

WCAE 2002

Workshop on Computer
Architecture Education

Anchorage, Alaska

May 26, 2002

Workshop Proceedings

Welcome Message

Welcome to the Workshop on Computer Architecture Education! By my unofficial count, this is the tenth such workshop since the series began at HPCA-1 in January 1995. I am pleased to announce that this WCAE received the second largest number of submissions ever, and the overall quality of the papers appears to meet or exceed that of all past workshops. This year's keynote address will be given by Fayé A. Briggs, who has a long and distinguished career as a researcher and textbook writer in academia, and now as director of chipset architecture for the world's largest manufacturer of processor chips.

A new feature of the workshop is discussion periods in every session, giving participants a chance to explore important aspects of teaching with the presenters. I hope that this will give you ideas you can take back to use in your own teaching, and then report on them at future WCAEs. I look forward to excellent presentations and excellent interactions with all of you.

Edward F. Gehringer, Workshop Organizer
Dept. of Electrical & Computer Engineering
Dept. of Computer Science
North Carolina State University
<http://www4.ncsu.edu/~efg>
efg@ncsu.edu

Table of Contents

Session 1. Welcome and Keynote 8:30–9:20

8:30	<i>Welcome</i> Edward F. Gehringer, workshop organizer	
8:35	<i>Keynote address:</i> “Introducing new variables and constraints into computer architecture education,” Fayé A. Briggs, Intel Corporation	3

Session 2. Teaching New Perspectives on Computer Architecture 9:20–10:00

9:20	“Teaching processor architecture with a VLSI perspective,” Mircea R. Stan and Kevin Skadron, University of Virginia	4
9:35	“Teaching students computer architecture for new nanotechnologies,” Michael T. Niemier and Peter M. Kogge, University of Notre Dame.....	10
9:50	Discussion	

Break 10:00–10:30

Session 3. Teaching with Custom Computer Architectures 10:30–11:15

10:30	“Using custom hardware and simulation to support computer systems teaching,” Murray Pearson, Dean Armstrong, and Tony McGregor, University of Waikato	19
10:45	“On the design of a new CPU architecture for pedagogical purposes,” Daniel Ellard, David Holland, Nicholas Murphy, and Margo Seltzer, Harvard University	27
11:05	Discussion	

Session 4. Active Learning 11:15–12:00

11:15	“Questions to enhance active learning in computer architecture,” Mark Fienup and J. Philip East, University of Northern Iowa	34
11:25	“An active learning environment for intermediate computer architecture courses,” Jayantha Herath, Sarnath Ramnath, Ajantha Herath, and Susantha Herath, St. Cloud State University	41
11:35	Discussion	

Lunch (on your own) 12:00–1:30

Session 5. Simulators and Tools 1:30–3:30

1:30	“Effective support of simulation in computer architecture instruction,” Christopher T. Weaver, Eric Larson, Todd Austin, University of Michigan	48
1:50	“Web-based training on computer architecture: The case for JCacheSim,” Irina Branovic, University of Siena, and Roberto Giorgi and Antonio Prete, University of Pisa.....	56
2:10	“Digital LC-2: From bits & gates to a Little Computer,” Albert Cohen, INRIA, and Olivier Temam, Université Paris-Sud	61
2:30	“MipsIt—A simulation and development environment using animation for computer architecture education,” Mats Brorsson, KTH, Royal Institute of Technology	65
2:50	“CoDeNios: A function-level co-design tool,” Yann Thoma and Eduardo Sanchez, Swiss Federal Institute of Technology, Lausanne	73
3:10	Discussion	

Break and Poster Session 3:30–4:15

	“How computers really work: A children's guide,” Shirley Crossley and Hugh Osborne, University of Huddersfield, and William Yurcik, Illinois State University	79
	“Update Plans: pointers in teaching computer architecture,” Hugh Osborne and Jiří Mencák, University of Huddersfield	84
	“CASTLE: Computer Architecture Self-Testing and Learning System,” Aleksandar Milenkovic, University of Alabama in Huntsville, Bosko Nikolic and Jovan Djordjevic, University of Belgrade	89
	“Development of a digital instrument as a motivational component in teaching embedded computers,” Gracián Triviño and Felipe Fernández, Universidad Politécnica	93
	“ILP in the undergraduate curriculum,” Daniel Tabak, George Mason University	98

Session 6. Resources for Architecture Courses 4:15–6:00

4:15	“PECTOPAH: Promoting Education in Computer Technology Using an Open-Ended Pedagogically Adaptable Hierarchy,” Hugh Osborne, Shirley Crossley and Jiří Mencák, University of Huddersfield, William Yurcik, Illinois State University	102
4:35	“Read, use, simulate, experiment and build: An integrated approach for teaching computer architecture,” Ioannis Papaefstathiou and Christos Sotiriou, University of Crete	105
4:55	“An integrated laboratory for computer architecture and networking,” Takamichi Tateoka, Mitsugu Suzuki, Kenji Kono, Youichi Maeda and Kôki Abe, University of Electro-Communications	110
5:10	“A lab course of computer organization,” J. Real, J. Sahuquillo, A. Pont, and A. Robles, Technical University of Valencia	118
5:30	“A survey of Web resources for teaching computer architecture,” William Yurcik, Illinois State University and Edward F. Gehringer, North Carolina State University	125
5:45	Discussion	

Visit the workshop on the Web, <http://www4.ncsu.edu/~efg/wcae2002.html>. PDF of all the proceedings, color screenshots, and more!

Introducing New Variables and Constraints into Computer Architecture Education

Keynote Address

Fayé A. Briggs

Director of Chipset Architecture, Intel Corporation

Abstract: Computer architecture education has evolved significantly over the last 30 years, especially in academia. Businesses have often sought to provide their internal education on various aspects of computer architecture. The goal of this talk is to provide an overview of many other variables and constraints that could further enrich the education of computer architecture. The intent is to suggest some new aspects of computer architecture education curriculum that will enrich the development of architecture & associated evaluation criteria

Teaching Processor Architecture with a VLSI Perspective

Mircea R. Stan
ECE Department
University of Virginia
Charlottesville, VA 22904
mircea@virginia.edu

Kevin Skadron
CS Department
University of Virginia
Charlottesville, VA 22904
skadron@cs.virginia.edu

Abstract—This paper proposes a new approach to teaching computer architecture by placing an explicit emphasis on circuit and VLSI aspects. This approach has the potential to enhance the teaching of both architecture and VLSI classes, to improve collaboration between CS and ECE departments and to lead to a better understanding of the current difficulties faced by microprocessor designers in industry.

Keywords: computer architecture, microprocessor design, VLSI design

I. INTRODUCTION

The teaching of computer architecture typically focuses on the interaction of instruction set architecture (ISA), instructions per clock cycle (IPC), and processor clock rate. Yet the circuit-design exigencies that profoundly impact the implementation of architecture-level concepts often receive little consideration. For example, the popular Hennessy and Patterson textbooks [1], [2] and others, despite their many strengths, have very limited information about logic and circuit issues. On the other hand, the VLSI and digital integrated circuit textbooks [3], [4] rarely consider the implications of their methods for microprocessor design at the architecture level. This division is often perpetuated by traditional academic boundaries. **In this paper we make the case that a new course is needed that crosses these boundaries and teaches computer architecture with an explicit VLSI perspective and vice-versa.**

A. Why teaching computer architecture with a VLSI perspective

Teaching computer architecture, as any other discipline, is different from school to school, but there have been attempts to unify it, either in an informal, grassroots way, e.g., by the increased popularity of some textbooks that are widely adopted and dominate the field; or in a formal way by the different accreditation mechanisms, e.g., ABET,

* This work was supported in part by NSF CAREER grant CCR-0133634, NSF CAREER grant MIP-9703440, and by a research grant from Intel MRL.

CSAB, and the creation and publication by IEEE/ACM of generic curricula for Computer Science and Engineering degrees.¹ In such a proposed curriculum, the main computer architecture concepts are covered in a “core” class, CS 220 - Computer Architecture, with more detailed microarchitecture and circuit issues being left to the non-core, “advanced” classes, *CS 320 - Advanced Computer Architecture* and *CS 323 - VLSI development*. We agree that not all students can, or should, learn all the details normally presented in these three classes, but we also think that it is important to teach the microarchitecture and VLSI aspects *together* for those students that elect to learn the advanced concepts and prepare for careers as microprocessor architects or circuit designers. In brief, we propose the creation of a combined class, *CS 320/323 - Advanced Computer Architecture: a VLSI Perspective*, see figure 1.

Such a class would be useful from many points of view. First, it breaks the artificial boundary between microarchitects and circuit designers. Both in industry and in academia, such differences clearly exist but are mostly detrimental. When architects do not have a good understanding of VLSI/circuit issues, they may take unwise decisions that penalize overall cost and performance; when circuit designers don’t understand the overall architecture, they cannot fully take advantage of the degrees of freedom in design or exploit synergistic design choices across multiple levels of abstraction. A course like *CS 320/323 - Advanced Computer Architecture: a VLSI Perspective* would prepare students with a complex view of both architecture and circuit aspects.

Second, the class would also help as a bridge for academic programs in Computer Science, Computer Engineering and Electrical Engineering. A quick search of different existing classes and programs at different universities reveals that Computer Architecture classes are many times taught in both CS and ECE departments, with more of them on the CS side, while VLSI classes are mostly taught in ECE and EE departments, with few CS departments offering them. This is exactly the case at the Univer-

¹<http://www.computer.org/education/cc2001>

sity of Virginia, where there are two classes in Computer Architecture, one in the CS, the other in the ECE department, but only one VLSI class, in the ECE department. A course like *CS 320/323 - Advanced Computer Architecture: a VLSI Perspective* would be equally attractive to both CS and ECE students and departments.

The third and final point is that such a class would bring new ideas and excitement into teaching both Computer Architecture and VLSI. While in industry the emphasis on circuit design aspects is clearly required for the high-performance microprocessors of today and tomorrow (as supported by the many publications at ISSCC and in JSSC), this trend is not yet fully reflected in the computer architecture classes being offered in academia. The situation with the VLSI classes is even more serious as very little progress has been made in the teaching VLSI since the seminal textbook by Mead and Conway. Even the newest VLSI textbooks still use the same bottom-up approach of first presenting device physics, followed by simple logic circuit design, combinational and sequential, followed by layout and finally a few case studies [3], [4]. Such an approach, quite successful in the past, has become slightly dated as it clearly targets “hard-core” Electrical and Computer Engineering students and is not interesting to most Computer Science students. Even the VLSI textbooks focusing on ASIC design are not appropriate for microprocessor designers, who need a balanced approach that combines both custom and semicustom design methods. A course like *CS 320/323 - Advanced Computer Architecture: a VLSI Perspective* would make both Computer Architecture, and especially VLSI design, more attractive to a wider spectrum of students and give them greater breadth of training.

II. COMPUTER ARCHITECTURE WITH A VLSI PERSPECTIVE: A BIRD’S EYE VIEW

The goal of the class is to give equal weight to both computer microarchitecture and circuit design aspects. In order to do this effectively the topics will be presented in parallel, with architecture concepts being used to provide a “natural” way to introduce VLSI and circuit design concepts. Accommodating both architecture and VLSI will necessarily entail sacrificing some material from traditional advanced-architecture and VLSI syllabi. Our philosophy is that with a sound training in fundamentals, the details are easily learned independently. For example, once the fundamentals of branch prediction and caching are understood, students can as needed teach themselves the various advanced branch-prediction and caching schemes, as well as variations like value prediction and prefetching. As another example, once the fundamentals of [MIRCEA:

VLSI].

To minimize the need for pre-requisites, the class will assume only a sophomore-level assembly-language and introductory computer-organization course as pre-requisite. CS 320/323 will start with a quick overview of Architecture (“Computer Architecture 101”) and VLSI (“VLSI Design 101”) to introduce the main ideas.

A. Overview: Processor Architecture

The overview of processor architecture topics will start with a classic, single-issue (scalar) processor. We plan to use a modern embedded processor example, like the Digital StrongArm, or its successor, the Intel XScale. This will include the basic operations of instruction fetch, instruction decode, register file access, integer and floating-point execution, and result writeback. In a generic fashion, we will also introduce the notions of pipelining and pipeline control, result forwarding, instruction and data caches, control and data hazards, exceptions, etc. The quantitative evaluation of performance through benchmarking and simulation will also be introduced here.

B. Overview: VLSI design

The overview of basic VLSI design concepts will start with a brief introduction of active device behavior and circuits, first at the switch level and only later with more detailed analysis and circuit-level modeling and simulation. Next we will touch on combinational vs. sequential logic and circuits, static vs. dynamic circuit concepts, and basic ideas of possible design flows, including custom, semicustom, and fully automated. We also briefly touch on the idea of a layout, and the corresponding CAD steps of floorplanning, placement and routing.

Following this quick introduction the course will get into more detailed discussion of each processor architecture topic and its “associated” VLSI circuit concept. There will be a clear attempt to present both architecture and circuit issues in a logical manner, in general by following the typical order of a processor pipeline for the architecture concepts, and associating the most important and natural circuit issue to the architecture, such that there are few or no repetitions and all the important aspects are covered.

III. INSTRUCTION FETCH AND DECODE: COMBINATIONAL LOGIC DESIGN

The classical processor pipeline starts with instruction fetch and decode, so it is natural to start our detailed treatment here as well. Since caches are used both for instruction and data, and since memory structures are not naturally best suited as a first introduction to circuit concepts,

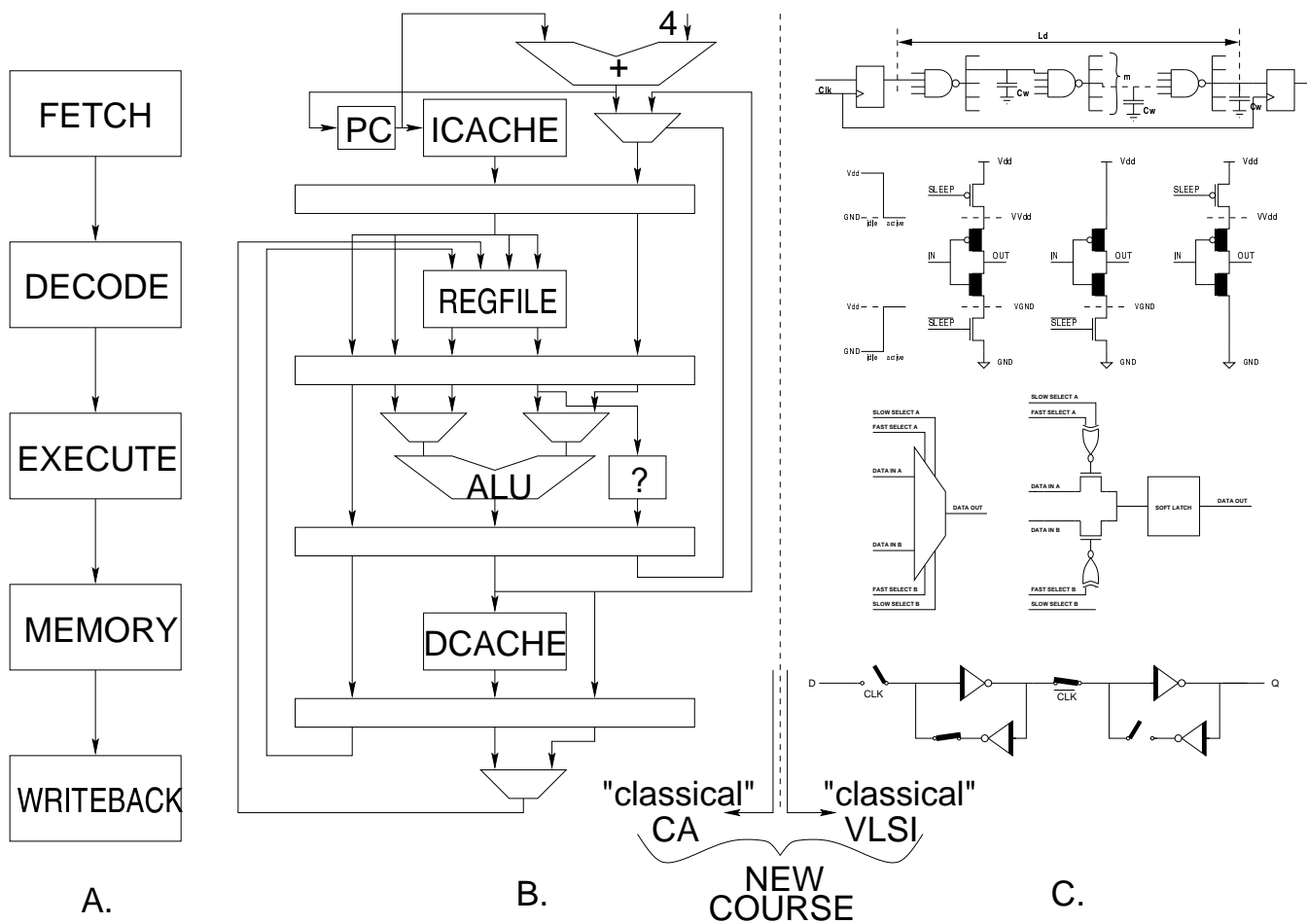


Fig. 1. New class on Computer Architecture with a VLSI perspective will combine elements from “classical” Computer Architecture classes and from “classical” VLSI design classes: A. typical 5-stage processor pipeline, B. typical simplified microarchitecture, C. typical VLSI concepts at the logic and circuit levels.

we postpone the actual discussion of VLSI concepts for memories to a later section.

A. Architecture: Instruction Fetch and Decode

Here we start with a quick discussion of instruction formats, the CISC vs. RISC debate, and how decoding a RISC instruction set is “easy” compared to a CISC. We will use MIPS as an example RISC architecture (we would have used Alpha, but now it is a “defunct” processor line) and x86 as an example of CISC, thus covering both extremes.

B. VLSI: Basic Combinational Logic

We can use decoding as a typical example of combinational logic circuits, and use it to illustrate the most important circuit design concepts. We start with simple static circuit techniques, including complementary static CMOS, pass-transistor and pass-gate logic, and show how they apply to simple logic gates with muxes and decoders

as a typical example. We then explain the advantages of complementary static CMOS (general applicability, robustness, regeneration of logic levels) as well as its disadvantages (size, suboptimal performance). We then show that other particular logic styles can outperform complementary CMOS in special cases, and exemplify with pass-transistor logic and pseudo-NMOS for muxes and decoders. We postpone the issue of dynamic combinational circuit design to a future section.

C. VLSI: Layout

Here we introduce layout techniques and the main figures of merit for VLSI circuits: performance (propagation delay), area (cost), power dissipation, reliability, robustness to noise, etc. We also introduce the notion of digital design as a trade-off among the possible figures of merit. We show simple bottom-up “polygon pushing” design steps.

IV. PIPELINING: SEQUENTIAL LOGIC DESIGN

One of the most effective ways to increase processor performance is to use pipelining of the various operations. This provides the perfect motivation for looking at the circuit design of sequential circuits.

A. Architecture: Pipelining

We first present the “classical” 4-stage and 5-stage pipelines, demonstrate the increased throughput that pipelining achieves, and explore the tradeoffs between latency and throughput for a pipelined processor. We follow up with more advanced concepts like superpipelining, and show the trade-offs due to an increase in the work per pipeline stage vs. overhead due to latch overhead.

B. VLSI: Floorplanning

The simple existence of a pipeline gives a level of regularity to the design that can be used for top-down floorplanning. Here we explain the importance of block adjacencies for reduced area (less routing) and increased performance (shorter wires).

C. VLSI: Synchronous Sequential Circuits

A pipeline is based on the overlap (in time) of the different functions; this overlap can be achieved with either synchronous or with asynchronous methods. Virtually all current processors are synchronous, so we start by explaining simple synchronous design concepts such as setup and hold times and propagation delay, edge-triggered flip-flop vs. transparent latch vs. pulsed register, etc. We present simple static CMOS implementations of such flip-flops, registers and latches, then introduce dynamic logic, followed by dynamic versions of these state elements for higher performance but also higher power and less noise immunity. We exemplify with a few of the most important types of flip-flops used in several microprocessors, including TSPC, the Earle latch, etc.

D. VLSI: Clocking

The issues of clock generation, clock distribution and their influence on clock skew are explained. We explain the trade-offs for clock-spines, clock-planes, H-tree and X-tree clock distribution schemes, as well as the notions of centralized and distributed clocking schemes. Here we also discuss the issue of optimally driving large loads through the placement and sizing of buffers.

E. VLSI: Low-Power Design

We explain the differences between dynamic and static power, power consumption and power dissipation, etc.

More advanced concepts like time-borrowing and dynamic voltage/frequency scaling are also presented here, as well as clock-gating and other low-power methods. We also introduce the energy-delay product.

F. VLSI: Asynchronous Design

In order to provide a balanced view, we also present asynchronous design concepts such as micro-pipelines, wave pipelining, and “hybrid” methods such as globally-asynchronous, locally-synchronous approaches. We also give the (few) examples where such methods have actually made it into real commercial microprocessors (e.g., wave-pipelining for address decoders).

V. EXECUTION UNITS: DATAPATH STRUCTURES

After instruction fetch and decode, the next step in a simple, scalar, processor is register read and execution. We postpone discussing register file issues to the next section and discuss execution units here.

A. Architecture: Integer Execution

Here we discuss briefly different issues related to integer datapaths, especially microarchitecture and logic level computer arithmetic algorithms, including addition, subtraction, multiplication, division and transcendental operations. Two’s complement notation is introduced as part of this topic. We also briefly explain MMX and other signal-processing enhancement techniques for general-purpose processors.

B. Architecture: Floating-Point Execution

We follow the integer datapath issues with the more complex issues related to FP arithmetic, including data formats like IEEE.

C. VLSI: Datapath and Computer Arithmetic

Here we explore in more depth the differences between static and dynamic combinational logic circuits, with the higher performance of dynamic logic being widely used for datapath circuits. We then present different adder circuit styles (e.g. Kogge-Stone), multiplier circuit styles, shifter styles, etc.

D. VLSI: Placement

The VLSI structures presented in previous sections were more or less “random” logic. For datapath circuits there is an obvious one-dimensional regularity (the number of “bits”) that can, and should, be exploited as bit-sliced design. Bit-slices are an example of regular placement of logic along one dimension. Here we also discuss

about custom and semicustom design methodologies and give examples of custom datapath design and semicustom standard-cell-based random logic.

VI. CACHES AND REGISTER FILES: MEMORY DESIGN

Finally we present caches and data-array structures. Caches are used for instructions and data, while data arrays are used for register files, queues, etc.

A. Architecture: Caches

We start by presenting issues related to cache associativity, first the two extremes, direct-mapped cache and fully-associative cache, followed by “in-between” cases like set-associative cache and CAM-RAM structures. We consider the issues of write-through vs. write-back, fills and write buffers. TLBs and generic buffers are other types of memory structures that are presented here. As advanced topics we present non-blocking caches and multi-level cache hierarchies.

B. Architecture: Register Files

For register files we start by presenting architectural registers and their implementation. We consider multi-porting as well as split-phase register access.

C. VLSI: Memories and Data Arrays

In order to implement memories and data arrays we present the main circuit building blocks. We start with the row and column decoders, followed by memory-cell design. Static/6T vs. dynamic/1T or 4T as well as wordlines and bitlines, precharging, for read and write are then discussed. Sense amp design and issues related to leakage and threshold wrap-up the design aspects. We follow by a brief discussion of defects, yield, and redundancy methods (spare rows and columns with reconfiguration) for increasing yield for memory structures.

D. VLSI: Routing

Physical design issues for memories are extremely important, in particular the issue of pitch-matching for the various subsections. This is an example of self-routing by abutment which shows the importance of regularity for VLSI design. General routing for “random” logic is a much more difficult problem.

VII. PIPELINE CONTROL: STATE MACHINES

A. Architecture: Pipeline Control

We first show how forwarding works and how the PC gets updated. We then introduce branch prediction and show how instructions get “squashed”. As an advanced

topic we introduce multiple (in-order) issue-superscalar and the associated scoreboarding and contrast this with VLIW techniques.

B. VLSI: State Machines

Here we discuss difficulties of longer pipelines in terms of forwarding complexity and misprediction penalty. We introduce PLAs as an alternative for combinational logic implementation.

VIII. VLSI: INTERCONNECT, BUSSES AND I/O

We present major difficulties related to long interconnect, RC and RLC delay issues, and revisit buffer-insertion to reduce quadratic delay. We also show I/O design and system-interconnect issues, including the need for multi-voltage design.

IX. WHEN THINGS GO WRONG: EXCEPTIONS, VERIFICATION, TESTING

A. Architecture: Exceptions

An essential part of architecture is exception handling. We discuss precise vs. imprecise exceptions, explore the challenges of exception handling from the ISA level, and then proceed to describe the requisite hardware structures. We first present interrupt/trap hardware, supervisor mode, exceptions and trap vectors. We then trace the sequence of steps for syscall trap, I/O interrupt. For dealing with exceptions while already handling an exception we explain the need for interrupt masks, processor status word, etc. As an advanced topic we present the BIOS and describe the process of bootstrapping the computer.

B. VLSI: Verification, Testing, and Packaging

We explain the issues related to verification and validation (making sure that the design is correct) as well as to testing and built-in self test (BIST—making sure that a correct design is correctly fabricated). The notions of defects, faults and errors is explored in more detail. A brief overview of manufacturing, packaging, binning is also presented here.

C. VLSI: Power distribution

With reduced voltages and increasing power, the currents that need to be distributed on chip are increasing at an alarming rate. Here we discuss issues related to IR-drop, electromigration, and their influence on performance and reliability. We briefly mention aluminum and copper interconnect and SOI.

X. OUT-OF-ORDER EXECUTION: VLSI METHODOLOGY

A. Architecture: *Out-of-Order Execution, Register Renaming*

Here we explain the benefits of out-of-order execution (OOE) and the need for renaming. We briefly describe basic OOE structures (register update unit vs. issue queues, etc.) as well as wakeup and select logic and renaming logic.

B. VLSI: *Queues and VLSI Methodology*

The issue queue has become one of the most complex structures in a modern out-of-order superscalar microprocessor. We choose the issue queue to do an in-depth analysis and exemplify with multiple case studies of real designs. We use this as a motivation for a look at different design methodology alternatives with their advantages and disadvantages.

XI. CONCLUSION

We have made the case for a class that teaches processor architecture with a VLSI perspective. We believe that such a class would have a strong impact in academia and will also better prepare students for jobs as either architects or circuit designers. We expect the class to be quite popular with a wide spectrum of students in CS and ECE departments. Since no current textbook uses this approach we also believe that there are significant opportunities for filling this void with a “new and improved” textbook that could be used, either for teaching Computer Architecture with a VLSI perspective, or, alternatively, for teaching VLSI for Computer Science students.

REFERENCES

- [1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition*, Morgan Kaufmann Publishers, 1995, ISBN 1-55860-329-8.
- [2] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann, San Mateo, 1993.
- [3] Jan M. Rabaey and Massoud Pedram, Eds., *Low Power Design Methodologies*, Kluwer Academic Publishers, Boston, MA, 1996.
- [4] Neil Weste and Kamran Eshraghian, Eds., *Principles of CMOS VLSI Design*, Addison Wesley, Reading, MA, 1993.

Teaching Students Computer Architecture for New, Nanotechnologies

Michael Thaddeus Niemier
University of Notre Dame
Dept. of Comp. Sci. and Eng.
Notre Dame, IN 46545
mniemier@nd.edu

Peter M. Kogge
University of Notre Dame
Dept. of Comp. Sci. and Eng.
Notre Dame, IN 46545
kogge@wizard.cse.nd.edu

Abstract:

Given the potential limitations facing CMOS, there has been an influx of work and research in various nano-scale devices. Most of the work related to nanotechnology has been done strictly with devices, with little attention given to circuits or architectures of them – the desired end result. In the past, these studies have usually lagged device development by many years. However, we propose a curriculum to help integrate the communities – device physicists and computer architects – earlier. One goal of such a curriculum would be to teach students how to generate a “Mead/Conway” methodology for a given nanotechnology. This would teach students not only how to help technology change and evolve, but eventually teach students how to adapt to changes after a technology evolution. Another goal would be to facilitate more (and earlier) interaction between device physicists and computer architects to prevent these two groups from developing diverging views of what is physically and computationally possible in a system of nano-scale devices.

1. Introduction:

Consider the following “quote” from the preface of a future book on nano-scale design:

“Until recently the design of integrated circuitry for nano-scale devices has been the province of circuit and logic designers working within nanotechnology firm research laboratories and select “pockets” of academia. Computer architects have traditionally composed systems from standard self-assembled nano-circuits designed and manufactured by these entities but have seldom participated in the specification and design of these circuits. Nano-engineering and Computer Science (NE/CS) curricula reflect this tradition with courses in nano-scale device physics and integrated circuit design (if any at all) aimed at a different group of students than those interested in digital system architecture and computer science. This text is written to fill a current gap in the literature and to introduce all NE/CS students to integrated system architecture and design for emerging nano-technologies. Combined with individual study in related research areas and participation in large system design projects, this text

provides the basis for a course-sequence in integrated nano-systems.” (Mead/Conway v)

With the potential physical and economic limitations facing CMOS, there has been a recent proliferation in research related to nano-scale devices – particularly those targeted toward computational systems. Much of this early work has been relegated to the development of the physical devices themselves; and while circuits and systems have probably been envisioned within each specific nanotechnology being considered, their development has usually not progressed beyond the conceptual stage. Furthermore, historically, computer architects have been disjoint from the process of actual circuit designs, and in the case of CMOS, comprehensive and integrated architectural and circuit design methodologies were not published until the late 1970s when Carver Mead and Lynn Conway's groundbreaking work appeared [1].

Interestingly, the above paragraph of *this* work is essentially verbatim from the preface of Mead and Conway's VLSI text. While written almost 25 years ago, it illustrates a problem that they faced – computer architects, who might be the “lowest common denominator” in designing a system to perform useful and efficient computation, did not take part in the development of the devices and basic circuits with which they were required to design. We are beginning to face this same problem now with regard to nano-scale devices, and this paper will propose the beginnings of a curriculum to help alleviate it.

At a recent NSF sponsored workshop on molecular scale devices and architectures [2], Lynn Conway reiterated that during the early years of CMOS development, while architects would sometimes work with MOS technologists, as a “group”, most individuals did not span the whole range of knowledge required to design a complete computer system. Likewise, the scope required to do complex designs is large and it is not completely feasible for a device physicist to understand all of the issues a computer architect must consider. In the pre-Mead/Conway era, the development flow was for system architects to express a design at a high level, such as Boolean equations, and then turn it over to logic designers who converted the designs into “netlists” of basic circuits. Fab experts would then lay out implementations of the individual

logic blocks, and “just wire them together.” Interaction between the architects and fab experts was limited. In terms of technology, MOS FETS were considered “slow and sloppy,” and real design was in sophisticated bipolar devices.

The invention of the self-aligning FET gate allowed Mead and Conway to bridge this gap by changing the focus of fab from considering chips “in cross section” to an “overhead view” where it is the interconnect that is most visible. They did this by developing a set of design rules and abstractions that a computer architect could use to involve himself or herself in the circuit design process. They reduced the physics-dependent device descriptions to a scale-independent set of parameters based largely on area and shape, with some simple rules for first order modeling of how such devices would interact in combination with each other. They also introduced some simple but useful circuit primitives that changed the discussion from isolated logic gate performance to interconnect. This allows architects, who are experts in hierarchical designs, to extend their hierarchies one level down – to potentially new basic structures, and then take advantage of these structures in implementing larger and larger systems. The introduction and use of clever circuits using pass transistors is just one example of such an insight.

When coupled with the ability to cheaply fabricate real chips through MOSIS, this revolutionized the academic computer architecture community. Now, inexpensive, but adventuresome, prototyping could be carried on in an academic setting, by students (and faculty) whose growing expertise was in expressing and analyzing novel regular and hierarchical designs.

Before proposing any new and targeted curriculum for nanotechnologies, we will first revisit the existing core of the computer architecture curriculum at the University of Notre Dame – a representative subset of courses that would be taken by a student wishing to specialize in computer architecture. Also, because we propose that in the future there should be greater integration between communities of computer engineers/architects and those actually working on nano device development, we will include an overlay of relevant electrical engineering curriculum – especially that which is targeted toward electrical engineers interested in computer systems. This will be used to show how electrical and computer engineering curricula currently interact and will help define a base for an integrated curriculum targeted toward nano-scale architectures.

Fig. 1 illustrates the existing curriculum. It also includes a listing of goals and topics relevant to each course, shows any overlap between the two curricula, documents popular course sequences, and highlights available course sequences. By examining this figure

one can clearly see that all of the pieces are in place to facilitate interaction and understanding between electrical and computer engineers (or device physicists and architects!). A set curriculum is already in place for electrical engineers who have an interest in computer systems and several course sequences are available for computer engineers interested in the “physics” of logic. (Note: an interesting side project might be to integrate this “roadmap” into the first course, Logic Design (CSE 221), of this sequence to help students see and understand the “bigger picture” earlier.)

At the same workshop mentioned above, when speaking of nano-scale devices, Conway also posed the question of when will there be some emerging areas where designers will be able to compile enough basic information to start generating interesting circuits. At the University of Notre Dame, we believe that one promising “emerging area” is the Quantum-dot Cellular Automata (QCA). QCA stores information within “cells” consisting of multiple quantum dots via the positions of single electrons, and performs logic functions not by electron flow, but by Coulombic interactions between electrons in neighboring QCA cells. Real QCA cells have been fabricated by Notre Dame device physicists that demonstrate the key properties of computation, information transfer, and storage. Also, researchers are on the verge of creating QCA cells consisting of single molecules which may be “self-assembled” into larger structures via attachment to DNA tilings. Truly, QCA is in the nano-scale realm and a subset of actual devices – both theoretical and experimentally proven – exists.

Prior to the beginning of the authors’ research on design with QCA, little work had been done in considering systems of, circuits for, or an architecture for QCA devices. Ironically (and rather unintentionally), our initial work mimicked the experiences of Mead and Conway in more ways than one. First, our interactions with technologists were not as successful as they could have been – because “as a group, most individuals did not span the range of knowledge required to design a complete computer system.” As a particular example, recently we discovered that a QCA circuit characteristic that we (as architects) deemed essential for useful and efficient circuits, was not a priority for device physicists. Clearly, this illustrates the need for better communication and understanding between the two communities. Second, when examining our design process, it has by in large mirrored the path proposed by Mead and Conway to help circuit designers understand the architectural possibilities of a technology.

Now, with many other nanotechnologies consisting of at least a subset of experimental devices, we propose

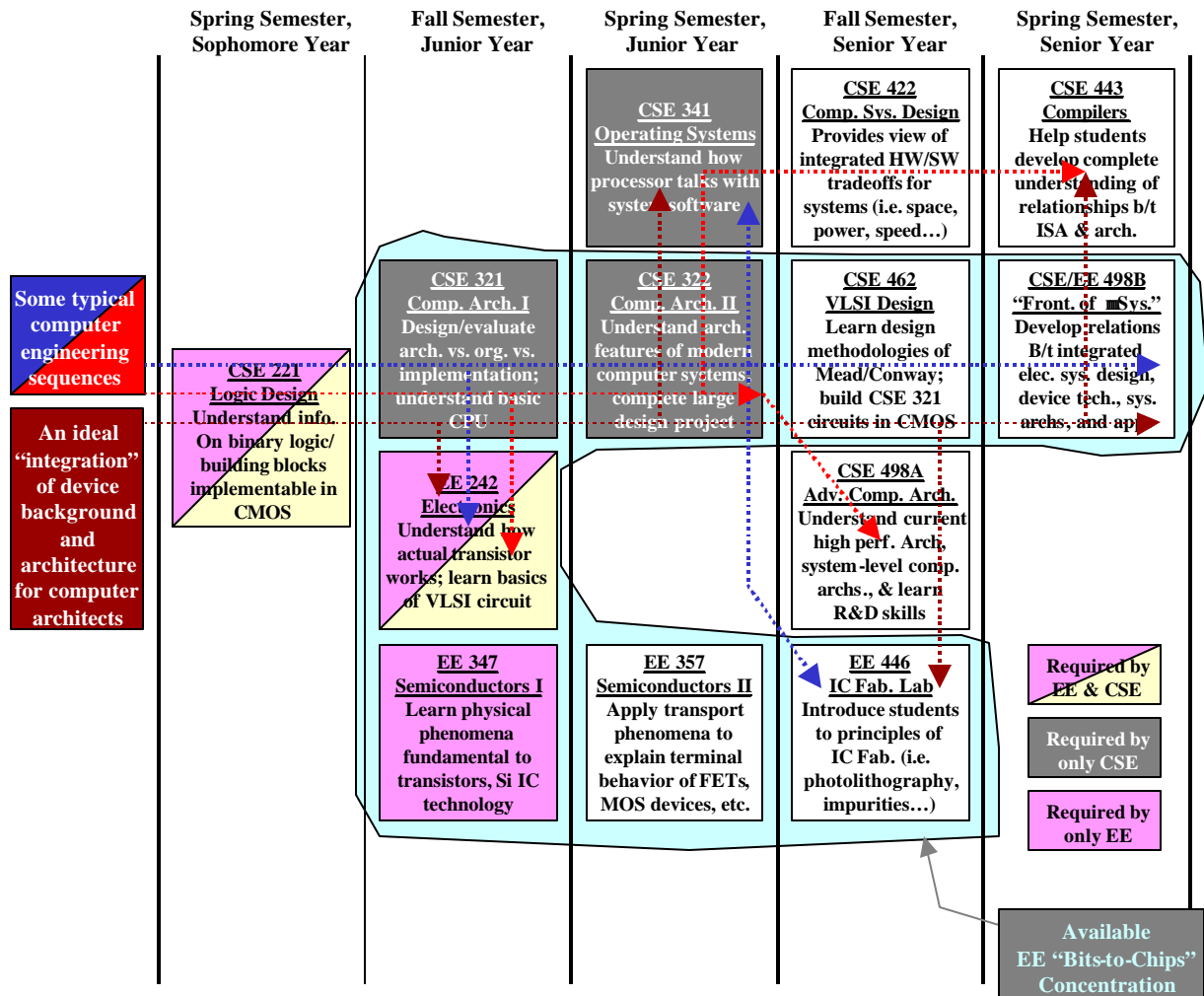


Fig. 1: Existing “core” of “conventional” computer architecture curriculum.

developing a curriculum to teach students how to develop a set of guidelines for computer architects and circuit designers for a specific nanotechnology. The context will include our experiences with QCA and the proven methodologies proposed by Mead and Conway for one of the most commercially successful computational mediums -- CMOS. Eventually an end result might be a “Mead/Conway” study for a specific nanotechnology. However, another (and earlier) goal of the curriculum is to teach students *how* to actually develop a “Mead/Conway” study for *any* nanotechnology. We also propose an extension of their work – namely, preparing computer architects and circuit designers to work with device physicists during actual device development. The end result envisioned is as a group, individuals who span the range of knowledge required to design better devices and complete computer systems.

With these thoughts in mind, Fig. 1 has been augmented in Fig. 2 to show a parallel curriculum that

will end with a “Frontiers of Nano-Systems course” and accomplish one of the first goals stated above – namely educate students on *how* to develop a “Mead/Conway” for any nanotechnology. Interestingly, the second goal (preparing computer architects and circuit designers to work with device physicists during actual device development) should be accomplished by the course sequence itself as a.) it (like a VLSI or logic design course) would be targeted toward both electrical and computer engineers and b.) “the big picture” detailed in the figure below will be explained to students at the beginning of the sequence and act as a roadmap to help the students understand what they are working toward. Finally, Fig. 2 illustrates an approximate time sequence as to where these courses would fit into existing electrical and computer engineering curriculum. They could easily occur simultaneously with or after an appropriate course in “conventional” electronics and architectures. However, they could also be taught *before* the similar “conventional” course. This is based

on the idea that someone who is trying to develop an architecture for a specific nanotechnology might have better success with *less* knowledge of previous design evolutions and/or design methodologies. Would a potential computer architect be better off with just a sound basis of knowledge in the nanotechnology that he or she is trying to develop a “Mead/Conway” for? Would this lead to the best possible design methodology and architecture for *that* particular nano-scale device? Arguments will be made for both cases based on our experiences with QCA.

The rest of this paper will discuss the “CMOS independent” parts of our current curriculum, and what needs to be kept intact from it – largely the hierarchical design approach. We will also detail how we propose to educate students to accomplish the above goals. We will first discuss our proposed curriculum in detail and discuss what background students should bring to it and

learn from it. The next section will discuss why we should – and how to – encourage students to think “outside the box” with regard to circuits and architectures for nanotechnologies. Next, we will consider mechanisms, examples, etc. for introducing students to the actual development of circuit design rules, techniques, and architectures. Finally, we will conclude and discuss future work. Interestingly, each of these sections will be introduced with an excerpt from the text of the Mead/Conway preface indicative of the fact that architects studying nanotechnology will have to face and solve many of the same problems that were first experienced during the last technology evolution.

2. (Student) Background:

“We have chosen to provide and assume that students will bring with them just enough essential

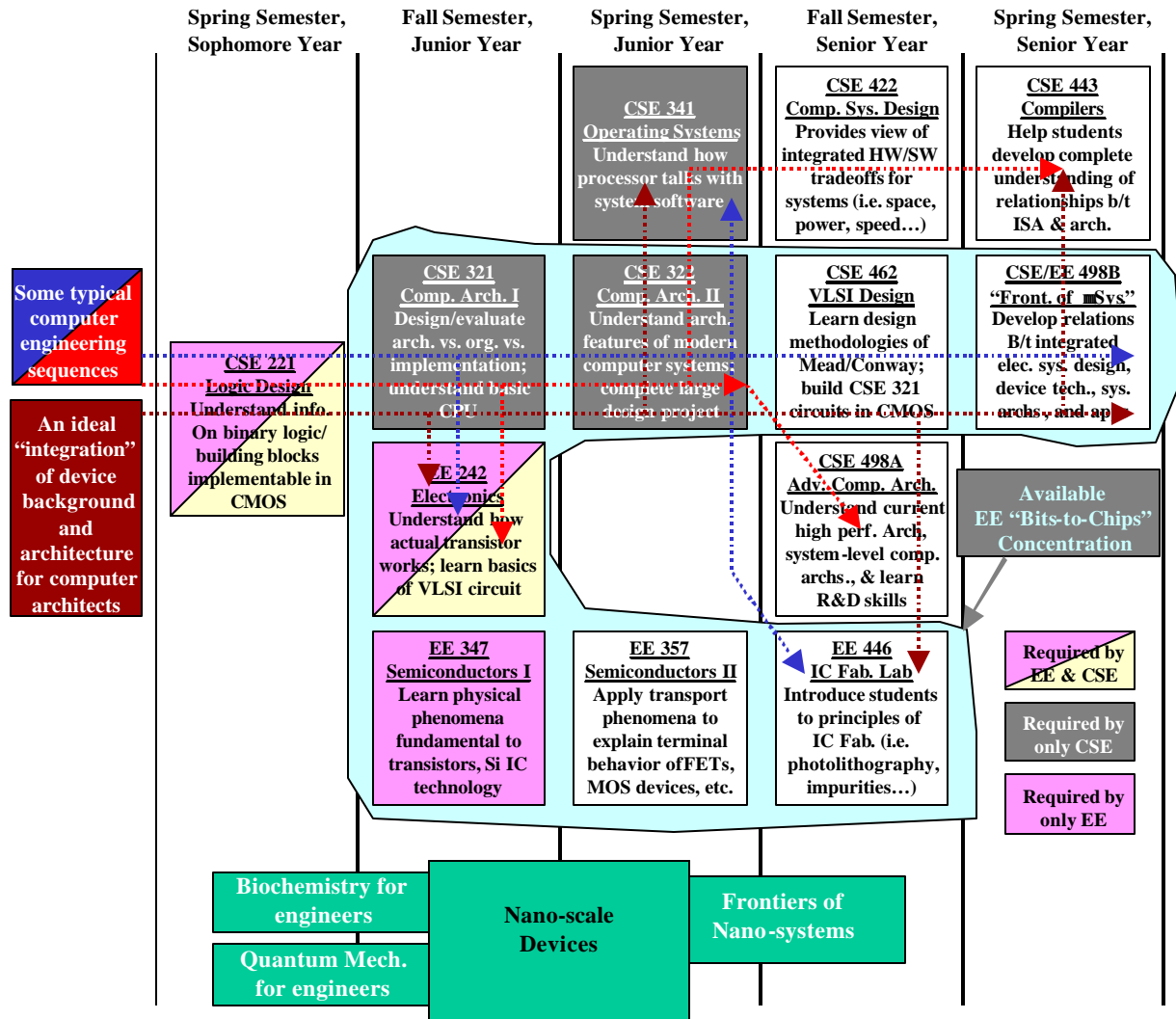


Fig. 2: Existing “core” of computer architecture curriculum augmented with proposed “nano”-curriculum.

information about devices, circuits, fabrication technology, logic design techniques, and system architecture to enable them to fully span the entire range of abstractions from the underlying physics to complete VLSI digital computer systems.” (Mead/Conway vi)

As stated in the introduction, an initial end goal of our curriculum is to teach students how to design a Mead/Conway study for any nanotechnology. The above excerpt from the actual Mead/Conway preface describes what knowledge the authors expected students (including computer architects!) to have in order to understand the design rules provided for VLSI systems. While the existing and “conventional” computer architecture course sequences will provide some needed background for a concentration in nano-scale design, clearly, preparing students for a technological evolution will require additional and different fundamental information as well.

It should be reemphasized that Mead and Conway were proposing a “capstone” class in VLSI design, while we are proposing a curriculum to teach the development of their methodologies as an end goal (which will hopefully, eventually lead to an analogous “capstone” course for a specific nanotechnology). Consequently, we must also define what background – devices, logic design methods, fabrication techniques, etc. – students will need to meet this goal. This “background” must be provided in two different ways. First, an entirely new subset of courses must be developed to teach students the fundamentals of *nano-scale* devices and *nano-scale* fabrication techniques. What should such a sequence entail? This question can best be answered by looking at the different disciplines that are part of various nano-scale device developments. For example, in addition to electrical engineers, physicists, and computer architects, *chemists* are an integral part of the development of QCA. Additionally, other emerging nanotechnologies – DNA-based computing, carbon nanotubes, etc. – all have roots in chemistry. With this in mind we believe that any curriculum designed to teach students how to develop systems of nano-scale devices should include a course in biochemistry – but targeted toward engineers.

