code.

Virtual addresses are 32 bits in length. The top two bits of a virtual address determine the segment that the address maps to. When the processor is in user mode, only segment 0 is accessible, but all the segments are accessible in supervisor mode. Addresses in segments 0 and 1 are mapped to physical addresses via the TLB, while addresses in segments 2 and 3 are mapped directly to physical addresses. Accesses to memory locations in segment 2 may be cached (if the implementation contains a cache) but accesses to memory locations in segment 3 may not be cached.

4.2 The Exception Architecture

A realistic but tractable exception architecture is essential to any processor used by an operating system course. Exception handlers, and particularly their entry/exit code, are among the most difficult parts of the operating system to code, test and debug. For most real 32-bit processors, searching the documentation to learn how to save and restore all the necessary aspects of the CPU state is a daunting task.

For Ant-32, our goal was to design an exception architecture that is realistic and complete, but also easy to understand and allows a simple implementation of the necessary glue routines for handling exceptions and saving and restoring processor state.

In Ant-32, interrupts and exceptions are enabled and disabled via special instructions. Interrupts from external devices are treated as a special kind of exception. Interrupts can be disabled independently of exceptions.

When exceptions are enabled, any exception causes the processor to enter supervisor mode, disable exceptions and interrupts, and jump to the exception handler. If an exception other than an interrupt occurs when exceptions are disabled, the processor resets. If an interrupt occurs while exceptions or interrupts are disabled, it is not delivered until interrupts and exceptions are enabled.

System calls are made via the `trap` instruction, which triggers an exception. The transition from supervisor mode to user mode is accomplished via the `rfe` instruction.

The Ant-32 exception-handling mechanism consists of eight special registers. These registers are part of the normal register set (and therefore can be addressed by any ordinary instruction), but they can only be accessed when the processor is in supervisor mode. Four of the registers are scratch registers, with no predefined semantics. They are intended to be used as temporary storage by the exception handler. The other four registers contain information about the state the processor was in when the exception occurred. These four registers are read-only, and their values are only updated when exceptions are enabled. When an exception occurs, further exceptions are immediately disabled, and these registers contain all the information necessary to determine the cause of the exception, and if appropriate reconstruct the state of the processor before the exception occurred and restart the instruction:

$\boxed{\textbf{e0}}$ When exceptions are enabled, this register is updated every cycle with the address of the currently executing instruction.

When an exception occurs, `e0` contains the address of the instruction that was being executed. Depending on the exception, after the exception handler is finished, this instruction may be re-executed.

$\boxed{\textbf{e1}}$ When exceptions are enabled, this register is updated every cycle to indicate whether interrupts are enabled.

When an exception occurs, interrupts are disabled, but `e1` tells whether or not interrupts were enabled before the exception occurred. This allows the exception handler to easily restore this part of the CPU state.

$\boxed{\textbf{e2}}$ When exceptions are enabled, this register is updated with every address sent to the memory system. If any memory exception occurs, this register will contain the memory address that caused the problem.

$\boxed{\textbf{e3}}$ This register contains the exception number and whether the processor was in user or supervisor mode when the exception occurred. For exceptions due to memory accesses, the value of this register also indicates whether the exception was caused by a read, write, or instruction fetch.

Disabling interrupts automatically whenever any exception occurs provides a way to prevent nested exceptions and an unrecoverable loss of data: if an interrupt is permitted to occur before the state of the processor has been preserved, then the state of the processor when the first exception occurred may be lost forever. By disabling interrupts until they are explicitly re-enabled, we can prevent this from happening.

The benefit of this arrangement is that the only way to fatally crash the processor is to have a mistake which causes an exception to occur in the exception entry/exit code. The drawback of this scheme is that the exception handler entry/exit code (and all the memory addresses referenced by this code) must generally be located in an unmapped memory segment, because otherwise a TLB miss could occur during execution of the exception handler.

5 Future Directions

Although completing the specification of our architecture was an important step towards our goal of making Ant-32 a widely valuable educational tool, we acknowledge that there is much more to do. From our experiences with Ant-8, we know that educators will not use Ant-32 in their curricula unless the benefits of using Ant-32 are obvious, and the cost of transition to Ant-32 is very low.

To minimize the transition costs, we have already implemented a reference assembler, simulator, and debugger for the Ant-32 architecture, an assembly-language tutorial and hardware specification. This software and documentation has already been used, with positive results, by a compiler course at Boston College. We are currently working on extending this material into full suite of educational materials for the Ant-32 architecture, including extended tutorial and reference texts, example code, lecture materials, problem sets and exercises with detailed solutions, and pre-compiled distributions for easy installation on popular platforms, in the same manner as we have done with our earlier eight-bit architecture. All of this material will be freely available from our web site, `http://www.ant.harvard.edu/`.

We are also planning a project to build a complete GNU tool-chain (`gcc`, `gas`, `gdb`, and complete libraries) for Ant-32 so that it can be used to write a complete operating system for Ant-32 with only a small amount of assembly language programming. This is a huge undertaking, and we invite anyone interested in helping to develop this material in any way to contact the Ant-32 team.

6 Related Work

Many simplified or artificial architectures have been created for the purposes of pedagogy or separating conceptual points from the details of implementation, beginning at the foundation of computer science with the the Turing machine [6] and continuing to the present day. Attempting to survey this field in the related work section of a five page paper is futile; in the last ten years SIGCSE has published at least 25 papers directly related to this topic, and we suspect that for every architecture documented in the literature there are at least a dozen toy architectures that are never publicized outside of the course they were created for.

The continued and vigorous activity in the development of simplified architectures, simulators for existing architectures, or extended metaphors for computation such as *Karel the Robot* [5] or the *Little Man* [7] computer simulators strengthens our belief that these are powerful pedagogical tools, and that they are worth further development.

All of the pedagogical systems of which we are aware focus on a single conceptual domain, instead of trying to work well across a spectrum of topics. One standout has been the MIPS architecture, which has served as a useful tool in the domains of both operating systems and machine architecture pedagogy. This is demonstrated by the number of educational projects based on MIPS, such as SPIM [3], MPS [2], Nachos [1], and descendants of MIPS such as DLX [4]. Once again, however, the sheer number and diversity of tools based on this architecture seems to imply that the situation could be improved. With Ant-32, we plan to combine the educational features of most of these tools into a single, coherent framework that can easily be adapted to a broad range of educational purposes.

7 Conclusions

We believe that Ant-32 will allow educators to streamline their courses by using the same architecture (and tools) in several courses, because Ant-

32 is well-suited to many different different educational purposes.

We recognize that educators will disagree in whole or in part with some of our assumptions, opinions, and conclusions, but when this happens, we hope that sharing our experiences in designing a 32-bit architecture for pedagogical purposes will be helpful to them as they develop or refine their own designs.

References

[1] W. A. Christopher, S. J. Procter, and T. E. Anderson. The nachos instructional operating system. *Proceedings of the USENIX Winter 1993 Conference*, 1993.

[2] M. Morsiani and R. Davoli. Learning operating systems structure and implementation through the mps computer system simulation. *Proceedings of SIGCSE 1999*, 31(1), 1999.

[3] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, 1994.

[4] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach, 2nd edition*. Morgan Kaufmann Publishers, 1996.

[5] R. Pattis. *Karel the Robot*. John Wiley and Sons, Inc, 1981, 1995.

[6] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[7] W. Yurcik and L. Brumbaugh. A web-based little man computer simulator. *Proceedings of SIGCSE 2001*, 33(1), 2001.