Other background information can most likely be derived from existing courses, albeit retargeted for different ends. For example, many emerging nanotechnologies are also rooted in quantum mechanics – Q-bits, QCA, etc. – and at the University of Notre Dame a course in quantum mechanics is available as part of the electrical engineering graduate curriculum (and available to interested undergraduates as well). Part of this existing course could easily be augmented/spun-off and should be targeted toward

engineering students who are interested in circuit and system design.

Together, these two courses – biochemistry for engineers and quantum mechanics for engineers – would provide the foundation for a course in nano-scale devices which would eventually segway into a course intended to teach the development of Mead/Conway-esq design rules and methodologies. This specific course sequence is highlighted in Fig. 3 and each course is paired with its “conventional equivalent”. By examining Fig. 3, one can conclude that the sequence of biochemistry for engineers and quantum mechanics for engineers would provide the same functionality for students desiring to study systems of nano-scale devices that the electrical engineering semiconductors course currently provides for students desiring to study systems of MOS devices. Namely, both teach students about the materials from which computational devices and their substrates can be built.

In the existing curriculum at the University of Notre Dame a course in electronics, which teaches students how computational devices constructed with various semiconductors actually function, occurs in parallel with the semiconductors course. Our proposed and parallel course in nano-scale devices fills the same role as a course in MOS electronics but occurs only after students have studied the fundamentals of how various *nano-scale* devices can actually be constructed. We believe that sequencing these course sets will provide engineering students with the greatest level of understanding about the computational devices.

Our course sequence concludes with a “Frontiers of Nano-Systems” course. The particular class is currently “paired” with the existing VLSI course (which employs and teaches the Mead/Conway design rules and methodologies for MOS) as well as the Frontiers of Microsystems course (which seeks to help students understand the relationships between integrated circuit design, device technology, system architecture, and applications for MOS devices) [3]. However, because there are many promising nano-scale devices and no heir-apparent to CMOS, our proposed “Frontiers of Nano-Systems” currently exists essentially as a combination of its two MOS equivalents. While it might involve case studies of architectures and design rules for existing and promising computational devices, it is more targeted toward helping students understand how such design rules were actually developed. Essentially, the goal of this course is to teach students how to help technology *evolve*.

Ideally, work completed and skills learned in a Frontiers of Nano-Systems course will someday lead to a specific Mead/Conway-esq course for a specific nanotechnology. Such a course might be offered when a nanotechnology has evolved enough that a MOSIS-like conglomerate exists for it. For MOS devices,

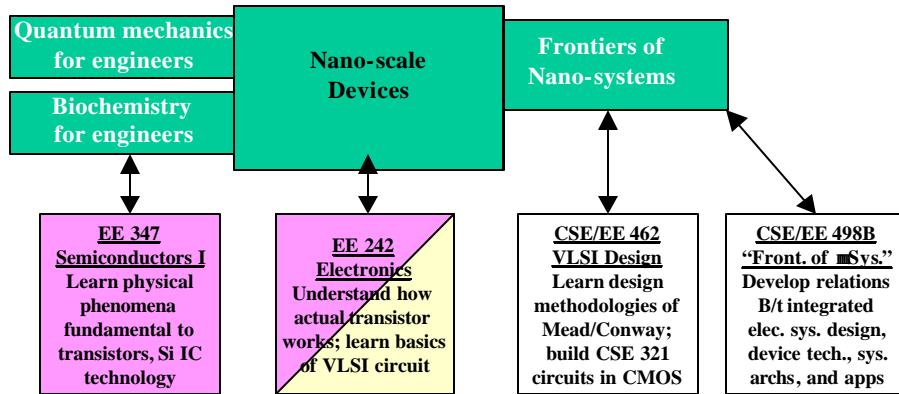


Fig. 3: The core of the “nano”-curriculum with “conventional” curriculum equivalents.

MOSIS (Metal Oxide Semiconductor Implementation Service) provides system designers with a single interface to the constantly changing technologies of the semiconductor industry and allows for the fabrication of their circuits. Were an “NIS” (“Nanotechnology Implementation Service”) to exist, a set of design rules (or single interface) for a specific nanotechnology would also exist. It is these design rules that would form the core of a course that would **not** teach students how to help technology *evolve*. Instead, such a course would not only allow computer architects to prototype and analyze novel and regular devices for a nanotechnology, it would also help a community *adapt* to a new computational medium. Essentially Frontiers of Nano-Systems would become two courses – one to teach students how to adapt, the other to teach students how to keep evolving. (Also, even those who do not participate in an eventual “NIS-targeted” course will have at least seen and experienced what is required to *adapt* to a new technology).

Finally, there are three important generalizations to make about our proposed curriculum for designing with nano-scale devices. First, when examining its “conventional equivalent” one can see that it consists of a mix of electrical engineering and computer engineering courses – specifically one electrical engineering elective, one electrical engineering and computer engineering requirement, and one computer engineering elective. Note that it contains no explicit or existing computer architecture courses (more on this next). However, it does contain a significant “deviation” from the “conventional” curriculum. Namely, previously, a semiconductors course was not a requirement or even a common elective for computer engineers (i.e. computer architects). However, because we want to facilitate closer interactions between electrical and computer engineers (device physicists and computer architects) who are trying to develop nano-scale devices, we believe a semiconductors-like course should be part of the core curriculum. Here this takes the form of biochemistry and quantum mechanics

for engineers which will help ensure that computer architects understand the limits and constraints of what can be built, constructed, or designed with a specific nano-scale device.

Second, as mentioned above, there are no explicit logic design or computer architecture courses that are part of our proposed curriculum. New or retargeted courses are not proposed because in order to understand a simple CPU or build simple computational logic circuitry students still must learn basic logic design techniques and hierarchical design methodologies that “conventional” classes like logic design and computer architecture provide. Now, if a semiconductors-like background should be a requirement for any computer architect working on developing nano-scale devices, then similarly a background in logic design/computer architecture would be ideal for device physicists. While a bits-to-chips sequence for electrical engineers is highlighted in Fig. 1, the front-end of that sequence – logic design and computer architecture – is most essential for cementing a close working relationship.

Third, and finally, in the introduction we posed the question of whether it would be best for a student to take part in this curriculum with either a thorough or a minimal background in logic design, device physics, and principle of VLSI design methodologies. Until now, we have left our proposed course sequences vague with regard to where they would fit into an academic timeline. One could make the argument that it would be best to teach students how to design for a nano-scale device before little or any of the “conventional curriculum” is taught (where “conventional curriculum” refers to MOS equivalent courses as well as courses in computer architecture or VLSI design). This way a student would have no preconceived notions of what a circuit or system must look like or has looked like and might develop the best set of system design rules for a particular nano-scale device. However, an argument against this approach would obviously be that a student would have little if any knowledge about basic design or even how a simple CPU works, severely limiting

what he or she might design. One could also argue that it would be best to prepare students for a technology change only after they have experienced all of the “conventional curriculum”. Then, they would have not only learned basic principles of logic and CPU design, but also will have learned advanced architecture techniques and studied design rules and methodologies for a proven computational medium – CMOS. However, this approach does not separate the process of technology from the process of design and may cloud students’ thinking by teaching them one way to design and study large systems of integrated circuits. Would *this* result in the best, original set of design rules for a particular nano-scale device?

A better answer might actually be a mix of the two arguments. A nano-engineering course in quantum mechanics and/or biochemistry should take place concurrently with a “conventional” semiconductors or electronics class. This way, students will learn the fundamentals of each technology in parallel and will be less inclined to “think” in terms of one technology over another. Similarly, a nano-scale device course should take place concurrently with a computer architecture course sequence and after a “conventional” electronics class. This will allow students to consider how basic CPU requirements and hierarchical design methodologies learned in computer architecture might apply to nano-scale devices. Also, the “conventional” electronics course will provide a student with a good foundation of what a computational device **has** to do, but not necessarily **how** it must do it. Finally, the Frontiers of Nano-Systems class could take place in conjunction with a “conventional” VLSI class (so students thinking is left “unclouded”) or after it (for a better foundation in what designing a Mead/Conway set of design rules is all about). However, we would suggest that students take it before some of the more advanced computer architecture classes. Why? Students will have a generic idea of what a CPU must do but will not be tied to more complex architectural techniques – hopefully leading to a set of original, targeted, and unclouded set of design rules for a particular nano-scale device. Additionally, one could always take advanced architecture courses later and apply techniques learned in them to an existing nano-scale system.

3. Out of the Box:

“VLSI electronics presents a challenge, not only to those involved in the development of fabrication technology, but also to computer scientists and computer architects. The ways in which digital systems are structured, the procedures used to design them, the trade-offs between hardware and software, and the design of computational algorithms will all be greatly

affected by the coming changes in integrated electronics.” (Mead/Conway v)

This Mead/Conway excerpt essentially describes what biochemistry for engineers, quantum mechanics for engineers, and nano-scale devices must teach students to do in our new and parallel curriculum. Obviously a major purpose of these classes and the case studies that will be analyzed in them will be to help students learn “the basics” of the promising nanotechnologies and initialize a close working relationship between device physicists and computer architects. This relationship is critical to prevent these two groups/entities from developing diverging views of what is physically and computationally possible in a system of nano-scale devices. It is best illustrated and explained here (and eventually to students in a class) with a short case study from our experiences with QCA.

Earlier, we alluded to the fact that a QCA circuit characteristic that we (as architects) deemed essential for useful and efficient circuits was not a priority for device physicists. Specifically, an idealized QCA device (or cell) can be viewed as a set of four charge containers or “dots” positioned at the corners of a square. The cells contain two extra mobile electrons which can quantum mechanically tunnel between dots but, by design, cannot tunnel between cells. The configuration of charge within the cell is quantified by cell polarization, which can vary between $P=-1$, representing a binary “0”, and $P=+1$, representing a binary “1”. Unlike CMOS (in which multiple layers of metal can facilitate data routing), there really is no “third dimension” in which to route wire in QCA. However, a wire formed by QCA cells rotated by 45 degrees can cross a wire formed by 90-degree (unrotated) QCA cells in the plane with no interference of either value on either wire. Early in our architectural/circuit design study of QCA, this property was considered to be of the utmost importance as it provided our only other “dimension” of routing. However, when discussing our designs with chemists (who are working on DNA substrates on which QCA molecules could be attached) we realized that they had not yet even considered the interaction of 45-degree cells with 90-degree cells (as for them, this was a very complex design problem). This early collaboration has resulted in some relatively minor changes in the way our circuit and system designs will be structured and has led the device physicists and chemists to reconsider this problem. The result should be a more feasible design with potential for earlier implementation.

Now, we also mentioned in the previous section that the courses in our sequence discussed above would and should take place in parallel with “conventional” logic design and computer architecture curriculum. This should allow and facilitate student thinking about

how the fundamental computational and CPU requirements detailed in these courses could best be mapped to systems of nano-scale devices. This brings us to the second purpose of this course sequence and one that was alluded to when detailing the nano-scale devices course. Namely, by now students will have realized that computational devices *have* to do certain things. However, with nanotechnology, how they do them is very much “up in the air”. Students must be taught to embrace this and how to think outside of the box. Again, this is best presented with a short case study.

An important feature of MOS electronics is a pass transistor that essentially allows current (i.e. binary information) to flow between *a* and *b* in either direction. However, in QCA information is not moved by electron flow but rather by Coulombic interaction between electrons in quantum dots. Because nearness between QCA cells is required to move information from *a* to *b* there is no obvious way to create the equivalent of a pass transistor (either bi- or uni-directional) using only QCA devices. (For example, this would make generating the equivalent of a switching matrix – i.e. for a simple FPGA – in QCA much more difficult – although not impossible). Also, unlike the standard CMOS clock, the QCA clock is not a signal with a high or low phase. Rather, the clock changes phases when potential barriers that affect a group of QCA cells (a clocking zone) pass through four clock phases: *switch* (unpolarized QCA cells are driven by some input and change state), *hold* (QCA cells are held in some definite polarization -- i.e. some binary state), *release* (QCA cells lose their polarization), and *relax* (QCA cells remain unpolarized). One clock cycle occurs when a given clocking zone has cycled through all four clock phases. To understand how the equivalent of at least a uni-directional QCA pass transistor or switch might be implemented, it's worthwhile to consider the exact purpose of the *relax* clock phase. Without it, QCA cells in the *switch* phase could be driven from two different directions (i.e. from cells with a definite polarization in the adjacent *hold* phase and cells with an initial polarization in the adjacent *release* phase). The *relax* phase acts as a buffer to ensure that this does not occur. Thus, the *relax* phase has the effect of “removing” a group of QCA cells from a given design. Using this idea, routing could be accomplished by using the clock to selectively “turn off” groups of QCA cells to create switches.

The timeline of this integrated, “conventional” curriculum and “nano” curriculum is ideal because students will have acquired some knowledge about the fundamental requirements for a CPU and logic as well as what devices are commonly used to implement them in their “conventional” courses. However,

simultaneously, courses such as nano-scale devices will teach students what is and what is not physically possible in the “nano”-realm. One lesson might show how some functionality and logic will certainly map from a standard technology to an evolved technology (i.e. CMOS → QCA). However, another lesson might best be summarized as follows: “You understand device *X*, you’ve used *X* a lot, well, now *X* is no longer physically possible and you’ll need to find a new way to either recreate its functionality or a completely different way to do task *Y*.”

4. Frontiers:

“In any given technology, form follows function in a particular way. The most efficient first step towards understanding the architectural possibilities of a technology is the study of carefully selected existing designs. However, system architecture and design, like any art, can only be learned by doing. Carrying a small design from conception through to successful completion provides the confidence necessary to undertake larger designs.” (Mead/Conway vii)

The above quotation from the Mead/Conway preface actually describes both courses which could eventually result from the sequence biochemistry/quantum mechanics for engineers and nano-scale devices. In the nearer term, a Frontiers of Nano-Systems course will teach students how to develop a set of design rules and system architecture using the methods described in the above excerpt. Explaining how this will be done will best be accomplished (and illustrated) via a series of case studies and comparisons between them.

For example, let’s revisit our work with QCA. Prior to our research, little work had been done in considering systems of, circuits for, or an architecture for QCA devices. Consequently, as with other technologies that preceded it, and like Mead and Conway proposed above, initial studies of QCA started off by designing basic circuit elements that would be needed for a processor. Next, it was determined that a simple microprocessor should be constructed QCA cell-by-QCA cell (essentially in the same manner in which many of the early Intel microprocessors were designed). The processor of choice was simple enough to be designed by hand, yet it still contained the basic elements that are part of any microprocessor (i.e. arithmetic and logic units, registers, latches, etc.). Hence, solutions to the difficulties encountered and overcome in this design would be applicable to even more complex systems and processors. Problems encountered during this design process were largely related to floorplanning – which in turn arose from the interdependence of layout and timing with QCA. As we saw above, the nature of the QCA “clock” leads to

an inherent self-latching of the QCA device. Given this constraint, and before making any further attempts at a large scale design, we felt the need to develop methods to successfully factor the constraints generated by the inherent self-latching of QCA out of the “equation” of a design and furthermore find a means to exploit it. Thus, an extensive study of floorplanning was conducted and several viable floorplans for QCA circuits were developed. After the floorplanning study was conducted, a complete layout of the dataflow for our microprocessor was finished. During this design process, register designs, feedback mechanisms, interconnect problems, etc. were developed and/or identified. Design rules were compiled and formed the engine of a simulator written to test circuits for logical correctness. These design tools were then used to simulate and reanalyze existing design schematics.

Work then proceeded to studying control flow. Interesting results from this work include the lack of a need for an explicit flip-flop to hold a bit of state information in a QCA state machine (the inherent latching in wire stores the bit), more intelligent floorplans to ensure that QCA cells representing bits of state actually change clock phases and polarizations at the proper time, an algorithm for intelligent state placement, and a one-hot state machine that could properly control a QCA dataflow and yet not maintain the “classical” properties of a “true” one-hot (i.e. all bits of state switch at a time *relative* to a set of inputs that determine state). While physically unrealizable in the short-term, when this work is finished the first complete QCA microprocessor will have been designed. Most importantly, this effort will provide the first real insight into how an architecture for a (self-latching) nanotechnology should be organized. Furthermore, as discussed in the third section of this paper, work with hand-crafted designs resulted in the opportunities to review them and collaborate with device physicists which in turn led to a more physically realizable near-term implementation target.

A next logical step will be to examine similar design rule evolutions and compare and contrast them – particularly determining and teaching the characteristics and needs for common threads between existing “Mead/Conway”s (i.e. floorplanning). Finally, as mentioned in the second section of this paper, when an NIS conglomerate exists for a specific technology, this class can itself evolve into a course that specifically teaches that set of system design rules – and helps students adapt to a new computational medium.

5. Wrap-up:

“The general availability of courses in VLSI system design at major universities marks the beginning of a new era in electronics. The rate of system innovation using this remarkable technology need no

longer be limited by the perceptions of a handful of semiconductor companies and large computer manufacturers. New metaphors for computation, new design methodologies, and an abundance of new application areas are already arising within the universities, within many system firms, and within a multitude of new small enterprises. There many never have been a greater opportunity for free enterprise than that presented by these circumstances.”

After changing “VLSI” to “nanotechnology” in the above Mead/Conway excerpt, nothing else need be said.

Acknowledgements:

The authors would like to emphasize the depth of insight we owe to Lynn Conway, whose comments at the MAW workshop and in email exchanges during the preparation of this paper were invaluable.

References:

- [1] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Publishing Company, Inc., Philippines, 1980.
- [2] Molecular Architecture Workshop, Univ. of Notre Dame, Nov 12-13, 2001, www.cse.nd.edu/cse_proj/maw
- [3] G.H. Bernstein, and J.B. Brockman, and G.L. Snider, and P.M. Kogge and B.E. Walvoord. “From Bits to Chips: A Multidisciplinary Curriculum for Microelectronics System Design Education”, American Society for Engineering Education IL/IN Sectional Conference, April 12, 2002 – Illinois Institute of Technology, Chicago, IL 2002.

Using Custom Hardware and Simulation to Support Computer Systems Teaching

Murray Pearson, Dean Armstrong and Tony McGregor
Department of Computer Science
University of Waikato
Hamilton
New Zealand
{mpearson,daa1,tonym}@cs.waikato.nz

Abstract

Teaching computer systems, including computer architecture, assembly language programming and operating systems implementation, is a challenging occupation. At the University of Waikato we require all computer science and information systems students study this material at second year. The challenges of teaching difficult material to a wide range of students have driven us to find ways of making the material more accessible. The corner-stone of our strategy for delivering this material is the design and implementation of a custom CPU that meets the needs of teaching. In addition to the custom CPU we have developed several simulators that allow specific topics to be studied in detail.

This paper describes our motivation for developing a custom CPU and supporting tools. We present our CPU and the teaching board and describe the implementation of the CPU in an FPGA. The simulators that have been developed to support the teaching of the course are then described.

The paper concludes with a description of the current status of the project.

1 Introduction

Teaching computer systems is a challenging but vital part of the computer science curriculum. In 1997 the Department of Computer Science at the University of Waikato decided that computer systems was important to all computer science and information science students and made its computer systems course compulsory for all second year students. Like most computer systems courses Waikato's uses assembly language programming as a vehicle to understanding the inter-relationships and interactions between the different components of a computer system. The brief of the course is quite differ-

ent to an introductory computer architecture course, even though it contains many of the same components. The difference lies in the audience and motivation. Our course is intended to be useful to all computer professionals, not just those who specialise in computer architecture. Our use of assembly language programming is an example of the impact of this difference. Very few of the students will continue to program in assembly language after the course, however, we believe that it is important that they have an understanding of computer operation at this level of abstraction. While we want to teach a coherent and realistic architecture we have no fundamental interest in details such as delay slots, addressing modes and word alignments. These are important topics for a specialist, but do not significantly add to the understanding of the operation of a computer system as a whole, which is the goal of our course. Assembly language is essential to this goal but many students find assembly language programming difficult and this detracts from the main thrust of the course, which is not to teach assembly language per say.

We wish to focus on the role of the machine and the interactions between the hardware and software components including compilers, libraries and the operating system, rather than spending a lot of time describing a particular manufactures performance oriented features. This has led us to develop our own instruction set architecture called WRAMP. As described later the course has a practical component; practical exercises reinforce the content of the lecture material. To support the practical component of the course using the WRAMP instruction set has required the development of a platform to allow students to assemble and execute WRAMP programs. The two choices considered were the development a WRAMP simulator or a custom hardware platform.

Using a simulator is easier and cheaper however we believe that the lack of real hardware distorts the learning

environment by adding an extra, unnecessary, abstraction when many students are struggling to come to grips with the essential content of the course. A simulator is itself a program running on a computer. This makes it difficult for students to readily identify the target system and they tend to confuse the role of components of the system. When this happens there is a risk that students will focus on the most obvious difference between practical work in this area and others: the programming language. When real hardware is used, the real focus is more likely to be on the target system.

For this reason, we believe that students should have the benefit of real hardware when they first learning assembly language programming. Until recently this would have excluded a custom CPU design, however it has been made possible by advances in reconfigurable logic. We have used FPGA technology to develop a single board computer (called REX) with with our own custom designed CPU and IO devices.

Once the students have developed a clear mental model of the components of a computer system, simulation can be used to enhance their understanding of the more complex topics in the course. To this end we have developed simulators for use in the course, two of which are presented here. The first of these, called RTLsim, is used to simulate a simple non-pipelined MIPS processor to demonstrate how instructions can be fetched from memory and executed. The second of the simulators is a multi-tasking simulator that introduces students to the ideas behind task swapping in a multitasking kernel.

The next section gives an outline of our computer systems course. Section 3 then describes, in more detail, the motivation for developing a processor and board to support the teaching this course. Sections 4 and 5 describe the design of the CPU and board. Section 6 then describe the simulators that that are used in the course followed by Section 7 which briefly describes the exercises carried out by students on the course.

A brief description is then given of how we intend to use the board in the third and fourth year computer architecture courses.

2 Course Outline

When the Department decided to make the second year computer systems course compulsory, its curriculum committee established a set of key topics that should be covered by the course. These included: data representation, machine architecture (including assembly language programming), memory and IO, operating systems and data communications.

Figure 1 shows the order of the topics that make up the course and the relative levels of abstraction used to describe them. The main content of the course can be

broken into two parts. The first part illustrates what happens to a high level program when it is compiled and executed on a computer system. This serves two purposes. First, it demonstrates some of the major issues which determine the performance of a computer system. Second, it shows the likely consequences of writing a particular construct in a high level programming language in terms of speed and size of the code generated.

The aim of the second part of the course is to produce an understanding of operating system principles and components, their role in supporting the user, and in the execution of programs written in high level languages such as C (the starting point of the course). The focus is on achieving an understanding with the operating system and the implications of hardware and software choices, rather than an ability to write a new one.

There is a strong theme of interactions and relationships between the components of a computer system. To support this we base the whole course around a single processor architecture so that the students could more easily see the way the individual components of the system contribute to the complete computer system.

3 Background

Because the goal of the course is to explain the role and interaction of the components of a computer system, not to teach assembly language programming for its own sake, there are two main requirements for a architecture:

1. a simple, easy to learn instruction set
2. an architecture that can easily demonstrate the relationship between high and low level languages, and user and kernel space.

These goals are at odds with most modern CPU architectures which have been optimised to maximise performance and not simplicity. To help achieve high performance modern CPUs contain many performance oriented techniques including the use of reorder buffers, register renaming and reservation stations[6]. Because of the complexity of these architectures it would not be possible to fully describe the structure and functionality of one of them in an introductory course.

While most architectures are optimised for performance some (such as the 8-bit processors -e.g. the Motorola HC11) are designed to be very cheap and simple. However, this very simplicity often raises the complexity required to program the CPU. For example, performing 16-bit indexed address access on an 8-bit processor that only has an 8-bit ALU requires a series of instructions to support the 16 bit addition rather than the single instruction available on larger word sized machine. Because of the way CPUs developed through the late '80s and early '90s, processors with a large enough word size to make

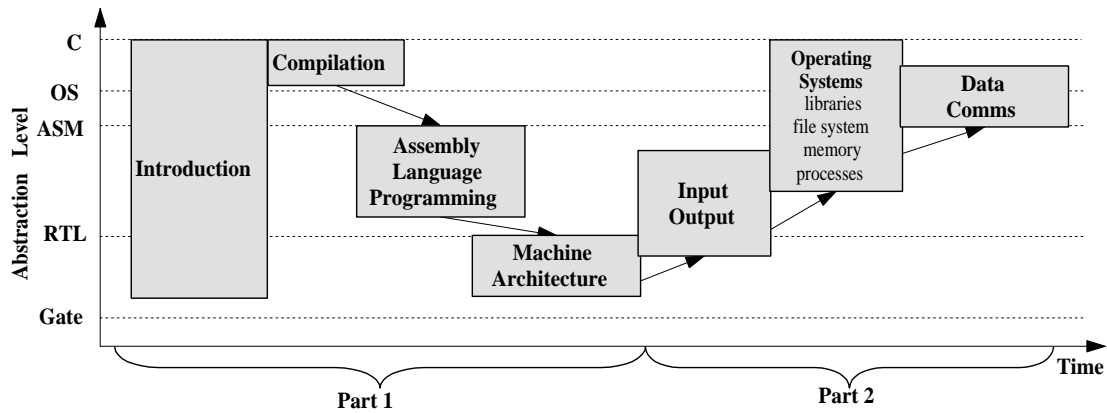


Figure 1. Topics Covered in the Course

those aspects of programming easy have other complexities, such as many addressing modes, that are not available across all instructions or complex interrupt processing. Although many modern CPUs are simpler, because of the influence of the RISC philosophy, they have other disadvantages, including branch and load delays as described below.

In the past, we have used the MIPS R3000 family as a compromise between the needs of our course and available CPU designs [4]. The MIPS CPUs have a relatively simple programmer's abstraction. The teaching process is also supported by a number of very popular text books including those written by Hennessey and Patterson [3] [2] and Goodman and Millar [1]. For this reason our computer systems course has been based around this processor for the last six years. While we have found this processor reasonably well suited to our needs, we have identified a number of aspects of the architecture that many students find difficult to understand and which are not central to our teaching goals. These include:

- the presence of *load delay slots* which mean that the instruction directly after a load instruction cannot use the result of the load as isn't available yet.
- the presence of *branch delay slots* which mean that the instruction directly after a branch instruction is always executed regardless of whether the branch is taken or not.
- the use of an *intelligent assembler* which is capable of reordering instructions and breaking some assembler instructions in two so that they can all be encoded using a single 32-bit word.
- the requirement that all *memory accesses to word values are word aligned*.
- the *parameter passing conventions* that are designed

to minimise the number of stack manipulations in a MIPS program.

While we do not believe that the complexities described above are insurmountable, they do detract from the goal of the course, that is to give a complete coverage of the computer systems area at an introductory level without being distracted by the complexities associated with describing a particular manufacturers quirks. This is in keeping with the introductory level and broad audience that this course is intended for. Other courses at the University are intended for students who will specialise in computer architecture, and these do cover commercial architectures, including exposure to many of these issues.

We have been unable to find a suitable commercial CPU architecture to support the teaching of our computer systems course so we developed our own.

Before discussing the architecture of the CPU we have designed we consider the question of whether to use a real CPU or a simulator. Most courses that teach computer architecture or assembly language teaching make use of CPU simulators. Using a simulated system offers two main advantages. Firstly, it is possible to develop a simulator for any CPU. This allows a CPU that is tailored to the goals of the course to be used rather than being limited to those that are available commercially. The second advantage of using a simulator is that simulators normally offer better debugging facilities and visualisations of a program. These can be used to help reinforce important concepts.

As noted in Section 1, using a simulator also introduces difficulties for students. It is more likely that students will confuse the boundaries between the host system and the simulated system. Our experience suggests there is a tendency for students to focus on the programming language when a course introduces a new language, rather than conceptual material in the course. The use of real hardware makes the distinctions between the target

system and the development tools concrete. The work presented in this paper largely removes the disadvantages of using a real CPU and enables both a simpler working model and a CPU designed to meet the needs of teaching. This includes good debugging facilities such as the ability to single step and observe register and memory values as the system executes.

4 Processor Design

In designing the processor a great deal of care has been taken to keep the design as simple and regular as possible while still being able support the complete range of practical experiences we wish the students to be exposed to. These experiences start with the writing of simple assembly language programs and build up to the development of a very simple multi-tasking kernel.

The resulting CPU design uses a 32 bit word, and is based around a register-register load-store architecture, very similar to the MIPS and DLX [5] processors. Most computational instructions have a three operand format, where the target and first source are general purpose registers, and the second source is either a register or an immediate value. Regularity of the instruction set was a key factor in maintaining the simplicity. Immediate flavours of all computational instructions are provided, as well as unsigned versions of all arithmetic instructions.

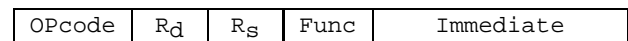
Care was taken to keep the correspondence between assembly language instructions and actual machine instructions as a one-to-one relationship. To this end a major feature of this CPU is the reduction of the address width to 20 bits, and the number of registers to 16. This allows an address, along with two register identifiers and an opcode to fit into a single instruction word, removing the need for assembler translation when a program label is referenced.

The other main differences from MIPS and DLX are the removal of the branch and load delay slots, and the fact that the CPU is 32 bit word addressable rather than byte addressable. Making the machine word addressable only, greatly simplifies the operation of the CPU, and allows us to present students with an easily understandable model of it. Another advantage of a word addressable machine is that it removes the possibility of word access alignment problems which new students frequently encounter on a byte addressable machine.

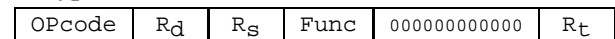
The CPU only supports three instruction formats as shown in Figure 2. It can also be seen from this figure that the instructions have been encoded to allow for easy manual disassembly from a hexadecimal number, with all fields aligned on 4 bit boundaries.

While the CPU has been made as simple as possible for the tasks we require it does include external and software interrupts and has supervisor and user modes with protection. These mechanisms are accessed through a

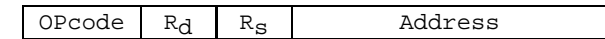
I-Type instruction



R-Type instruction



J-Type instruction



OPCode	4 bit operation code
R _d	4 bit destination register specifier
R _s	4 bit source register specifier
R _t	4 bit source register specifier
Func	4 bit function specifier
Immediate	16 bit immediate field
Address	20 bit address field

Figure 2. Instruction encoding formats

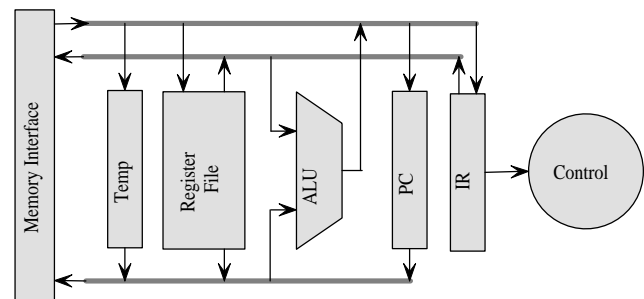


Figure 3. Processor Block Diagram

special register file, similar to the MIPS' coprocessor 0. This means that these concepts need not be discussed for students to begin programming in assembler, and when desired, they can be introduced by describing the special register file, and the two instructions needed to access its contents.

The data-path of the processor is based around a three-bus structure (as shown in Figure 3) and instructions take multiple clock cycles to execute. As can be seen from Figure 3 the CPU's data-path is very simple making it possible to completely explain the operation of the data-path to second year students. In particular it is possible to explain in detail how machine code instructions stored in memory can be fetched, decoded, and executed on the data-path.

The CPU has been represented in VHDL so that it can be targeted to a reconfigurable logic device. The CPU design when synthesised consumes a large portion of a 200 thousand gate Xilinx Spartan II FPGA device.

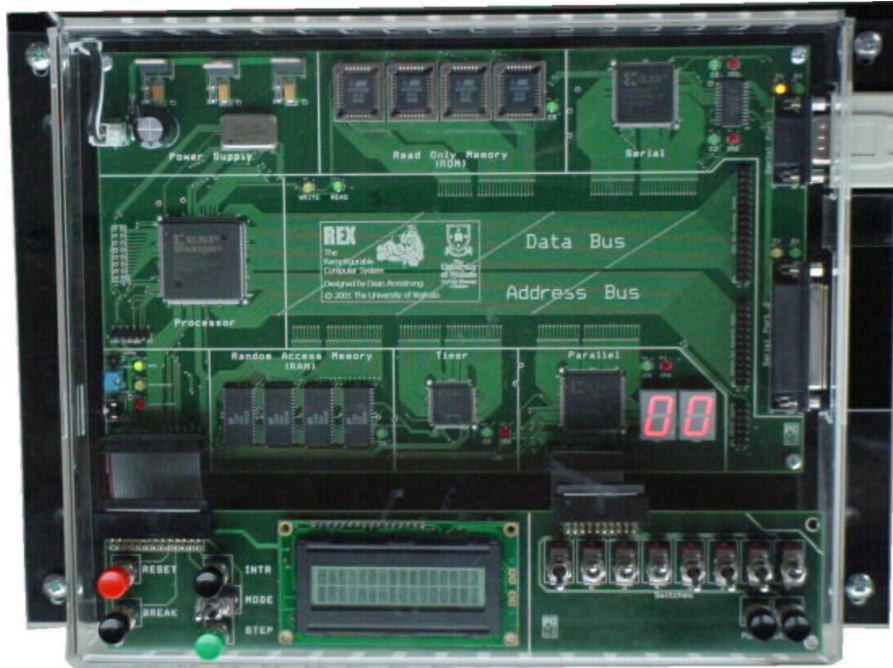


Figure 4. The Teaching Kit

5 Board Design

Figure 4 shows the REX board designed to support the CPU described in the previous section. As can be seen from the picture we have been careful to layout the board so that the main components that make up a computer system can be clearly identified. The main data-paths that connect these components are also visible on the board.

Reconfigurable logic is used wherever possible on the board to allow it to be as flexible as possible. In addition to making the design of our own CPU and IO devices possible, this allows the architecture of these components that students are presented with to be fine tuned as the course develops. As explained later, it also allows the board to be used for multiple teaching functions, including FPGA and CPU design.

While it would have been possible to place most or all of the reconfigurable designs into a single chip the decision was made to use a separate chip for each IO device and the CPU, making it possible for the students to physically identify each of these devices on the board. The choice to use multiple RAM and ROM chips to provide the 32 bits of data rather than employing multiple accesses to a single chip was also made with the intention of clarifying the operation for the students. Effort was made, however to keep the number of non-essential support components to a minimum.

The boards are intended to be connected to a workstation where students can write and assemble programs,

which can then be loaded and run on the board. Because we wanted to build a laboratory for a large class it was important to make reconfiguration easy. In particular we designed the board to support remote reconfiguration of all programmable devices and the stored bootstrap program code. Scripts have been developed that enable all of the REX boards in a laboratory environment to be completely reconfigured from a single command. Cost has also been kept to a reasonable level.

Although there are a number of features that support teaching, one that had a large impact on both the board and CPU design is support for cycle-by-cycle stepping of the processor with an LCD display to indicate bus contents, and LEDs to show device selection and exceptions. We believe this feature will be a major asset for students struggling with the many new abstractions and concepts presented by the course.

6 Simulators

In the course we use a number of simulators to reinforce some of the more complex conceptual material. The first simulator (RTLsim) has been developed to reinforce the ideas associated with the execution of machine code instructions on a data-path. The second simulator is a multi-tasking simulator that introduces students to the ideas behind task swapping in a multitasking kernel.

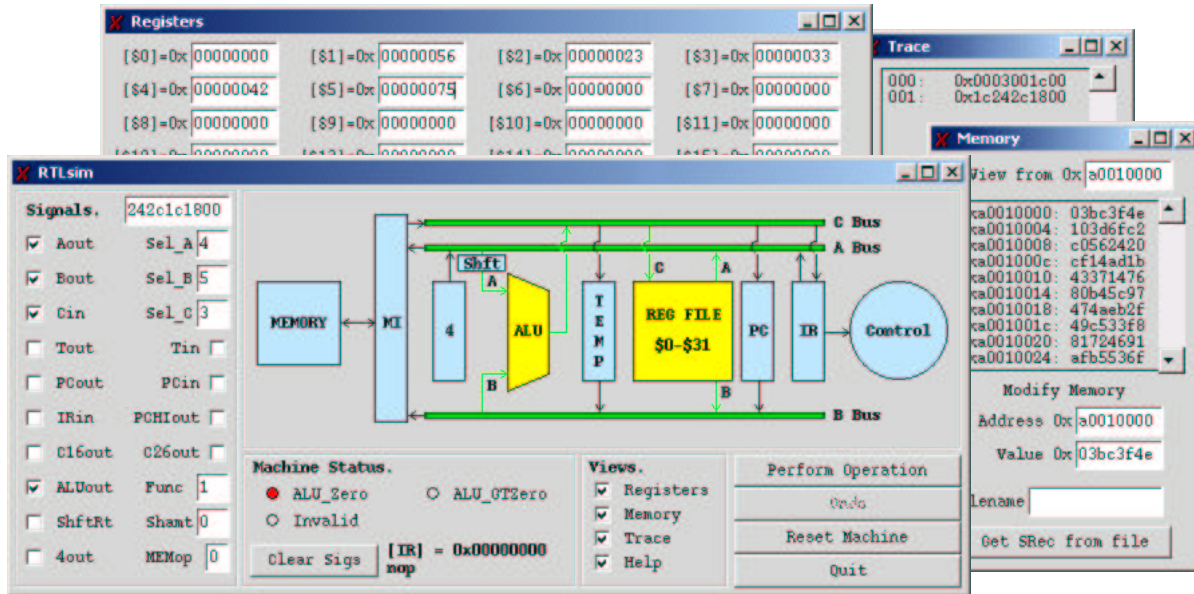


Figure 5. Screenshot showing RTLsim in operation

6.1 RTLsim

In the first part of the course the students learn the relationships between a program written in a high level language such as “C” and its representation in assembler and machine code. Following on from this we show the students how a machine code instructions can be executed on a simple processor data-path. In previous years a simulator called RTLsim which simulates the data-path of a simple non-pipelined MIPS like processor has been used to support the teaching of this component of the course. Currently we are in the process of developing a WRAMP version of the simulator. The rest of this section describes the MIPS version of RTLsim.

RTLsim is written in C for a UNIX system running X-windows. When the simulator is run the student (user) acts as the control unit for the data path by selecting the control signals that will be active in each control step.

Figure 5 shows the main window for the simulator that comprises of two main components, a visual representation of the data-path and a control signals window. The data-path is made up of a 32-register register file, ALU, Memory interface and a number of other registers to store values such as the program counter and the current instruction being executed. Three internal buses are used to connect to connect these components together. This combination of components and buses is sufficient to fetch and execute most of the instructions in the MIPS R3000 instruction set. The control signals section of at the left hand end the main window is used by the student to set the values of control signals that are going to be ac-

tive in the current control step. For example consider the execution of the MIPS instruction `add $3, $4, $5` that adds the contents of register 4 to register 5 and store the result into register 3. Assuming the instruction has been fetched into the instruction register during earlier control steps then the settings shown in the controls signals window of 5 would cause the necessary actions to occur to execute this instruction. As the student sets the control signals for a control step they are given visual feedback on the data-path of what will occur when the control step is executed. For example if the PCout signal is selected the colours of the PC register and the B Bus would change to show that the PC register is going to output a value onto the Bbus. If two components try to output to the same bus at the same time the bus would turn red to indicate an illegal operation.

From the main window, other windows may be opened that show the contents of memory and the register file. In the case of the memory window it is possible to preload memory image from an file in s-record format before starting a simulation. This is the same file format used to upload programs to the MIPS board. This enables the students to upload and execute the same program on both a MIPS board and RTLsim, allowing the executions to be compared.

The simulator can also record a trace of the operations that are performed in each control step. This trace can be used by the student to playback the operations in the simulator or used as input to an automated marking system.

Before RTLsim was introduced to the course the students were given a paper-based exercise where they had

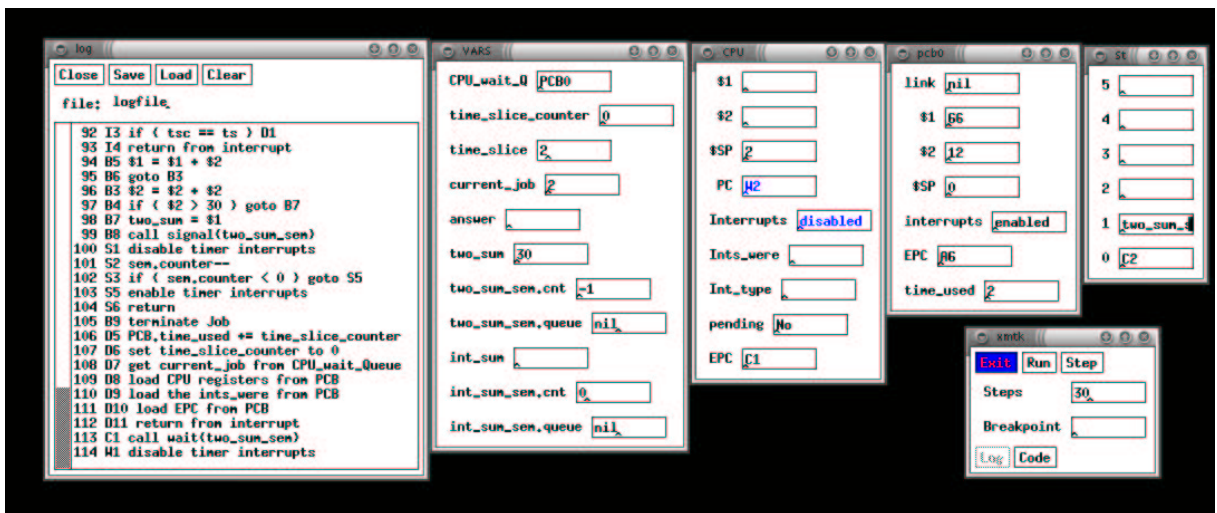


Figure 6. Multi-tasking simulator

to define the sequence of control steps necessary to execute a set of MIPS instructions they were given. If the students had not grasped the main concepts they completed the entire exercise incorrectly and were not given any feedback until the assignments were marked and returned to them several weeks later. However with the introduction of RTLsim the students are given immediate feedback at several levels. Firstly as the students set the control signals they are given visual feedback on the data-path. Once they believe they have the necessary signals to execute the control step they can try it and observe the outcome in the registers and memory. If the outcome is incorrect the simulator provides undo operations so they can try again. Lastly, an automated marking system is used. If the exercise is not completed correctly the marking system generates a set of comments that tells the students where they went wrong so they can try again.

6.2 The Multi-tasking Simulator

One of the assignments undertaken by students in the second year course using the boards is the development of a very simple multi-tasking kernel. The kernel does not include memory management, task creation or termination but it does share the CPU between three tasks, including the saving and restoring of state and changing of stacks between tasks. The tasks are designed to use different parts of the hardware. One reads the switches and writes the value read to the seven segment display, another reads characters from the secondary terminal and writes the uppercase values to the terminal. The third task displays the time on the primary serial port. Students have already written these tasks in a single task environment, in earlier assignments.

Although the multi-tasking kernel does not require

very many lines of code, there are conceptual and coding barriers to its implementation. We address these issues in classes but have found it useful to re-enforce the ideas using a multi-tasking simulator, before students attempt their own implementation. The simulator is written in C for X-windows and creates a number of windows. An example of the windows is shown in figure 6. Each task has two windows associated with it, the first is the stack and the second is the saved state of the task (its process descriptor). An example for one task is shown in the right most two windows in figure 6. When the students use the simulator there are three tasks; two have been omitted here to save space. The link field is used to form a linked list of tasks waiting for the CPU or waiting on a semaphore for an event.

Moving to the left in figure 6 the middle window shows the CPU registers. The simulated machine has only two general purpose registers, a stack pointer, a program counter, a status register and a saved program counter which shows the value of the program counter as it was at the last interrupt. The status register is divided into the interrupt status (masked or enabled), the interrupt status before the last interrupt (software interrupts are taken even if interrupts are masked), the type of interrupt (e.g. timer interrupt) and whether there is an interrupt pending (when interrupts are disabled).

The window second to the left shows the values of some shared memory variables. These include the head of the CPU wait queue, the number of interrupts left in this time slice, the job currently using the CPU the output of two of the tasks (answer and two_sum), and semaphores that hold task 3 until these two tasks are completed.

The left hand window, which gives a trace of the in-

structions that have been executed. The simulator executes pseudo-code which has been designed to be close enough to WRAMP assembly code that it is easy to imagine the assembly code that matches a pseudo-code instruction, but without some of the confusing detail of assembly code. The number at the left of the log window indicates the sequence number of the instructions that have been executed. The letter/number code next to the sequence number is the address of the instruction. The letter in the address indicates what part of the code (A = task A, F = first level interrupt handler, W = wait, S = signal, etc.) the instruction belongs to.

As each step of the simulation is executed the values that change are highlighted in red in the appropriate window. Students are able to change the values at any time to alter the course of the simulation. The assignment encourages them to do this, including altering the time-slice length.

Readers interested in obtaining the simulator should contact the author at `tonym@cs.waikato.ac.nz`.

7 Assignments

The assignments that make up the practical component of the course are shown in Table 7. Of particular note is the implementation of a multitasking kernel by the students. Given that most students are not computer technology students and that most successfully complete this exercise we believe this is a major indication of the success of the course.

No.	Name
1	Introduction to Unix
2	Data Representation
3	Introduction to REX
4	C and WRAMP assembly
5	RTL Design Exercise
6	Parallel and Serial IO
7	Interrupts
8	Multitasking Kernel Simulator
9	Multitasking Kernel Coding
10	Error Detection

Table 1. Assignments

8 Use of the Board by 3rd and 4th year Students

We are currently teaching students in a third year computer architecture course about design using VHDL. By the end of the course the students will be able to design the main components (ALU, registers, finite state machines, etc) that make up a CPU. In future years we plan to use the REX boards to support the teaching of this course.

In our fourth year computer architecture course, students design and implement their own CPU. Last year the students used a prototype version of the REX board to implement their CPUs. With the introduction of the new board and the experience gained using the board in the second and third year courses, we hope to extend the complexity of the project undertaken in this course.

9 Conclusions

This paper described the range of hardware and software tools that have been developed to support the teaching of the introductory Computer Systems course at the University of Waikato.

There is much merit in the design of custom CPU and IO devices for teaching purposes. Current reconfigurable hardware devices have made it possible to build a single board computer, with a custom CPU and IO devices, to support the teaching of computer systems courses. Using this approach we have removed some of the ‘sharp edges’ of assembly language programming, like branch delay slots and complex CPU status control, that add complexity to introductory teaching but do not add significant value.

An additional advantage is that the board will provide a consistent teaching platform across a range of courses. We expect that this will considerably enhance the students learning experience.

We have just installed 25 REX boards in one of the Departments Computer Labs. Supporting tools such as a monitor program for the board, a C compiler, an assembler and linker are now largely complete. Over the past couple of weeks students have been using the REX boards to complete their assignments. All of the feedback we have had from the students todate has been very positive and encouraging.

References

- [1] J. Goodman and K. Millar. *A Programmer’s View of Computer Architecture with Assembly Language examples from the MIPS RISC Architecture*. Oxford Press, 1992.
- [2] J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan-Kaufman, 1995.
- [3] D. A. Patterson and J. Hennessy. *Computer Organisation and Design: The Hardware/Software interface*. Morgan-Kaufman, 1994.
- [4] M. Pearson, A. McGregor, and G. Holmes. Teaching computer systems to majors: A MIPS based approach. *IEEE Computer Society Computer Architecture Technical Committee News Letter*, pages 22–24, Feb. 1999.
- [5] P. M. Sailer and D. R. Kaeli. *The DLX Instruction Set Architecture Handbook*. Morgan-Kaufmann, 1996.
- [6] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11, pages 25–33. 1967.

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 inter-

esting. 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 pos-

itive 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 original 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 ac-

tually 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. Ad-

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 han-

dlers. 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:

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.

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.

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.

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 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.

Improving Computer Architecture Education Through the Use of Questioning

Mark Fienup and J. Philip East
Computer Science Department
University of Northern Iowa
Cedar Falls, IA 50614-0507
fienup@cs.uni.edu
east@cs.uni.edu

Abstract

Learning is not a spectator sport! Yet, the majority of classroom time is spent lecturing. While traditional lecture might be useful for disseminating information, textbooks and web pages already do that. Why spend valuable class time telling students what the book says. Students need to be more engaged than listening and note taking allow! In-class questioning can be very effective at actively engaging students. This paper provides some background information about questioning, supplies some process suggestions for those wishing to enhance their use of questions, and provides some Computer Architecture specific examples of questions.

1. Introduction

For several years we have realized that traditional lecture is too passive and probably is not the best use of in-class time. Studies have shown that after 10-15 minutes of lecturing students essentially stop learning, but their attention-span clock is reset by interjecting activities to break up the lecture. (Stuart & Rutherford 1978) Additionally, Students retain only a small fraction of the material covered, attendance only has a marginal effect on performance, and learning via lecture is independent of the lecturer's quality. (Stuart & Rutherford 1978) The bottom line is that lecture is not very effective!

We accept as fundamental that it is desirable to have "engaged" students who "actively" process the content we attempt to teach them. Active learning (rather than passive memorization of content) should be the goal of instruction. Achieving active learning is, however, not necessarily easy. Our goal became to better understand the art and science of asking questions in class so that our students would learn more or better by being actively engaged in the content of our courses. At WCAE 2000, Fienup (2000) explored the use of active and group learning in Computer Architecture. This paper is an extension of that work by providing some background information about questioning, supplying some process suggestions for those wishing to enhance their use of questions, and providing some Computer Architecture specific examples of questions.

We discovered that there are a variety of goals that one might have when asking questions. The next part of the paper will discuss various goals for questions and other insights we gained from the literature and our conversations. The bulk of the paper will include exemplar questions and attendant goals. We hope they will be useful to readers who wish to include more questioning in their Computer Architecture teaching (and allow some to skip the step where you say "duh" and hit yourself on the forehead for not realizing that there is more to questioning for active learning than just blithely asking questions).

2. Background RE Questioning

We used several techniques for gathering information about questioning. We examined readily available literature, reflected on our prior experiences with questioning, and talked about our experiences. From these activities, we identified several goals of questioning in the Computer Science classroom:

- to have students practice a skill
- to grade student performance
- to provide students with practice on applying knowledge
- to motivate a topic
- to motivate students
- to gauge student understanding
- to engage students in active learning
- to develop students' meta-knowledge
- to regain/reset student attention spans

In examining the literature (e.g., Dantonio & Beisenherz, 2001, Chuska, 1995, Wasserman, 1992; Wilen, 1991), we encountered similar lists. For example, Wilen (1991) indicates that

Although the two major enduring purposes of teacher questions are to determine student understanding of basic facts associated with specific content and to have students apply facts using critical thinking skills, educators have suggested other related purposes:

- to stimulate student participation
- to conduct a review of materials previously read or studied
- to stimulate discussion of a topic, issue, or problem
- to involve students in creative thinking
- to diagnose students abilities
- to assess student progress
- to determine the extent to which student objectives have been achieved
- to arouse student interest
- to control student behavior
- to personalize subject matter to support student contributions in class (p. 8-9)

Both these lists can probably be condensed. They do, however, suggest rather strongly that a variety of goals may be achieved via questioning

and that the questioning activity is not simple. Additionally, we also note that the results of questioning activity can probably be classified as recall of knowledge and application of knowledge (understanding).

From our perspective, recall of knowledge is important but probably does not constitute active learning (which is our goal). We might, however, legitimately use a recall question to achieve a goal such as assessing student knowledge and understanding, or as a motivational lead-in to stimulate student interest in or attention to upcoming topics.

The goal in which we are most interested is that of engaging students' minds on the current lecture topic in a relatively restricted way. We see the role of in class questions to be one of initiating intellectual activity in student minds. In general, such activity might involve:

- practice of some specific intellectual activity, e.g., designing, testing, debugging, interpreting specifications, etc.
- applying specific knowledge
- having students examine their own knowledge and understanding

While we have approached this goal from the point of view of questioning, we assume we are not restricted to oral questions or even to questions. Asking students to engage in an intellectual activity can be construed as asking a question.

3. Process Suggestions

Obviously, we suggest that questioning (and other activity) be used to engage students more actively in the content of Computer Architecture. But that is not as simple as asking questions. It must be planned. The planning may need to involve a variety of issues and occur at various times and levels in a course.

Before the course begins, we recommend familiarizing yourself with the various goals and types of questions that can be asked and

considering the impact on course planning. For example, we believe that there are benefits to having small (4-5 students) groups working together on questions. Group formation can be left to students or dictated by the instructor. We prefer the latter. If the "better" students are spread throughout the groups, there is potentially a teacher per group. Weaker students are more likely to ask questions of their peers. Because students' mental contexts have more in common with students than the professor, the student "teacher" in the group may be in a better position to communicate effectively. We believe that the better students also benefit by trying to explain concepts to weaker students. Think about how much you learned about the material of a course the first time that you taught it.

You should also consider addressing your goals for the in-class questioning activity in your syllabus and, occasionally, in class. If students understand why you are asking so many questions and not just "telling" them what they are supposed to know, they may well participate more fully and learn more. You may also wish to incorporate some aspect of grading (e.g., class participation) to reflect your opinion of the importance of active learning. We would suggest about 10% of the course grade be based on in-class participation of the questions. We base this portion of their grade on student evaluations from peers within their in-class groups.

Before each class or unit, plan your questions. Questions should be used to enhance the learning of the most important topics of each class. Identify the most important content goals or ideas in the lesson. Then proceed to planning your lesson (and the questioning you will use in it). It is as important to consider what you are going to ask as it is to consider what you are going to tell. Do not treat your questions lightly. Consider the goal(s) you wish to achieve with each question. Think carefully about how students will respond to the question.

- Are they likely to just turn off and wait until the "real" classwork starts back up? If so,

can you ask the question differently or do something in class that short-circuits that reaction?

- How much time is necessary for them to formulate a reasonable response?
- Is the question clear and unambiguous?
- Is the question too easy or difficult?
- Will students be adequately prepared when the question is asked?

Additionally, consider using non-oral questions. Placing questions on a transparency or handout will demonstrate that you consider them important. Doing so may also communicate to students that you expect them to spend some time on the question while at the same time encouraging you to wait until students have had time to process it. Many students have commented that revisiting questions asked in class an effective way to prepare for examinations since they focus on the important skills and concepts of the course.

What you do during class can affect the success of your plans. When you ask questions, allow students a chance to respond. If students don't respond, wait. If students still don't respond, wait! Eventually, they will respond (if not in today's class, then in tomorrow's). Also, after a student response, wait and think. We find that our first impulse is often less useful than we would have liked. Consider what the student might have been thinking and whether and how you might follow up on the response to enhance the learning of both that individual and other students. If nothing else, when you pause, the students will think you are taking the response seriously.

Be careful how you respond to student answers. You want to foster an atmosphere where students do not feel threatened by answering the questions. Even comments like "that's not quite on the mark, Bob" can be enough to make students hesitant to respond to questions. Since we tend to have groups answering a question, we might simply ask what another group thought.

However, it is important that the correct answer is identified as such.

Finally, it is important to spend time after class reflecting on what happened. (Schon, 1983) We often find this hard to do. But, it is necessary, we believe, in order to achieve success at changing our teaching behavior. The up-front planning is quite important, but will be mostly wasted if we do not take time to analyze how well the plans worked. In essence, the reflection assesses how well reality matched the plans and, if so, whether the desired outcomes were achieved. Did we actually follow our plans? If not, is that good or bad? Did the students behave or respond as anticipated? Does the planned questioning appear to achieve the desired results? If not, what other questioning or activity might be better? The goal of the reflection is to make us aware of what we do. We suggest a brief reflection time, perhaps keeping a journal or annotating the lesson plan. Of course this data will need to be fed back into the planning process of the next iteration of the course and indirectly for future lessons in the current and other courses.

4. Sample Computer Architecture Questions

In the discussion below, we provide some examples of questions or class activities. Along with the examples we provide some discussion of our intended goals and of the processes we experienced or expected with the questions. We do not limit ourselves to positive examples. It seems useful to supply some examples of not so good questions so that others might learn from our mistakes.

4.1 Knowledge Recall Questions

Knowledge recall questions are relatively easy to ask. Often, however, they do little to enhance instruction. The following questions are probably not particularly helpful, even though they exactly address what we want to know.

- What did you learn in this chapter?
- What are the main points in the reading?
- Do you have questions over the chapter/section?

A small set of quick-check kinds of questions, however, might be useful. They could provide examples of some types of test questions as well as a review of important points in the content. For example:

- What is a cache?
- What is the purpose of the (shift left logical) "SHL" assembly language instruction?
- What is an operating system?
- How is bus skew handled in the PCI protocol?

Even though these questions do have some utility, we are inclined to believe they should probably be subsumed into the next category of question in which skills are practiced.

4.2 Skill Demonstration Questions

Many relatively simple skills such as converting from a decimal number to binary, or using a newly introduced assembly language instruction are often just demonstrated by professors with the assumption that students have mastered the skill since they did not ask any questions about it. Worse yet, students might fool themselves into thinking they have mastered the skill too. Life would be much easier if we could learn to swim by watching someone swim. Demonstrations of even the simplest skills by the professor should be followed up by practice questions for the students. The development of skill requires practice, and feedback as to the correctness of practice. Some examples here are:

- Converting between base 10, 2, and 16.
- Addition of two binary numbers
- Trace the assembly language program containing the newly introduced (shift left logical) "SHL" to showing the resulting register values.
- Use the newly introduced (shift left logical) "SHL" assemble language instruction to calculate....
- Draw the timing diagram for the code segment on the given pipelined processor.

- If the given cache is direct-mapped, what would be the format (tag bits, cache line bits, block offset bits) of the address?
- What does the given assembly language code "do"? Similar in nature to tracing, this question requires students to abstract from code to a general statement of code purpose. Tracing is necessary for understanding a program and, we believe, skill at abstraction is necessary for coding skill to progress to design skill.
- Using the given hit ratio and access times for the cache and memory, calculate the effective memory access time.

Other courses have similar examples of relatively low-level skills necessary for competence in the subject—various proof techniques in discrete structures, using syntax diagrams to see if a block of code is syntactically correct, and counting statements in algorithms.

4.3 Questions Drawing on Personal Experience

Questions asking students to draw on their past experiences can often be used instead of asking a more direct, but too complex or abstract, question. For example in Computer Architecture, when discussing immediate-addressing modes with respect to instruction-set-design issues, you might be tempted to ask the question: "How many bits should be used for an immediate operand?" It is more constructive to make the question more concrete by asking students to draw on past experiences by asking questions like the following:

- From your programming experience, what range of integer values would cover 90% of the constant integer values used in all the programs you have ever written?
- How many binary bits would you need to represent this range of values?

The sequence of questions focuses the discussion on the sought after answer.

Questions requiring students to examine their own knowledge and understanding can often be

used to motivate a deeper understanding of a topic, but the instructor must be careful that the intended point is made by the activity. To motivate hardware support for operating systems in a Computer Architecture course, I often ask the following sequence of questions:

- What is an operating system (hardware/software, goals, functionality)?
- How does OS/hardware protect against a user program that is stuck in an infinite loop?

The first question motivates the students to think about operating systems and their role. They usually decide that an operating system is software used to provide services such as security, file access, printer access, etc. On the second question, students typically answer that the system allows users to break/interrupt a program after a while. Having good oral questions to follow up on student answers is important. Asking about "what happens in a batch system?" steers the discussion back toward the desired answer of a "CPU timer". Other times students respond to the second question with answers like "the operating system will be watching for infinite loops." The instructor might follow up with a question like, "In a single CPU system, how many programs can be executing at once?" If the students answers "one", then you might ask, "If the user program with the infinite loop is running, then how can the operating system (which we decided was a program) be running too?" This gets the discussion back to the need for the CPU-timer hardware support.

4.4 Questions to Create Cognitive Dissonance

An Earth Science colleague once told me that students in his crystallography course did not have preconceptions about the content in his course. He was wrong. Students may come to us with little knowledge and incorrect assumptions about word usage and meaning, but they will always have some preconceptions about our content. Often the preconceptions will be inaccurate and hard to replace. Identifying

and attempting to fix them and to short-circuit the establishment of new misconceptions are critical aspects of teaching. The strongest learning occurs when we are able to produce cognitive dissonance in student minds. We need this kind of learning to alter misconceptions— weaker techniques will not work. Additionally, it would be nice if we were able to generate such a mindset at will. Probably we cannot, but we can try.

The last example from the previous subsection is a good example of creating cognitive dissonance in student minds. By asking the question "If the user program with the infinite loop is running, then how can the operating system (which we decided was a program) be running too?"

Along the same lines, other questions that can create cognitive dissonance when teaching about hardware support for operating systems would be:

- Since a user's program needs to be allowed to perform disk I/O, how does the OS/hardware prevent a user program from accessing files of other user?
- Since a user program needs to be able to perform memory accesses, how does the OS/hardware prevent a user program from accessing (RAM) memory of other user programs or the OS?

4.5 Questions to Motivate a Topic

Before discussing a new topic it is often useful to ask a question related to the topic to get students curious. Alternatively, it is sometime useful to ask a question about a topic's prerequisite knowledge. This kind of question is an advance organizer and should serve to establish cognitive hooks into students' past experience. For example, before taking about parameter passing in assembly-language ask questions about how students view the run-time stack in their most familiar high-level language.

Clearly, our lists of questions are incomplete. Space concerns make that necessary. So too

does our level of progress. Frankly, we have only begun the work necessary to become better questioners (and, thus, better teachers). Many more examples of Computer Architecture questions can be found on-line at Fienup (2001).

5. Conclusions

Our most significant insight is that asking good questions takes work. We had to (and may still need to) read about questioning and apply what we read to teaching Computer Architecture. Additionally, relatively significant planning is necessary. In essence, we need to plan for questions, much as we plan for lecture.

We are still convinced that doing the extra work pays off. We think student learning has improved, i.e., more students are learning more of the material at a level we think is good. Additionally, we believe the "extra" work in planning will lessen, and perhaps disappear. As we learn more and practice questioning (and planning for it), the time requirements will be less. Also, as questioning becomes a bigger part of our teaching, the planning of telling is replaced by planning for questioning.

Should you decide to include more questioning in your teaching, we have some advice beyond that of reading and planning. Reflect on your questioning behavior. Explicate your goals and plans before teaching. After teaching, reflect on how well you implemented your plans and on how well the questioning worked. Then introduce those conclusions into your future planning. (This may require some record keeping.) Finally, do not expect perfection. Like all other human endeavors, you will get better with practice, particularly with good (reflective) practice.

6. References

Chuska, K. R. (1995) Improving classroom questions. Bloomington, IN: Phi Delta Kappa.

- Dantonio, M & Beisenherz, P.C. (2001) Learning to question, Questioning to learn. Boston: Allyn and Bacon.
- East, J. P. (2001) Experience with In-person Grading. Proceedings of the 34nd Midwest Instruction and Computing Symposium, April 5-7, 2001. Cedar Falls, IA.
- Felder, R. & Brent, R. (1996). Navigating the bumpy road to student-centered instruction. College Teaching, 44, 43-47.
- Fienup, M. (2000) Active and group learning in the Computer Architecture classroom, Proceedings of the Workshop on Computer Architecture Education, June 2000, Vancouver, B.C., Canada.
- Fienup, M. (2001) Fall 2001 Computer Architecture course home page. <http://www.cs.uni.edu/~fienup/cs142f01/in-class-materials>.
- Frederick, P. (1986). The lively lecture - 8 variations. College Teaching, 34, 43-50.
- McConnell, J. (1996). Active learning and its use in Computer Science. SIGCSE Bulletin, 28, 52-54.
- Schon, D. A. (1983). The reflective practitioner: How professional think in action. New York: Basic Books.
- Silberman, M. (1996). Active learning: 101 strategies to teach any subject. Boston: Allyn & Bacon.
- Stuart, J. & Rutherford, R. J. (1978, September 2). Medical student concentration during lectures. The Lancet, 514-516.
- Wasserman, S. (1993) Asking the right question: The essence of Teaching. Bloomington, IN: Phi Delta Kappa.
- Wilten, W. W. (1991) Questioning Skills, for Teachers. Washington, D.C.: National Education Association.

An Active Learning Environment for Intermediate Computer Architecture Courses

Jayantha Herath, Sarnath Ramnath, Ajantha Herath*, Susantha Herath

St. Cloud State University, St. Cloud, MN 56301

*Marycrest International University, Davenport, IA 52807

herath@stcloudstate.edu

<http://web.stcloudstate.edu/jherath/CompArch-2>

Abstract

Most computer science, information systems and engineering programs have two or more computer architecture courses but lack suitable active learning and design experience in the classroom. Computer architecture at the intermediate level should focus on the implementation of basic programming constructs in different instruction set architectures. To accommodate such features we developed an undergraduate computer architecture course with hands-on classroom activities, laboratories and web based assignments. To assess the course we distributed the course modules among 200 computer architecture instructors. This paper describes our experience in developing active learning course modules.

1. Introduction

During last fifteen years, we have been experimenting with methods to improve the quality and efficiency of teaching computer architecture courses for undergraduate computer science and engineering students. Our goal has been and continues to be to help them become good computer scientists in a relatively short period of time with both theoretical understanding and practical skills so that they can enter and make an effective contribution to the profession. Traditionally, computer architecture subject matter has been presented to a less than enthusiastic student body in a relatively passive classroom environment. In general, this chalk-

talk instructional process consists of multiple copying stages: the instructor first copies notes from a textbook to his note book, then the instructor copies those notes onto the blackboard, thereafter the students copy notes into their note books. Moreover, each instructor allocates considerable chunk of his/her time to prepare or update the same course material in each offering. In addition, there is both local and national need for high-quality trained labor with the ability to stay current with the technological advances in the computer architecture field .

Growth of any undergraduate computer science or engineering program will largely depend on the strength of the computer architecture curriculum. To address the deficiencies in traditional curriculum [4-10] and to satisfy the current needs, we redesigned our computer architecture course sequence with fundamentals to incorporate rapidly changing computer related technologies so that our graduates will be current with the technologies before they graduate. It is hypothesized that the learning rate can be increased if both the instructor and the student are active at the same time. Thus the performance of the students can be improved dramatically by converting the traditional passive classroom into an active hands-on learning environment. Designing a course with learning-by-doing modules and making it available for all the

instructors on-line [1] reduces the course preparation time for instructors, reduces multiple copying steps in the learning process, strengthen the abilities and increase the enthusiasm of both traditional undergraduate students as well as the adult learners.

Goals and Objectives

The main objective of this project was to develop computer architecture course modules for intermediate level undergraduate students and the faculty. These active learning modules are central to achieve the following goals:

- To provide the students an efficient, rigorous and engaging learning environment with necessary tools and training to become proficient in the computer architecture subject matter in a relatively short period of time.
- To provide architectural details necessary to implement basic programming constructs learned in CS-1 and CS-2 with hands-on skills, integration, team-work and hence to enhance the quality of the graduates.
- To use performance focused learning at all levels of curriculum to illustrate the principles of computer architecture.
- To provide the faculty and students modifiable on-line courseware with state-of-the-art hardware and software practice.

Following sections outline the details of course plan, goals achieved, difficulties encountered, assessment plan future work and summary.

2. Detailed Course Plan

The course, outlined below, will address the ways of reducing the deficiencies in the existing curriculum [4-10]. When developing and delivering the computer architecture subject matter for computer science majors, we believe that the prime factor to be focused on in any step is processor performance in implementing programming language constructs. Our curriculum consists of three semester courses to help master the computer architecture subject matter in a technology integrated classroom laboratory. First course of this sequence will cover fundamentals of architectural design [11-13]. The laboratories for this course consist of hardware and software simulations of combinational and sequential digital circuits. This foundation will help to develop the skills from gate level to register transfer level component integration in design. The intermediate level course that we designed introduce both complex instruction set and reduced instruction set processor architectures, instruction set manipulations with I/O, memory, registers, control and procedures [1-2][14-16] to the students. The laboratories for this course consist of hardware and software simulations of programming constructs in CISC and RISC architectures. After completion of intermediate course the students will be able to learn architectural details of any other processor. The third course is focusing on the advanced concepts in architecture involving parallel/distributed computations and special purpose architectures to provide both depth and breadth to the subject matter. Parallel processing and special purpose processing concepts in the undergraduate curriculum has been the focus of several curriculum improvement efforts for some time [3][17-18]. The students should be able to understand the importance of parallelism in enhancing performance and its benefits as an application programmer, a systems programmer, an

algorithm designer, and a computer architect. A course sequence with the features outlined above could help our students develop design skills in several different architectures before their graduation. The undergraduate curriculum, graduate programs and industry will definitely appreciate the graduates with such design skills.

Topics for Computer Architecture II

At the intermediate level we introduced processor design and focused on the implementation techniques of basic programming constructs such as I/O, arithmetic expressions, memory operations, register operations, if-else-for-while control and functions in several different instruction set processor architectures. Two complex instruction sets and one reduced instruction set processor architectures were introduced in our course. Students learned that proper instruction set design, memory management and I/O management techniques will also lead to the performance enhancement. Increasing performance of the processor by reducing the program execution time is considered at each design and implementation. Focusing on the importance of performance when designing the processor helped to maintain the momentum and enthusiasm in the classroom. Often the students were excited to observe the register level manipulations in the processors. They also enjoyed discovering the processor and controller designs in an active classroom. Comparing different architectures including pipeline techniques and abstracting the essentials of the processor architectures at this level generated the required enthusiasm to the learning and teaching. The required textbook for this course is Paterson and Hennessey [2]. We are looking for ways to integrate rapid prototyping of the systems to the course using web based tools.

Hardware/Software Laboratories

To provide architectural concepts with hands-on skills, integration, team-work and hence to enhance the quality of the graduates, we added pre-lab, in-lab and post-lab assignments to complement the classroom activities. Table 2 summarizes the educational experience gained from these laboratories.

Table 2. Educational experience

Experience	Level	Application
Prelab	Analysis, synthesis	Design circuits and programs to perform a specific simple task
Closed Labs	Application, analysis, synthesis, evaluation	Design, implement and test circuits and programs to perform a specific task within a given period of time
Open Labs	Application, Analysis, synthesis, evaluation	Design circuits and programs to perform a difficult task
In-class activities	Application, Analysis, synthesis	Cost reduction, performance improvement, integration
Tests	Analysis, synthesis	Architecture design related questions

To reflect student-centered design and analysis processes, classroom activities were modified to accommodate skills in performance improvement and cost reduction when designing processors. In general, pre laboratory assignments helped the students explore and create on their own. They synthesized the classroom instructions with other resources to produce hardware and software and then to test and to debug. In the classroom, each student provided with a computer and tool kit to extend the concepts they learned in the pre lab assignment. Less challenging design problems

that can be solved within a given period of time were assigned as in-class closed-laboratory assignments. A post-lab assignment helped the students to analyze the use of in-class activities. More challenging and time consuming problems were assigned as post laboratories. Students were active in both laboratory and in the classroom while thinking and experimenting on a machine with the architectural concepts introduced in the classroom. After completing each project, students submitted a report discussing their experience. First, each student worked alone as the sole developer of the hardware and software. Towards the end of the semester two to four students were allowed to work in a team to design, construct and complete the projects. The group was responsible for developing definitions and specification of a larger hardware/software product to solve a problem as their final project. The course helped students become proficient in the subject matter in a relatively short period of time.

3. Goals Achieved

We created the active learning course material that will enhance students' high level skills: teamwork, analysis, synthesis, performance comparison and active participation in the classroom. To reflect the inclusion of several different instruction set architectures we created hands-on hardware and software laboratory assignments. Our computer science students received the instructions based on the course material developed in Spring 2002 and Fall 2001.

Our active learning course modules enabled students to learn architectural concepts more effectively and efficiently thus providing students an opportunity to function well in an increasingly competitive technical society. The classroom activities provided the students with opportunities for analysis, synthesis, and verification of correctness in building larger systems all

during traditional class time. Such modifications increased the enthusiasm in the classroom, addressed the needs of both traditional undergraduates and adult students, the needs of the industry and provided necessary tools and training for the student to become proficient in the computer architecture subject matter in a relatively short period of time. To our knowledge, no other computer architecture course used our approach. Therefore, our course modules and experimental results will be very useful for the other computer science and engineering programs nationally. Table 3 depicts the Indicators/Measurements of goal attainment of all three courses.

Table 3. Indicators/Measurements of goal attainment

Entry level	Intermediate level	Goal Attainment
Gate level design and analysis	Design, analysis and performance improvement of architectural components, processors, controllers	Parallel processing, system design, analysis and performance improvements
Exams	Exams	Exams

Architecture Symposium

A computer architecture symposium [21-22][24] was organized at the end of the Spring'02 semester to stimulate our undergraduate and graduate students, computer science and engineering faculty in tri-state and the local industry. We invited five excellent speakers from MIT, University of Minnesota, IBM T.J. Watson center and Oracle to deliver lectures based on their work. The symposium was well attended by the students, faculty and industry. Spring'02 semester started with introduction seven trillion FLOPS machine, then the students learned about 35 trillion FLOPS machine. At the end of the semester in the symposium students learned about the 185 trillion FLOPS machine

under development at IBM. This conference also helped our efforts to develop a core curriculum for Computer Science that presents an integrated view of hardware and software to the undergraduate students.

Difficulties

Incorporating several architectures into one course seemed overloading the students and faculty at the beginning. However, making our course modules available for the students at the beginning of the semester via web helped to eliminate this difficulty. Selecting a series of projects that increases enthusiasm in a diverse body of students was also a difficulty we encountered. Observing, helping and verifying the correctness of weekly work focusing on the analysis and synthesis of components was a time-consuming task. Trained student assistants helped in scheduling the laboratories and reduced the burden. However, attracting suitable student assistants and paying them sufficiently to keep them was also another difficulty we faced. Identifying suitable modern educational circuit boards for our experiments was another difficulty we faced.

St. Cloud State University, with six colleges, is the second largest university in Minnesota. The university enrollment is approximately 15,000 students drawn from MN, rest of the USA and foreign countries. The computer science department, among the 10 departments of College of Science and Engineering, is one of the two CSAB accredited departments in MN. The department consists of 180 undergraduate major students, 30 graduate students and 10 full time faculty members. We have two departmental laboratories with 50 PCs for introductory programming classes, architecture and operating systems. Most of the graduates enter industry or graduate school after graduation. Computer science department offered the

computer architecture I course twice during the academic year 2002/03 for about ninety students. Computer architecture II course is offered twice a year during the academic year 2002/03 for about sixty students. Computer architecture III course is offered once during the academic year 2002/03 for about forty students. We graduated 25 students this year.

4. Course Assessment

The course material developed was evaluated by soliciting the criticism from the faculty and students. Student learning was evaluated using many different ways. The background knowledge and preconception checks were performed in the form of a simple questionnaire/worksheet that the students will fill in prior to working on the lab assignments. The students were asked to explain the concepts they have learned so that the instructor can measure student learning. Faculty and teaching assistants regularly observed the team work. Recording experiences from laboratory assignments was an essential part of the student work. Student groups submitted weekly project reports. Group-work evaluations were also used to assess the course. In the larger lab projects, students worked together in groups. Each member turned in an evaluation of his/her own learning experiences gained by being part of a team. To reinforce the learning, a test was scheduled after the completion of each module. Excellent students performed well in all levels and had complete understanding of the subject matter. Very good students were strong in many areas but weak in some. Average students showed weaknesses in some levels. Poor students could not perform well in many areas. Classroom opinion polls and course-related self confidence surveys were also performed to receive the feedback. In the future, comments from the industrial advisory committee and accreditation board member's

site visit and reviews from other instructors will be used to evaluate the project performance. Within our large university system we will have opportunities to test our designs which could possibly extend to other faculty and students. We are currently in contact with many computer architecture instructors to find ways to improve the courses we teach.

Dissemination of Course Modules Among Instructors

To disseminate the findings of this project, laboratory manuals, course notes and other related information, the web is heavily used. Before the start of Spring'02 semester, we contacted approximately 600 computer science departments using our distribution list and informed the availability of our course modules for their classroom use and review with no charge. More than 200 computer architecture instructors requested the course modules. We distributed our lecture notes among them via e-mail. A better version of our course material is now available to others for classroom use [19-20]. It is important to note that we have successfully completed the introduction to computer architecture project earlier and distributed the course material to more than 200 instructors. We will continue assessing the course material through faculty and student feedback for next few semesters. We will continue to share the experience gained from this experiment with the rest of the computer architecture community. Progress of this project will be reported to the MnSCU Center for Teaching and Learning.

5. Summary and Future Work

Traditionally, computer architecture courses are presented, with complexity and confusion, to a less than enthusiastic student body and often delivered in a relatively passive classroom environment. In general, learning takes place if both the instructor and the

student are active at the same time. To promote this in the classroom and to overcome the above mentioned deficiency, we developed an intermediate computer architecture course with hands-on classroom activities, laboratories and web based tools and distributed among many computer architecture instructors. Other deficiencies encountered in the traditional learning environment such as instructor's preparation time and multiple copying stages involved in the learning process were also addressed. Availability of properly designed and developed on-line course materials, with a series of hands-on laboratories as well as classroom activities will definitely reduce both instructors' preparation time and multiple copying stages, and increase student learning rate. Such on-line courses could help both traditional students and adult learners to explore the computer architecture area while developing their design and analysis skills. Modifiability and flexibility of course material at the instructor's end will contribute very much to the faculty development. Often, the students are confused because of not having a well-defined focus in the classroom activities. This computer architecture course is designed to complement the activities performed in CS-1, CS-2 and computer architecture-1 courses. The subject matter provides the gateway for advanced studies in computer architecture and other areas. The course helped to understand the implementation details of basic programming constructs in CISC and RISC architectures. Performance issue is considered in all alternative designs. This courseware helped students to be active in the classroom and increased the enthusiasm in learning computer architectures. Hardware description programming experience allows description of the structure, specification using a familiar programming language and simulation before being manufactured. As a result, students as designers can quickly compare alternatives

for high performance and test for correctness. We are planning to use a industry-standard hardware description programming language [23] in both first and second level courses. Developing a clustered computing environment will be useful for the laboratories in the third course of the sequence. Educational circuit boards with several processors that communicate with each other through dedicated channels will be a good alternative for the advanced course. Virtual environments with variety of visualization systems are matured enough to aid students' understanding of miniaturized complex processor architecture. Through such platforms students will learn to appreciate the instruction set architecture. In the future revisions we will explore the feasibility of incorporating such virtual environments in the computer architecture classroom [7][8] and then improving upon them in successive iterations.

Acknowledgments

This project has been supported by the MnSCU Center for Teaching and Learning through the Bush/MnSCU Learning by Doing Program.

6. References

1. A Web-Based Computer Architecture Course Database, Edward F. Gehringer
<http://www.csc.ncsu.edu/eos/users/e/efg/archdb/FI/E/2000CACDPaper.pdf>
2. Computer Organization and Design: Hardware/Software Interface, Second Edition, John L. Hennessy and David A. Patterson, 1997
<http://www.mkp.com>
3. Computer Architecture: A Quantitative approach, Third Edition, John L. Hennessy and David A. Patterson, 2002 <http://www.mkp.com>
4. The Undergraduate Curriculum in Computer Architecture, Alan Clements,
<http://www.computer.org/micro/mi2000/m3toc.pdf>
5. Teaching Design in a Computer Architecture Course, Daniel C. Hyde,
<http://www.computer.org/micro/mi2000/m3toc.pdf>
6. Rapid Prototyping Using Field-Programmable Logic Devices, James O. Hamblen,
<http://www.computer.org/micro/mi2000/m3toc.pdf>
7. PUNCH: Web Portal for Running Tools Nirav H. Kapadia, Renato J. Figueiredo, and José A.B. Fortes,
<http://www.computer.org/micro/mi2000/m3toc.pdf>
8. Building Real Computer Systems Augustus K. Uht, Jien-Chung Lo, Ying Sun, James C. Daly, and James Kowalski,
<http://www.computer.org/micro/mi2000/m3toc.pdf>
9. HASE DLX Simulation Model Roland N. Ibbett,
<http://www.computer.org/micro/mi2000/m3toc.pdf>
10. An Integrated Environment for Teaching Computer Architecture Jovan Djordjevic, Aleksandar Milenkovic, and Nenad Grbanovic,
<http://www.computer.org/micro/mi2000/m3toc.pdf>
11. Digital Design, 3/e 2002 Morris Mano,
<http://prenhall.com>
12. Digital Design: Principles and Practices, Updated Edition, 3/e 2001 John Wakerly,
<http://prenhall.com>
13. Digital Design Essentials and Xilinx 2.1 Package, 1/e 2002 Richard Sandige,
<http://prenhall.com>
14. Computer Systems Organization and Architecture John Carpinelli (2001), <http://awl.com>
15. COMPUTER ORGANIZATION, Fifth Edition V. Carl Hamacher, Zvonko Vranesic, Safwat Zakay,
<http://mhhe.com>
16. Computer Systems Design and Architecture, 1/e 1997 Vincent Heuring , Harry Jordan,
<http://prenhall.com>
17. Parallel Computer Architecture: A Hardware/Software Approach David Culler and J.P. Singh with Anoop Gupta, August 1998
<http://www.mkp.com>
18. Readings in Computer Architecture, Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, September 1999, <http://www.mkp.com>
19. Computer Architecture I Preliminary version
<http://web.stcloudstate.edu/jherath/CompArch-1>
20. Computer Architecture II Preliminary version
<http://web.stcloudstate.edu/jherath/CompArch-2>
21. Hardware/Software Interfacing for High Performance Symposium -02
<http://web.stcloudstate.edu/jherath/Conference.htm>
22. The RAW Microprocessor, M.B. Taylor etal, MICRO 2002March
http://dlib2.computer.org/mi/books/mi2002/pdf/m2_025.pdf
23. VHDL Primer, A, 3/e 1999 Jayaram Bhasker,
<http://prenhall.com>
24. VLSI Digital Signal Processing Systems: Design and Implementation, K. K. Parhi, 1999,
<http://wiley.com>

Effective Support of Simulation in Computer Architecture Instruction

Christopher T. Weaver, Eric Larson, Todd Austin
Advanced Computer Architecture Laboratory, University of Michigan
{chriswea, larsone, austin}@eecs.umich.edu

Abstract

The use of simulation is well established in academic and industry research as a means of evaluating architecture trade-offs. The large code base, complex architectural models, and numerous configurations of these simulators can consternate those just learning computer architecture. Even those experienced with computer architecture, may have trouble adapting a simulator to their needs, due to the code complexity and simulation method. In this paper we present tools we have developed to make simulation more accessible in the classroom by aiding the process of launching simulations, interpreting results and developing new architectural models.

1 Introduction

The use of simulation tools in computer engineering is essential due to the time overhead and cost of manufacturing prototypes. To better prepare the student, we and many others have integrated the use of architectural simulation tools into our computer organization curriculum. However, detailed simulators can be very daunting to the beginner, as they typically possess hundreds of options and thousands of lines of code. In this paper we discuss how simulators can be made more approachable to both students who are learning the fundamentals of computer architecture and those that are investigating a particular issue in the field.

In our introductory courses, users who are learning the fundamentals are more concerned with running simulations, than understanding or modifying its implementation. We have found the best way to aid novice students, is to provide tools that have a simple interface and an output that allows them to clearly see what is going on. We present

two graphical tools (SS-GUI and GPV) and a backend perl script that decrease the complexity of using architectural simulators.

In our more advanced courses, we often ask our students to add performance enhancing features to a microarchitectural simulator. We have found the students are best served by a simulator that is modular and simple to alter. In addition, they require a verification method to ensure their changes do not break the simulator. If bugs are detected the infrastructure should have methods to expedite the detection and correction of the error. We present the features of the Micro Architectural Simulator Environment (MASE) that make it ideally suited for class projects.

The rest of the paper is structured as follows. First we discuss the tools (SS-GUI and perl script backend) that we have developed that simplify the running of a simulation. Next we talk about the graphical pipetrace viewer (GPV) which simplifies the simulation analysis process. We then focus on MASE, which aids more advanced students in developing new architectural models. Finally, we give some concluding remarks on these tools and their use in education.

2 Launching Simulations

SS-GUI, shown in Figure 1, is a user-interface form that contains all of the fields necessary to launch a simulation. The save and load options make it possible for an instructor to setup a template for the class to use as the basis of their simulations. Presently the environment is customized to the SimpleScalar toolset [3], however the only non-generic field is the simulator options field. These fields are constructed by parsing a global configuration file that specifies the options available for the simulator. Additional features of the

GUI are enumerated below with corresponding marks on Figure 1.

1. File options- This menu allows for the loading and saving of the GUI form contents. This allows the system admin or class instructor to fill in a base line form that the student can load and alter.
2. Setting Menu- This menu bring up prompts for the form comments.
3. Simulation Settings- This section contains all the paths to the necessary components to run a simulation. This can be classified as three different types of data: configuration of the simulator, run setup, and benchmark specification. The configuration of the simulator requires the user to supply the path to the actual simulator and any configuration file to use. The run setup requires the user to supply

the path of the backend run script (talked about in the next paragraph), where to run the simulation, where to store the results and how to tag the results for later inspection. Finally, the user must supply the benchmark to execute, the path to the executable and type/path of the input set to use.

4. Benchmark Selection window- The user has the option to select the benchmark from a list or type the benchmark and its options in manually. The pop-up window contains information about each of the different benchmarks that are supported (currently spec2000, spec95 and a few others). A global benchmark configuration file specifies how to run the experiments.
5. Simulator Option Scroll Window- This window contains all of the simulator options that

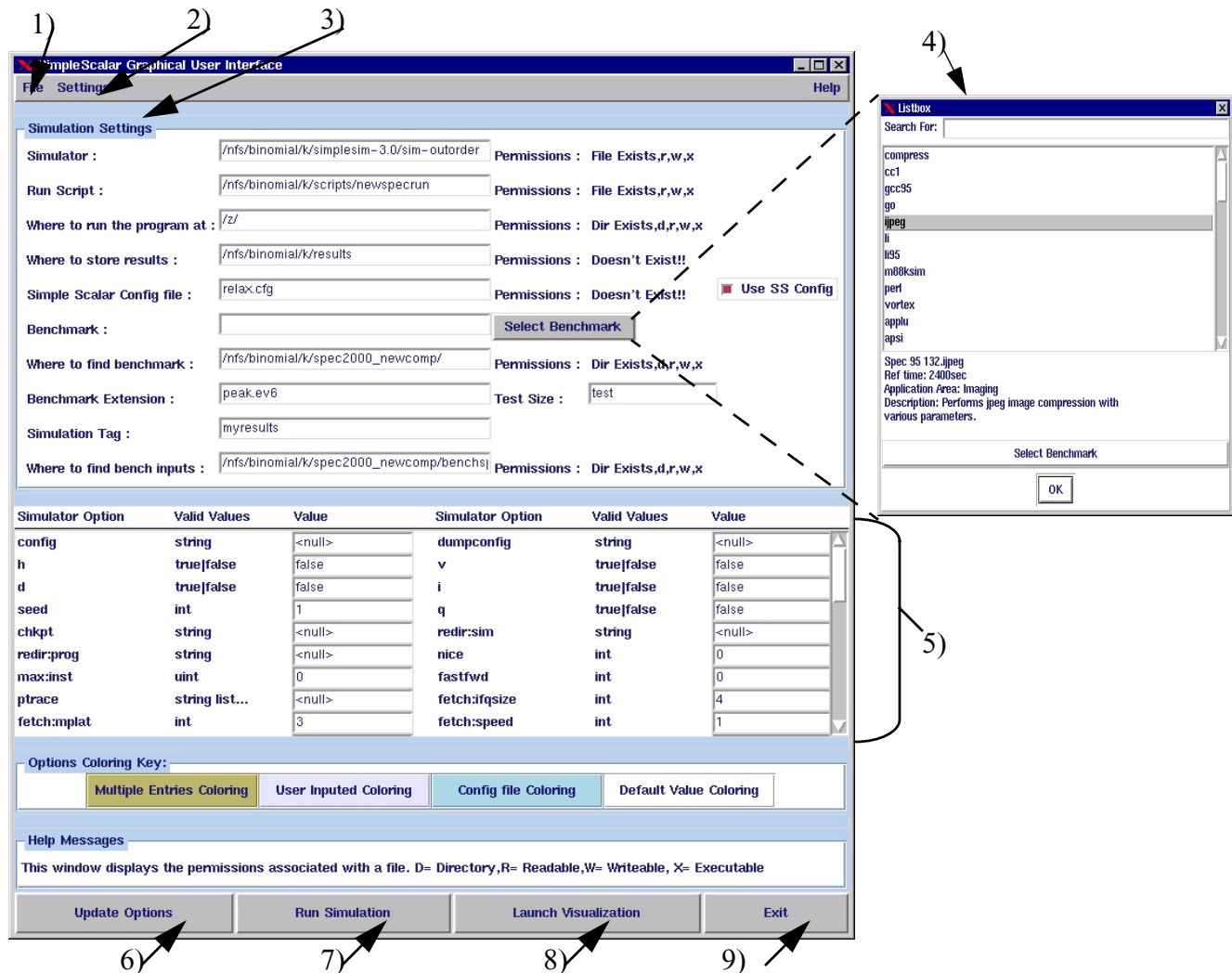


Figure 1: SS-GUI - a frontend for running simulations

are available for the current simulator. If a configuration file is specified, the options will display this value. The entries can also be modified by the user. A color guide is used to illustrate whether the value is the default, specified in the config file, entered by the user or contains multiple entries. The multiple entry fields are reserved for future usage, where the GUI can be used to generate test queues for a variety of simulator options.

6. Update Options Button- This button will run the simulator without any arguments, so that the available options are reported. The reported options are then parsed and reloaded into the Simulator Option Scroll Window.
7. Run Simulation Button- This button will run the backend perl script with the options setup in the GUI form.
8. Launch Visualization Button- The launch visualization option will run the backend perl script with a flag that causes the output to be streamed into GPV (described in the next section).
9. Exit- Exit the GUI environment.

The backend perl script contains a variety of features, however its basic function is to copy all of the simulation files to a experiment directory, launch the simulation, and copy back the results. The script contains all of the arguments need to launch the supported benchmarks (currently spec2000, spec95 and a few other benchmarks). The run script can optionally check that the simulator gave the correct output. The logs generated by the script expedite the diagnosis of run failures.

3 Interpreting Results

Figure 2 gives an overview of GPV, our pipeline viewer. An architectural simulator is used to produce a pipetrace stream. This stream contains a detailed description of the instruction flow through the machine, documenting the movement of instructions in the pipeline from “birth” to “death”. In addition, the pipetrace stream denotes various other events and stages transitions that occur during an instruction’s lifetime.

The pipetrace stream from the architectural simulator can be sent directly into GPV or buffered in a file for later analysis. GPV digests this information and produces a graphical representa-

tion of the data. The graph generated by GPV plots instructions in program order, denoting over the lifetime of an instruction what operation it was performing or why it was stalled. In addition, the tool is able to plot any other numeric statistics on a resource graph.

Multiple traces can be displayed on the screen at any given time for easy analysis. GPV also supports both coarse and fine grain analysis through the use of a zoom function. Color coded events, which are user definable, makes spotting potential bottlenecks a simple task. The remainder of this section will outline the tool in detail, including the main view, advanced features, trace file format, and other infrastructure with which GPV has been designed to communicate.

3.1 Main Visualization Window

The main GUI window of GPV is illustrated in Figure 2. The GUI has two main graphical display windows, the instruction window and the resource window. The instruction window plots instructions in program order on a time axis (measured in cycles). For example, the third instruction bar in Figure 2, shows the execution of an ADDQ instruction on a 4-wide Alpha simulator. As shown in the figure, this instruction is stalled in Fetch (IF) until the stall in the internal ld/st is resolved, after which it continues to completion.

This method for graphing instructions as they flow through a pipeline is a common visual representation, used in many textbooks including Hennessy and Patterson [6]. The instruction axis contains tick marks to indicate the cycle count. Additionally, the vertical axis will also display the instruction mnemonic when the window is zoomed in enough to fit legible text aside each instruction mark (typically two zooms from when the pipetrace is first loaded).

The right panel provides a legend of the coloring that is used to illustrate the instruction’s flow through the different stages of the pipeline. Significant events, such as branch mispredictions or cache misses, are displayed in conjunction with the instruction’s transitions through the pipeline. The use of color (with a user configurable palette) provides an effective means for spotting potential bottlenecks. A highlight option, which can flash the occurrences of a particular event, can be used as an alternative method of locating bottlenecks.

The bottom window, the resource view, displays graphs of any numeric statistic provided in the pipetrace file. GPV has been designed to plot both integer and real statistics. Up to four data sets (our current development extends this to ten) can be displayed simultaneously with color coded axes that indicate the range of the variable. Since there can be a wide variation in the data range of a statistic, a separate x-axis is provided for each one of the four resources that can be displayed at a time. Both the resource and instruction views are plotted against simulator time on the x-axis. This permits widely varying statistical data sets to be plotted within the same window. To avoid clutter, the GUI allows the selective hiding of individual resource views.

The resource view in Figure 2 is shown plotting the IPC of a simulated program. As shown in the figure, the IPC of the program starts to drop during the cache miss. Once the miss has been handled and instructions start to retire, the IPC begins to recover. The flexibility of the resource view allows the user to choose the statistics that are most valuable for performance analysis and correlate these statistics to instructions flowing through the pipeline. This simplifies the task of identifying bottlenecks, as illustrated by the rela-

tionship of the cache miss to the IPC drop in Figure 2.

The GUI provides several additional features that assist in diagnosing performance bottlenecks. The display can be zoomed in and out to trade off detail for trend analysis. When the display is zoomed out it is straightforward to determine areas of low performance by locating pipeline trace regions with low slope. The slope of the line is given by ¹:

$$slope = \frac{\Delta y}{\Delta x} = -(IPC)$$

Thus for a perfect single wide pipeline (no data, control or resource hazards) with no multi-cycle stages the IPC would be 1 (slope of -1). The display will show the areas of low performance with a gradual (more horizontal) slope and areas of high performance with a steep (more vertical) slope.

GPV also allows users to select instructions for more information. Selecting an individual

1. The negative sign is because instruction progress in the negative y direction.



Figure 2: GPV Display Window. This example shows the execution of instructions on a 4-wide Alpha ISA model. (Note: Internal micro-code operations, *i.e.* internal ld/st, are allowed to finish out of program order.)

instruction displays the cycle time of execution and the instruction mnemonic. This makes it possible to get information about single instructions when the pipeline display is too small to label each individual instruction. Similarly, the resource view allows resource graph lines to be selected, which returns the label, cycle number and instantaneous value. Since the resource graphs are displayed as continuous lines from discrete data in the pipetrace file, intermediate points are calculated by linear interpolation.

4 Developing New Models

MASE (Micro Architectural Simulation Environment) is a flexible performance infrastructure to model modern out-of-order microarchitectures. It is a novel performance modeling infrastructure that is built on top of the SimpleScalar toolset [3]. MASE is most appropriate for advanced computer architecture courses where students are adding enhancements to a baseline microarchitecture and analyzing their results. MASE simplifies this process by adding a dynamic checker that can detect implementation errors, modularizing the code base improving code readability and understanding, and adding support for optimizations that are difficult to implement. Additional information on MASE can be found in [7].

4.1 Dynamic checker

The dynamic checker is used to verify that any changes or enhancements to the simulator code are indeed correct. Since not all errors directly cause an error in the output, it provides extra security that a model enhancement did not violate any microarchitectural dependencies or program semantics. In most simulators, it is difficult to determine precisely where an error occurred when there is a difference in the output. The checker will pinpoint the first instruction where a mismatch occurs, greatly reducing debugging time.

The checker resides in the commit stage, monitoring all instructions that are committed. It compares values produced from the core to the correct value. The correct value is obtained by the use of an oracle in the fetch stage. The oracle is an in-order functional simulator that has its own architectural state and memory. The oracle data is passed to the checker using a queue. In addition to checking the output value, the checker will also

check (if appropriate) the PC, next PC, effective memory address, and any value written into memory. If the results match, the result will be committed to architectural state and the simulation will progress as normal. If the results do not match, an error message is printed out indicating the failing instruction along with the computed and expected values. The simulation may continue or be aborted depending on a user-controlled flag. If the simulation is allowed to continue, the oracle result will be committed to architectural state and a recovery will be initiated. The instruction with the bad result is allowed to commit (with its result corrected) in order to ensure forward progress. The remaining instructions in the pipeline are flushed and the front-end is redirected to the next instruction.

Our experience with the checker has been very positive, starting when we were implementing MASE itself. The first bug we found involved failing instructions that referred to Alpha register \$31 (the zero register). Almost immediately, we were able to determine that the processing of this special register was incorrect. Once that problem was flushed out, we noticed that most of the problems dealt with conditional move instructions and how the output was incorrectly zero most of the time. We concentrated our debugging efforts at the conditional move and quickly identified that when the conditional move was not executed, it was not handled properly.

The checker was also useful in implementing a blind load speculation case study¹. As one might expect, loads were the only instruction that failed so the error message provided by the checker did not provide as much insight as in the previous cases. Instead, we focused on the first error that was signalled. We used *gdb* to debug the simulator and set a breakpoint on the failing instruction. Once we arrived at the failing instruction, we analyzed the state of the machine at that time and were able to isolate the problem relatively quickly.

4.2 Modularized code

The MASE performance model has been divided into several files, summarized in Table 1. The rest of the SimpleScalar infrastructure is well

1. Loads are allowed to speculatively execute once their addresses are known regardless if earlier stores could overwrite the data the load is accessing [9].

Table 1: Description of MASE files

mase-checker.c	Oracle and checker.
mase-commit.c	Backend of the machine: writeback, commit, and some recovery routines
mase-debug.c	MASE-specific support for SimpleScalar’s DLite! debugger
mase-decode.h	Macros used for decoding an instruction
mase-exec.c	Core of the machine: issue and execute
mase-fe.c	Frontend of the machine: fetch and dispatch
mase-macros-exec.h	Execution macros for the execute stage
mase-macros-oracle.h	Execution macros for the oracle
mase-mem.c	Memory interface functions
mase-opts.c	File contains all MASE-related options and statistics
mase-structs.h	Common MASE data structures
mase.c	Initialization routines and main simulator loop

modularized with separate files for branch predictors, caches, and memory systems. This organization allows users to focus on the part of the simulator they plan to work on without requiring intimate knowledge of the other sections. It also allows different users to work on different files without having to worry about combining changes within in a single file later¹. It is straightforward to add enhancements since most of the new code can be placed in separate files usually requiring only slight modifications to the existing code.

Many of the features in MASE were added to make the model more realistic and representative of modern microarchitectures. A side effect of this is that it makes it easier for new users to understand how the provided code works. For example, one of the main obstacles to understanding how sim-outorder works is due to the fact that the core only simulated timing - there is no execute stage. The core of MASE executes instructions, allowing new users to track an instruction from fetch to commit without wondering where the execute stage is. To further improve readability, the execution and decoding macros have been placed into separate file, removing machine-dependent code from the bulk of the core.

1. sim-outorder.c is 4,692 lines long!

4.3 Modernized microarchitectural model

One of the goals of MASE is to modernize the baseline microarchitectural model, allowing for the creation for more accurate models. To accomplish this, we added support for several different types of optimizations or analyses that would be difficult to implement in the previous version of SimpleScalar. This section outlines some of the things we added.

A micro-functional core is added that executes instructions instead of just timing them. This allows for timing dependent computation which is necessary for accurate modeling of the mispredicted instruction stream or multiprocessor race conditions. Lastly, it is necessary to execute instructions in the core in order to use the checker to find implementation errors such as violating register dependencies.

An oracle sits in the fetch stage of the pipeline and is a functional emulator containing its own register file and memory. Oracles are commonly used to provide “perfect” behavior to do studies that measure the maximum benefit of an optimization. A common case of this is perfect branch prediction where all branch mispredictions are eliminated. In order to provide this capability, the oracle resides in the fetch stage so it knows the correct next PC to fetch.

We added a flexible speculative state management facility that permits restarting from any instruction. The ability to restart from any

instruction allows optimizations such as load address speculation and value prediction to be implemented. In these optimizations, instructions other than branches could be mispredicted, making it necessary to restart at the offending instruction. This approach also simplifies external interrupt handling since any instruction could follow an interrupt request, forcing a rollback. The checker also uses this mechanism to recover from any errors that are detected since any instruction could potentially cause an error.

MASE uses a callback interface is used that allows the memory system (or any resource) to invoke a callback function once the memory system has determined an operation's true latency. The callback interface provides for a more flexible and accurate method for determining the latency of non-deterministic resources.

5 Related Work

There are a number of performance modeling infrastructures available to instructors today that implement various forms of these technologies. The Pentium Pro simulator [12], Dinero [5], and Cheetah [15] are examples of simulators that read external traces of instructions. Turandot [10], SMTSIM [16] and VMW [4], are simulators, like SimpleScalar, that generate instructions traces through the use of emulation. RSIM [11] is an example of a micro-functional simulator; instructions are emulated in the execution stage of the performance model. Unlike MASE, it does not have a trace-driven component in the front-end. This prevents oracle studies such as perfect branch prediction. The idea of dynamic verification at retirement was inspired by Breach's Multiscalar processor simulator [2]. Other simulation environments include SimOS [13] and SimICS [8] which focus on system-level instruction-set simulation. MINT [17] and ATOM [14] concentrate on fast instruction execution.

There are also numerous visualization infrastructures available today. The tools range from pedagogical aids to comprehensive performance analyzers. DLXview [18] is a tool that depicts the DLX pipeline that is outlined in Computer Architecture: A Quantitative Approach by John Hennessy and David Patterson [6]. It was created as part of the CASLE (Compiler/Architecture Simulation for Learning and Experimenting) project at Purdue. Another common method for visualizing the performance of a simulator is to abstract away

the architecture and provide statistics based on the actual code running. CPROF [20][21] and VTUNE[19] are two examples of programs that display information such as cache misses or branch mispredictions for specific segments of code. RIVET [22-24] is a powerful display environment developed at the Stanford Computer Graphics Laboratory. The tool provides a very detailed time line view to identify problem areas. This view uses multiple levels of selection to gradually decrease the area of code being viewed, while simultaneously increasing the detail. Further background information on these tools and how GPV differs can be found in [25]. This paper also illustrates how visualization can be used for performance analysis.

6 Conclusion

We have introduced three tools in this paper that aid students using simulation in the classroom. The SS-GUI and backend perl script make it simple to launch simulations, by allowing the user to graphical select the simulator options and benchmark to simulate. The graphical pipeline viewer (GPV) aids the student in analyzing the simulation results. Finally, MASE's modularized code base and built-in checker mechanism make it ideally suited for efficient architectural model generation.

SS-GUI and GPV can be downloaded from <http://www.eecs.umich.edu/~chriswea/visualization/vis.tar>. The MASE toolset and documentation can be downloaded from <http://www.simplescalar.com/v4test.html>.

Acknowledgments

This work was supported under a National Science Foundation Graduate Fellowship and by the NSF CADRE program, grant no. EIA-9975286. Equipment support was provided by Intel.

References

- [1] T. Austin. DIVA: A Dynamic Approach to Microprocessor Verification. *Journal of Instruction-Level Parallelism Vol. 2*, Jun. 2000.
- [2] S. Breach. Design and Evaluation of a Multiscalar Processor. *Ph.D. thesis, University of Wisconsin-Madison*, 1999.
- [3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. *University of Wisconsin Computer Sciences Technical Report #1342*, June 1997.
- [4] T. Diep. VMW: A Visualization-based Microarchitecture Workbench. *Ph.D. thesis, Carnegie Mellon University*, June 1995.
- [5] J. Edler and M. Hill. Dinero IV Trace-Driven Unipro-

- cessor Cache Simulator. <http://www.neci.nj.nec.com/homepages/edler/d4>.
- [6] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, 1996.
- [7] E. Larson, S. Chatterjee, T. Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, Nov. 2001.
- [8] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. *Usenix Annual Technical Conference*, June 1998.
- [9] A. Moshovos and G. Sohi. Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors. *The 6th Annual Int. Symposium on High Performance Computer Architecture*, Jan. 2000.
- [10] M. Moudgill, J. Wellman, J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, May/June 1999.
- [11] V. Pai, P. Ranganathan, and S. Adve. RSIM Reference Manual. Version 1.0. *Technical Report 9705, Department of Electrical and Computer Engineering, Rice University*, July 1997.
- [12] D. Papworth. Tuning the Pentium Pro Microarchitecture. *IEEE Micro*, April 1996.
- [13] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SIMOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, Winter 1995.
- [14] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *Proc. of the 1994 Symposium on Programming Language Design and Implementation*, June 1994.
- [15] R. Sugumar and S. Abraham. cheetah - Single-pass simulator for direct-mapped, set-associative and fully associative caches. *Unix Manual Page*, 1993.
- [16] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proc. of the 22nd Annual Int. Symposium on Computer Architecture*, June 1995.
- [17] J. Veenstra and R. Fowler. MINT: a front end for efficient simulation of shared-memory multiprocessors. *Proc. of the 2nd Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, Jan. 1994.
- [18] Intel. VTune: Visual Tuning Environment, 1997. <http://developer.intel.com/design/perftool/vtune/index.htm>.
- [19] DLXView.[online] Available: <<http://yara.ecn.purdue.edu/~teamaaa/dlxview/>>, cited June 2001.
- [20] A.R. Lebeck, "Cache Conscious Programming in Undergraduate Computer Science," ACEM SIGCSE Technical Symposium on Computer Science Education, SIGCSE '99.
- [21] A.R. Lebeck and David A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," *IEEE COMPUTER*, 27(10):15-26, October 1994.
- [22] Robert Bosch, Chris Stolte, Gordon Stoll, Mendel Rosenblum and Pat Hanrahan, "Performance Analysis and Visualization of Parallel Systems Using SimOS and Rivet: A Case Study," *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, January 2000.
- [23] Robert Bosch, Chris Stolte, Diane Tang, John Gerth, Mendel Rosenblum, and Pat Hanrahan, "Rivet: A Flexible Environment for Computer Systems Visualization," *Computer Graphics* 34(1), February 2000.
- [24] Chris Stolte, Robert Bosch, Pat Hanrahan, and Mendel Rosenblum, "Visualizing Application Behavior on Superscalar Processors," *In Proceedings of the Fifth IEEE Symposium on Information Visualization*, October 1999.
- [25] Chris Weaver, Kenneth C. Barr, Eric D. Marsman, Dan Ernst, and Todd Austin, "Performance Analysis Using Pipeline Visualization," *2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2001)*, Nov 2001.

Web-based training on computer architecture: The case for JCachesim

Irina Branovic¹, Roberto Giorgi², and Antonio Prete³

^{1,2} Dipartimento di Ingegneria dell' Informazione
Facoltà di Ingegneria
University of Siena, Italy
branovic@dii.unisi.it, giorgi@unisi.it

³ Dipartimento di Ingegneria dell'Informazione
Facoltà di Ingegneria
University of Pisa, Italy
prete@iet.unipi.it

Abstract

This paper describes possible advantages of adding an interactive tool with log capabilities, in an online learning environment. We describe the interactive, Java-based tool named JCachesim, which is used for experimenting cache behavior with simple assembly programs while varying cache features. The tool has embedded features that allow the teacher to monitor the progress of each individual student.

1. Introduction

Internet offers the technology that supplements traditional classroom training with Web-based components and learning environments, where the educational process is experienced online. The objective is not to duplicate the characteristics of an ordinary class, but to use the possibilities of the computer to actually do better than what normally occurs in the face to face class.

According to projections, by 2004, 75 percent of US college students will have taken at least one online course. The number of colleges and universities offering e-learning will more than double, from 1,500 in 1999 to more than 3,300 in 2004. Student enrollment in these courses will increase 33% annually during this time [1].

Educational advantages that arise when supplementing a course with Web-based tools include:

- Enabling student-centered teaching approaches
- Providing 24/7 accessibility to course materials
- Providing just in time methods to assess and evaluate student progress
- Reducing “administrivia” around course management

There are also other, less obvious, but equally compelling advantages in favor of online teaching. Students are judged solely by their submitted work and their participation in online discussion forums, not by how they look. This "anonymity filter" has proven to have a positive effect on shy students, who are more likely to respond in class discussions and debates when they have the time to think beforehand, and to compose answers they feel good about.

Navigating through the screens of an interesting, colorful Web site maintains students' interest and can keep their brains active. Students can see other students' work and profit from their inspiration and understanding. Using conferencing, e-mail and other Internet features, students can also comment on each other's creations and discuss variations and other possibilities.

Structured note taking, using tools such as interactive study guides, and the use of visuals and graphics as part of the syllabus and presentation outlines contribute to student understanding of the course. Student discussion records, groups and project work and commentaries can be used to add to the content of the course.

2. Virtual classroom on computer architecture

There are a number of possible solutions for building a Web-based course on computer architecture. Detailed explanation about creating a virtual classroom, as well as examples of Web teaching environments can be found in [2]. For the purpose of creating our computer architecture classroom, we used a similar environment.

The consistent interface of distance learning environments speeds up the process of learning, and does not intimidate instructors and students with the ordeal of learning to use a new software

application each time a new tool is incorporated into the course.

Although our students found reading text in lectures via computer screen sometimes tedious, they liked integrated simulators, prerecorded lectures, and quizzes.

One of the most interesting enhancements that we added recently is the possibility of using interactive tools based on Java applets. These tools allow the students both to exercise and to learn. We also embedded a facility to automatically log the student's use of the tool, and create a personalized record to make sure that he used the basic functionalities of the tool. In the following, we consider such interactive tool for a lesson regarding cache memories.

3. The JCachesim tool

One of the lessons in our computer architecture classroom allows students to use an interesting tool for studying and analyzing a computer with cache memory, called JCachesim. It is based on previous experience on non Web-based tools [3].

JCachesim is a simulation environment of a computer with a cache memory. It allows the student to observe the CPU and the cache activities during the execution of a program, and in particular during read or write memory operation, to evaluate the system performance, to analyze the reference locality and the distribution of memory accesses due to the program execution. An exercise is organized in three phases: configuration, simulation, and analysis.

In the first phase, students write a program in an assembly language, and then configure the system. For cache memory, the student chooses the cache capacity, the placement policy (direct, full, or set-associative mappings), the cache block size, the number of ways. Main memory size can be chosen, the main memory update policy and, finally, the block replacement policy (FIFO, random, or LRU).

For I/O devices, the student specifies the I/O type (monitor, keyboard or general purpose), the synchronization scheme (none or handshake), the interrupt scheme (none, vectored interrupt or non vectored interrupt), and the addresses of the relative device registers (Figure 1).

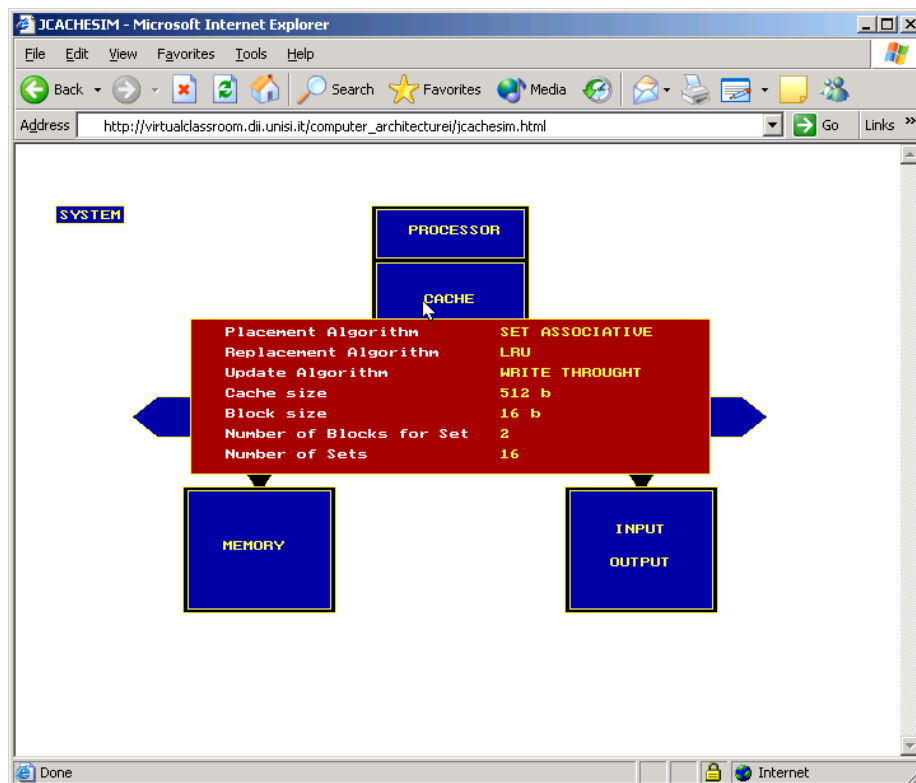


Figure 1: An example of configuring parameters, by clicking on the selected system component.

In the simulation phase, JCachesim can work in one of following three modes:

- Single – the student can ask for the execution of a memory operation by specifying the memory address and the operation type. In the single mode, JCachesim executes a single memory operation and shows, through an animation, the cache and main memory events, and the sequence of actions necessary to perform the required memory operation (Figure 2).
- Trace – the student can execute a program step-by-step, and examine cache or memory contents.

- Exe – the student can ask for the execution of the whole program (or a portion).

The student can watch the statistics regarding cache operations at any time (Figures 3, 4).

JCachesim tool is written in the form of interactive applets that allow us to train students. However, one of the most useful features of JCachesim is its ability to create a log of student's activities. The log file contains the student name, the time he or she took the test, the chosen settings and what kind of experiments were performed. The log file is automatically stored and available to instructors (Figure 5).

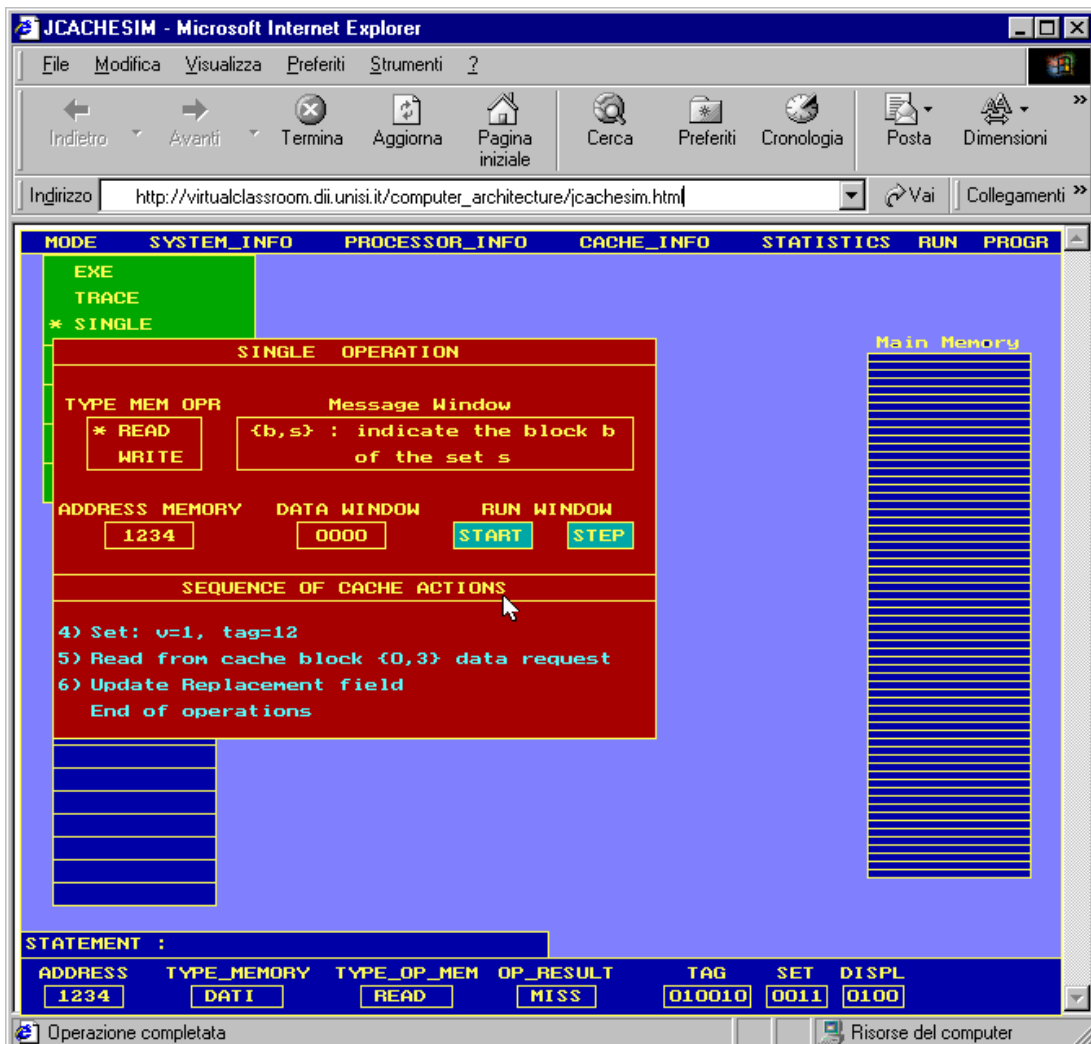


Figure 2: An example of JCachesim working in single mode.

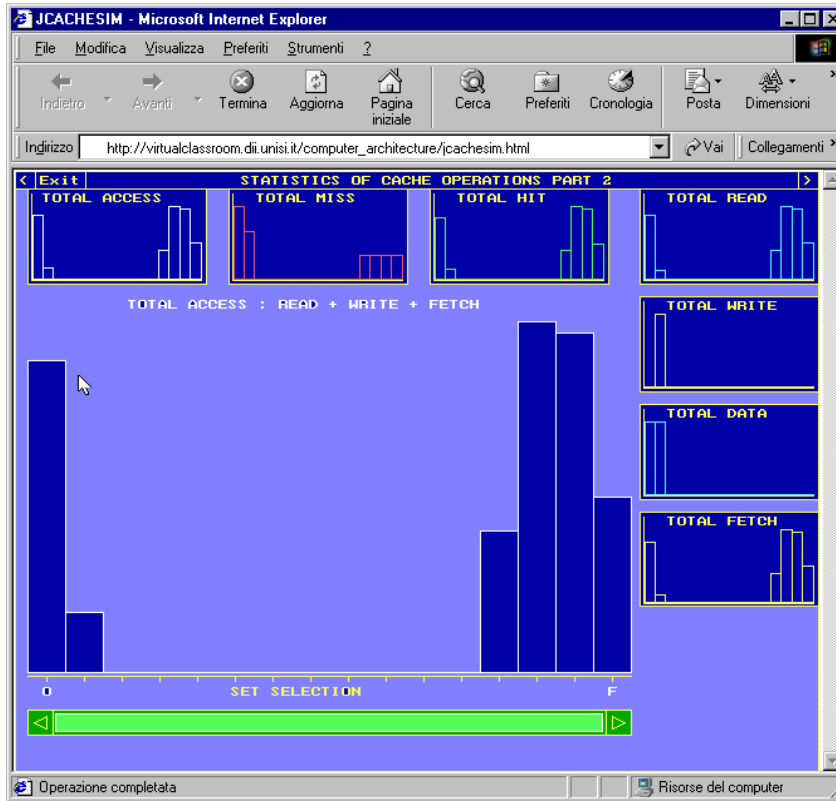


Figure 3: Pictures showing the locality of accesses in various memory areas.

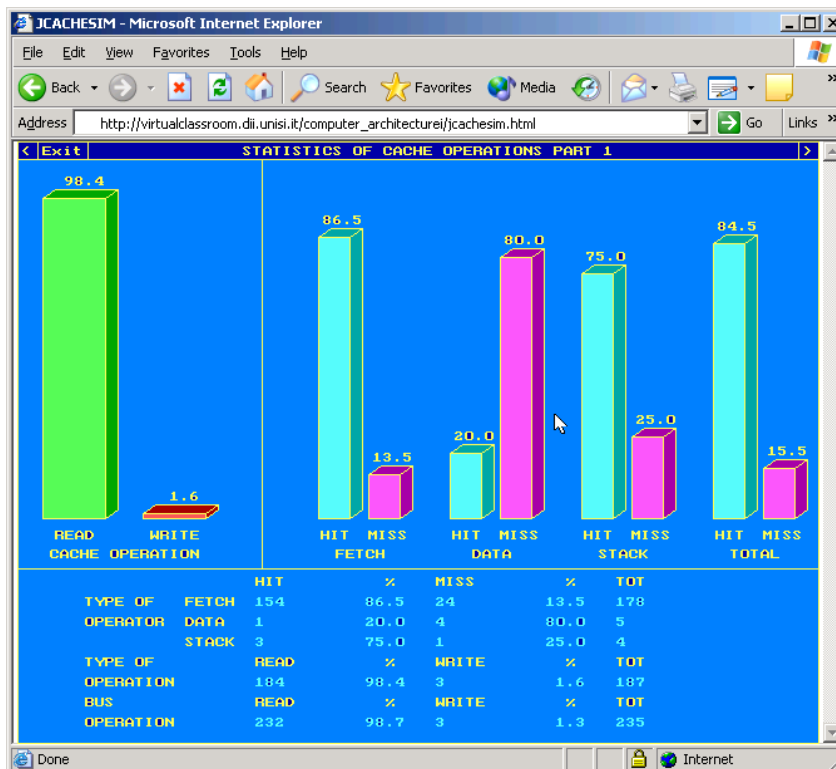
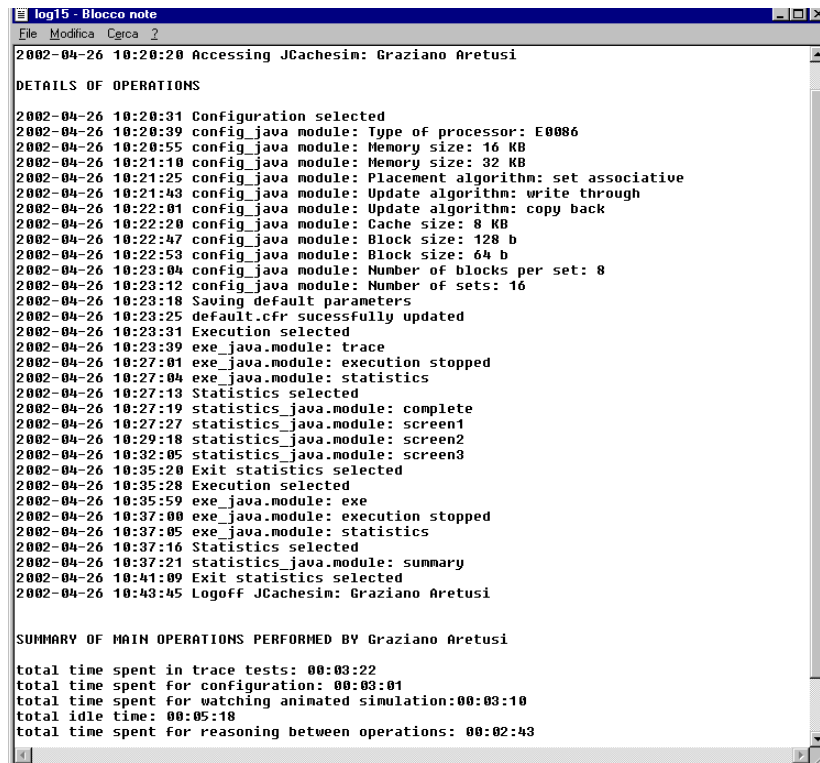


Figure 4: Global statistics of cache operations.

The generated log file also contains following information: total time spent for reasoning between operations, and total idle time. These data are used solely for tracking time the student spent

using the tool, and not for measuring his performances.

The JCachesim tool is still in the prototype phase, but we hope to deliver a final version to the public as soon as possible.



```
log15 - Blocco note
File Modifica Cerca ?
2002-04-26 10:20:20 Accessing JCachesim: Graziano Aretusi

DETAILS OF OPERATIONS
2002-04-26 10:20:31 Configuration selected
2002-04-26 10:20:39 config_java module: Type of processor: E0086
2002-04-26 10:20:55 config_java module: Memory size: 16 KB
2002-04-26 10:21:10 config_java module: Memory size: 32 KB
2002-04-26 10:21:25 config_java module: Placement algorithm: set associative
2002-04-26 10:21:43 config_java module: Update algorithm: write through
2002-04-26 10:22:01 config_java module: Update algorithm: copy back
2002-04-26 10:22:20 config_java module: Cache size: 8 KB
2002-04-26 10:22:47 config_java module: Block size: 128 b
2002-04-26 10:22:53 config_java module: Block size: 64 b
2002-04-26 10:23:04 config_java module: Number of blocks per set: 8
2002-04-26 10:23:12 config_java module: Number of sets: 16
2002-04-26 10:23:18 Saving default parameters
2002-04-26 10:23:25 default.cfr successfully updated
2002-04-26 10:23:31 Execution selected
2002-04-26 10:23:39 exe_java.module: trace
2002-04-26 10:27:01 exe_java.module: execution stopped
2002-04-26 10:27:04 exe_java.module: statistics
2002-04-26 10:27:13 Statistics selected
2002-04-26 10:27:19 statistics_java.module: complete
2002-04-26 10:27:27 statistics_java.module: screen1
2002-04-26 10:29:18 statistics_java.module: screen2
2002-04-26 10:32:05 statistics_java.module: screen3
2002-04-26 10:35:20 Exit statistics selected
2002-04-26 10:35:28 Execution selected
2002-04-26 10:35:59 exe_java.module: exe
2002-04-26 10:37:00 exe_java.module: execution stopped
2002-04-26 10:37:05 exe_java.module: statistics
2002-04-26 10:37:16 Statistics selected
2002-04-26 10:37:21 statistics_java.module: summary
2002-04-26 10:41:09 Exit statistics selected
2002-04-26 10:43:45 Logoff JCachesim: Graziano Aretusi

SUMMARY OF MAIN OPERATIONS PERFORMED BY Graziano Aretusi
total time spent in trace tests: 00:03:22
total time spent for configuration: 00:03:01
total time spent for watching animated simulation:00:03:10
total idle time: 00:05:18
total time spent for reasoning between operations: 00:02:43
```

Figure 5: An example of student log file, to be considered by the teacher.

4. Conclusions

Computer architecture requires understanding a wide variety of issues and interactions among them. One important step is that the student makes use of simulation tools to understand concepts otherwise difficult to experience.

Our internal research indicates that teaching and studying at a distance is equally effective, even better than traditional instruction, provided there is timely teacher-to-student feedback.

We have described the possible advantages of integrating an interactive tool with log capabilities into a virtual classroom environment. Using an interactive tool like JCachesim allows students to indicate the settings of a cache memory, to observe the cache activity needed for a memory operation, to evaluate the system performance by varying the parameters, and to analyze the program behavior by the memory references. One of the most important features of this tool is the ability to generate log files, which can be used to monitor students' progresses and track their

activities. Future plans for improving the JCachesim include providing more Java modules for enabling immediate interaction between students and instructors.

5. References

- [1] International Data Corporation: *Distance Learning in Higher Education: Market Forecast and Analysis, 1999-2004*.
- [2] Branovic, I., Milutinovic, V., Tutorial on Advances in Internet-based Education, (<http://galeb.etf.bg.ac.yu/~vm/tutorials>), School of Electrical Engineering, University of Belgrade, Serbia, Yugoslavia, 2001.
- [3] Prete, A., "Cachesim: A graphical software environment to support the teaching of computer system with cache memories", Proceedings of 7-th SEI Conference on Software Engineering Education, Springer-Verlag, January 1994.

Digital LC-2

From Bits & Gates to a Little Computer

Albert Cohen
A3 group, INRIA Rocquencourt

Olivier Temam
LRI, Université Paris-Sud

May 26, 2002

Abstract

This paper describes *DigLC2*, a gate-level simulator for the *Little Computer 2* architecture (LC-2) [3] which serves to strengthen the bottom-up approach to teach computer architecture. *DigLC2* is based on *Chipmunk*'s digital circuit simulator [1]; the circuit is freely available on the web and ready to use.

1 Context and Presentation

The principle of our approach is to combine a bottom-up presentation of computer architecture (from digital gates to processor and system) with an intuitive graphical gate-level design tool. This combination enables students to truly understand the logic behind processor design and internal processor workings, and simultaneously to gain confidence in the acquired knowledge thanks to experimental validation of concepts with a gate-level processor simulator (*DigLC2*). Based on this solid knowledge, we believe students are much more likely to quickly grasp and master new information about the evolution of processor design.

DigLC2 [2] is a gate-level simulator for the *Little Computer 2* (LC-2), as described by Patt and Patel in their introductory textbook on architecture and programming [3]. Unlike the existing LC-2 functional simulator [4, 5], it provides a detailed description of all processor components at the gate-level, so that students can themselves build a full processor using only elementary gates (AND, OR, NOT and Tri-State), thereby demystifying processor architecture.

The *DigLC2* simulator started as a support tool for a course at École Polytechnique (France) [6]. Designed to cooperate with the LC-2 functional simulator and assembler environment [4, 5], we wanted it robust and modular for practical lectures, as intuitive as possible to serve as a basis for student projects, and versatile enough to explore fundamental architecture and programming concepts. *DigLC2* contributed to our teaching experience in

the following ways:

- to understand the detailed sub-cycle behaviour of a realistic 16-bit processor;
- to experiment custom processor components in the context of a whole processor;
- to compare multiple data-flow and control models;
- to execute sample LC-2 programs, displaying processing stages from instruction-fetch to write-back;
- to play with basic input/output and interrupt mechanisms (they were not supported in the functional simulator [4]).
- to understand simple operating systems concepts;
- to extend the processor with hardware devices and off-chip controllers;
- to design and implement architecture enhancements for performance.

We followed the bottom-up approach advocated by Patt and Patel: students have been directly involved in the design of each processor component exploring multiple design issues. They achieved a finer understanding of the data-path and control structures, with a broader view of processor and system construction. Based on these fundamental concepts, the course diverted towards high-performance designs, program optimization techniques, and the foreseeable future of micro-architectures.

The students were already familiar with C, object-oriented and functional programming (OCaml) on one side, and analog electrical engineering on the other, but they had no experience in digital systems. Our intent was neither to bridge the gap between assembly and high-level languages nor to describe the mapping of ideal transistors to silicon wafers — both topics being taught in

the following semesters. We focused instead at the intermediate levels of the design, demystifying the building blocks of a microprocessor: from gates to combinatorial and sequential logics to data-paths and microprogrammed control to the instruction set architecture to assembler programming [6].

2 Technical Overview

The LC-2 system [3, 5] comprises a simple 16-bit microprocessor and a basic terminal emulation hardware. The instruction set is load/store¹ with 8 registers and 3 operands; it appears as a tradeoff between control-friendly and education-friendly features. The data-path is based on a 16-bit bus to connect almost all components *and* to communicate outside of the chip. Control is microprogrammed (fifty 39-bits wide microinstructions) and relies on a dedicated microsequencer for fast instruction selection and compaction. The LC-2 instruction set is very sketchy but supports a universal machine (e.g., no subtract, no OR operator, no shift...), forgetting about efficiency considerations. In comparison, system and device interaction is rather realistic and complete for such an educational architecture: both polling and interrupt-driven mechanisms are supported, and system calls (TRAPs) are clearly distinct from subroutine calls (yet the system does not address memory protection and address translation). Thanks to the original and efficient teaching model proposed by Patt and Patel, more and more introductory architecture courses are being built on the LC-2; the clean educational design of this processor is obviously a major incentive to do so.

The *DigLC2* simulator is free software (GPL), available online at

<http://www-rocq.inria.fr/-acohen/teach/diglc2.html>.

It is fully reusable, adaptable, and ready to use. Installation and usage documentation is available. The user should be familiar with the LC-2 specification, signal names and processor structures, as defined in Patt and Patel's textbook [3] (along with its appendices). *DigLC2* still lacks a technical manual, but the circuit is simple and most of the design is a straightforward implementation of the LC-2 specification. It runs over *DigLog*, Chipmunk's digital circuit simulator (GPL) [1]. We implemented the complete LC-2 architecture, including I/O terminal-emulation devices, interrupt vectors and memory (with customizable latency). Except for the SRAM memory chips and terminal device, every component of the LC-2 is built of elementary gates. The data-path and microsequencer are identical to the LC-2 specification.

¹Plus indirect load and store operations — for programming convenience — that we personally would not have provided and that we intentionally avoided in the course and application exercises.

We rewrote the microprogram from scratch — see the *DigLC2* documentation — and applied large-scale tests on sample codes and student projects. The “boot-time” memory structure (vector table, operating system, boot ROM and memory-mapped I/O) is almost identical to the functional simulator's model [5], except that the initial PC is 0x0000 and that some I/O routines have been optimized.

Concerning I/O operations, the LC-2 description is not complete and we had to make a few implementation choices: the interrupt vectors for keyboard input and CRT output (0x0010 and 0x0011, respectively) and the detailed implementation of I/O registers (interrupt control bits, strobe signals, device operation latency).

Figure 1 shows the control panel of the LC-2 simulator. It displays every addressable and internal register, the full microinstruction, and many other signals. It also provides keyboard and screen emulations (standard *DigLog* components) for interactive terminal operations.

As one may expect, performance is much lower than Postiff's functional simulator: approximately 20 cycles per second on a 500 MHz pentium III (interactive run, maximum details displayed): gate-level simulation of big programs is not realistic. However, we found these performances quite reasonable for the educational purposes of the LC-2 architecture:

- target codes implement short-lived classroom algorithms, toy programs and simple I/O operations;
- the most tedious part is linked with string processing and printing, e.g., the full CRT synchronization protocol proposed by Patt and Patel leads to a very slow implementation; still, choosing pragmatic parameters (short strings) and optimizing the code of display-oriented subroutines is usually satisfactory;
- in many cases, the user may even want to watch the real-time execution of the program, looking for errors in the assembly code, in a processor component, or in some custom additional circuit.

Eventually, we found only two architecture faults during circuit implementation: the first one is about choosing latches or flipflops and has been (arguably) corrected in recent online errata, the second one is a tricky page/PC-incrementation bug in conditional branch instructions. Considering the overall design, the detailed implementation choices and our teaching experience, we believe that the LC-2 architecture is a significant progress over previous educational systems; but we also hope that feed-back from professors and students around the world will be taken into account in future versions of the Little Computer and contribute to further improvements.

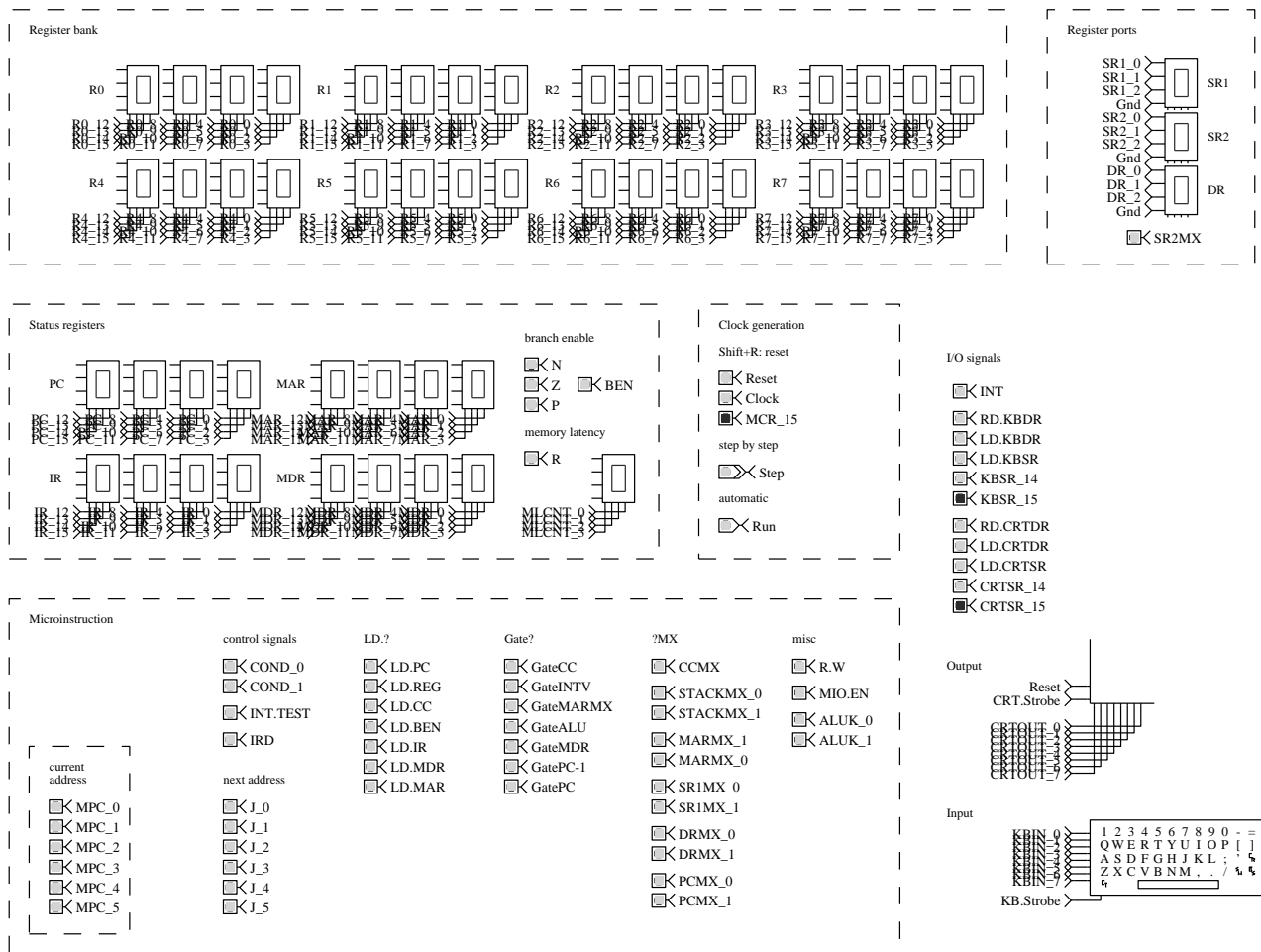


Figure 1: *DigLC2* control panel.

3 Student Projects

Three application projects have been proposed based on this digital simulator.

- Pipelining the LC-2, with a simple hazard detection and branch prediction mechanism. One student implemented a prototype version of a pipelined LC-2 (hardwired control, no indirect memory instructions). As a side-effect, simulator performance was significantly improved...
- Implementing a DMA controller for video output and experimenting a few bus protocols. This kind of extension is greatly simplified by the modular structure of *DigLC2*. For example, every memory control signal has a LC-2 side and a SRAM-chip side, and the LC-2 is designed to cope with an arbitrary/unknown memory latency.
- Adding an instruction cache and/or a data cache to

the LC-2; trying various associativity and replacement policies.

We believe that many existing student projects could benefit from *DigLC2*, focusing on the most interesting part of the project without the overhead of building a full processor or the complexity of a real-world processor. It can also be used to investigate the detailed implementation of processor performance enhancements — such as pipelining, superscalar and out-of-order execution — in the context of interrupts, and to interact with an existing assembler and legacy source code.

4 Conclusion and Future Work

DigLC2 is an interesting compromise between high-level structural modeling of digital circuits and expensive hardware test-beds. It is a useful tool for architecture courses, practical lectures, student projects and tutorials.

DigLC2 is an intuitive and modular implementation of the complete LC-2 system; it does not intend to be a fully realistic view of the actual silicon mapping, but provides a full gate-level simulation. By combining the bottom-up approach with *DigLC2* within the course and classes, students were able to progressively build their own full processor, using components they themselves designed session after session, and then they were able to visualize the execution of simple assembly programs at the gate-level.

Still, we would like to emphasize on the preliminary nature of this work. We believe that the tool might become even more beneficial if provided with multiple alternative implementations of each component, variations on the instruction-set architecture, and performance enhancements. We are not acquainted with processor verification techniques and did not address the testing and/or formal validation issues. Thanks to the wide distribution of Patt and Patel's textbook, we strongly encourage a community effort to contribute to the *DigLC2* project, as well to the underlying *DigLog* simulator [1].

References

- [1] The Chipmunk system (specifically *DigLog*, the digital part of the *Log* simulator).
Available online at
<http://www.cs.berkeley.edu/~lazzaro/chipmunk>.
- [2] A. Cohen. *DigLC2: a gate-level simulator for the little computer 2*.
Available online at
<http://www-rocq.inria.fr/~acohen/teach/diglc2.html>.
- [3] Y. N. Patt and S. J. Patel. *Introduction to computing systems: from bits & gates to C & beyond*.
McGraw-Hill, 2001.
<http://www.mhhe.com/engcs/compsci/patt>.
- [4] M. Postiff. LC-2 simulator (and assembler).
Available online at
<http://www.mhhe.com/engcs/compsci/patt/lc2unix.mhtml>.
- [5] M. Postiff. *LC-2 Programmer's Reference and User Guide*. University of Michigan (EECS 100), 1999.
<http://www.mhhe.com/engcs/compsci/patt/lc2labstud.mhtml>.
- [6] O. Temam and A. Cohen. Cours d'architecture. École Polytechnique 3^{ème} année majeure 1, 2001–2002.
<http://www.lri.fr/~temam/X/index.html>
(in French).

MipsIt—A Simulation and Development Environment Using Animation for Computer Architecture Education

Mats Brorsson

Department of Microelectronics and Information Technology,
KTH, Royal Institute of Technology
Electrum 229, SE-164 40 Kista, Sweden
email: Mats.Brorsson@imit.kth.se

Abstract

Computer animation is a tool which nowadays is used in more and more fields. In this paper we describe the use of computer animation to support the learning of computer organization itself. MipsIt is a system consisting of a software development environment, a system and cache simulator and a highly flexible microarchitecture simulator used for pipeline studies. It has been in use for several years now and constitutes an important tool in the education at Lund University and KTH, Royal Institute of Technology in Sweden.

1. Introduction

In order to learn computer architecture and systems you need to learn how to master abstractions. A computer system is one layer of abstraction on top of the other. At one end you have digital electronics, which in itself can be seen as several layers, and at the other you have complex applications using perhaps techniques such as polymorphic inheritance which needs to be resolved in run-time.

For students studying computer organization and architecture, these abstractions are often confusing as they are not always that distinct. Furthermore, given the high level of integration in modern computers, it is also quite difficult for students to get a good understanding of what is happening deep down in the black centipedes on the motherboard.

At Lund University, and now also at KTH, Royal Institute of Technology, both in Sweden, I have taken part in the development of a set of courses in computer systems, organization and architecture in which laboratory exercises and simulation tool support are used extensively to support learning. In this paper I describe some of the simulation tools that were developed during this process.

The MipsIt set of tools is part of a bigger laboratory exercise environment with a hardware platform, software development tools and a number of simulators. Many of the simulators support *animation* as a support for the students to understand the works of a relatively complex structure.

I first describe some of the trade-offs between hardware platforms and simulators that we considered in developing

our exercise material. Next I present an overview of the software system of MipsIt followed by a more detailed description of the animated simulators in sections 4 and 5.

2. Hardware vs. Simulation

I think that most instructors would agree with me that exercises where the students get real hands-on experience with digital electronics, assembly level programming, data representation etc. are crucial for the students' learning. Furthermore, it is my firm belief that students must touch, feel and smell¹ the real hardware in a computer organization course. Some universities let the students study computers using only simulated hardware. In my experience, this might lead to a confusion as to what is really happening. Is there really another machine inside this PC, workstation or whatever is used as simulation host?

Therefore, we use real, naked hardware—naked in the sense that there is no operating system on the hardware—to aid the students in understanding computer systems. The current system consists of a development board with a MIPS processor, some memory and a few simple I/O devices [2]. Unfortunately, this does not entirely solve the problem of abstraction. When you connect a development board to a host computer through a terminal program this might be a problem as well. I have had students that answer the question on where the program is executed by pointing to the window on the host computer screen instead of on the processor chip on the board on the desk beside the computer. It is anyway less abstract than if the program executes on a simulator and it is possible to remove the cable between the development board and the host computer and verify that the program still executes with simple I/O devices on the board.

This works well for students to learn about data representation, assembly level programming, simple I/O structures (polling and interrupts) and general low-level computer system design. It is, however, not well suited to study cache memories or processor hardware design as these structures are buried deep inside the processor.

1. Hopefully they smell burned electronics before it breaks!

We have previously let the students build simple micro-programmed processors using discrete components during laboratory exercises. Even though this has been very effective for the understanding of how simple hardware can be organized to execute instructions, we have abandoned it for the first compulsory course.¹ The simplifications that we needed to make in the instruction set that we could implement were to big in order to relate to modern computer instruction set architectures and it was not possible to do off-line development with the hardware used.

So how do we then support the study of hardware structures such as cache memories and pipeline design when we cannot build hardware for it. This is where animated simulation fits in. We have also resorted to simulation of the development board to let the students work at home preparing for lab. exercises and to support distance courses.

I have during the course of being a university teacher found that many students have difficulties of really understanding how cache memories work. They can easily understand the concept, but when it comes to how you should build the hardware to actually implement the concept, it becomes difficult. The concept of pipelining—which is the dominant implementation method of processors today—has similar characteristics. It is easy to understand in principle, but when it comes to the actual hardware design, it becomes tricky. This was the motivation to why we developed the animated simulators described in this paper.

3. The MipsIt system

The MipsIt system consists of a development environment, a hardware platform and a series of simulators. The topic of this paper is mainly the animation support in the simulators for cache memory and pipeline simulation, but for completeness I also describe the other parts. All software developed for the MipsIt system are targeted for the Windows (95-XP) platform as host machine.

3.1 Development environment

The MipsIt development environment is used to develop software for the hardware platform as well as for the various simulators. It targets the development board shown in section 3.2 but the same binary can be used to execute on the various simulators as well. Figure 1 shows an example of how the development environment might look for a software project with mixed C and assembler files.

The compiler, linker and other tools are standard tools in the gcc tool chain configured for cross-compilation to a MIPS target platform. What we developed was the graphical user interface mimicking the MS Visual DevStudio as a

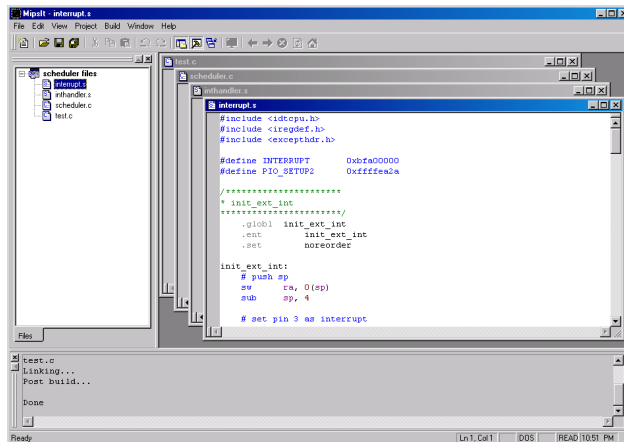


Figure 1. The development environment is inspired by Visual DevStudio.

front-end to gcc and which also replaces Makefile by handling projects of source codes and their dependences.

Although not tested, the same front-end should be possible to use with any gcc cross-compiler. The system is highly configurable and there is also an option to run an arbitrary command before or after linking. We use this feature to create an S-record file that can be used for downloading to the development board. We later modified the on-board monitor to use the ecoff-file produced by the compiler/linker.

3.2 Hardware

Figure 2 shows a photograph of the development board from IDT that is in use at Lund University [2]. It contains an IDT 36100 micro controller with a MIPS32 ISA processor core [3]. We deliberately chose the MIPS architecture as our instruction ISA because of its simplicity.

Another advantage of the MIPS ISA at the time was also that it is used in the textbooks of Hennessy and Patterson [1, 4]. These textbooks are used at Lund University as well as in many other universities and it makes it easier for the students if they can relate the laboratory exercise material to the textbook directly. The abstractions needed are difficult enough anyway.

The evaluation board itself, as shown in figure 2, contains the processor (chip-select logic, timers and two serial port UARTs are also integrated on-chip), some SRAM, slots for DRAM (not used in our configuration) and EEPROM which contains a simple monitor. The monitor contains some routines from libc which can be taken advantage of for small footprint C programs. The routines include partial functionality of printf. There are also routines to install normal C functions as interrupt routines.

All micro controller signals also appear on the edge connectors of the development board. We developed a simple daughter board containing one eight-bit and one 16-bit parallel bi-directional I/O port. It also contains a simple interrupt unit with three interrupt sources (two push-buttons and

1. The course is given in a 4.5 year programme leading to an M.Sc. in computer science, electrical engineering or information technology engineering.

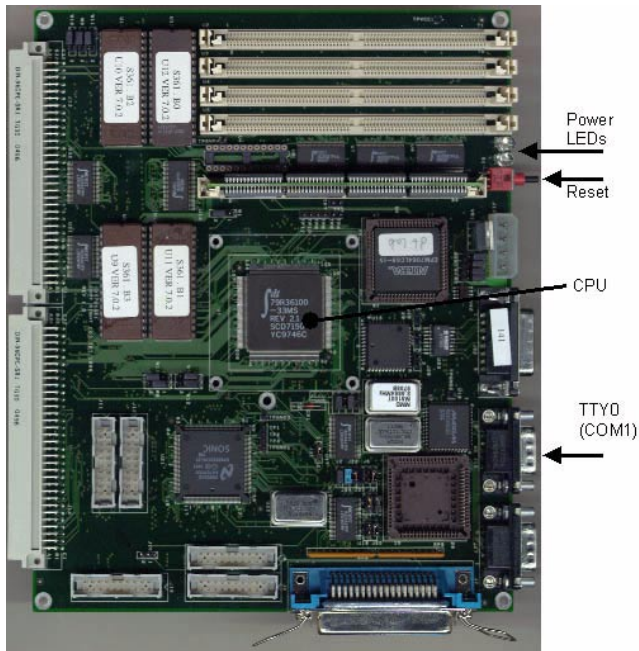


Figure 2. The IDT development board used in the exercises.

one adjustable pulse source) that also could be read as a six-bit parallel input port. Three of the bits contain the current status of the three input sources and the other three are latched versions of the same input. These bits retain their value until reset by writing (any value) to the port.

3.3 The simulators

The third part of the MipsIt environment is a set of simulators. There is one system simulator which mimics the evaluation board as faithfully as possible. While developing this simulator it was our goal to be able to execute any binary code developed for the target hardware platform. We therefore have to simulate the on-board monitor and all I/O devices, including on-chip timers.

The full code compatibility has been achieved in the system simulator which is described in section 4. This simulator also contains an animated cache simulator. We also wanted to use simulation for microprocessor implementation studies. This resulted in a general micro architecture simulator which is controlled by a simple hardware description language and animation control so that many different microprocessor implementations can be illustrated. This simulator is described in section 5.

4. The system simulator

4.1 Overview

Figure 3 shows the system simulator with a few of its windows open. The top left window shows a simplified view of the entire system: CPU, instruction and data caches, some

RAM, a console window and some I/O devices. The window at bottom left shows the register contents of the CPU. The top right window shows the eight-bit parallel I/O device which consists of eight LEDs and eight binary switches. Just as what we have in hardware. The 16-bit parallel I/O-port is the only one from the hardware that is not implemented in the simulator. The bottom right window shows the simple interrupt sources. Two push-buttons and an adjustable pulse timer.

The main reason for developing this simulator is because it simplifies for the students to study computer organization on their own, at home. Most students have access to PCs with Windows and it is therefore easy for them to download the simulators and development environment to start work on their own.

However, as we designed the laboratory exercises, we found that the simulator could actually be used also in the class-room. Figure 4 shows the memory view in the simulator. The memory addresses are shown to the left and the contents is shown to the right as hexadecimal numbers and an interpretation. In the current view the interpretation is the disassembler view but other possible views are interpretations as unsigned integers, signed integers, single precision floating point numbers and ASCII characters.

The dot to the left in the memory view shows a break point and the line, also to the left, signifies that these instructions are currently in the instruction cache. The darker line shows the current instruction being simulated.

With this view, the simulator became a powerful tool to study instruction execution and the effect each instruction had on the registers etc. Since the MIPS architecture does not have vectored interrupts, it became cumbersome to single-step interrupt routines on the hardware and we therefore used the simulator to study interrupt routines in this detail. The students could also experiment with instruction coding, making hexadecimal instruction codes by hand, entering them in the memory and immediately see if they had coded the instruction correctly. Floating point number coding could be studied in the same way.

4.2 Cache simulator

Even with all the benefits as described above, these were not the only purposes for developing the simulator. The major driving force was to introduce animation to aid the students to really understand the inner workings of cache memories. We used the figures of a textbook as an inspiration as how to present the caches graphically [4].

Figure 5 shows the cache view in the simulator. It shows the current configuration of the data cache. The data and the instruction caches can be configured independently. It shows the entire cache contents with both tag and data store. It also shows how the address presented by the processor is divided into different fields for indexing, word select and tag

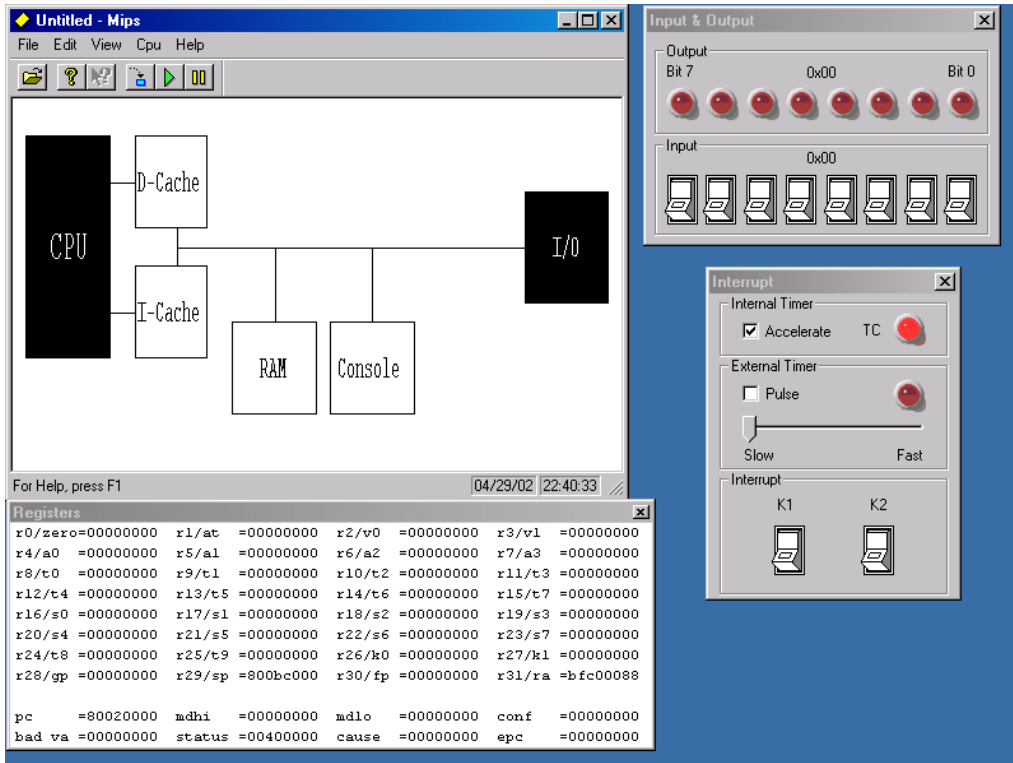


Figure 3. The system simulator with CPU register and I/O-device windows open.

Address	Content	Label		
80020A8C	8F BF 00 1C	LW	\$31, 0x1c(\$29)	
80020A90	8F BE 00 18	LW	\$30, 0x18(\$29)	
80020A94	8F B1 00 14	LW	\$17, 0x14(\$29)	
80020A98	8F B0 00 10	LW	\$16, 0x10(\$29)	
80020A9C	03 E0 00 08	JR	\$31	
80020AA0	27 BD 00 20	ADDIU	\$29, \$29, 0x20	
80020AA4	3C 02 80 02	__main[]	LUI	\$02, 0x8002
80020AA8	8C 42 0E 70	LW	\$02, 0xe70(\$02)	
80020AAC	27 BD FF E8	ADDIU	\$29, \$29, 0xffe8	
80020AB0	AF BE 00 10	SW	\$30, 0x10(\$29)	
80020AB4	03 A0 F0 21	ADDU	\$30, \$29, \$00	
80020AB8	14 40 00 06	ENE	\$00, \$02, 0x6	
80020ABC	AF BF 00 14	SW	\$31, 0x14(\$29)	
80020AC0	24 02 00 01	ADDIU	\$02, \$00, 0x1	
80020AC4	3C 01 80 02	LUI	\$01, 0x8002	
80020AC8	AC 22 0E 70	SW	\$02, 0xe70(\$01)	
80020ACC	0C 00 82 80	JAL	0x8280	
80020AD0	00 00 00 00	NOP		
80020AD4	03 C0 E8 21	ADDU	\$29, \$30, \$00	
80020AD8	8F BF 00 14	LW	\$31, 0x14(\$29)	
80020ADC	8F BE 00 10	LW	\$30, 0x10(\$29)	
80020AE0	03 E0 00 08	JR	\$31	
80020AE4	27 BD 00 18	ADDIU	\$29, \$29, 0x18	
80020AE8	00 00 00 00	NOP		
80020AEC	00 00 00 00	NOP		
80020AF0	3C 02 BF C0	/cirdan/cailin/ldt	LUI	\$02, 0xbfc0
80020AF4	34 42 00 38	ORI	\$02, \$02, 0x38	

LUI \$01, 0x8002 ; \$1=0
Address mode: Virtual View mode: Assembler Tracking PC

Figure 4. The memory view in the simulator.

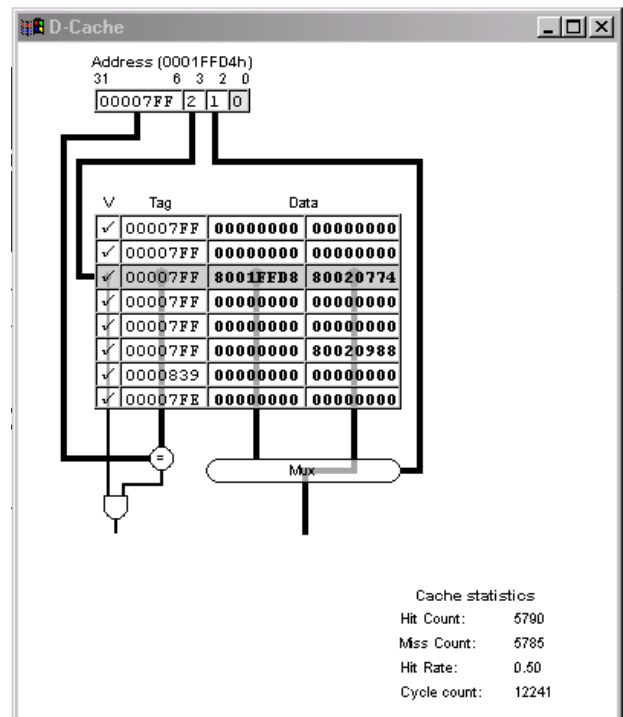


Figure 5. The animated cache view.

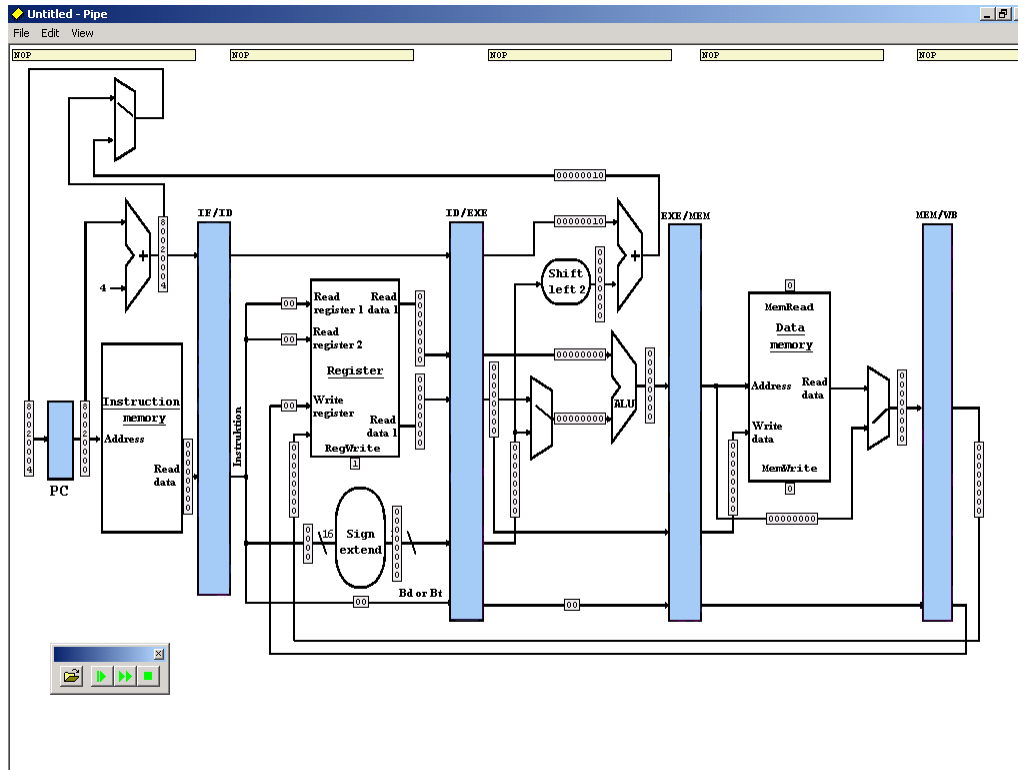


Figure 6. An example of a simple pipeline simulator view. The simulator is only a shell which can be loaded with arbitrary pipeline structures.

check. The students can single-step their programs and follow the cache access for every memory reference and therefore gain a deeper understanding of how the cache works.

The simulator also keeps some simple statistics, as shown at the bottom right in the figure. This can be used to compare different cache settings for more longer-running programs.

Memory access penalty can also be configured and it is thereby possible to perform direct comparison with the hardware which contain small instruction and data caches.

5. The pipeline simulator

At Lund University, we had a long experience of using animation and graphical representation of pipeline execution [5]. We wanted to make use of this experience, but retain the compatibility with the hardware that we developed for the system simulator as described previously. Another design goal was to make a flexible design that could be run by students at home on their PCs. The existing software was for Sun/Solaris and neither very portable nor flexible.

5.1 PipeS and PipeXL

Instead of having a hardwired pipeline design in the simulator software, we developed a flexible simulation shell which could be loaded with different micro architecture implementations. The simulator shell can be used to load programs into memory, to drive the simulated clock signal and to mon-

itor register and memory contents, as in the previously described simulator. However, when the program starts, it contains no description of the simulator hardware. This has to be loaded from a file which describes the micro architecture in a hardware description language (see next section). Figure 6 shows an example in which a simple five stage pipeline without forwarding is shown.

The students can load the memory with a program, just as before, and starts to single step the execution. As the program advances, the pipeline graphics is changed. Muxes are set in the positions needed for execution, data values on buses and inputs are shown and the instruction currently executing in each pipeline stage is shown at the top.

Our experience is that this tool has been tremendously powerful to convey the concept of pipelining to the students. The way that the pipeline is graphically represented is an invention of a student at Lund University in the late 80s but was independently later discovered for use in major textbooks [4, 6].

Figure 7 shows another example of a micro architecture implementation. This is much more complex and complete. In addition to what is present in figure 6, it also contains the control signals, data forwarding and hazard control. We will now see how we can represent different pipeline structures to be used in the simulator.

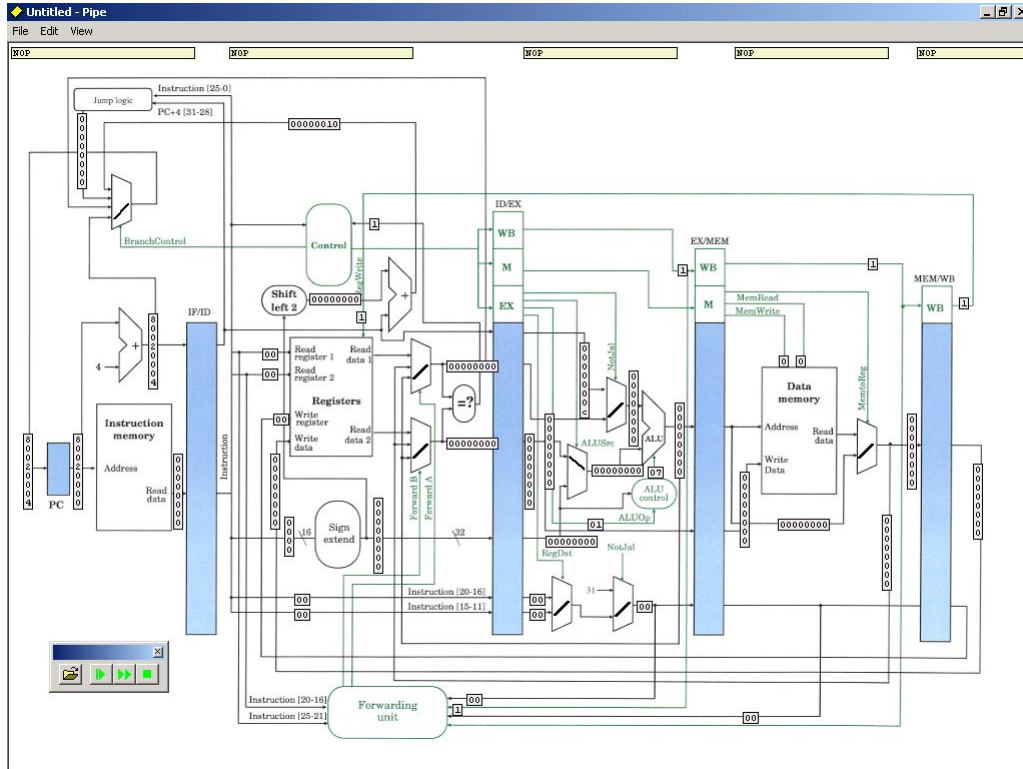


Figure 7. A more complex pipeline structured using the same simulator shell.

5.2 Some Hardware Description Language

The micro architectural structure of the processor is described in a simple hardware description language which is described here shortly. The pictures shown in figures 6 and 7 are not derived from this language but simple bitmap files that are shown in the window.

The original aim was to use a subset of VHDL as description language to be able to leverage on the body of text written about this language. However, it turned to be cumbersome to parse and instead we developed a simple object oriented HDL where each component is a class which can be instantiated to an object and the inputs and outputs of an object are connected to the inputs and outputs of other objects. Almost any synchronous hardware structure can be expressed in this language. There are also hooks to the simulation framework in the language. Most notably for the clock signal, memory accesses and to the register and memory views of the simulator.

5.3 Components

Below is the code for a simple component; the two-input mux:

```
class CMux2
{
  in In0
```

```
  in In1
  in Control:1
  out Out

  script
    function OnChange()
    {
      if ( Control.Get()==0 )
        Out.Set( In0.Get() );
      else
        Out.Set( In1.Get() );
    }

    Out.Set(0);
  end_script

  event ALL OnChange()
}
```

At first a class description is used to give the component a name. Next the interface of the component is specified. The default width of inputs and outputs is 32 bits. In this case only the control signal deviates in that it is specified as one bit only.

Then follows a script which describes the behavior of the component. The function OnChange is executed whenever

any of the input signals changes state as described by the last statement in the component description. Input signal values are retrieved by accessing a member function `Get()` and similarly output signal values are set by the member function `Set(x)`. The last lines of the script can be used to set the initial state of the output signals.

A mux is a combinational component and does not contain any state. If any of the input changes, the output is immediately changed also. In contrast, the program counter is a simple component which is clocked. As shown by the example text below, the PC is clocked by a simulated two-phase clock.

```
// this is the Program Counter
class CPC {
  in Ph0:1
  in Ph1:1
  in In
  out Out

  script
    var rPC = 0;

    function OnPh0() {
      rPC = In.Get();
    }

    function OnPh1() {
      Out.Set(rPC);
    }
  end_script

  event Ph0 OnPh0()
  event Ph1 OnPh1()
}
```

The signals `Ph0` and `Ph1` are driven by a clock object and are used to derive the two-phase clock. The value of the PC is stored in an internal variable (`rPC`) which is read from the input on one clock phase and output on the second clock phase.

5.4 Connecting components together

When the entire micro architecture has been suitably broken down in components, clocked or not, they can be connected together. The following piece of code shows the connections for the ID-stage in the simple pipeline of figure 6.

```
// components:

object CPC PC
object CInstrMem InstrMem
object CMux2 PcMux
object CiAdd4 IfAdd4
```

```
object CRegIfId RegIfId

// Net list:
// to PC
connect PcMux.Out PC.In
// to InstrMem
connect PC.Out InstrMem.Address
// To pc+4 thing
connect PC.Out IfAdd4.In
// to the pipeline register
connect InstrMem.ReadData
                                RegIfId.in_Instruction
connect IfAdd4.Out RegIfId.in_PC
// to the pc mux
// (only connections from this stage
// are done here)
connect IfAdd4.Out PcMux.In0
// connect the clock
connect clk.Ph0 RegIfId.Ph0
connect clk.Ph1 RegIfId.Ph1
connect clk.Ph0 PC.Ph0
connect clk.Ph1 PC.Ph1

// and now some probes:
probe InstrMem.ReadData 173 393 16 8 1
probe PC.Out 70 355 16 8 1
probe IfAdd4.Out 148 171 16 8 1
probe PcMux.Out 14 355 16 8 1
probe InstrMem.ReadData 2 4 16 30 2
```

Note how the components first are defined and then connected together in simple connect-statements. At the end, graphical probes are defined. It is these that makes up the animation in the final simulation picture. After the key word `probe`, a signal name is given. This is the signal to monitor. The next two arguments are the x- and y-coordinates of where the probe is to be shown in the pipeline picture. The third argument is the number format for the probe data, the fourth argument is the number of digits to use, and the final argument is the direction of the probe: 0 for horizontal and 1 for vertical. The last probe is different. It is used to show the assembler format of the instruction read from memory. The mux direction is also a probe, but since the mux in this pipeline is controlled in the EX-stage, the probe is also defined there.

We have not yet let students define their own pipeline designs. What we have done is to provide them with a skeleton and let them work out the instruction deciding, hazard control, and forwarding unit for themselves. It has been amazingly simple for them to iron out these details, once they got the hang of pipelining in the first place.

6. Conclusion

The software described in this paper has been used in computer organization and architecture education at Lund University and at KTH for several years now. It is now mature enough that we feel it is time to share our experiences which have been very good. We have received quite encouraging feedback from students who both find it useful to work with laboratory exercises at home and who appreciate the graphical animation in the user interface.

Acknowledgements

The work reported here was performed while the author was affiliated with the department of Information Technology at Lund University. The laboratory exercises were made together with Jan Eric Larsson. Coauthors of the software were: Ola Bergqvist, Georg Fischer, Mats Brorsson, Martin Andersson, Joakim Lindfors and Tobias Harms.

A binary version of the MipsIt package for Windows 95-xp with suitable exercises can be retrieved for educational and personal use from the following web site:
<http://www.embe.nu/mipsit>.

References

- [1] J. L. Hennessy and D. A. Patterson, *Computer Architecture — A Quantitative Approach*, 3rd ed., Morgan Kaufmann Publishers, 2002.
- [2] Integrated Device Technology, Inc., *79S361 Evaluation board: Hardware User's Manual*, ver 2.0, Sept. 1996.
- [3] Integrated Device Technology, Inc., *IDT79RC36100, Highly Integrated RISController: Hardware User's Manual*, ver 2.1, Aug. 1998.
- [4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd Ed., Morgan Kaufmann Publishers, 1997.
- [5] P. Stenstrom, H. Nilsson, and J. Skeppstedt, Using Graphics and Animation to Visualize Instruction Pipelining and its Hazards, in *Proceedings of the 1993 SCS Western Simulation Multiconference on Simulation in Engineering Education*, 1993, pp. 130-135.
- [6] B. Werner, K. Ranerup, B. Breidegard, G. Jennings, and L. Philipson, *Werner Diagrams - Visual Aid for Design of Synchronous Systems*, Technical report, Department of Computer Engineering, Lund University, November 1992.

CoDeNios: A Function Level Co-Design Tool

Yann Thoma and Eduardo Sanchez
Logic Systems Laboratory
Swiss Federal Institute of Technology
1015 Lausanne, Switzerland
{yann.thoma,eduardo.sanchez}@epfl.ch

Abstract

The need of co-design systems, along with the FPGA complexity, is increasing dramatically, both in industrial and academic settings. New tools are necessary to ease the development of such systems. Altera supplies a development kit with a 200'000 equivalent gates FPGA; combined with its proprietary Nios configurable processor, it allows co-design and multi-processor architecture creation. In this paper, we present a new tool, CoDeNios, which lets a developer partition a C program at the function level, and automatically generates the whole system.

1 Introduction

Until recently, co-design[4] was limited to complex industrial projects. The high cost of such systems did not allow academic projects to use co-design. Now, with the development of Field Programmable Gate Arrays (FPGAs), the conception of such systems is easier. The reprogrammable capability of FPGAs permits prototyping at a low cost, which is very important for universities and industries. The problem now is the lack of tools aiding development of these systems. With this aim in view, Altera supplies the Nios processor family. This soft IP core is a configurable RISC processor which can be used in any design.

In this paper we present CoDeNios (**CO-DE**sign with a **NIOS**), a new tool based on a Nios processor, which helps a developer make a hardware/software partition[3] of a C program. This partition is made at the function call level. For each function declared like `void fname(...)`, the user can force it to be calculated either by the main processor, by a slave processor, or by a hardware module. For the last case, the developer has to write a VHDL file to define the function behavior. Apart from this human intervention, the whole interface between hardware and

software is automatically generated (C and VHDL files).

Contrarily to other systems like COSYMA [2], which automatically makes a partition, our software lets the user choose it. This particularity allows the developer to test any hardware module by automatically interfacing it to a processor. It is also useful for academic courses, where students can do the partition themselves, and evaluate their work. P. Chou, R. Ortega and G. Borriello [1] have created a system to synthesise a hardware/software interface for a micro-controller. Their work is made for peripherals present outside the chip which contains the controller. With our tool, the processor and its user-defined peripherals are implemented in the same chip. Thus, CoDeNios is better suited for system prototyping and hardware module evaluation.

This paper is structured as follows: Section 2 describes the APEX20K^R FPGA family supplied by AlteraTM and the Nios processor used by CoDeNios. Section 3 focuses on CoDeNios itself, explaining its possibilities, while section 4 explores the performances of a design generated by our application. Finally section 5 concludes by discussing current and future work.

2 APEX20K family and Nios

Altera, with the APEX20K family, offers FPGAs with densities ranging from 30'000 to over 1.5 million gates. It is built for system-on-a-programmable-chip (SOPC) designs, with embedded system blocks used to implement memories as dual-port RAM, ROM, CAM, etc. For our application, we use a development board with an APEX20K200E, from the APEX20K family (cf. figure 1). This FPGA contains 106'496 configurable memory bits, and 200'000 equivalent gates, which is enough to implement a 3-processor design.

Along with these new FPGAs which allow SOPC

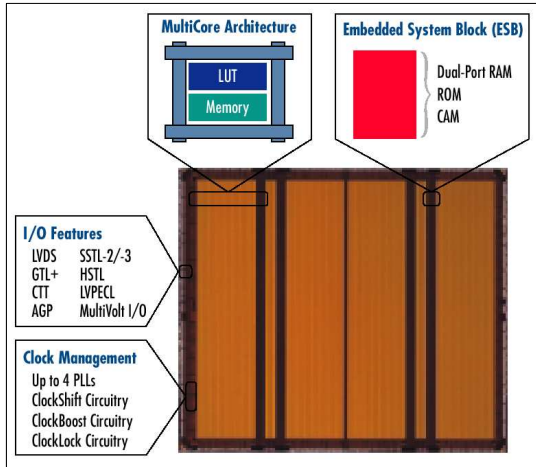


Figure 1: APEX Device Features

designs, Altera supplies a new processor. The Nios (cf. table 1) is a configurable RISC processor, working with 16 or 32 bits (instruction and data). A wizard helps create a Nios with all the necessary parameters. The size of instructions, as well as the number of registers, is decided by the user. A multiplication unit can be added to speed up multiplications, with a cost in term of gates. The most interesting possibility is the ability to add as many peripherals as needed. Many of them are already supplied by the wizard: memory interfaces for ROM or RAM, UART to manage a serial COM, IDE controller, timer, etc. All these peripherals are memory mapped for the processor. User-defined peripherals can also be added, by specifying the address range, an optional interrupt number and the number of clock cycles for write and read operation. When all the processor parameters are set, a VHDL entity is generated, which can be included in any design.

As CoDeNios supports a multi-processor architecture, we chose a 16 bit Nios, so as to allow a maximum of processors in a design. One single special peripheral was added, which contains all hardware and slave processor calculated functions. It has an address range of 2, used to access a counter (1 address for a 32 bit counter accessible in 2 read cycles) and to define a protocol for calling functions and pass parameters.

3 CoDeNios

The hardware/software partitioning of a task aims to accelerate it, by taking advantage of hardware speed. An important issue is therefore to be able to find bottlenecks where hardware can speed up a sys-

Table 1: Nios processor characteristics

Feature	Description
type	RISC
pipeline	4 levels (5 for load/store)
instructions and data size	16 or 32 bits
number of registers	128, 256 or 512
frequency	< 50 MHz
place	approximately 26'000 bits for the 16 bits version

tem. Then the new solution needs to be evaluated in order to prove it is better than the original software execution. Currently there is no theory to calculate precisely the execution time of a co-design system, so many experiments and measures have to be run.

A second co-design problem is the interface between hardware and software. For each new hardware module connected to a processor a protocol has to be defined. The conception of this part of a system can be very time-consuming, so automating this task would be a great advantage for a developer.

CoDeNios proposes to solve both problems. This tool, based on the Nios processor described above, has a graphical user interface which enables a developer to make a partition of a C program, at the function level, simply by click, drag and drop operations. This partition allows a function to be calculated by the main processor, by a slave, or by a hardware module. Once the choices are validated, an interface between the different processors and the hardware modules is generated in the form of VHDL and C files. The original C code of the main processor is transformed to call slave modules, while for a slave Nios, the whole C code is generated. For the hardware, the whole system is generated, except the architecture of hardware modules. For them, a template is generated, letting the developer describe the function behavior.

3.1 Function Selection

At the beginning of a project, the developer writes a C program for a 16 bit Nios. The C file can be opened with CoDeNios. A graphical user interface (GUI), as shown in figure 2, lists all functions returning `void`¹ in a rectangle representing the main processor. It is then possible to drag and drop a

¹This limitation will be reduced, by also allowing functions returning an integer.

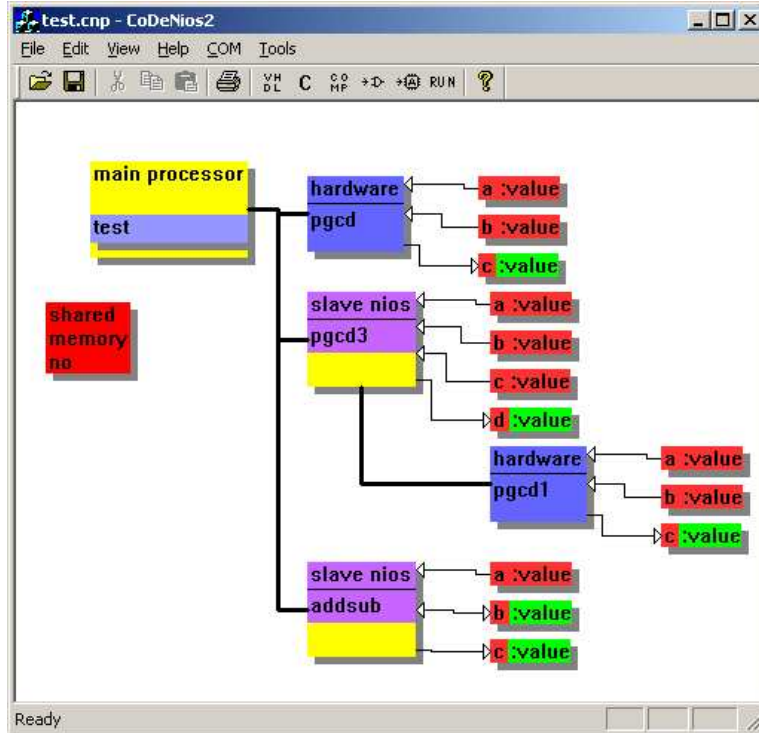


Figure 2: CoDeNios graphical user interface

function outside this rectangle to make it a hardware module. By clicking on it, a hardware module can be turned into a slave processor, and vice versa. For both entities, all input and output parameters are listed, connected by an arrow. For a parameter passed in C by reference (`int *a`), the direction (input, output, input-output) can be changed by the user, by clicking on the arrow. The value or the reference can be sent to the slave module, allows the use of pointers to access a shared memory.

When the whole system is configured correctly, buttons on the GUI can launch VHDL and C file generation, hardware synthesis, placement and routing, C compilation, and finally start up the execution of the program on the board, assuming the FPGA is configured. This command sends all different executable codes for every processor on-chip. Then, with a terminal, CoDenios installs a communication between the main processor and the user, who can view `printf()` results and type characters which are sent to the FPGA.

3.2 Automatic Interface Generation

As explained above, CoDeNios generates VHDL files implementing a protocol between all processors and hardware modules. For a Nios-to-Nios communica-

tion, no intervention of the user is required, whereas he has to write VHDL for a Nios-to-hardware communication. In this last case, a template is generated, declaring the entity and implementing a small state machine. The state machine corresponds to the protocol the developer has to respect. First, every input and output parameter of the function is declared as ports. For an output parameter, an additional port, called `load_x` (where `x` is the name of the parameter), is used to load the result value in a register outside the entity. An input signal called `start` goes to '1' for one clock cycle, indicating that the input parameters are loaded, and that the entity can start the calculation. An output signal called `done` has to be put at '1' during one clock cycle to inform an external controller that all output parameters are loaded, and that the calculation is over.

As an example, the Greatest Common Divider (GCD) function is declared like this: `void gcd(int a,int b,int *c)`. Figure 3 shows the template generated, which implements a state machine waiting for the `start` signal to be '1'. When this event occurs, it loads the value 0 in the output register of `c` and sets `done` to '1' to signify the treatment is finished. From this template, the developer only needs to change the architecture, or to map an existing VHDL file into the architecture.

```

library ieee;
use ieee.std_logic_1164.all;

entity gcd is port (
  -- input parameter
  a_in: in std_logic_vector(15 downto 0);
  -- input parameter
  b_in: in std_logic_vector(15 downto 0);
  -- output parameter
  c_out: out std_logic_vector(15 downto 0);
  -- put it at '1' to load the output
  -- parameter
  load_c: out std_logic;

  clk: in std_logic; -- clock signal
  rst: in std_logic; -- reset, '0' active
  -- '1' during one clock cycle to begin
  -- the treatment
  start: in std_logic;
  -- put it at '1' during one clock cycle
  -- when the treatment is finished
  done: out std_logic
); end gcd;

architecture struct of gcd is

  type state_type is (s0,s1);
  signal state,n_state: state_type;

begin

  process(state,start)
  begin
    -- default output values
    done<='0';
    c_out<=(others=>'0');
    load_c<='0';
    n_state<=state;

    case state is
      when s0=> -- wait for start
        if start='1' then
          n_state<=s1;
        end if;
      when s1=> -- treatment finished
        done<='1';
        load_c<='1';
        n_state<=s0;
      end case;
    end process;

  process(rst,clk)
  begin
    if rst='0' then
      state<=s0;
    elsif clk'event and clk='1' then
      state<=n_state;
    end if;
  end process;

end struct;

```

Figure 3: Generated VHDL file for GCD function

3.3 Parallelism

Regarding the C files, each original function which is chosen to be calculated by a slave (processor or hardware) is replaced by two new calls, one to start the function calculation, and one to wait for its termination. Continuing with the GCD example, `gcd(a,b,&c)` will be replaced by:

```
hcall_gcd(a,b,&c);hwait();
```

`hcall_gcd()` launches the new hardware function calculation, and `hwait()` waits for its termination and retrieves the output parameters. This call/termination splitting allows us to take advantage of the hardware parallelism. It is possible to call several independent² functions and then to wait until they are all finished. By calling the most time-consuming functions first, the total execution time can be dramatically reduced (cf. figure 4).

3.4 Execution Time Evaluation

As presented above, one important aspect of CoDeNios is its capacity to evaluate the execution time of

²Two functions are said to be independent if they are called consecutively, and no output parameters of the first are input of the second.

a hardware or software function. With this aim in view, some counters are automatically placed in the system. One global 32bit counter is directly accessible by the main processor. It is set to 0 with a soft reset of the FPGA, and counts the clock cycles. It makes it possible to evaluate the total time of different (parallel or not) function calls. A counter is attached to each co-design module, in order to evaluate the real number of clock cycles of a function execution. It does not take into account the time to pass parameters and to call the function. Its value is retrieved by the master after the output parameters.

The global counter value is accessible via a function `void GetTime(time_t *t)` and the module counters are accessible by `void GetFuncTime(int FUNCID,time_t *t)`. They are declared in an automatically generated file which contains all the procedures responsible for the co-design function calls.

3.5 Memories

As multi-processor architectures are possible with CoDeNios, several memories are used. The main processor places its executable code in the onboard SRAM of 1MB. The slaves each use only one on-chip RAM of 1KB. This limitation is due to the number

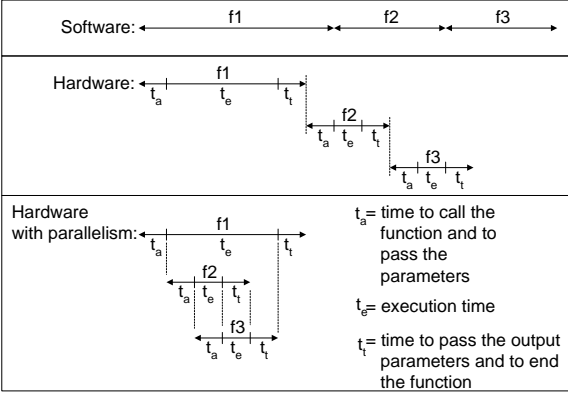


Figure 4: 3 types of executions

of embedded system blocks³ of the APEX20KE200 (52 blocks of 2048 bits). A larger RAM for each would have prevented having 3 processors on-chip. A shared memory of 1KB can be added automatically in order to pass arrays to co-design functions (by passing a pointer). It is shared between the main processor and all co-design modules. To manage this RAM, a simple arbitration is implemented, giving a different priority to each module.

4 Performance

The performance of a design made with CoDeNios depends on the hardware implementation written by the user for the hardware functions. The total execution time depends on the parameter passing time, the calling time, and the hardware calculation time. The parameter passing time is very small; a write instruction for an input parameter, and a read one for an output. On the other hand, to call and then to wait for a function costs 113 clock cycles. Because of this, the efficiency of the hardware user-defined modules is very important. One single addition would be slower by hardware, the latency of 113 clock cycles being too long, whereas a mathematical series calculation could be more efficient in hardware. Note that for an industrial purpose these 113 clock cycles could be reduced, by changing the generated C code. Currently, this code is split into different functions (one to call, and one to wait). As a software function call costs time, by putting all operations inline we could gain a lot of time. This has not been done yet, because of the C code clarity, which is important for student projects. Another way to save time would be not to allow exact calculation of hardware function

³The embedded system blocks are used to implement memories.

execution time. In the current version, this value is retrieved after the function termination. By deleting it, 4 clock cycles could be spared, but, because they allow the developer to evaluate the software solution as well as the hardware one, this deletion was not done.

Finally, the performance of a system depends on the parallelism imposed by the developer. If more than one function can be launched at the same time, the execution time can be dramatically reduced.

5 Conclusion

In this paper we presented a co-design tool called CoDeNios. This pedagogic tool helps a developer make a hardware/software partition of a C program, and generates the interface between the hardware and the software. A multi-processor architecture is also possible, sparing the user the task of interfacing the different processors.

CoDeNios, in its present state, can be used as a teaching tool. The students can rapidly test hardware modules by integrating them in a co-design system, without having to develop a protocol to synchronize the hardware and the software. To evaluate the efficiency of their hardware modules, C functions permit to retrieve counters values. Therefore it is possible to compare a software solution with a hardware one.

In the latest version of Nios (v2.0), the developer can add a user-defined module inside the core processor, a feature which overlaps a subset of the CoDeNios possibilities. This new development is interesting in that it highlights the importance of the current co-design trend that our project follows. Even though the performances of the Nios add-on are better in term of speed, our system allows for a much richer and wider range of applications. In effect, the Nios system is limited to the call of one module at a time and a maximum of two operands per module. In contrast, it is possible to implement fully parallel module calls with CoDeNios, with as many arguments (input/output) as desired, and add extra features, such as shared memory access from the hardware module. This higher flexibility and wealth of potentialities make CoDeNios a perfect tool for teaching applications.

In addition to the educational function of CoDeNios, an industrial use is possible. Having the possibility to create a complete system mixing hardware and software implies a small development time. To make this even easier, a tool to generate VHDL from C functions is currently being developed in our lab.

It will be able to transform a subset of C (if, for, while, +, -, *, /) calculating with 16 bit integers into a hardware pipeline. Integrated with CoDeNios, it will complete the automation of the system generation. The development process will also be totally automated based on the user choices.

Finally, besides the C to VHDL translation, we will add new possibilities to CoDeNios. First, functions which return an integer will be potential slave calculated functions. For instance, a developer will be allowed to use a co-design function in a conditional statement, or in an expression. Second, the function parameter size is currently fixed to 16 bits. This limitation will be removed, allowing different types of data to be sent to a co-design module.

References

- [1] P. Chou, R. Ortega, and G. Boriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *Proceedings of the International Conference on Computer Aided Design*, pages 488–495, Los Alamitos, California, 1992. IEEE Computer Society Press.
- [2] J. Henkel, T. Benner, and R. Ernst. Hardware generation and partitioning effects in the COSYMA system. In *Proceedings of the International Workshop on Hardware-Software Code-sign*, 1993.
- [3] A. Kalavade and E. A. Lee. The extended partitioning problem: Hardware/software mapping, scheduling, and implementation-bin selection. In G. De Micheli, R. Ernst, and W. Wolf, editors, *Readings in hardware/software co-design*, Series in Systems on Silicon, pages 293–313. Morgan Kaufmann, June 2001.
- [4] G. De Micheli and R. K. Gupta. Hardware-software co-design. In G. De Micheli, R. Ernst, and W. Wolf, editors, *Readings in hardware/software co-design*, Series in Systems on Silicon, pages 30–44. Morgan Kaufmann, June 2001.

How Computers Really Work: A Children's Guide

Shirley Crossley and Hugh Osborne
School of Computing & Mathematics
University of Huddersfield
Queensgate
Huddersfield HD1 3DH
U.K.
shirleycrossley@blueyonder.co.uk
h.r.osborne@hud.ac.uk

William Yurcik
Dept. of Applied Computer Science
Illinois State University
Normal
Illinois
USA
wjyurci@ilstu.edu

ABSTRACT

Current Information Technology teaching at elementary school level concentrates on teaching pupils "application skills". Very little time is spent in teaching pupils the fundamentals of "how a computer works" — computer architectures.

One source of this lacuna is the lack of suitable support material for teaching the basic concepts of computer architecture to this age group. This paper reports on the investigation, development and evaluation of a pilot computer architecture CD-ROM, aimed at 7–11 year olds.

1. INTRODUCTION

Children of today are different from those of 10–20 years ago. The majority of children going through the education system at the moment are the first generation having grown up with computers. Children need to be prepared for future employment, and to become full members of the "Information Society". They need not only to be able to use a computer, but also to understand the basics of how it works — e.g. to be able to name the major components of the machine and to understand how they affect its efficiency. However, current teaching at this level concentrates on application skills. Little, if any, time is dedicated to computer basics.

This is a missed opportunity. In a survey of teachers of this age group we found that 80% felt that children should be taught basic computer architectures. Just as children may learn a foreign language most fluently at an early age [14], so they may also be most receptive to the "foreign language" of computers at elementary school level. The difficulties that undergraduates often experience in understanding computer architectures are not due to any inherent difficulty of the subject matter, but are caused by having to "unlearn" incomplete, unrealistic and sometimes just plain weird misconceptions of how a computer works. Currently there is a shortage of useful, up-to-date information for children in the 7–11 age group on "how computers work". The majority of books and on-line resources are either outdated or too in-depth and complicated for this age group, or both.

Traditional text-based computer architecture teaching can be hard for children to digest. The use of multimedia can aid learning, as children can not only read and listen but also

watch animations of computer components, thus enhancing the learning process. They can learn at their own pace and in a non-linear fashion.

In this paper we report on a pilot interactive CD-ROM aimed at teaching the basics of computer architecture to children aged 7–11.

2. BACKGROUND

2.1 How Children Learn

John Holt in "How Children Learn" [8] writes that children do not need to be made, told or shown what to learn as they are already active learners. Children learn from hands-on experiences that involve all their senses. Early attitudes and perceptions influence a child's learning. When children are learning new information and skills, they are also developing attitudes towards learning. Children learn well through play [8, 11]. Play is the primary way that children gather and process new information, learn new skills and practice old ones. It is also important for children to be able to reflect on what they know and how they solved a problem. "All of us, not just children, learn more effectively when we are at our most "playful" . . . when we are actively participating in an enjoyable experience." [6] Games and game like learning environments can provide children with a challenge. Children delight in "new styles of learning" that can be delivered with new technologies.

2.2 How Computers Aid Learning

It has long been suggested that the computer will change how children learn. In 1966 Suppes [15] predicted that developments in educational technology, and especially computer usage, would change the face of education. He saw the computer as a tool that can be used interactively and present materials in different and novel ways. In 1981 Papert [12] also discussed the promise of classroom computers, suggesting that we can confidently allow children's minds to "develop through the exploration of computer simulated 'microworlds'".

Today these visions are closer to reality. The development of multimedia technologies offers new ways in which learning can take place. It has the potential to reduce the need for subject specific teacher expertise [13]. Schools across the world are embracing this new technology and looking for

ways to use it to enhance learning. At the same time many of these institutions are attempting to manage on small budgets, inferior technology, limited access and a teaching staff that does not have up-to-date training in computer use. A recent *British Educational Communications and Technology Agency* (BECTA) report [5] reported that only 29% of children had used the Internet at school, and only 4% had used email. This suggests that a “low tech” CD-ROM that can be used as a stand alone program on a low specification machine is more likely to be successful than a “high tech” product requiring Internet access. In order to address the problem of teachers’ lack of familiarity with the subject matter, any such teaching material must also be self explanatory, allowing children to use it without requiring too much teacher expertise.

2.3 Other Products

Although the market is flooded with educational software there is very little available in this area for this age group. In our survey a typical comment was “[. . .] terminology [is] aimed at older students”.

2.3.1 Textbook Related Products.

[17] is a book and CD package that can be used in (pre-)university education. The book is expertly illustrated for visualization of computer processes at different layers of abstraction. The interactive CD provides a tour through the interior and exterior of a computer with animated explanations, video interviews, and computing tips. However, the CD does not include all of the information contained in the book and is in fact no longer being sold bundled with the book. [17] also does not synthesize the many separate components it describes, instead leaving as a mystery how all the parts of a computer work together in unison.

2.3.2 Stand Alone Products.

Two products are in the “Techno Quest” range produced by EagleMoss Publications Ltd.: *scuz quest* is a game that covers binary code, computer memory, programs and operating systems; *bug quest* addresses integrated circuits, calculators, computer logic, the CPU and AI. Both are aimed at beginners, but have various shortcomings: “The game has to be played successfully right through in order to progress.”; “After three incorrect attempts you are thrown out.”; “The level is too low.” [3, 4]. A similar program is “KeyBytes Plus for Windows”. This is a heavily text reliant program [2]. It is therefore probably unsuitable for the target age range.

2.3.3 Internet Based Products.

The British Broadcasting Corporation gives a brief overview and explanation of each part of a computer [1], but this is totally text based and again unsuited to the age range. Another text-based source of (teacher support) material is the Computer Museum [16].

PV Insight [10] has a downloadable program explaining computer architecture. The program consists of very dense images of various parts of the computer with hotspots. When the user clicks on a hotspot a text box with very dense text appears giving an explanation of the corresponding component. This text window also covers the image so that the

user cannot view both at once. This is a very complex program that children would struggle to use and understand.

EasyCPU is an Internet-accessible, simplified CPU simulator of the Intel X86 processor family designed for pre-university education [18]. EasyCPU uses colour changes and animations to illustrate data path operations during the execution of each instruction. Results from over 2000 students measuring the effectiveness of EasyCPU indicated that an interactive and animated software tool enhances both motivation and debugging skills [18].

3. THE DESIGN

The software is designed for children aged 7–11, who have been classified as novice users. With this in mind interaction between the child and the computer is limited to mouse clicks and rollovers. Interaction between the computer and the child is through animation, sound and text. Visual aids, such as animation, are used to highlight and to distinguish between different actions and areas of information. A schematic layout of the pages is shown in Figure 1.

Navigation has been kept simple, consistent and intuitive. At all times the user can easily decide where they are, where they can go and how they can get there. Navigation incorporates the well documented “three-step rule”, where the user only needs to jump three levels to get to any point in the CD-ROM. To help with this every page contains a shortcut back to the “welcome page”.

Because of the age of the intended audience, buttons and icons include sound as well as text. As the product was aimed specifically at children all voiceovers and sound effects were provided by children from the target age group. A nine year old girl recorded the sounds naming the buttons. An eleven-year-old boy read the voiceovers for the introductory text in each section. There is a facility for toggling the voiceovers.

4. THE MAIN AREAS

4.1 The Welcome Page.

A classroom metaphor has been chosen for the main “welcome page” (see Figure 2). Rather than having a standard text menu, images of a computer and its main components are placed in the classroom. By clicking on any of these images the user will be taken to the pages explaining that component. A bookshelf containing books links to the quiz pages, a world map to the Internet section, and the blackboard to the history pages.

Children are guided throughout by “Chip”, the animated floppy disk that can be seen near the bottom right hand corner of Figure 2. An element of fun is added by making Chip ticklish — if the mouse pointer touches Chip he will laugh or tell the user to stop tickling him!

4.2 Peripherals

Each of the main peripherals — keyboard, mouse, printer and monitor has its own area in the CD-ROM. In each of these areas the operation of the peripheral device, and its role in the operation of the computer system is explained using a combination of text, images sound and animation.

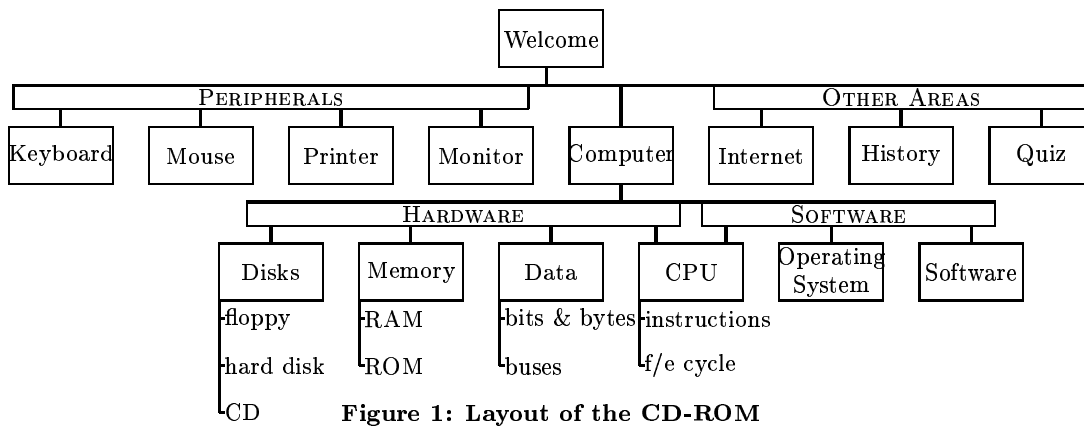


Figure 1: Layout of the CD-ROM

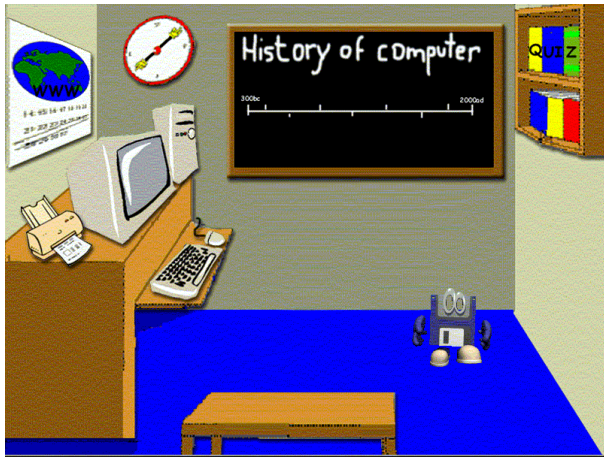


Figure 2: The Main Welcome Page

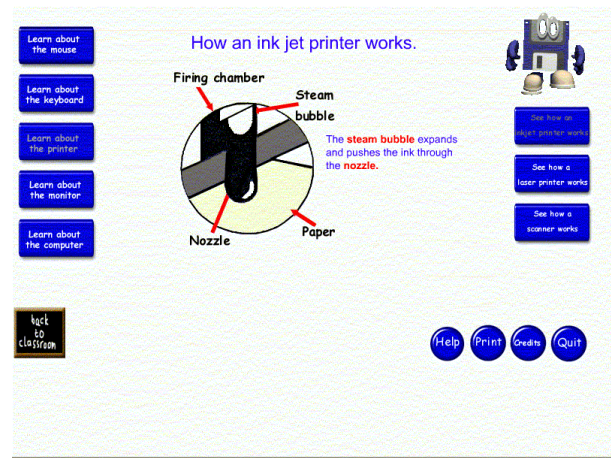


Figure 3: An Inkjet Printer

The printer area, for example, contains links to two animations illustrating the operation of laser and inkjet printers (see Figure 3, which shows a bubble of ink forming in an inkjet printer).

4.3 The Computer.

This is the largest section of the CD-ROM. The main areas here are the CPU, the Data area and Memory.

The CPU. This section is based on the Little Man Computer developed by Stuart Madnick at MIT in 1966 and described in e.g. [7], and its extension, the Postroom Computer [19, 9]. In this paradigm a “computer” consists of a walled mailroom, 100 mailboxes numbered from 00 to 99 (the memory), a calculator (the Arithmetic Logic Unit), a counter (the location counter) and input and output baskets. The animation first shows the “Little Man” explaining what each of the components does. He then explains that he has a short list of basic instructions that he can execute. Children can then choose from this list and the animation will explain the relevant behaviour. Currently there is no facility for composing your own programs within this animation, but we hope to integrate this with a simplified version of the Postroom Computer undergraduate teaching aid.

Data. The data area illustrates the (ASCII) encoding of a keypress in a byte, and shows how the byte is composed of bits. The roles of the data, control and address buses are explained.

Memory. The difference between RAM and ROM is shown metaphorically — the word ROM is shown engraved in stone, while RAM is being wiped off a blackboard. Each subsection also contains a more extensive description of the uses of each type of memory.

4.4 Other Areas

The Internet. A metaphor of posting a letter containing a request for a specific web page was used here to explain what “happens” when a web page is requested, and the problems that may arise such as time out (due to congestion or computer failure) and restricted access.

History. The history is presented as a simple time line. Children can either progress through the timeline sequentially or pick and choose time slices. Key events in the development of the computer are presented for the time slice(s) selected.



Figure 4: The Quiz

The Quiz. The quiz consists of a selection of multiple choice questions covering material from all sections of the CD-ROM. “Silly” answers are included as a fun element — see Figure 4. User involvement is also increased by adding “congratulation” and “commiseration” sounds.

5. EVALUATION

During production alpha multimedia testing — testing the functionality, interaction and navigation — was performed by a small group of children who reviewed and evaluated the product. Once the pilot CD-ROM was ready for release a group of potential users — children in the target age group and their teachers — were used for an opportunistic test. Data collection methods were by observation and semi-structured interviewing of the participants while being recorded on video.

The children were asked to work through the package and give feedback. Before starting the package there was some discussion with the children about their current knowledge of “how computers work”. The majority of the children said that they knew how a computer worked, but further questioning revealed that this meant that they knew how to use them — e.g. how to log on to the Internet — not that they understood computer architecture concepts. After using the package the children were asked for their overall impressions and whether they enjoyed this method of learning.

The quiz and animated aspects of the product were the most popular with the children. Half of the children went straight to the quiz and then realised that they had to actually go through the package before being able to answer the questions. The colours, fonts etc. were chosen with ease and speed of reading in mind. The children all enjoyed the interface and liked the classroom metaphor. The bold colours and larger fonts were popular and not considered patronising. The teachers found it bright and cheerful, and felt that the animations — especially Chip — would catch the children’s attention. All of the children said that they were happy working with a computer program of this nature. Most of them felt that it was better than traditional learning methods, though some felt that it should only be used

to complement traditional classroom learning.

Generally the product was well received by both the children and the teachers. The most successful parts of the product were clearly the animated components. There is some evidence that they aided understanding and offered a clearer explanation than is available on paper-based materials, or even achievable by a teacher in a traditional classroom situation. The children had the most difficulty in understanding the “Little Man Computer” section of the package, though some of them did successfully remember (part of) the instruction set. Some of them stated that they would prefer it if they could interact with the “Little Man Computer” — i.e. create Little Man Computer programs. This would almost certainly help to increase their understanding. As mentioned above, we hope to integrate the Postroom Computer with the CD-ROM to offer just this capability.

6. CONCLUSIONS

We have piloted a prototype of a teaching aid aimed at 7–11 year olds which allows them to explore the various components of a computer. The CD-ROM is user oriented and based on what children like and want. Interactivity is provided through animation, text and speech. It is designed to provide a game like environment and to create opportunities for children to be involved in the learning process, and to allow them to work things out for themselves. It is, of course, intended to be enjoyable and to relate to the target age group.

Our survey of teachers suggests strongly that there is a market for this type of product, and the results of tests of the pilot version are very encouraging. Children and teachers were enthusiastic about the design and content of the software.

Though our reasearch has been heavily aimed at the British market we believe that the lessons we have learned are applicable to a much wider market. We now hope to develop the CD-ROM further, extending both the breadth (the target age group) and the depth.

7. REFERENCES

- [1] BBC¹. Inside your computer. www.bbc.co.uk/education/archive/multimedia_biz/inside1.shtml.
- [2] BECTA². Review: KeyBytes for Windows, 1998. www.becta.org.uk/information/cd-roms/1998/1259.html.
- [3] BECTA². Review: Techno Quest Computer Special — Disk 1, Scuz Quest, 1998. www.becta.org.uk/information/cd-roms/1998/1359.html.
- [4] BECTA². Review: Techno Quest Computer Special — Disk 2, Bug Quest, 1998. www.becta.org.uk/information/cd-roms/1998/1360.html.
- [5] BECTA². Survey of information and communications technology in schools 1999, 2000. www.becta.org.uk/news/keyictdocs/0100survey.html.

¹British Broadcasting Corporation

²British Educational Communications and Technology Agency

- [6] D. Chandler. *Young Learners and the Micro-computer*. Open University Press, 1984.
- [7] Irv Englander. *The Architecture of Computer Hardware and Systems Software*. John Wiley & Sons, New York, 2000. Second edition.
- [8] J. Holt. *How Children Learn*. Penguin Books, 1991. 2nd edition.
- [9] Hugh Osborne. The Postroom Computer: Teaching introductory undergraduate computer architecture. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE 2002)*, 2002.
- [10] Informative Graphics Corp. PC Insight Demo. www.kidsdomain.com/down/pc/pcinsightp1.html.
- [11] V. Lee. *Children's Learning in School*. Open University, 1990.
- [12] S. Papert. *Mindstorms: Children, Computers and Powerful Ideas*. Harvester Wheatsheaf, 1981.
- [13] pjb Associates. Multimedia in education: The transition from primary school to secondary school — European Parliament STOA 1997, 2001. www.pjb.co.uk/mmeduc.htm.
- [14] David Singleton. *Age Factor in Second Language Acquisition: A Critical Look at the Critical Period Hypothesis*. Multilingual Matters, 1989.
- [15] P. Suppes. The uses of computers in education. *Scientific American*, 215, 1966.
- [16] The Computer Museum. Educational activities packet. www.tcm.org/html/resources/ed-packet/epintro.html.
- [17] Ron White, Timothy Downs, and Stephen Adams. *How Computers Work*. Que Publishing, 1999.
- [18] C. Yehezkel, W. Yurcik, and M. Pearson. Teaching computer architecture with a computer-aided learning environment: State-of-the-art simulators. In *Proceedings of International Conference on Simulation and Multimedia in Engineering Education (ICSEE)*. Society for Computer Simulation Press, 2001.
- [19] William Yurcik and Hugh Osborne. A crowd of Little Man Computers: Visual computer simulator teaching tools. In *Proceedings of 2001 Winter Simulation Conference*, New York, 2001. ACM.

Update Plans: Pointers in Teaching Computer Architecture

Hugh Osborne and Jiří Mencák
School of Computing & Mathematics
University of Huddersfield
Huddersfield HD1 3DH, U.K.
{h.r.osborne,j.mencak}@hud.ac.uk

ABSTRACT

Pointers are intrinsic to Computer Science. Each field of Computer Science seems to use its own more or less ad hoc notation for describing pointers and operations on pointers, thus impeding crossover of students' skills from one area to another.

This paper describes *Update Plans*, a “universal” pointer specification language, and its application to teaching Computer Architectures. Consistent use of Update Plans as a supplement to traditional notations can greatly enhance students' ability to apply skills learned in Computer Architecture courses to other pointer applications, and this is also illustrated.

1. INTRODUCTION

Pointers are innate to computer science in general, and to Computer Architectures in particular. Students will be confronted with pointers even at an introductory level (although possibly implicitly) in discussions of e.g. [indirect] addressing modes, and again at a more advanced level — e.g. [vectored] interrupts. Students often experience difficulty in recognising the same concepts when they encounter them in other areas of Computer Science: e.g. in data structures and in compiler construction to name just two. Each field of computer science seems to have its own notations and conventions for describing pointer structures — e.g. Register Transfer Language in computer architectures, informal diagrams when describing data structures, pseudo-code with explicit pointers in compiler construction. While this may arguably have the advantage of providing notations particularly suited to each application, it does impede a crossover of students' understanding of pointer applications and operations from one subject to another — it is quite common for students to understand the abstract notion of pointers in data structures but to have difficulty in implementing them in a high-level language, let alone relating such an implementation to addresses and indirection in low level code. What is needed is a “universal” pointer notation that can supplement, if not replace, the profusion of conventions currently in use, and which emphasises the “low-level” nature of most pointer operations while still allowing a high level of abstraction in their description.

This paper describes *Update Plans*, a “universal” pointer specification language, its application to teaching Computer Architectures, and its rôle in facilitating crossover of students' skills. The Update Plan formalism can be used to

specify a wide range of pointer applications. Abstraction mechanisms within the language allow the appropriate level of information hiding, but in contrast to informal notations the hidden information can be fully recovered from the Update Plan specification.

The remainder of this paper is organised as follows. Section 2 contains a brief introduction to Update Plans. This paper is, however, not intended as a tutorial in Update Plans, and the reader is referred to [4, 3, 5] for more information. Sections 3 and 4 concentrate on the mechanics of Update Plan descriptions in describing and teaching concrete and abstract machine architectures. Section 5 illustrates how the same formalism can be used to describe pointers in abstract data types thus encouraging students to identify the relationship between high level and low level operations involving pointers. These three sections are again not intended as tutorials in the particular pointer operations, but as illustrations of the didactic application of Update Plans. The aim of this paper is to show that Update Plan descriptions are at least, if not more suitable than traditional methods as a tool for teaching pointers. Furthermore they can be used in a wide range of areas while emphasising a Computer Architecture perspective. The overall approach to teaching pointers need not change — a change of tool, rather than a change of method is being proposed here. Section 6 summarises the use of Update Plans as an educational tool in many fields of Computer Science, and discusses possible further developments.

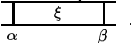
2. UPDATE PLANS

The Update Plans formalism was originally introduced [8] as a ‘target language’ in the framework of the design of a translator generator. It has since been extended for the general description of machines and algorithms, and as a tool for completion of formal proofs of program implementations [4, 5]. It has also been used as a didactic tool at various universities.

The basic concept underlying the Update Plan formalism is that of an *update* of a machine configuration, each possible update being specified by an *update rule*.

An update plan is a set of *update schemes*, each of which may contain unspecified values. A update scheme containing no unspecified values is called an *update rule*. Update schemes yield update rules by instantiation. A scheme consists of a left-hand side and a right-hand side, both being sets of

locator expressions.

A locator expression is a triple, written $\alpha[\xi]\beta$, where α and β are addresses or *locators* in one of a set of stores in an underlying machine model. Each store is a linear countably infinite sequence of memory cells (e.g. bytes or machine words). The above locator expression expresses the fact that $\alpha \leq \beta$ and that the cells between the addresses α and β contain (a particular representation of) the value of ξ . The notation of a *locator expression* is chosen such that it looks like the picture 

An update scheme states that if it is applicable (i.e. if all locator expressions in its left-hand side are satisfied), the memory may be minimally updated such that thereafter all locator expressions in its right-hand side are satisfied. The left and right-hand side of an update scheme are separated by an arrow (\implies) which optionally carries a *guard* ($\lnot \gamma \rhd$) which is an additional applicability condition.

For instance, the two-scheme update plan in Figure 1 implements Euclid's algorithm for computing the greatest common divisor of the number initially between A and B and that initially between B and C. Capitalised words denote constants: A, B and C are fixed locators and x and y are unspecified values. In fact, if at any stage of the computation the machine configuration contains $A[9]B$ and $B[6]C$, (only) the second update scheme can be instantiated to an applicable update rule, whereupon the 9 is replaced by a 3. An unspecified value on the left hand side of an update scheme can be considered a variable which obtains its value by means of instantiation.

Simple notational conventions acknowledge the existence of a *programme counter* at a (hidden) fixed locator (by convention PC), and allow the omission of irrelevant addresses and the combination of adjacent locator expressions. The update schemes in figure 2 for example, which specify the *push* and *add* instructions on some zero address machine are examples of applications of these conventions. The locator SP is the stack pointer.

The expressive power of Update Plans is greatly increased by the use of a macro-like mechanism known as *archetypes*. Using the archetype mechanism complicated pointer structures, families of such structures, and even infinite classes of arbitrarily large structures may be replaced by a single archetype call, thus making it possible to express many update schemes as one.

Archetypes are inspired by macro mechanisms. Their parameter resolution system is purely “macro” in flavour, though their expansion may be context driven, i.e. dependent on the configuration in which the macro is expanded.

An archetype definition defines a left and right hand side in the archetype *body*. When an archetype call is expanded the left and right hand sides of its body are included in the left and right hand sides respectively of the update scheme in which the call appears. There is a parameter resolution mechanism to ensure consistent replacement of archetype parameters.

An example of an archetype definition, and of a possible application is given in Figure 3. The archetype definition (1) defines a *pop* which pops the value x from the stack (addressed through the stack pointer SP). In (2) this archetype is applied in an update scheme defining an *ADD* instruction that will pop a value from the stack and add it to the value in the accumulator (ACC). Finally (3) shows the same update scheme, but with the *pop* archetype replaced by its expansion. Note that the information hidden by the *pop* archetype has now become explicit.

Larger examples of Update Plan specifications can be found in, e.g., [2, 3, 5].

3. CONCRETE ARCHITECTURES

Consideration of pointers is unavoidable when teaching Computer Architectures. Even the simplest addressing mode — direct addressing — involves a pointer. The *object* represented by a direct addressed operand is the *address* at which the *value* of that operand can be found. For example, in 68K assembler *MOVE 1234, 5678* means “copy the value at address 5678 to address 1234” — 1234 and 5678 are pointers to the data. More complex addressing modes — e.g. register indirect — can involve multiple indirections and even side effects — e.g. predecrement addressing mode. A popular notation for explaining these indirections and side effects is Register Transfer Language. An example is given in Figure 4 (adapted from [1]). While this notation is reasonably transparent, and is relatively succinct for a single opcode and operand combination, it intermingles the effects of the opcode and the addressing modes, requiring a separate description for each possible opcode and addressing mode, leading to a combinatorial explosion of definitions as the number of instructions and addressing modes increases.

The Update Plan formalism separates the definition of the opcode from the definitions of addressing modes, making it easier to teach these as separate concepts. For example part of the definition of an operand in some assembly language might be as shown in Figure 5. This defines the archetype *oprnd* with two parameters. The first parameter, *ea*, is the *effective address* of the operand; the second, *v* is the value of the operand. Line (4) defines a register indirect addressing mode, with *r* as the register identifier (e.g. R6). The locator expression $r[ea]$ states that the effective address can be found in this register, while the locator expression $ea[v]$ describes the indirection needed to find the value *v* at effective address *ea*. Line (5) provides an alternative definition of an operand, this time in predecrement addressing mode. Again *r* is the register identifier. Access to the effective address and value is similar to the previous case, except that the value in the register must be decreased (moved left across the value *v*) to give the effective address. The locator expression $r[ea]$ to the right of the guard (\implies) expresses the update of the contents of the register. These two lines would be part of a longer archetype definition defining all possible addressing modes in terms of their effective address and value.

Once the addressing modes have been defined they can be used to define the effect of opcodes. For example, the *ADD* opcode is defined in Figure 6. This definition explicitly shows the effect of instruction execution on the programme counter

Figure 1: Euclid’s Algorithm for the Greatest Common Divisor, its Implementation in Update Plans, and an Instantiation of an Update Scheme

```

while (x ≠ y) {
  if (x < y) y = y - x;
  else x = x - y;
}
return x;

```

$$\left\| \begin{array}{l} A[x]B \ B[y]C \ \Rightarrow \ B[y-x]C. \\ A[x]B \ B[y]C \ \Rightarrow \ A[x-y]B. \end{array} \right\| \left\| A[9]B \ B[6]C \ \Rightarrow \ A[3]B. \right.$$

Figure 2: PUSH and ADD Instructions Specified in Update Plans

$$\begin{array}{l} \text{PUSH } x \ \text{SP}[q] \ \Rightarrow \ \text{SP}[p] \ p[x]q. \\ \text{ADD } \text{SP}[q] \ [x \ y]q \ \Rightarrow \ \text{SP}[p] \ p[x+y]q. \end{array}$$

Figure 3: Definition, Application and Expansion of a pop Archetype

$$\begin{array}{ll} \text{(Definition)} & \text{pop}(x) = \text{SP}[s] \ s[x]t \ \Rightarrow \ \text{SP}[t]. \quad (1) \\ \text{(Application)} & \text{ADD } \text{pop}(x) \ \text{ACC}[y] \ \Rightarrow \ \text{ACC}[x+y]. \quad (2) \\ \text{(Expansion)} & \text{ADD } \text{SP}[s] \ s[x]t \ \text{ACC}[y] \ \Rightarrow \ \text{SP}[t] \ \text{ACC}[x+y]. \quad (3) \end{array}$$

Figure 4: RTL Definitions for Typical 68K Instructions

```

MOVE Di,Dj [Dj] ← [Di]
MOVE P,Di [Di] ← [M(P)]
MOVE Di,N [M(N)] ← [Di]
EXG Di,Dj [Temp] ← [Di], [Di] ← [Dj], [Dj] ← [Temp]
SWAP Di [Di(0:15)] ← [Di(16:31)], [Di(16:31)] ← [Di(0:15)]
LEA P,Ai [Ai] ← P

```

Figure 5: Definition of an Operand in Update Plans

$$\begin{array}{ll} \text{oprnd}(ea, v) & = \dots \\ & = \text{REGIND } r \ r[ea] \ ea[v] \ \Rightarrow \ . \quad (4) \\ & = \text{PREDEC } r \ r[b] \ ea[v]b \ \Rightarrow \ r[ea]. \quad (5) \\ & \vdots \end{array}$$

Figure 6: Definition of a Two Address ADD Instruction

$$\text{PC}[pc] \ pc[\text{ADD } \text{op}(ea_x, x) \ \text{op}(ea_y, y)]qc \ \Rightarrow \ \text{PC}[qc] \ ea_x[x+y].$$

Figure 7: Alternative Definition of a Two Address ADD Instruction

$$\text{ADD } \text{op}(ea_x, x) \ \text{op}(ea_y, y) \ \Rightarrow \ ea_x[x+y].$$

(PC). The notational conventions mentioned in Section 2 also allow an alternative form as shown in figure 7 in which the emphasis is on the functionality of the instruction.

By separating the definitions of the opcodes from the definitions of the addressing modes not only has the specification of the instruction set become much more compact and manageable (a simple two address machine with only 16 opcodes and 8 addressing modes which would require 1,024 (16 × 8 × 8) RTL definitions for a full specification only needs 24 (16 + 8) lines in an Update Plan specification) but also it becomes much easier to teach these two concepts independently and in an incremental fashion. Note that the order of presentation of these elements of the instruction set would typically be reversed when presenting them to students. The opcodes can be introduced using a reduced set of addressing modes (e.g. only direct address) as shown in Figure 8, and only when students have mastered this reduced instruction set will the full set of addressing modes be introduced. This

approach can be reinforced by the use of a suitable assembler emulator, such as the Postroom Computer [6, 10], which supports this incremental approach. The Postroom Computer is presented to students using the Update Plan formalism to reinforce informal descriptions of the machine. It also uses the Update Plan formalism to describe its internal state when students trace execution of Postroom Computer programmes. Experience has shown that students soon master an (informal) understanding of the meaning of Update Plan notation.

4. ABSTRACT ARCHITECTURES

Pointers are also unavoidable when teaching abstract machine architectures, whether the machine is for a procedural, functional, logical or object oriented language. This section presents an example from an implementation of a functional language. A functional language implementation has been chosen because the data structures in implementations of functional languages are typically small, and very limited in

Figure 8: Definition of a Simple Two Address ADD Instruction

$$\text{ADD } ea_x \ ea_y \ ea_x[x] \ ea_y[y] \implies \ ea_x[x + y].$$

number. The methods illustrated here could also be applied to more complex abstract machines.

Peyton Jones [7] describes function evaluation by graph reduction using pointer reversal by a combination of informal diagrams and a rather clumsy notation for describing the pointer reversal itself, which uses the implicit function ‘Left’ which hides an essential indirection, requires inspection of the corresponding diagram to show that the operation can only be applied if the forward pointer ‘F’ points to an *application node*, and needs an explicit statement of the simultaneity of the components of the pointer reversal operation (see Figure 9 — adapted from [7]). Not only can pointer reversal be expressed much more succinctly in Update Plans (see Figure 10), but all of the hidden information in Figure 9 is now explicitly present in the description.

Note also that the implementational structure of an application node is much clearer in the Update Plan specification — an application node being a structure containing a constant (APP) identifying it as an application node, and two pointers (c and d on the left hand side, b and d on the right hand side).

Relatively simple Update Plan specifications of the other operations involved in graph reduction can be given, supplementing an informal diagrammatical explanation, and allowing students to experiment with the implementation. A prototype implementation of (a subset of) Update Plans is currently being used in an advanced level course on the implementation of functional languages. Students can, for example, be given Update Plan implementations of standard graph reduction operations and be asked to develop a λ -calculus to graph expression compiler, or they can be asked to develop a compiler from λ expressions to Update Plan specifications of supercombinators.

It should again be emphasised that Update Plan specifications are not proposed as a *replacement* for informal notations, but as a *supplement*. The advantages are threefold. Update Plans have a formal semantics, making specifications precise. The existence of an implementation (albeit currently limited) makes it possible for students to experiment with the construction of graph manipulation primitives. In addition, the students following this course have encountered Update Plans earlier in their studies in an introductory course in Computer Architectures, making the crossover of skills easier.

5. ABSTRACT DATA TYPES

Data structures are another area of Computer Science where pointers are rife — both explicitly in e.g. lists, stacks, queues, trees etc., and implicitly in arrays, records, structures, objects, etc. This is illustrated here by using Update Plans to describe binary trees and operations on them. This example shows how the *single rotate left* operation in AVL trees can be defined using (only) Update Plans. The most common way of explaining such structures and operations is by a combination of informal diagram and (pseudo) code, as

shown in Figure 11 (adapted from [9]). In the *single rotate left* operation an unbalanced node having no children on the left, but both a child and a grandchild on the right is balanced by promoting the right hand child to the root node, with the original root node as the new root node’s left hand child.

In Update Plans the data structure can be defined as an archetype. Figure 12 contains an example defining a binary tree. The first archetype (6) defines the *abstract* structure of a binary tree. This archetype can be read as: “A binary tree is either the empty tree, or a node containing a key and two subtrees”. Archetypes (7) and (8) then define the *concrete* structure of binary trees, defining the empty tree to be the NULL pointer, and a node to be a pointer to a data structure containing a key, and two pointers to the node’s subtrees. A definition of the single rotate left operation is given in Figure 13. Note that this definition can be read as the textual representation of the tree diagram shown in Figure 11. In other words, this definition of the operation *implicitly* contains the pointers. In contrast to the usual style of explanation of the operation as shown in Figure 11 there is, however, no need to change the representation to make the pointers explicit — simple expansion of the `node` archetypes suffices, as shown in Figure 14. It should be emphasised that this is the *same* update scheme as in Figure 13, only after archetype expansion. The abstract data structure in Figure 13 has been transformed into a concrete representation using pointers while staying in the same paradigm.

6. CONCLUSIONS

The previous three sections have demonstrated the application of Update Plans to teaching Computer Architecture. By using a unified notation that is not only applicable to teaching Computer Architectures, but that is also suitable for describing pointer applications in other areas of Computer Science crossover of students’ skills and understanding throughout the curriculum is greatly facilitated. Also, students often find it difficult to relate the high level concept of *pointers* to the low level concept of *addresses*. By using the same formalism to describe both the relationship is made explicit, strengthening students’ understanding of pointers from a Computer Systems Architecture perspective.

Update Plans cannot completely replace the other methods discussed here, especially informal graph diagrams — a picture is, after all, worth a thousand words, or even Update Plans — but the formalism should be used to complement and unify explanations of the rôles of pointers in Computer Science and to emphasise the low level nature of most pointer operations.

The Update Plan formalism has been used successfully as a descriptive tool in teaching introductory Computer Architectures. An implementation of a subset of Update Plans is also available, and this is being used to provide more advanced students with hands-on experience — designing and implementing their own instruction sets, and building and using an abstract intermediate code machine.

Figure 9: Pointer Reversal in an Implementation of a Functional Language

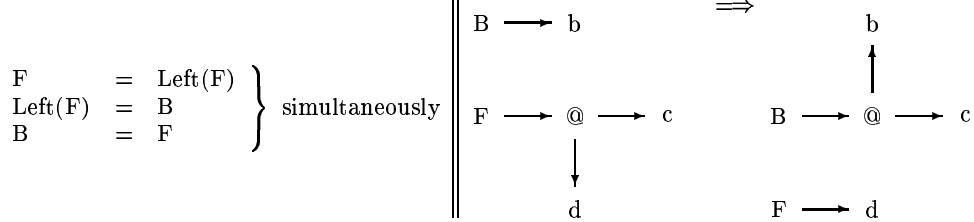


Figure 10: Update Plan Specification of Pointer Reversal

$$F[f] B[b] f[APP c d] \Rightarrow F[c] B[f] f[APP b d].$$

Figure 11: Single Rotate Left in an AVL Tree

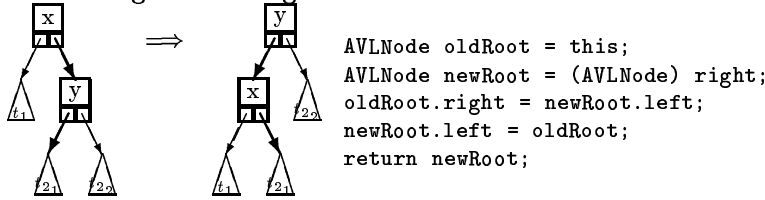


Figure 12: Defining a Binary Tree in Update Plans

$$\begin{aligned} \text{tree}() &= \text{empty}(). \\ &= \text{node}(\text{key}, \text{tree}(), \text{tree}()). & (6) \\ \text{empty}() &= \text{NULL}. & (7) \\ \text{node}(\text{key}, \text{left}, \text{right}) &= \text{a a[key left right]}. & (8) \end{aligned}$$

Figure 13: The Single Rotate Left Operation on AVL trees

$$\begin{aligned} \text{ROL oldroot}(x, \text{tree}()_1, \text{newroot}(y, \text{tree}()_{2_1}, \text{tree}()_{2_2})) \\ \Rightarrow \text{newroot}(y, \text{oldroot}(x, \text{tree}()_1, \text{tree}()_{2_1})). \end{aligned}$$

Figure 14: The Update Scheme from Figure 13 after Expansion of the node Archetypes

$$\begin{aligned} \text{ROL oldroot oldroot[x tree}()_1 \text{ newroot]} \text{ newroot[y tree}()_{2_1} \text{ tree}()_{2_2}] \\ \Rightarrow \text{newroot newroot[y oldroot tree}()_{2_2}] \text{ oldroot[x tree}()_1, \text{tree}()_{2_1}]. \end{aligned}$$

7. REFERENCES

- [1] Alan Clements. *The Principles of Computer Hardware*. Oxford University Press, 2000.
- [2] Hugh Osborne. The semantics and syntax of update schemes. In *Code Generation — Concepts, Tools, Techniques*, Workshops in Computing. Springer Verlag, 1992.
- [3] Hugh Osborne. Update Plans. In *Proceedings of the 25th Hawaii International Conference on System Sciences*. IEEE Computer Society Press, 1992.
- [4] Hugh Osborne. *Update Plans — A High Level Low Level Specification Language*. PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1994.
- [5] Hugh Osborne. Update Plans for parallel architectures. In M. Kara, J.R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 79–90, Amsterdam, 1996. IOS Press.
- [6] Hugh Osborne. The Postroom Computer: Teaching introductory undergraduate computer architecture. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE 2002)*, 2002.
- [7] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
- [8] Hans Meijer. *Programmar: A Translator Generator*. PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1986.
- [9] Russel Winder and Graham Roberts. *Developing Java Software*. John Wiley & Sons, 1998.
- [10] William Yurcik and Hugh Osborne. A crowd of Little Man Computers: Visual computer simulator teaching tools. In *Proceedings of 2001 Winter Simulation Conference*, New York, 2001. ACM.

CASTLE: COMPUTER ARCHITECTURE SELF-TESTING AND LEARNING SYSTEM

Aleksandar Milenkovic^a, Bosko Nikolic^b, Jovan Djordjevic^b

^aElectrical and Computer Engineering Dept., University of Alabama in Huntsville

^bComputer Engineering Dept., School of Electrical Engineering, University of Belgrade

E-mail: {milenska@ece.uah.edu, nbosko@etf.bg.ac.yu, jrdjrdjevic@kiklop.etf.bg.ac.yu }

Abstract. *The paper introduces the CASTLE, a Web-based system for Computer Architecture Self-Testing and LEarning. The CASTLE offers self-testing and learning facilities meant to be used by students at home and/or in lab in the process of studying and exam preparation. It also offers a rich set of facilities to help the system administration and to provide feedback to instructors. The core of the CASTLE tool is developed using zero-cost environment, in such a way that it could be easily modified and used for teaching other courses.*

I INTRODUCTION

The Internet has dramatically changed the way instructors teach Computer architecture and organization, and the way students learn. Modern software tools enable the development of Web-based graphical animations to illustrate complex topics [1], advanced computer architecture CAD tools have become available via Web browsers [2], and collections of course material including tests and exams can be shared between instructors [3].

Web-based testing plays an important role in distance learning. For example, the IEEE Computer Society has recently started to offer to its members various courses as a part of the Distance Learning Campus [4]. We feel that “classic” classroom- and lab-based courses could also benefit from the opportunity of online testing and self-assessment and that is why we are building the CASTLE, a Web based software system for Computer Architecture Self-Testing and Learning.

Previously we made some efforts in the similar direction by developing the CALKAS [5]. However, it uses rather expensive commercial environment, and it is primarily targeted to the assessment of the student knowledge during labs.

The CASTLE offers the students an opportunity for online testing on various topics in computer architecture and organization, anytime, anywhere. Using this tool, students can continuously reinforce their classroom learning, and can get valuable feedback

about their course advancement. The CASTLE allows students to choose the level of testing. At the beginning they can start with elementary questions, and as they progress through the course they choose more complex tests at the medium and advanced levels. Each question is tagged with an explanation field, which includes a full explanation or a link to the corresponding textbook or material on the Web.

The CASTLE allows instructors a Web-based administration by using simple forms to insert, edit, or delete information about students and questions. In addition to that, the CASTLE can generate various statistics from the database providing the instructors with valuable feedback about students’ advancement. Using these statistics, instructors can identify what is difficult for students to grasp. Often instructors have groups of students with different background and inhomogeneous knowledge. In such cases the CASTLE should help those with insufficient prerequisites to catch up. Thanks to explanations it provides, the CASTLE as a “virtual instructor” could improve the overall quality of teaching since it gives the instructor more time to spend on difficult topics.

The CASTLE is developed using Java Servlet/Java Server pages technologies and MySQL as a database. We have developed and tested the core of the CASTLE and now we are building the database with questions. The rest of the paper is organized as follows. In Section 2 we describe the facilities offered by the CASTLE. Section 3 gives a short overview of the CASTLE internals. Section 4 concludes.

II USING CASTLE

The CASTLE offers two levels of functionality:

- At the user level, it provides self-testing facilities to students, and
- At the administrator level, it provides administration facilities to instructors.

The user level

The first step in working with CASTLE is to login: a user enters her/his username and password, and

activates **Login** button (Figure 1). The system checks whether a user with that username exists in the database of users and whether the password is correct. If the login is successful, the system allows access to self-testing mode, and the Welcome screen appears (Figure 3). New users are asked to register first (Figure 2).

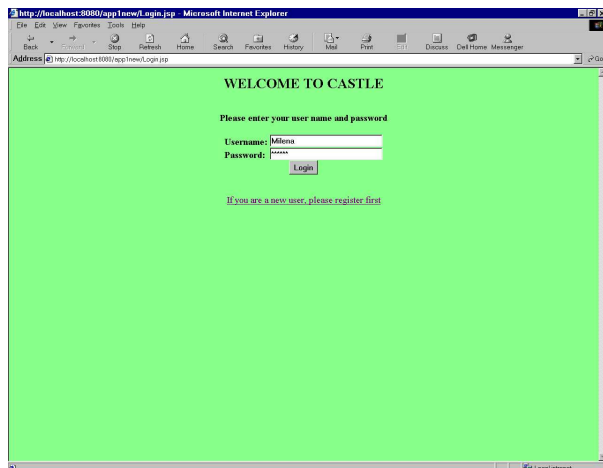


Figure 1. Login screen.

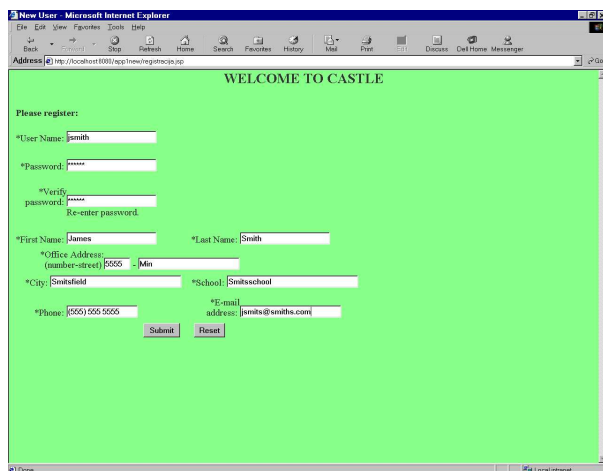


Figure 2. Register screen.

The Welcome screen offers the user possibility to select the type of testing; the current version of the CASTLE supports the following types: Comprehensive, Processor Architecture, Memory Hierarchy, and Multiprocessors. The user also defines test duration (test time per question), the number of questions in the test, and the level of testing. The CASTLE currently supports three levels of testing: *Elementary*, *Medium*, and *Advanced*.

The test is then activated using the **Start test** button. The CASTLE generates randomly requested number of questions with offered answers from the database; all questions generated have the same

difficulty tag (Elementary, Medium, or Advanced). The questions appear one by one. For each question the remaining time is counted-down in real time and displayed on the screen (Figure 4). Questions may include graphical content. The user answers the questions by activating the appropriate check box in front of the answer deemed to be correct.

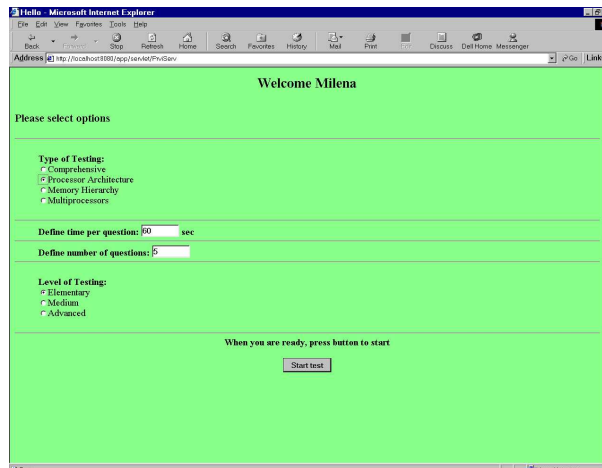


Figure 3. Welcome screen.

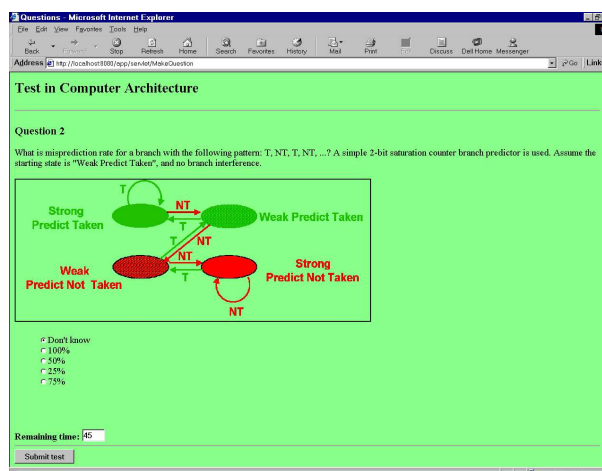


Figure 4. Test screen.

When the user has completed a question, even if the time predetermined for giving an answer has not yet expired, she/he can submit the test by activating the **Submit test** button. If the test has not been submitted within the predetermined period of time, the CASTLE stops the testing when the time expires and the user is asked to submit the answer. The CASTLE checks correctness of the given answer and generates a result screen including the question, given and the correct answer, and explanation for the correct answer (Figure 5). By activating the **Next Question** button the test continues.

The information concerning the completed test, such as the user's identification number, the date, the time, the generated questions, and the given answers, are saved in the appropriate database tables. Hence, any relevant information concerning all tests taken by any user can be obtained at any time.

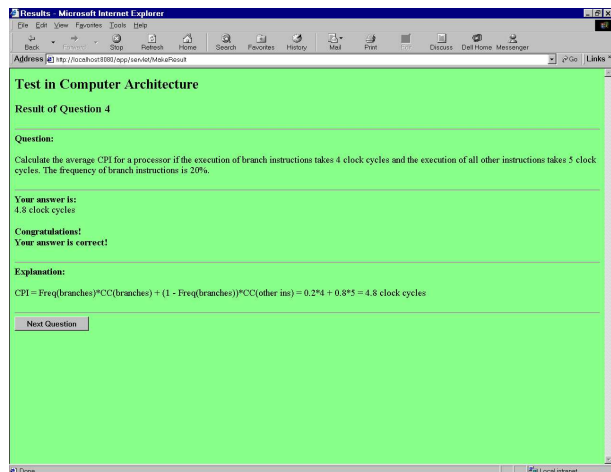


Figure 5. Result screen.

When the user has completed the test she/he can get the final test report. This report contains the score and a table with all questions from the test, the answers given and the correct ones. At the end of the test, the user can start new test session by activating **New test** button.

The administrator level

At the administrator level instructors use the CASTLE to maintain the database including information regarding users, questions and offered answers, and test sessions. The CASTLE provides simple forms that can be used to enter new questions and update the list with answers, modify the list of offered answers, add, edit, and remove users. In addition to that, the CASTLE allows instructors to generate and print the itemized reports including statistics - number of tests taken, percentage of correct answers, etc., for each topic (e.g., Memory hierarchy), and for each question. Finally, the CASTLE allows instructors to backup the whole database.

The first step for an instructor is to login by entering administrator username and password; Administrators use the same Login screen as ordinary users (Figure 1). After a successful login the Welcome administrator screen appears (Figure 6). From this screen the administrator can select any of the available functions: **Insert User** to add a new user, **Edit User** to edit information about a user, **Delete User** to remove a user from the database of users, **Insert Question** to add a new question in the database, **Edit Question** to

relevant fields of a question, **Delete Question** to remove a question from the database, **Define Queries & Printing Reports** to create and print various reports, and **Backup** to backup the database.

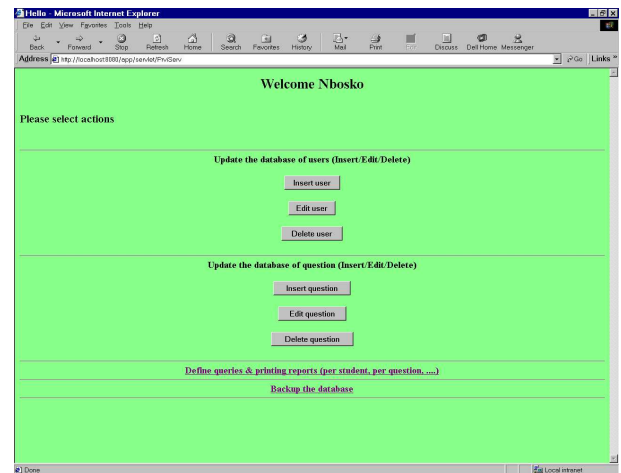


Figure 6. Administrator Welcome screen.

Insert Question and **Insert User** buttons bring screens containing all relevant fields to be defined for a new question and a new user, respectively. Figure 7 shows the form for entering a new question. The instructor enters relevant fields such as the text of the question, offered answers (up to four possible answers), Id for the correct answer, Id for the area, the level of difficulty, and the explanation. All fields are checked for consistency before the database is updated by activating **Submit** button.

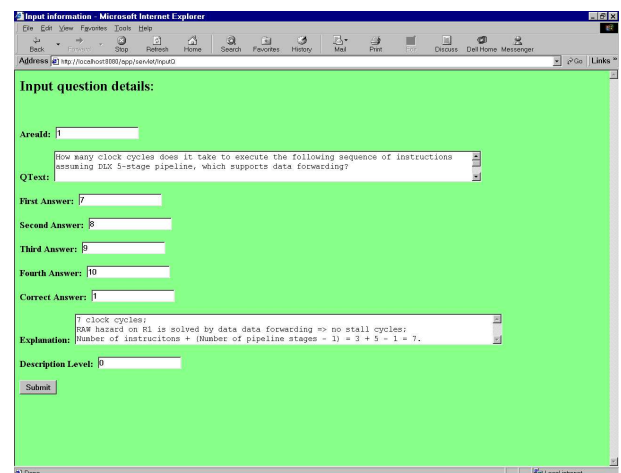


Figure 7. Insert Question screen.

Edit Question and **Edit User** forms require the questionID and username to be entered by the instructor, respectively. Complete record appears on the screen, and all fields can be changed. Changes become visible by activating the **Submit** button. Similarly, the instructor can remove a user or a

question using **Delete User** and **Delete Question** forms.

By activating **Define Queries & Printing Reports** button instructor opens a new form where she/he selects a query, such as global statistics, and statistics per area, per question and per user. The result screen will contain required information, including statistic charts (Figure 8). The instructor prints the report by activating the **Print** button.

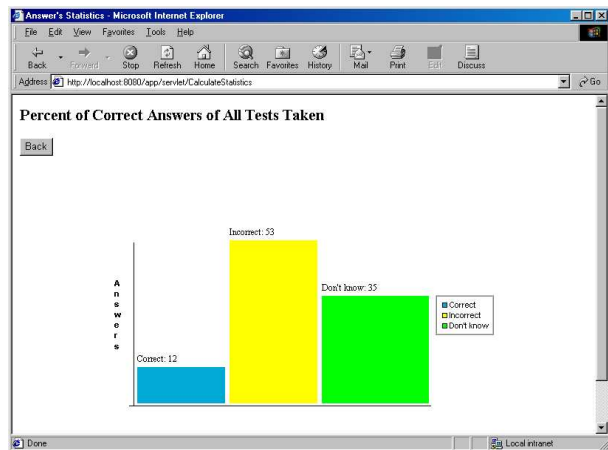


Figure 8. Statistics screen.

III INTERNALS OF THE CASTLE

Primary requests for the development environment were to support all facilities of the CASTLE as well as to minimize the cost. We use a zero cost environment based on Java Servlet and JavaServer Pages (JSP) technologies [6]. As a Web server we use Tomcat [7], a free open-source implementation of Java Servlet and JavaServer Pages technologies developed under the Jakarta project at the Apache Software Foundation. We use MySQL [8], a free, open source database available for many computing platforms. It represents the most affordable solution for relational database services available today. For communication between Java servlets and the database we use, a free JDBC driver mm.mysql-2.0.4-bin.jar [9].

Figure 9 shows the development environment and illustrates the data flow. We decided to implement a rather simple graphical interface, so the CASTLE can be accessed without any delay even over 56K modem connections.

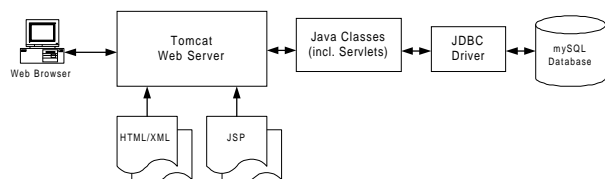


Figure 9. Development Environment.

IV CONCLUSION

This paper introduces the CASTLE, a Web-based system for testing in computer architecture and organization. It allows students to test their knowledge continuously throughout the course giving them full control over the number of questions they want to take, test difficulty, and course topics. In addition to that the CASTLE facilitates the system administration and provide valuable feedback about course advancement to instructors during the course. The development environment guarantees simple user interface, flexibility, and security of data, availability, maintainability and upgradeability.

The primary short-term goal is to expand the current number of questions and to support different question forms in addition to multiple choice. The next step will be to open the gates of the CASTLE to broader community, to all interested to improve their knowledge in computer architecture.

ACKNOWLEDGEMENTS

This work is partially supported by the UAH Provost's office through an Instructional Grant.

REFERENCES

- [1] J. Djordjevic, A. Milenkovic, N. Grbanovic, "An Integrated Environment for Teaching Computer Architecture," *IEEE Micro*, Vol. 20, No.3, pp. 38-47, May/June 2000.
- [2] N. Kapadia, R. Figueiredo, J. Fortes, "PUNCH: Web Portal For Running Tools," *IEEE Micro*, Vol. 20, No. 3, pp. 38-47, May/June 2000.
- [3] E. Gehringer, T. Louca, "Using the Computer Architecture Course Database," *IEEE TCCA Newsletter*, pp. 85-89, September 2000.
- [4] <http://www.computer.org/DistanceLearning>
- [5] J. Djordjevic, A. Milenkovic, I. Todorovic, and D. Marinov, "CALCAS: A Computer Architecture Learning and Knowledge Assessment System," *IEEE TCCA Newsletter*, pp. 26-29, June 2000.
- [6] <http://java.sun.com/j2se/>
- [7] <http://jakarta.apache.org/site/binindex.html>
- [8] <http://www.mysql.com/>
- [9] <http://mmysql.sourceforge.net/>

Development of a digital instrument as a motivational component in teaching embedded computers

Gracián Triviño¹, Felipe Fernández²

¹Universidad Politécnica, Madrid, Spain, gtrivino@fi.upm.es

²Universidad Politécnica, Madrid, Spain, felipe.fernandez@es.bosch.com

Abstract

Nowadays it is frequent that, at the first levels of Computer Engineering studies, the students have acquired some practical experience developing software projects. However, they have little or none experience facing up the project of designing and developing a simple computer electronic module.

On the other hand, due to the strong development of the area and its continuous presence in the media, students are especially motivated towards robotics and, in general, towards systems that can interact with the physical environment.

According to these circumstances, this paper describes the strategy followed to introduce a new subject denominated “Digital Instrumentation and Data acquisition”. This subject is an optional part of the curriculum area dedicated to Computers Architecture in the Faculty of Computer Engineering at the Polytechnic University of Madrid (Spain).

1. Introduction

Two years ago, we faced up the challenge of designing the layout of a new subject in the area of Computers Architecture. In teaching embedded computers, it is especially clear that the goals of teaching are not only a set of theoretical concepts. Together with them, you need to teach practical procedures, and to teach attitudes encouraging the students to develop a personal interest in studying the topics related with the subject.

Considering the characteristics of the current curriculum, the students acquire during the first courses some capabilities to develop software projects while they have little or none experience designing and developing computer electronic circuits. One elemental rule in teaching consists of using the available knowledge in the students’ minds as the basis to build the new knowledge structures. Therefore, a first idea was to find a way of using this capability for software development as one of the basis of the new subject structure.

On the other hand, the new subject was going to be one more in the set of optional subjects that are available for the students choice. Then it was necessary to think about special motivations, the marketing strategy that

would encourage the student to introduce our subject in his/her course configuration.

During last years, an increasing interest among students on topics related with robotics has been detected. More specifically, students are interested in computers that are able to interact with the physical environment. Therefore, a second idea was to use this interest as a motivation to convince the students to enrol in our subject.

We decided to name this undergraduate course “Digital Instrumentation and Data acquisition”. Under this denomination, our intention was to cover a space detected in our curriculum of the Computers Architecture area. A classification of the didactical contents of the new subject is the following:

Theoretical contents

- Different types of sensors making emphasis in the applied physical principles.
- Differential amplifiers, instrumentation amplifiers, A-D converters.
- Instruments, instrumentation systems, standard instrumentation platforms (GPIB, VXI, PXI)
- Instrumentation languages, instrumentation software environment.

Practical contents

- Design and building of an electronic circuit for data acquisition based in microcontrollers.
- Design and building of electronics circuits to handle the signal provided by different sensors.
- Design and building of software programs handling this hardware to create digital instruments.

The complete subject program can be found in the subject Internet web page [1]. The remainder of this paper is aimed to describe different aspects of the resources developed to support the teaching of the practical contents.

2. Practical Project

As a part of their learning activities, the students of “Digital Instrumentation and Data Acquisition” must develop a *practical project* that consists of building a digital instrument.

Through the analysis, design and building of a digital instrument the student learns practical knowledge that

complements the corresponding theoretical concepts and procedures.

When we faced up the development of some resources to support the students during the *practical project* development, we considered some special requirements having in account not only a teaching strategy but also an adequate marketing strategy:

- The student must have the possibility of using his/her own personal computer (PC) as an important component of the practical project.

The idea is to use the fact of that most of the activity of students developing software is currently performed using their personal computer.

- The whole hardware of the *practical project* must be compact and small enough to be portable.

This is not only, because that could make more comfortable the project development, but also, because the students' interest in this type of practical projects allows using the project itself as a marketing reclaim. The student will carry the circuit board with him/her having the possibility of talking about it and showing it to fellows.

- After the *practical project* is finished, the system must be able to be used for new projects. Consequently, all the resources used must remain available to the student. If the *practical project* hardware is of student's property, it will remain available not only for other projects at the university but also to develop his/her own home projects. To make this possible, it is desirable to maintain the whole project as low cost as possible.

3. Hardware resources

The main support of the *practical project* is a printed circuit board that we have called iFOTON. This circuit contains a microcontroller with few additional electronic devices and a free mounting area where it is possible to solder additional components. Figure 1 shows a iFOTON block diagram.

This circuit includes all that is necessary to make easy the development of an electronic digital instrument. Figure 2 shows a photo of the PCB with all the electronic components mounted. A brief description of iFOTON main features is the following:

Microcontroller

The PIC16F873/76 microcontroller from Microchip [2] has been chosen. This family of devices has an excellent cost-benefit rate and its use in the market has been growing up during the last years. This microcontroller has a set of interesting characteristics that we are only to enumerate: It has a RISC architecture, 4K of flash program memory, 193 bytes of RAM, 128 bytes of EEPROM. Moreover, the encapsulated peripheral devices are: A/D converter of 10 bits, 3 timers, PWM modules, communications modules (USART, I2C) and 3 parallel digital input output ports.

Serial port

It is the main mechanism of communication between iFOTON and the PC. The USART provided by the microcontroller has been used. A Maxim MAX232 serial signal adapter has been used in order to convert the TTL signal provided by this peripheral to the RS232 protocol logical level [3].

Programming port

An important advantage of the selected microcontroller is the so-called "In Circuit Programming" characteristic [4]. This allows, using only a 5 volts power supply and handling a limited set of connections (4), to read and write the microcontroller program memory without extracting it from the circuit socket.

The PC standard parallel port is used to handle these connections. This port fulfils the IEEE 1284 (SPP, EPP, ECP) standard signalling method for a bi-directional parallel peripheral interface for personal computers [5]. To make the connection the student must build the adequate cable following the instructions provided with the iFOTON documentation.

Input output port

The microcontroller provides three configurable input-output ports that have been situated near the free mounting area:

- All of them can be configured as digital inputs. They have associated internal programmable pull-ups and the possibility of associating encapsulated devices to interrupt processing and to perform pulse wide measurement.
- All of them can be selected as digital outputs. They have 25 mA sink/source current capability.
- Five pins of them can be configured as analogical

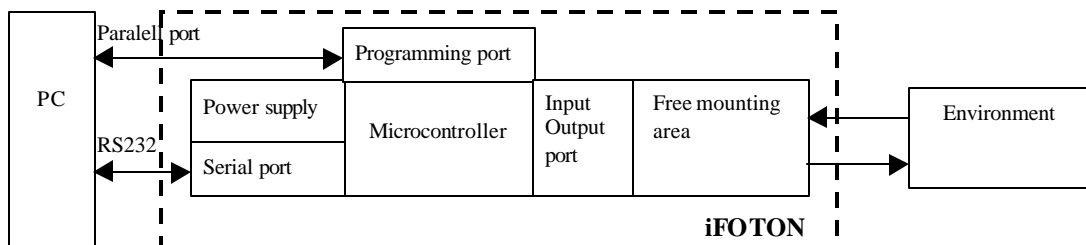


Figure 1. iFOTON Block diagram

inputs: These inputs share the multiplexed A/D converter.

Power supply

iFOTON uses the DC power supply that is generally provided with the PC loudspeakers. This device has a jack connector that provides 9 volts where the central connection is connected to ground. Then a simple 7805 voltage regulator and two capacitors are enough to complete the circuit power supply.

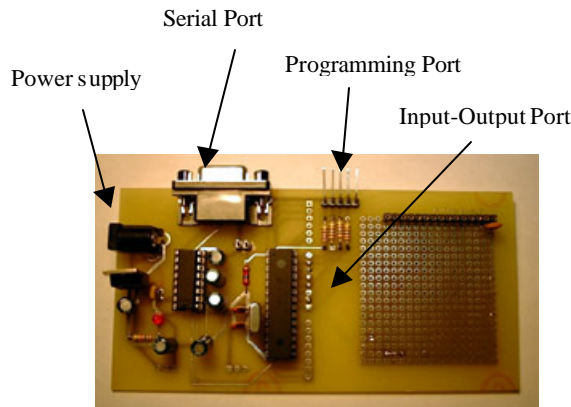


Figure 2. iFOTON PCB

4. Software resources

A set of software tools has been developed to support the design and building of *practical projects* with iFOTON: a Programmer, a Project configuration control, and an Interpreter of commands. The first two of them are executed in the student PC and the third one in the iFOTON microcontroller.

Code Programmer

This is a software tool that converts iFOTON in a microcontroller programmer [6]. This program uses the connection between the PC and the iFOTON Programming Port to allow the user reading and writing the microcontroller program memory. The object file to be programmed must be HEX formatted [7]. This format is generated for most of the assemblers and compilers available in the market.

Project Configuration Control

Together with the Code Programmer, a simple Project Configuration Controller is provided. This program allows creating a files structure for every *Practical Project*. It creates in the PC disk a directory with the project name containing a set of mandatory files: Requirements, Design, Diagrams and Software source. The Project Configuration Controller asks the user to associate these files with the corresponding software tool used to generate them.

Using iFOTON a fairly amount of different *practical projects* could be developed. The idea of this tool is to get a standardised set of documents for all these projects. On the one hand, this will make them easier to be analysed and evaluated by the teacher, and on the other hand, this will help to build a projects database easier to reuse.

Interpreter of commands

Taking into account the availability of the microcontroller programmer, a first approach in order to build a digital instrument could be to develop software to be loaded in the microcontroller memory. This allows building a stand-alone instrument. However, as mentioned above, the possibility of allowing the student to develop software to be executed in his/her personal computer is desirable. This avoids the need of knowing details about the microcontroller programming features. In order to make this possible, an Interpreter of Commands, that has been called iF, has been designed and implemented. This software, loaded in the microcontroller memory, allows managing the iFOTON input-output ports using commands, which are sent and received via the serial port.

<i>Header</i>	“\$”
<i>Command code</i>	Access to ports is performed using two characters: First: “L” means Read, “E” means Write Second: “D” means Digital, “A” means Analogical There are other special commands: “S” means Scanner “M” means Stepper Motor control Etc.
<i>iFOTON address</i>	This allows connecting several iFOTON systems to the same serial bus
<i>Port address</i>	Meaning the specific pin concerned
<i>Data</i>	Used with writing commands
<i>Final code</i>	“,”

Table 1. Commands format

To simplify the design of the Interpreter, a set of restrictions have been introduced assigning specific functions to every iFOTON Input-Output Port. The port A, pines 1 to 5 in the iFOTON Input-Output Port, is used as analogical input. The port B, pines 6 to 13, is used as digital input. And, port C, pines 14 to 22, is used a digital output.

Therefore, by executing the iF interpreter in iFOTON and running a terminal emulator as the Microsoft HyperTerminal in the PC, we can use the keyboard to send commands and the screen to read the answers.

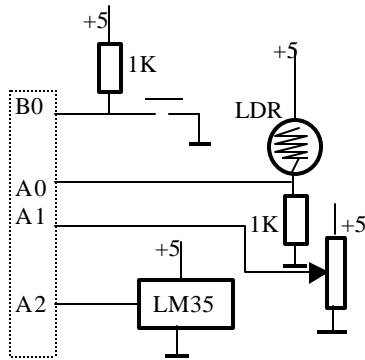


Figure 3. Sensor connections

A command is a string of ASCII characters with the format described in table 1.

The following are some examples of using the iF commands:

“\$LA00;” reads the analogical value of voltage in pin 0 of port A. The answer could be \$0127; a value in the range 0 -1025 meaning a value in the range 0 - 5000 mV.

“\$LD02;” reads the digital value in pin 2 of port B.

The answer could be \$0; meaning a low digital TTL level input.

“\$ED050;” writes the digital value 0 in the pin 5 of port C.

As far as commands that are more specific are considered useful, they are implemented expanding in this way the iF interpreter possibilities.

5. Documentation

Most of the documentation that is required to undertake the *practical projects* is provided to the students using the iFOTON web page in Internet [8].

The aim is to provide not only the necessary information to allow the students to solve their projects but also to help the advanced students to get detailed information and to explore new possibilities of use.

This set of documents includes iFOTON description, iFOTON building manual, iF commands description and some examples of *practical projects*. The hyper-textual documentation includes links to the main company providers of the electronic components used. Logically this information must be in continuous evolution.

6. Examples of practical projects

6.1. Environment data acquisition

This *Practical Project* consists of representing in the PC screen the data obtained from the following sensors: a push button, the position of a potentiometer, the room temperature, the light intensity in the room, and the atmospheric pressure.



Figure 4. Instrument user interface

All the necessary electronic components can be allocated in the free soldering area. The chosen temperature sensor was the LM35 from National that can be connected directly to an analogical port. A LDR sensor was used to measure the light intensity. Figure 3 shows the simple circuits that must be mounted in the free soldering area to handle these sensors.

The MPX2200A sensor from Motorola was chosen for absolute pressure sensing. The provided signal by this sensor needs to be amplified. The students must analyse different instrumentation amplifier circuits [9] and build one of them to handle this sensor signal.

Once the hardware components have been mounted, all the measures can be obtained directly using iF commands.

The software part of this *Practical Project* is aimed at learning how to program with a specific software tool for the development of the so-called *virtual instruments*. The students must learn to design and build one of these instruments using the visual language LabView from National Instruments [10]. They can use an available free cost Student Version of this program [11].

Figure 4 shows an example of this *Practical Project* user interface that has been developed using LabView.

6.2. Design and building of a simple Digital Analyser

The *project* consists of building a simple Digital Analyser with only one input and limited buffer of memory and resolution. The design of this second example of *practical project* is based on using the special iF command “\$S”. The Digital Analyser probe is connected to pins C1 and C0 that must be tied together. The “\$S” command uses the microcontroller capabilities for pulse wide measurement. It obtains in microseconds the time the digital signal remains high and the time the signal remains low. Using these values, the answer of this command is a text string that contains 40 values of time intervals starting with the first change of level. For example “\$U:64554C: 473U:50244C: 101...” where U: means up level, 64554 are the microseconds that the signal remains high, C: means down level, and so on.

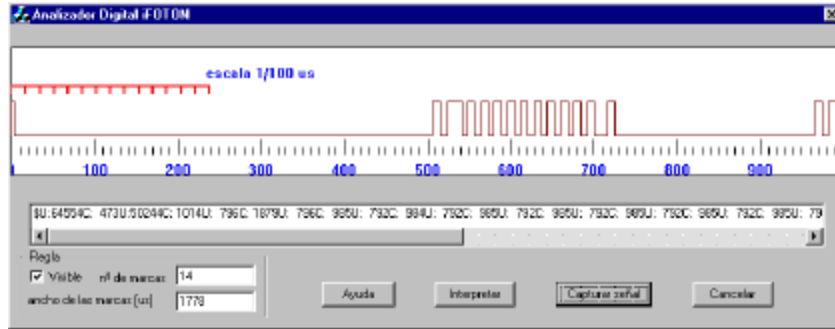


Figure 5. Digital Analyser user interface

Students will use these data as input to the PC software module.

For this *practical project*, students are required to develop the software using a generic programming tool as the C++ programming language. Using, for example, a tool as Microsoft Visual C++ software development environment, the design of a graphical user interface is not very difficult.

To test the Digital Analyser the student is asked to connect to the probe the signal provided by an Infrared Sensor (TFM5360 from TEMIC). When this sensor captures the signal emitted by a TV infrared remote control, it provides a TTL signal formed by a sequence of pulses.

Figure 5 shows an example of a Digital Analyser user interface. The screen shows the response obtained after receiving a signal from a TV PHILLIPS infrared remote control.

7. Conclusions

The following conclusions have been obtained after have experimented the new subject pedagogical structure for two courses.

“Digital Instrumentation and Data acquisition” is a subject especially orientated to students interested in having a double skill in hardware and software.

An important number of students of software would like to study and build hardware as well. They appreciate the possibility of building a physical system as opposite to developing systems exclusively made with software components. In fact, we think that the *Practical Project* is one of the causes for that, in this period of two years, the number of students that has chosen our subject has been duplicated.

iFOTON has demonstrated to be useful to support the students *Practical Projects* and also as a basis to de-

velop some others prototypes of data acquisition and control systems. It is meaningful that we have received from Internet some demands for purchasing iFOTON.

The developed set of resources provides a powerful tool for teaching digital instrumentation. However, we can make an additional effort improving the Project Configuration Manager. If we make an additional development, it will be possible to create a library of hardware-software reusable modules. This library will be available in Internet.

References

- [1] <http://www.dtf.fi.upm.es/~gtrivino/iad.html>
- [2] PIC16F87X. 28/40-pin 8-Bit CMOS FLASH Microcontrollers. Document DS30292B Microchip (1999).
- [3] <http://www.dtf.fi.upm.es/~gtrivino/IFOTON/1798.pdf>
- [4] EEPROM Memory Programming Specification. Document DS39025E Microchip (2000).
- [5] IEEE Standard Signalling Method for a Bi-directional Parallel Peripheral Interface for Personal Computers. IEEE 1284-2000.
- [6] Spur, R. *A PC-Based Development Programmer for the PIC16C84*. Application Note AN589, Microchip, 1999
- [7] Richey, R. (1998): Downloading HEX files to PIC16F87X PIC Microcontrollers. TB025. Microchip.
- [8] <http://www.dtf.fi.upm.es/~gtrivino/IFOTON>
- [9] Paton B.E. *Sensors, Transducers, and LabVIEW*. Prentice Hall, 1999
- [10] Bishop R.H. *Learning with LabView*. Addison-Wesley, 1998
- [11] Jamal R., Pichlick H. *LabVIEW applications and solutions*. Prentice Hall, 1999

ILP in the Undergraduate Curriculum

Daniel Tabak
ECE Dept., George Mason University,
Fairfax, VA 22030-4444
Tel. (703) 993-1598, FAX (703) 993-1601
e-mail: dtabak@osf1.gmu.edu

ABSTRACT

The paper discusses the teaching of instruction level parallelism (ILP) in undergraduate electrical engineering (EE) and computer engineering (CpE) curricula. An argument is made for justifying the teaching of this topic, usually taught in graduate courses, at the undergraduate level. A detailed account of the way this topic is actually taught at the author's University is given. The paper discusses the specific ILP subjects, presented to the students, along with the technical literature sources used.

1. Introduction.

The study of instruction level parallelism (ILP) has been relegated primarily to textbooks intended for graduate studies [1]. It is also the practice in many Universities to teach this topic at the graduate level in most cases. At the same time, it should be realized that practically all modern computers, be they RISC or CISC, are implementing ILP on a constantly growing scale. Some of the latest products, worth mentioning, are Intel Pentium 4, Intel and Hewlett-Packard (HP) IA-64 architecture Itanium, AMD Hammer (64-bit Intel x86, or IA-32, architecture), Sun Microsystems UltraSPARC, Silicon Graphics Inc.(SGI) MIPS R10000, and others.

The author's department of Electrical and Computer Engineering (ECE) at the George Mason University (GMU) has two engineering curricula: electrical engineering (EE) and computer engineering (CpE), leading to all three degrees (BS, MS, Ph.D.). The author has been teaching for many years a senior course on computer design. This course is required

for the BS degree in CpE, and it is a technical elective for the BS in EE.

It has been realized by the author, who developed this course from scratch, that students graduating with the BS degree and going into industry (in most cases) or to graduate studies, should be knowledgeable not only of the basic engineering principles of computer organization and architecture, but of the most recent design techniques and practices, implemented in modern processors. For this reason, the course content has been constantly changed and revised from year to year (sometimes, from semester to semester), to reflect the perpetual innovations in computer design.

As ILP began to be one of the main topics of research and practice of microarchitecture, it was introduced, in a timely manner, into the senior course on computer design. Recently, the subject of ILP also started to appear in textbooks intended primarily for undergraduate curricula, such as [2], chapter 8 and [3], chapter 5. The details of the ILP topics, covered in this course, are described in this paper. The course program and its literary sources are presented in the next section. Section 3 lists the examples of actual ILP processors, presented to the students. Section 4 includes concluding comments.

2. ILP in the Computer Design Course.

Prior to going into ILP, the students are exposed to a very detailed study of scalar pipelining. The primary textbook of the course is [1]. It was used in this course since its first edition in 1989. Chapter 3 in [1] has a very exhaustive coverage of pipelining. A good coverage of pipelining can also be found in [2], chapter 8, and [3], chapters 4 and 5.

After going over the basic principles of pipelining, using the examples in [1], chapter 3, the students are exposed to what can go wrong in pipelines; namely, to the possible pipeline hazards:

- Structural hazards
- Data hazards
- Control hazards

The above hazards, and some of their possible remedies, are discussed in detail. It is later pointed out that these hazards are only more serious in case of ILP.

Subsequently to pipelining, the discussion of ILP is initiated, using chapter 4 of [1] and other sources [4-6]. Sources [4,5] were chosen because they constitute extensive surveys on the subject with relatively large references lists. Report [6] was included because it contains very useful material on branch prediction, not available in such concentrated form elsewhere. In addition, material was taken from [7-9]. These are some of the earliest ILP publications, containing basic material. Superscalar, superpipelined, and very large instruction word (VLIW) operations are defined. However, the course concentrates primarily on superscalar operation, because of its prevalent implementation in industry. With the advent of the Intel-HP IA-64 architecture, more weight to VLIW may be given in the future.

Initially, problems involved with data dependence in ILP operations are discussed in detail. The concepts of name dependence, anti-dependence, output dependence, and control dependence, are defined, and some examples are given. The examples are taken both from [1] and some are supplied by the instructor. In addition, the following terms, associated with this topic, are defined and pointed out in the examples:

- Rear After Write – RAW
- Write After Read – WAR
- Write After Write – WAW

Subsequently, the following methods, approaches, and special data structures, having to do with data dependence, are studied in detail:

- Register renaming
- Speculative execution
- Out-of-order execution
- Scoreboarding
- Reorder buffer (ROB)
- Reservation stations (RS)
- Trace caching

All of the above topics are well covered and exemplified in chapter 4 of [1]. Other sources, such as [4,5,7] are also used. It is pointed out to the students that instead of using an RS in front of each functional unit (FU), one can use one central window with more entries, to forward operands to all FUs [7]. Some processors are indeed implementing this option.

Some topics in chapter 4 of [1], having a strong software “flavor”, such as loop unrolling, are skipped. It has been the experience of the author, that engineering majors do not willingly accept topics involving programming. Had the course been given to computer science majors, the above topics would also be included.

Problems due to branches in ILP, particularly those dealing with the conditional ones, are handled next. The topics of speculative and out-of-order execution are raised again. In addition, the following topics and data structures are studied:

- Branch prediction (local, global, bimodal)
- Branch target buffer (BTB)
- History table (HT)
- Counter structure (Counts)

This material is also covered in chapter 4 of [1]. In addition, references [4-7] are used. Of particular importance on this topic is the report [6].

The topic of data prediction is not covered in the undergraduate curriculum, since it belongs in the realm of basic research, as opposed to current industrial practice. It is relegated to a subsequent graduate course in computer architecture, along with other more advanced topics (such as explicitly parallel instruction computing - EPIC, for instance).

3. Examples of ILP Systems

Examples of actual processors, both of the RISC and CISC type, implementing ILP, are presented to the students. Special data structures and methods, discussed in the previous section, are pointed out to the students, as they are encountered in the processor examples. Some of the examples are brought up during the discussion of various topics in section 2. Reference [1] contains a number of examples. Reference [4] contains examples of SGI MIPS R10000, Compaq (Digital) Alpha 21164 (actually used in the primary computing system on GMU campus), and AMD K5. A number of ILP examples (including the R10000 and Alpha 21164) can be found in [10]. Another source to which students are directed is the Internet (websites such as www.intel.com, developer.intel.com, www.extremetech.com and others).

The main ILP implementation example, illustrated in detail in this course, is the Intel-HP IA-64 architecture with its first product, the Itanium. Most of this material comes from the Intel and HP websites on the Internet.

In conjunction with the study of the IA-64 architecture, the students are familiarized with the concept of predication, along with illustrative examples of its implementation. The concepts of EPIC [11,12] are briefly covered. The details of EPIC are relegated to a subsequent graduate course on computer architecture. In the Itanium example, ILP features, discussed in general earlier, such as register renaming, scoreboarding, branch prediction, and multiple FUs, are pointed out to the students.

The Intel IA-32 architecture products are also included in the examples, particularly the latest Pentium 4. Also here as for the Itanium, the ILP features such as out-of-order execution, trace caching, branch prediction, and multiple FUs, are stressed. The multiple register files (128 registers) in both Itanium and the Pentium 4 are pointed out to the students. In the Pentium 4, those are of course rename registers along with the old x86 architecture 8 “general purpose” registers (not quite “general”, because of their special tasks).

Other examples, such as the Alpha architecture processors (actually used on the GMU campus), the Sun UltraSPARC, and the SGI MIPS R10000, are also covered.

4. Concluding comments

Because of the prevalence of ILP implementation in industrial products, it is obvious that the topic should be included in undergraduate curricula, preparing engineers and computer specialist for the information technology industry. A sample of a possible undergraduate coverage of ILP, as practiced in a senior EE and CpE course at GMU, has been presented. This program has been constantly revised and modified in the past few years, to follow up the developments of the state of the art and engineering practice. This development and constant revision of the course is intended to continue.

REFERENCES

1. J.L.Hennessy, D.A.Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., M.Kaufmann, San Francisco, CA, 1996.
2. C.Hamacher, Z.Vranesic, S.Zaky, *Computer Organization*, 5th ed., McGraw Hill, NY, 2002.
3. J.P.Hayes, *Computer Architecture and Organization*, 3rd ed., McGraw Hill, NY, 1998.
4. J.E.Smith, G.S.Sohi, The Microarchitecture of Superscalar Processors, Proc.IEEE, vol.83, no.12, pp.1609-1624, Dec.1995.

5. A.Moshovos, G.S.Sohi, Microarchitecture Innovations, Proc.IEEE, vol.89, no.11, pp.1560-1575, Nov.2001.
6. S.McFarling, Combining Branch Predictors, WRL Technical Note, TN-36, June 1993.
7. M.Johnson, *Superscalar Design*, Prentice Hall, Englewood Cliffs, NJ, 1990.
8. N.P.Jouppi, D.W.Wall, Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines, In Proc. ASPLOS III, pp.272-282, Boston, MA, April 1989.
9. N.P.Jouppi, The Nonuniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance, IEEE Trans. on Computers, vol.38, no.12, pp.1645-1658, Dec.1989.
10. D.Tabak, *RISC Systems and Applications*, RSP, UK and Wiley, NY, 1996.
11. M.S.Schlansker, B.R.Rau, EPIC: Explicitly Parallel Instruction Computing, IEEE Spectrum, vol.33, no.2, pp.37-45, Feb.2000.
12. M.S.Schlansker, B.R.Rau, EPIC: An Architecture for Instruction-Level Parallel Processors, HP Laboratories Report, HPL-1999-111, Feb.2000.

PECTOPAH: Promoting Education in Computer Technology using an Open-ended Pedagogically Adaptable Hierarchy

Hugh Osborne, Shirley Crossley and Jiří Mencák
School of Computing & Mathematics
University of Huddersfield
Huddersfield HD1 3DH, U.K.
{h.r.osborne,j.mencak}@hud.ac.uk
shirleycrossley@blueyonder.co.uk

William Yurcik
Dept. of Applied Computer Science
Illinois State University
Normal
Illinois
USA
wjyurci@ilstu.edu

1. TEACHING COMPUTER ARCHITECTURE

1.1 Computer Systems Architecture

An understanding of Computer Systems Architecture (CSA) is essential to an understanding of Computer Science. There is however a tendency, at all levels, in teaching Information and Communications Technology (ICT) to neglect CSA, but teaching ICT without teaching CSA is like teaching Russian without teaching the Cyrillic alphabet — students may become reasonably fluent in the application of abstract high level skills (e.g. they know that the Russian for restaurant is *restoran*), but lack the basic skills needed to maintain and extend those skills (e.g. they cannot identify ПЕCТOПAН as being the “real” Russian for restaurant). There are two major reasons for the neglect of CSA in teaching ICT. There is a misconception of the effect of technological change, and there is a tendency to use inappropriate didactic tools.

1.2 The Rôle of Technological Change

“With IT technology developing so rapidly, is it really worth trying to teach something that will be out of date within a very short time?” As rapid as the developments have been it is the high level applications of IT that have changed — *the basic principles of CSA are essentially the same as they were 50 years ago.* Learning these basic principles allows students to build their understanding of high level IT applications on their knowledge of the basic principles of digital computers, confident that these principles are unlikely to change quickly and that they will be able to apply the same understanding to further developments and thus maintain a state of the art knowledge.

1.3 Using the Right Tools

CSA is traditionally considered to have a high learning threshold. This is due to the difficulty of teaching it in an incremental *and* hands-on fashion. For CSA hands-on experience should be provided by writing low level programmes, but low level programming already requires a solid grounding in aspects of CSA, e.g. internal data representation, the memory hierarchy, interaction with peripheral devices, etc.

This perceived difficulty is again a misconception, due to the use of inappropriate tools. A common approach is to use the inbuilt assembly language of some real machine to provide hands-on experience. These languages are not de-

signed as didactic tools, but as programming tools for experienced users who already have a thorough understanding of CSA, making such languages hard to understand, and making it hard for students to separate the (manufacturer specific) incidental from the (subject wide) essential. The frequently cryptic documentation only exacerbates the problems students have. Using such a tool to learn CSA is akin to trying to learn Russian using only a Russian/English-English/Russian dictionary — a very useful tool in skilled hands, but inappropriate as a beginner’s guide.

This abstract describes three tools that together provide a progressive hierarchy of teaching aids that can be used at many levels of teaching, providing students with a seamless incremental toolbox that can be used throughout their education.

The remainder of this abstract is organised as follows: Section 2 describes the teaching philosophy behind the toolbox; in Section 3 the three components of the toolbox are described; Section 4 discusses the use of these tools; and section 5 is a summary of their integration in the toolbox.

2. INCREMENTAL TEACHING

The aim of any good course must be to introduce students to new concepts in an incremental fashion. The subject matter must be analysed and a plan of delivery developed so that the *learning threshold* is at all times as low as possible. Learning of concepts is strongly reinforced by “hands-on” experience, and this should be introduced as early as possible — any course in which many weeks of background introduction are required before students can undertake realistic exercises is likely to be perceived as “difficult” or “too theoretical”.

Low learning thresholds by no means exclude high learning expectations. It is essential that the teacher has high expectations of the students’ learning, and communicates these to the students. Students perform to expectations, so high expectations bring out the best in achievement.

These two aims are strongly related to the need to structure courses so as to enable students of the widest possible range of abilities to profit to the maximum of their capabilities from the material on offer. A well structured course

will provide students with knowledge and skills at various levels. The course should contain enough advanced material to challenge the more able student, allowing them the opportunity to develop and prove their ability, while ensuring that the basics are covered in sufficient detail for the less able to provide them with the basic knowledge expected of them.

Incremental teaching is also the ideal on a longer timescale. Students making the transition from, for example, secondary to tertiary education often experience a “fault line” where there is a mismatch between their prior knowledge and experience and the prerequisites assumed by their new institution. While such problems are to some extent unavoidable, the development of national curricula and the provision of integrated tools and methodologies can help to alleviate them.

3. APPROPRIATE TOOLS FOR CSA EDUCATION

3.1 Primary and Secondary Education — “How Computers Really Work”

There is a shortage of appropriate material for teaching late elementary and secondary school pupils the essentials of Computer Systems Architecture. “How Computers Really Work” is a pilot interactive CD-ROM for teaching CAS to primary and secondary school children. Students are guided throughout by “Chip”, an animated floppy disk. There are areas on the CD covering peripherals, computer hardware and software, the internet and the history of computing. There is also a quiz consisting of multiple choice questions covering material from all other sections of the CD. The largest area of the CD is Computer Architecture area dealing with the CPU, memory and data. The description of the CPU is based on the Postroom Computer (see section 3.2).

3.2 Introductory Undergraduate Level — “The Postroom Computer”

The problem of teaching low level programming at an introductory undergraduate level was addressed as early as 1965 by Stuart Madnick and John Donovan (see e.g. [3]). In the *Little Man Computer* (LMC) they provided an extremely simplified model of low level programming and computer architecture. The LMC model has proven to be of lasting popularity, as the number of LMC emulation programmes currently in existence, 35 years after it was originally proposed, shows (see, e.g., [11] or [10] for a survey).

The *Postroom Computer* (PC) [8, 9] is an extended emulation of the LMC model, in which the emphasis is on flexibility and generality. It is designed to introduce aspects of CSA and low-level programming in an incremental way. The extensions are designed to provide a range of computing models within the LMC/PC paradigm. As they are introduced they can be related both to the LMC/PC paradigm and to “real” machines. The PC provides a powerful and flexible tool for teaching CSA. By adding orthogonal extensions to Madnick and Donovan’s basic LMC aspects of CSA can be introduced in a stepwise fashion, never overwhelming students with details, yet leading eventually to a full understanding of the principles of CSA.

The PC also introduces students to a more formal description (*Update Plans*, see section 3.3) of computer systems

architectures.

3.3 Advanced Undergraduate/Postgraduate — “Update Plans”

Update Plans (UP) is a formalism for the description of abstract machines and algorithms. UP is particularly suitable as a specification language for the description of large classes of computer systems architectures.

UP has didactic uses in many areas of Computer Science other than CSA — for example data structures and compiler construction. The common denominator in all these applications is that of a *pointer*. Pointers are intrinsic to Computer Science. Each field of Computer Science seems to use its own more or less ad hoc notation for describing pointers and operations on pointers, thus impeding crossover of students’ skills from one area to another. UP is a “universal” pointer specification language. Consistent use of UP as a supplement to the traditional notations can greatly enhance students’ ability to apply skills learned in one domain to other pointer applications.

4. USING THE TOOLS

4.1 How Computers Really Work

A pilot version of the interactive CD-ROM was tested on a group of children and their teachers. The children all enjoyed the package and liked the classroom metaphor used in the CD-ROM. The teachers found it bright and cheerful, and felt that the animations would catch the children’s attention. The children did have some difficulty in understanding the Postroom Computer section. However, in the pilot version, there was no facility for hands-on programming of the Postroom Computer. Fully integrating a user friendly GUI to the Postroom Computer would undoubtedly greatly enhance the usefulness of this tool.

4.2 The Postroom Computer

The PC is supported by both online documentation[1] and a fully integrated system of course materials [2], including a range of exercises allowing students of all levels to advance their knowledge and skills.

Experience has shown that after one semester (12 weeks), with one hour of lectures and one hour of supervised practical exercises per week, first year first semester undergraduates with no prior knowledge of the subject demonstrate a good understanding of the subject matter and can successfully undertake a range of ambitious low level programming exercises of a level normally considered to be too advanced for introductory CSA courses. The students typically achieve a higher level of understanding of the principles of CSA than usual at this level of instruction.

4.3 Update Plans

UP was not originally developed as a teaching aid, but as a theoretical tool. It has been the target of academic research [4, 5, 6, 7], which has shown that the formalism is a valuable aid for the description and analysis of a wide range of computer systems architectures. UP has also been used as a teaching aid at various universities, and has shown its worth in helping students to understand how the components of computers, both hardware and software, interact. An implementation of a subset of Update Plans is currently

being used as a teaching aid for an advanced course on the implementation of functional languages.

5. THE TOOLBOX

Each of the three components described is a useful tool in its own right. Together they provide a long term incremental tool for teaching CSA.

How Computers Really Work provides a basic introduction to CSA, including a first contact with the Postroom Computer model.

The Postroom Computer can be used as an incremental hands-on teaching aid for introductory undergraduate CSA. No prior exposure of the students to the Postroom Computer is required for successful deployment of this tool, though it would of course be an advantage. As well as introducing students to the important concepts of CSA, the Postroom Computer model will also probably be their first introduction to precise and formal descriptions of computer systems structures, using Update Plans.

Update Plans provide a tool for further development of student's understanding of CSA, allowing them to develop and test their own models of advanced computer systems architectures. Update Plans are also applicable to many other areas of computer science (e.g. data structures and compiler construction), thus facilitating crossover of students' skills and understanding. Further applications of Update Plans can be developed to encourage this crossover, and to provide an open-ended tool in teaching computer science.

6. REFERENCES

- [1] <http://scom.hud.ac.uk/staff/scomhro/Courses/PostroomComputer/>.
- [2] <http://scom.hud.ac.uk/staff/scomhro/Courses/CFS155/>.
- [3] Irv Englander. *The Architecture of Computer Hardware and Systems Software*. John Wiley & Sons, New York, 2000. Second edition.
- [4] Hugh Osborne. The semantics and syntax of update schemes. In *Code Generation — Concepts, Tools, Techniques*, Workshops in Computing. Springer Verlag, 1992.
- [5] Hugh Osborne. Update Plans. In *Proceedings of the 25th Hawaii International Conference on System Sciences*. IEEE Computer Society Press, 1992.
- [6] Hugh Osborne. *Update Plans — A High Level Low Level Specification Language*. PhD thesis, University of Nijmegen, 1994. <http://scom.hud.ac.uk/staff/scomhro/Papers/PhD/phd.html>.
- [7] Hugh Osborne. Update Plans for parallel architectures. In M. Kara, J.R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*. IOS Press, 1996.
- [8] Hugh Osborne. The Postroom Computer. *ACM Journal of Educational Resources in Computing*, 2(1), March 2002.
- [9] Hugh Osborne. The Postroom Computer: Teaching introductory undergraduate computer architecture. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE 2002)*, 2002.
- [10] Gregory S. Wolfe, William Yurcik, Hugh Osborne, and Mark Holliday. Teaching computer organization/architecture with limited resources using simulators. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE 2002)*, 2002.
- [11] William Yurcik and Hugh Osborne. A crowd of Little Man Computers: Visual computer simulator teaching tools. In *Proceedings of 2001 Winter Simulation Conference*, New York, 2001. ACM.

Read, Use, Simulate, Experiment and Build : An Integrated Approach for Teaching Computer Architecture

Ioannis Papaefstathiou and Christos P. Sotiriou

Department of Computer Science,
University of Crete,
P.O. Box 1385, Heraklion, Crete, GR 711 10, Greece.
{ygp,sotiriou}@ics.forth.gr

Abstract

In this paper we present an integrated approach for teaching undergraduates Computer Architecture. Our approach consists of five steps: “read”, which corresponds to studying the textbook theory, “use”, which corresponds to using a simulator with appropriate graphical features to visualise the application of the theory, “simulate”, which corresponds to developing an architectural simulation, “experiment”, which corresponds to modifying the architectural simulation and observing the impact that changes make to performance, and finally “build”, which corresponds to developing a low-level hardware model in a standard Hardware Description Language. In our experience, going down to the gate-level is of great importance, as students often find difficult to visualise how different architectural approaches affect the actual hardware (both datapath and control). By following this five-step approach in our teaching we observed a significant increase in both student performance and interest in Computer Architecture and hardware design.

1 Introduction

The subject of Computer Architecture is widely recognised as a significant and essential part of the undergraduate syllabus of university degrees related to computer or hardware design. One of the main problems with teaching Computer Architecture, is that students should not only understand the textbook theory, but more importantly its application in real systems and the impact that different architectural approaches have on the complexity and the performance of a system.

Thus, to make the teaching process more effective we have chosen to use an educational approach which we based on five steps: Read, Use, Simulate, Experiment and Build. In this paper we describe these five teach-

ing steps and focus on the ones we believe are yet uncommon, however have been very effective in our experience.

2 “Read”: Textbook Theory

Our Computer Architecture teaching is based on the Hennessy and Patterson Computer Architecture textbook, “Computer Architecture: A Quantitative Approach” [1], currently recognised as the most extensive and complete reference on the subject. Our course is taught in the last year of the Computer Science undergraduate degree, *i.e.* year 4, and runs for a duration of 14 weeks. As our teaching philosophy relies on combining theory with practice, we prefer to give students practical experience than a vast amount of theory. Thus, in 14 weeks we cover the first five chapters of the book, both in terms of theory and practice.

3 “Use”: HASE Simulator

After the “Read” stage, students are given simple exercises on a graphical simulator. Our simulator of choice is the HASE [2] environment. HASE (Hierarchical computer Architecture design and Simulation Environment) is a graphical design, simulation and visualisation environment that can be used for both teaching and research. We use the DLX HASE model developed at the University of Edinburgh. HASE allows students to visualise both the overall structure of the DLX architecture and the execution of instructions by observing the step-by-step progress of individual events. HASE also allows for students to explore the impact of architectural parameters to the performance of the architecture, as students can change these using only the GUI environment (Graphical User Interface) and then re-run the simulation.

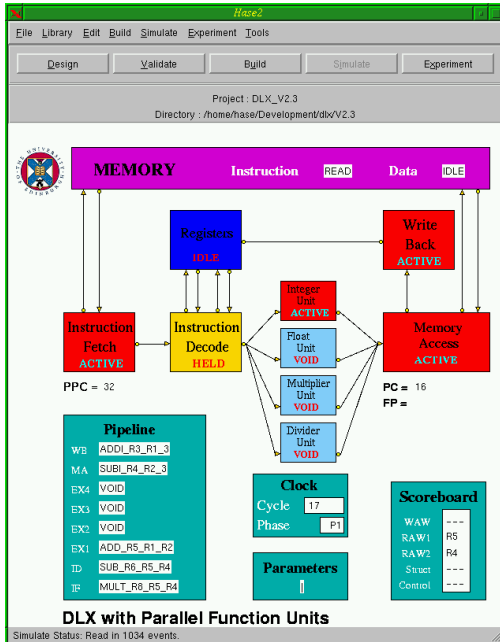


Figure 1: The HASE DLX Model

The DLX HASE exercises require students to write DLX assembly code and execute it on the HASE environment. With the help of the simulation environment students can measure the execution time, study the execution of each instruction in detail (passing through each pipeline stage) and the impact of architectural parameters. Students are asked to reason about the execution time of their program and to optimise their code based on their reasoning. They can experiment with different code schedules and different parameters and evaluate the execution time with the aim of finding the best possible cases.

Since using HASE as part of our teaching, rather than the standard pen-and-paper ones, we observed a significant increase in the students understanding and performance in the written examinations. This is probably due to the fact that by getting hands-on experience of the theory covered, students gain deeper and more thorough understanding.

4 “Simulate and Experiment”: Develop a Simulator

The next stage of the course requires for the students to implement their own architectural simulation using a standard Hardware Description language (HDL), *i.e.* Verilog in our case. In this stage the implementation of

the architecture is to be at the behavioural level. The students are asked to implement a RISC CPU called ARCP. The reason we chose an alternative to the DLX architecture was to give students something more challenging than simply re-implementing the DLX, which they already are familiar with at this stage from the HASE simulations.

4.1 ARCP - A 2-way Issue Architecture for Teaching

The ARCP architecture is based on the DLX, and has a very similar instruction set, however it is slightly more complicated, being 2-way superscalar. ARCP fetches two instructions at the same time from its instruction memory, which should be aligned and independent of each other for reasons of simplicity (students are given only 6 weeks of term for completing the whole project).

The main characteristics of the ARC architecture are :

- 64 General Purpose Registers.
- 32-bit address and word lengths.
- byte addressable, big-endian architecture.
- support for two data types: words (32-bits) and bytes (8-bits).
- 2-way fetch and execution of *independent* instructions; the independence of instructions must be ensured by the compiler/assembly programmer.
- only one control instruction (branch or call instruction) is allowed in an instruction pair and it must be placed in the first of the two instructions.
- only one memory reference instruction is allowed in an instruction pair and it must be placed in the second of the two instructions.
- any number of arithmetic/logical operations are allowed.
- same memory used for instructions and data and self-modifying code is not allowed.
- memory can only be accessed using load or store instructions.
- branches are not delayed.
- register 0 is hardwired to 0.
- there are no condition codes; comparison instructions write a 1 (for true) or a 0 (for false) at a destination register.
- conditional branches are PC-relative while unconditionals (call instructions) may be PC-relative or register-indirect; unconditionals store their current address in their destination register.

4.1.1 ARCP Instruction formats

The three different instruction formats and the format of an instruction pair are shown in Figure 2.

4.1.2 ARCP Instructions

All supported instructions along with their opcodes and formats are shown in Figure 3.

Most of these instructions are straightforward and found in the majority of RISC style architectures. The

4 MS bits	3 LS opcode bits							
	opcode	000	001	010	011	100	101	110
0000	add R	addi I	sub R	subii I	mul R	muli I		cmgti I
0001	cmeg R	cmegi I	cmne R	cmnei I	cmge R	cmgei I	cmlt R	cmlti I
0010	and R	andi I	or R	ori I	xor R	xori I	gcp R	cmlei I
0011	shru R	shrui I	shrs R	shrsi I	shl R	shli I		sethi L
0100	ldbu I	ldbs I	ldw I		stb I		stw I	
0101	breq L	brne L	brge L	brlt L			callr R	call L

Figure 3: ARCP Instructions and Opcodes

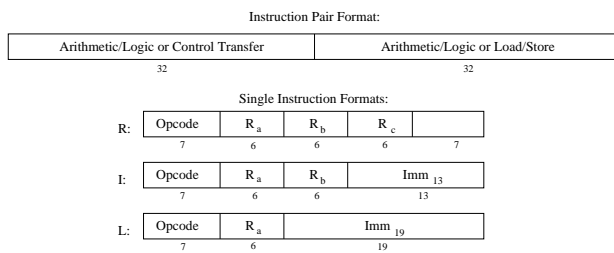


Figure 2: ARCP Instruction Formats

only unusual ones are the `sybii` and `gcp` instructions. The `sybii` instruction corresponds to a subtract immediate inverse operation, *i.e.* subtracts the register operand from the immediate, thus inverting the order of the subtraction. The `gcp` instruction corresponds to a guarded copy operation. A guarded copy operates using three registers and copies the source register into the destination if the third register, the guard, is not equal to zero. Guarded copy instructions can be used for implementing if-then-else blocks without branches and therefore can improve the efficiency and performance of the pipelining.

4.2 ARCP Simulation and Evaluation

In the “Simulate and Experiment” phase of the project the students are asked to build a behavioral simulation of this CPU and collect a set of measurements based on a number of small benchmark programs. Some of these benchmarks are provided by the lecturers, whereas the rest are to be developed by the students and are to be representative of typical applications. In our view, letting the student deal with the problem of finding the best benchmarks for evaluating the performance of the processor is really important, as it makes them really think hard of all the underlying issues involved. To help students achieve this, our research group has developed sim-

ple compilers and assemblers which students can use to produce their benchmarks.

The measurements that we are asking the students to provide (and we believe they are the most important for such a simulation) are the following:

- number of useful instructions executed (non NOOP).
- number of instruction pairs executed.
- average number of useful instructions.
- average number of memory reads per pair.
- average number of memory writes per pair; the last two are important for understanding the use of the memory hierarchy and the impact of having different data and instruction memories.
- number of taken and not-taken branches.
- percentage of useful instructions for each of the following groups: add/sub/mul, compare, and/or/xor, shift, gcp, load/store, branch, subroutine-call and jump.

Towards the end of the course students are asked to write a report which describes possible optimisations on the above architecture based on their simulation results. They are also asked to run new experiments on their architecture so as to support their claims for the possible optimisations. We believe that this idea of students proposing possible optimisations given an initial architecture is a crucial skill that a Computer Architecture student should acquire.

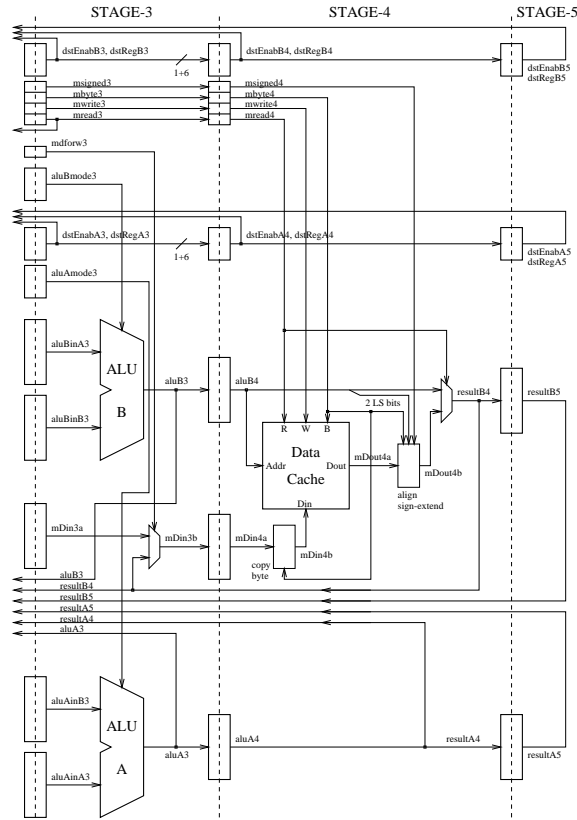


Figure 6: Stages 3, 4 and 5 of the pipeline

6 Conclusion

In this paper an integrated approach for teaching Computer Architecture was presented, which is currently used at our University, and has been found to be very effective. Its main advantages are the following:

1. It increases the interest of the students in Computer Architecture and hardware in general. There was a significant increase in the number of students concentrating on Hardware after we have adopted this approach either by taken their undergraduate thesis on a hardware subject or enroll on a hardware or semi-hardware oriented postgraduate program.
2. It gives the student a thoroughly comprehension of the main subjects of Computer Architecture
3. It enhances their performance in the exams which is probably due to the fact that they get a lot of hands-on experience on every aspect of Computer Architecture.
4. It provides them with skills that are very useful when designing hardware and not only when investigating the architecture of a system.

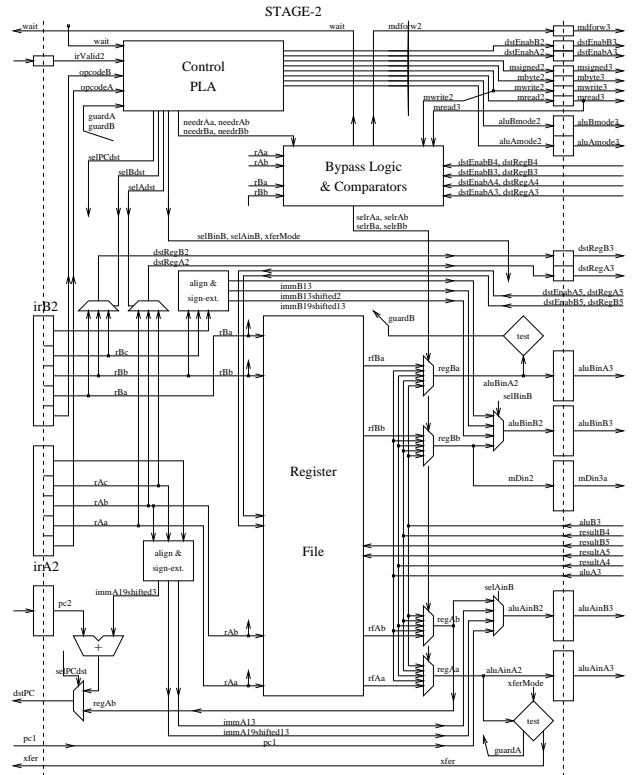


Figure 7: Stage 2 of the pipeline

This approach is, we believe, ideal for a course that is taken by students that might want to focus on hardware, or have already made such a decision and they would like to get a first idea of how a system is initially designed, then simulated and finally built and tested. Its main disadvantage is, we believe, that it relatively increases the work needed for the course and might not be that appropriate for cases where just an introduction to Computer Architecture is needed (maybe because there are a great number of more specialized hardware design courses in the syllabus).

References

- [1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.
- [2] P. S. Coe, F. W. Howell, R. N. Ibbett, and L. M. Williams, "A Hierarchical Computer Architecture Design and Simulation Environment," *ACM Transactions on Modelling and Computer Simulation*, vol. 8, Oct. 1998.

An Integrated Laboratory for Computer Architecture and Networking

Takamichi Tateoka, Mitsugu Suzuki, Kenji Kono, Youichi Maeda, and Kôki Abe

Department of Computer Science
The University of Electro-Communications
Tokyo, 182-8585 Japan
Email: `cnp@cacao.cs.uec.ac.jp`

Abstract

Processors, compilers, and networks – important materials covered by computer science curricula – are often treated independently in laboratories associated with corresponding lecture courses. An integrated laboratory called CNP for juniors majoring in computer science at the University of Electro-Communications has been developed and is now under way, where a networking protocol stack implemented by students is translated into object codes by a compiler implemented by students, which in turn are executed on a processor implemented also by students. The goals of the integrated laboratory are to deal with modern and attractive materials, to provide students with opportunities of collaborating in constructing a large system, as well as to have students share a feeling of accomplishments among them. Responses from students approved our intention and verified the effectiveness. In this paper, we describe the design and development of baseline components to be integrated, laboratory organizations and schedules, and results and evaluations of the laboratory.

1 Introduction

Processors, compilers, and computer networks are important materials covered by computer science curricula. They are often treated independently in laboratories associated with corresponding lecture courses. Many reports on laboratories dealing with microprocessor design and implementation have been published (eg.[1]). Exercises on compiler design are too common to mention. Some reports on computer networking laboratory exist[2], although it has been recognized in the Computing community that academic institutions should treat computer networking to more fully extents[5].

However, in order for improving cost performance of

a computer system, tradeoffs between hardware and software must be well understood and the characteristics of applications executed on the system need to be carefully examined. Adjusting interfaces between system components is also required. Thus taking a broad view of entire system is mandatory. For students to acquire the view, separate components need to be integrated into a complete system in a laboratory.

A design problem across areas can effectively be solved in a short term by teamworking, where the problem is divided into parts and works by team members are shared and combined. Providing students with opportunities to have experiences of such teamworking in university laboratories dealing with design and implementation of both hardware and software for modern and attractive applications is of key importance.

An integrated laboratory called CNP for juniors majoring in computer science at the University of Electro-Communications (UEC) has been developed and is now under way, where a networking protocol stack (called TinyIP) implemented by students is translated into object codes by a compiler (called TinyC) implemented by students, which in turn are executed on a processor (called MinIPS) implemented also by students. The whole system integrated by students in the laboratory is called TinyJ.

Students are organized into several teams. Members of a team cooperatively perform the laboratory experiments. The goals of the CNP laboratory are for each student to understand the interfaces between system modules, to design and implement an assigned one, to integrate cooperatively the components into a system, as well as to discuss and adjust their specifications.

In the following, Section 2 describes laboratory design and developments of baseline components to be implemented and integrated by students. Related courses offered to students are also stated in this section. Section 3 describes the details of the laboratory including stu-

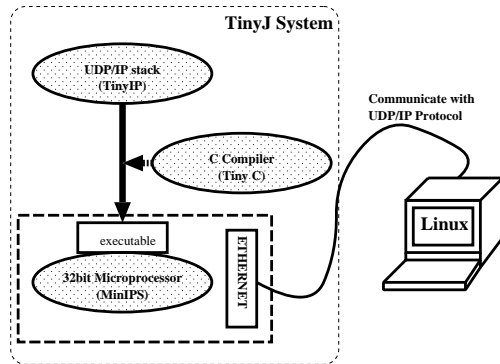


Figure 1: Illustration of the integrated laboratory.

dent organization and schedule of the laboratory course. Section 4 gives results and evaluations of the laboratory. Section 5 closes with a summary and future works.

2 Design and Developments

2.1 Laboratory Design

The laboratory is designed to amalgamate UDP/IP protocol stack, a C compiler, and a 32bit RISC processor. Students integrate these materials into a complete system (Tiny J) to construct a complete small computer system which is capable of communicating with Linux OS through Internet standard UDP/IP protocol. A simple protocol stack (TinyIP) coded in a simplified C language is translated into object codes by a compiler (Tiny C) implemented by students, which in turn are executed on a processor (MinIPS) implemented also by students. The overview of the integrated laboratory is illustrated in Figure 1.

We design the laboratory so that students can design, implement, and modify all parts of the system components. Students, however, are not assigned to design every submodule because of restricted laboratory hours. We provide students with information enough for them to inspect any part. For example HDL descriptions of peripherals used for console function and source codes of original compiler which are to be extended by students are given to them.

We summarize related lectures and laboratories referring to the Computing Curricula 2001 by IEEE CS[5]. Lectures covering AR1 to AR5 of Architecture and Organization and PL1 to PL6 of Programming Language are offered as core, accompanied with corresponding labs. AR6 (Functional organization) is covered by an elective sophomore course, where [6] is used as a text-

book. PL8 (Language translation system) is covered by an elective junior course, where a simplified C compiler TinyC[8] is introduced and designed. Fundamentals of OS are introduced in a requisite course but the topics are intensively treated in an elective junior course. Topics in Net-Centric Computing are treated in an elective senior course, where principles of communication networks with OSI layered architecture are introduced.

The CNP is offered as a requisite junior laboratory course. Since we can not expect special knowledges given by elective lectures or lectures offered in future courses, when necessary we provide students with practical lectures required to complete the assignments in laboratory hours.

2.2 MinIPS Computer System

Requirements for the processor to be developed are: 1)To be simple and modern as an educational processor given to computer science juniors; 2)To have enough performance that allows building a computer system using the processor as a CPU core in Tiny J System; 3)To conform to the processor dealt with by [6] used as a textbook in the corresponding lecture course.

Requirements for the computer system based on the processor are: 1)To accommodate a communication port through which the protocol stack transmits and receives packets; 2)To equip with enough amounts of memory for programming TinyIP; 3)To have functions for loading programs and acting as a console.

We utilize a SRAM based FPGA as the implementing device used by students. It makes students to redesign the processor any times without care of making errors. An evaluation board, system-on-a-programmable-chip (SOPC) Development Board by Altera, equips with an FPGA which is capable of realizing 400,000 gate logic circuits using Logic Elements (LEs) and 20 KB memory using Embedded System Blocks (ESBs). The board also equips with Synchronous SRAM (SSRAM), RS232C, and Ethernet transceiver, which enable to organize a system meeting the requirements with no other hardware supplements. For those reasons we have chosen to utilize the SOPC Board for implementing the system.

We do not use any commercially available intellectual properties for FPGA configurations such as Ethernet controllers because intellectual properties would introduce black boxes. Thus all the prototype modules including the peripherals have been designed by ourselves. Although students are not assigned to design the peripherals, our descriptions are given to students so that whoever interested is ready for reading.

The organization we have designed is shown in Figure 2.

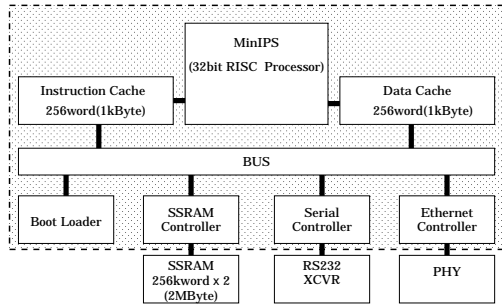


Figure 2: MinIPS system organization. (Enclosed with dashed box is the part implemented on an FPGA.)

The MinIPS processor[3] is a 32 bit RISC which conforms with MIPS[6]. The MinIPS instruction set is reduced to a minimum. For example, multiplication and division are not provided as machine instructions but are compiled to subroutine calls. The block diagram of the processor is shown in Figure 3. Conforming to the textbook, the structure is composed of five pipeline stages; instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write back (WB). A forwarding unit (FW Unit) is equipped. The load delay is one, and the branch delay is also one. The MinIPS processor conforms to the textbook but it is simpler.

Although the MinIPS processor is based on Harvard Architecture, it does not allow fetching instruction and data simultaneously because the memory is not physically divided into instruction and data submemories. As a solution to the problem we provide instruction and data caches which we implement using ESB memories on FPGA. Two SSRAM chips each of 256k words capacity mounted on the SOPC board are utilized for the main memory. It has enough capacity for programming TinyIP.

The specification of the RS232C controller is based on the simulator SPIM[6], enabling a smooth shift from simulations to executions on real machine. We use the RS232C for program loading and console function.

For the communication port, we adopt the commonly used Ethernet where a link layer address is provided. As the physical layer a PHY chip on the SOPC is utilized. Descriptions of the link layer for controlling the PHY has been developed by ourselves[4] and given to students. The interface is driven by hardware interrupts since polling is not a practical method for receiving Ethernet packets.

For loading programs and acting as a console, two programs, boot loader and monitor, are provided. The boot loader is stored in a ROM area implemented on ESB. It

is initially executed for loading programs upon turning on the power. The monitor is loaded through the boot loader. It provides such console functions as displaying the memory contents, loading programs and data into memory, displaying the contents of registers, modifying the contents of program counter, and handling interrupts as well as dispatching the corresponding processes.

Using Quartus II ver.1.1, a development software tool by Altera, we describe and compile the design in Verilog-HDL, configuring the FPGA on the SOPC board. About 57% of the LE resources have been used for the configuration. The compilation requires about 25 minutes with a platform of Pentium4 1.7GHz CPU with RDRAM 512MB memory. The MinIPS system is operating at a clock rate of 16.5MHz.

2.3 Tiny C Compiler

Requirements for the language compiler to be developed are: 1)To be simple enough to understand; 2)To have enough ability that allows compiling network protocol stack and applications for TinyJ System; 3)To conform to compiler design lectured in the corresponding lecture course.

Tiny C[8] is a small subset of C language developed by Prof. Watanabe as an illustrative compiler for his compiler course. It almost meets our requirements except for lacking of support for some operators such as bitwise operators. We supplemented support for unary address operator (“&”), bitwise operators (“&”, “|”), modulo operator (“%”) and shift operators (“<<”, “>>”) into Tiny C. The supplemented version is denoted by Tiny C hereafter unless otherwise noted.

We also introduced minor modifications into SPIM emulator[6] to use it as a MinIPS emulator. The modifications include appending a memory image snapshot function which is used as a substitution for assembler and linker to obtain MinIPS object codes. Signed multiplication and division routines were added into “trap.handler” containing startup codes for MinIPS, since MinIPS does not support these instructions.

2.4 TinyIP protocol stack

Requirements for the protocol stack to be developed are: 1)To be simple enough to understand and easy to describe in TinyC with simple syntax; 2)To be realistic and practical so that students feel a sense of accomplishment; 3)To be educational so that students understand features and benefits of the protocol with layered architecture; 4)To be extensible so that students can append their own ideas to the protocol; 5)To be independent of

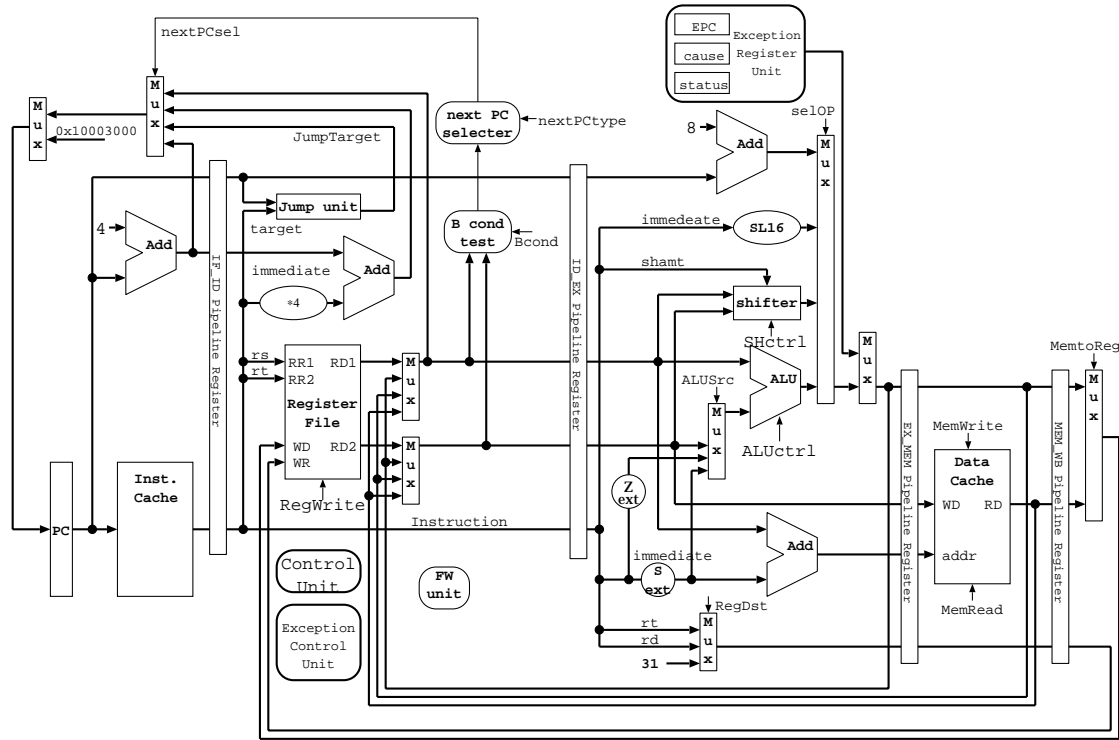


Figure 3: Block diagram of MinIPS processor.

hardware so as to allow testing before the processor becomes available.

To simplify the protocol stack, we use Internet Protocol (IP) as a network layer, and User Datagram Protocol (UDP) as a transport layer. We do not support packet fragmentation, Address Resolution Protocol (ARP), nor Internet Control Message Protocol (ICMP) processing. In spite of the limited functions, it is still capable of communicating with standard IP such as the one implemented in the Linux kernel.

We provide students with two versions of the implementations: one written in standard C language to illustrate the design of the protocol stack, the other written in Tiny C to be integrated into Tiny J System. Both of them have almost the same structure except that the latter calls for works on differences between standard C and Tiny C compilers. We basically describe the former in this section.

The stack consists of ten modules whose functions and calling flows are shown in Figure 4. They are described in separate C source files with well-defined interfaces. Receiving functions are driven by interrupts. The method of using interrupt mechanism is practical and keeps the control flow simple and conforming to the OSI seven-layer model. The structure facilitates append-

ing new features to the stack, making the stack extensible. Hardware dependent routines are collected into one module (`hardware.c`), resulting in portability to new hardware.

The core routine of the stack consists of about 800 lines in C language with additional 400 lines of instructive comments. Tiny C version consists of about 500 lines since some features such as generic FIFO routines were omitted.

We also supplied a TinyIP compatible library for standard UDP/IP stacks on Linux. Students can build and execute application programs before completion of their own stacks.

We developed a monitoring tool `etherpeep` which allows observing ethernet frames in hexadecimal numbers (and ascii characters). It is similar to `tcpdump` command on Linux with `-x` option except that `etherpeep` displays ethernet headers. It displays whole ethernet frames in simple format.

3 Integrated Laboratory

The laboratory course offered in second semester of junior year started from year 2001. Hours assigned to the

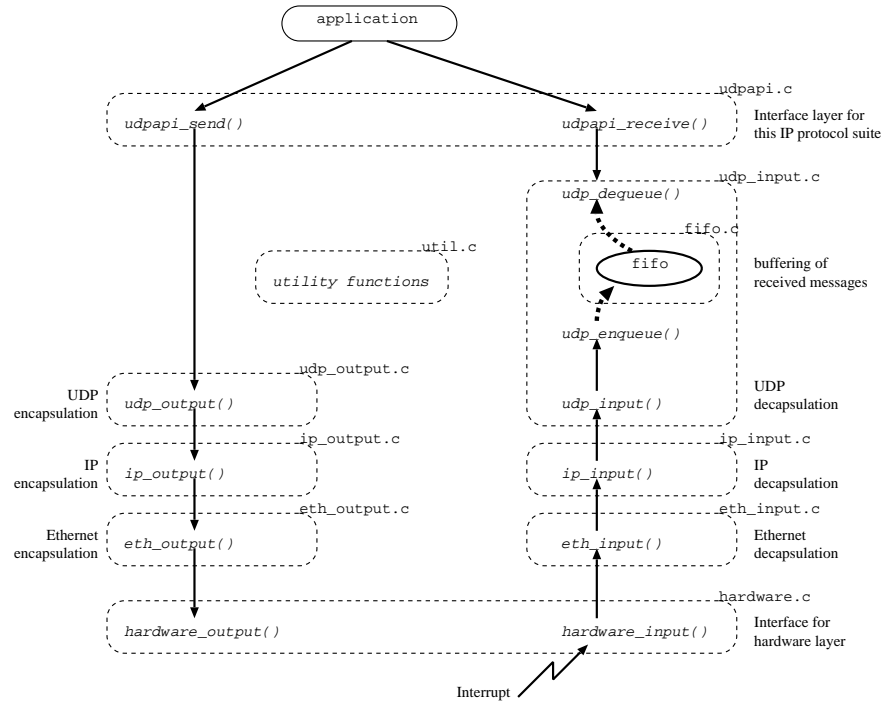


Figure 4: Functions and their calling flows of TinyIP modules.

course are divided into two periods each consisting of 12 three-hour classes. The laboratory course completes within individual periods. In each period 30 students take the course. Thus 60 students in total participate in the laboratory during the semester.

In each period students are grouped into five teams each of six students. A team consists of N (networking), C (compiler), and P (processor) subgroups who are in charge of working on corresponding sub-laboratories and cooperatively develop a complete Tiny J system.

In the P sub-laboratory students proceed along the following steps: 1) Learning how to use design tools; 2) Designing small submodules; 3) Designing arithmetic logic unit; 4) Designing pipelined processor; 5) Compiling the MinIPS system and verifying the function.

For coordinating them with other sub-laboratory assignees, we fix a minimum specification given to students. After getting familiarized with the development tools, students first design simple modules such as multiplexors and adders, and then gradually shift to designing more complex modules. When completing all the necessary modules, they start designing the entire processor.

In designing the processor they are not assigned to describe the whole of the processor. Instead, they are given a processor description with several parts taken out in such a way that the behavior of the pipelined processor

can still be understood. They are assigned to supplement the incomplete design with proper descriptions. For such modules as the RS232C and Ethernet controllers, descriptions are given to students so as to enable them to verify that the processor is operating.

For testing and verifying the design several tools are provided: 1) A graphical simulator embedded in Quartus has been used throughout the experiments; 2) LEDs equipped on the board are used in the preliminary experiments as well as for monitoring states of programming execution; 3) The boot loader given to students has been effective in checking whether the MinIPS system works as a whole; 4) Test programs such as calculating prime numbers in TinyC are given to students for more extensive debugging.

Discussions among team members on the specification of MinIPS are expected. For example, extra instructions may be added to MinIPS instruction set if an agreement is reached between C and P assignees.

Students start the C sub-laboratory by tracing the parser in the original TinyC source code to draw a chart illustrating the syntax of the language processed by the compiler. Then they are assigned to refine the original TinyC compiler so as to accept additional operators and literals required to implement TinyIP. The assignments are the necessary supplements described in the

previous section. Discussions among team members on the specification of their Tiny C is expected also in this sub-laboratory. Agreements in the team members may lead to changing the specification of their final version of Tiny C.

In the N sub-laboratory students proceed along the following steps: 1)Writing simple applications; 2)Analyzing ethernet frames; 3)Implementing and enhancing TinyIP for Linux; 4)Implementing TinyIP in Tiny C for MinIPS; 5)Combining TinyIP with other sub-laboratories. The fourth and the last are steps for the CNP integration requiring collaborations of team members. The integration steps are to be led by N assignees.

Students start the N sub-laboratory from learning basic network architecture and writing some simple applications. They connect two Linux boxes with an ethernet cross cable, configuring a LAN isolated from the campus network. We provide a TinyIP compatible library for Linux and a sample application with detailed documents. They utilize the library to write client and server programs satisfying echo protocol[7]. The programs are used as applications later in Tiny J.

Next they learn how the frame is encapsulated and decapsulated. They capture and analyze ethernet frames produced by standard UDP/IP stack with `etherpeep` command. They also get a good reference of working UDP/IP frames.

Students then proceed to implementing TinyIP in standard C. We provide a template of TinyIP implementation missing core functions such as encapsulation and decapsulation of ethernet, IP, and UDP frames. For verifying the implementation they connect two Linux boxes: one is configured to use standard UDP/IP stacks while the other is to use TinyIP. On both boxes they execute their client and server applications developed at the first step. They enhance their TinyIP implementation by adding some features such as ICMP and ARP, and/or by making improvements on memory consumption.

After discussions among team members to fix the final language specification of Tiny C, they implement TinyIP in Tiny C for MinIPS. They are given a template of TinyIP implementation in Tiny C and write the missing code in accordance with Tiny C specifications. They compile and test their implementations in the following three environments: 1)gcc for compilation and Linux for execution; 2)Tiny C for compilation and MinIPS emulator for execution; 3)Tiny C for compilation and MinIPS real hardware for execution. In the first environment, they can test TinyIP stack independently of Tiny C and MinIPS. In the second environment, they can test TinyIP stack and Tiny C independently of MinIPS real hardware.

Finally in the third environment they integrate results

developed by team members into a complete Tiny J System. The integration follows the steps: 1)N, C, and P subgroups demonstrate respectively that TinyIP is running on Linux, that Tiny C generates code executable on MinIPS emulator, and that the MinIPS processor works by executing LED blinking program. 2)They synthesize MinIPS and load the TinyIP compiled by Tiny C. 3)They execute an echo server on Tiny J System and confirm that it can communicate with an echo client on the Linux box.

Screens displaying UDP/IP communications between MinIPS and Linux are shown in Figure 5. A window of the MinIPS console displays loading and execution of Tiny J object codes for the echo server. The Linux screen displays execution of the echo client on a window while monitoring the communications by `etherpeep` on the other window. The echo server in this figure turns upper/lower cases of received alphabets in transmission for ease of verification.

4 Results and Discussions

In the P sub-laboratory, according to steps reached by individual students, we gave hints at early stages to adjust their paces. In the first and second periods, eight of ten and ten of ten assignees completed the P sub-laboratory, respectively. Even students who could not complete the laboratory expressed in their reports a strong sense of accomplishments.

We accepted seventeen reports from the C assignees. All of them completed all the requisite assignments, and twelve students tried the optional enhancements. Examples of the enhancements made by students are: adding pre-increment/pre-decrement operators; extending the lexical analyzer so as to accept various sorts of integer literals.

All of the twenty N assignees succeeded in implementing TinyIP stacks for Linux. They enhanced their stacks for Linux in various ways. Table 1 shows the enhancements and the numbers of students who tried and finished the enhancements. Some of them made multiple enhancements.

Tools for checking individual components, for testing effects caused by interactions between components, and for verifying Tiny J integration as a system are required. Although some of them have been provided for students as mentioned in the previous section, the testing environment is still poor as a whole. Particularly in verifying the integration, it is not easy for students to create programs for checking expected behaviors, because situations covering exhaustive failures are difficult to produce. If we provided better test tools, they could verify their implementations more easily.

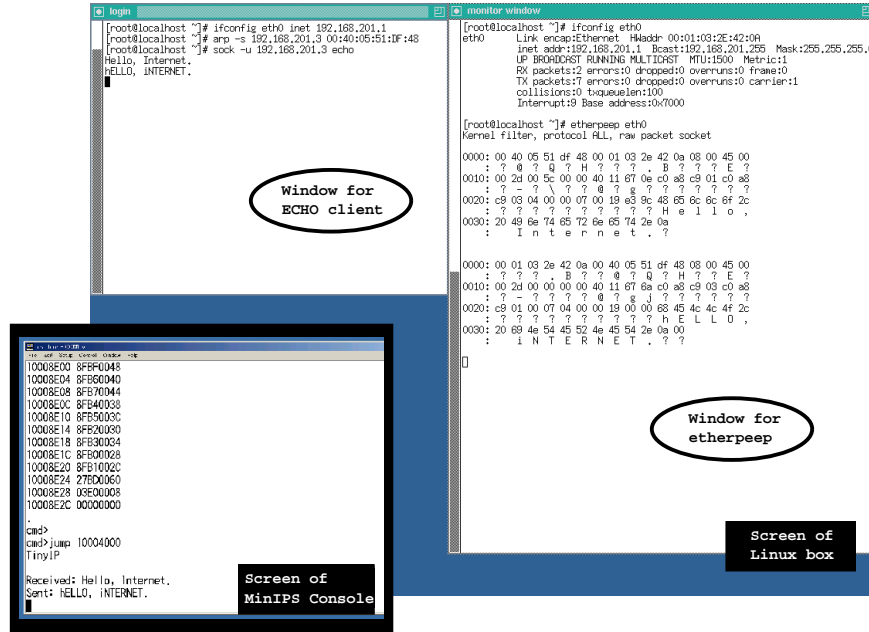


Figure 5: Screens displaying UDP/IP communications between MinIPS and Linux.

Table 1: Enhancements made by students.

Features	# of students who tried (finished)
Optimizing memory usage	8(8)
IP fragment transmission	4(2)
IP fragment reception	3(2)
ARP request	7(6)
ARP reply	5(5)
ICMP echo reply	4(3)
ICMP port unreachable	2(2)

We asked students to fill out a questionnaire provided by us to evaluate the laboratory from a student's point of view. A summary of the answers collected from P assignees after the first period of the laboratory is shown in Figure 6. The results show that it took long time for students to complete the laboratory compared to regular 36 hours: for example, 10 to 15 extra hours needed for 60% of the P assignees. However, we can see that 90% of the students understood the laboratory and 100% of them enjoyed it. Almost the same responses have been obtained about levels of understanding and attractiveness from other sub-laboratory assignees. In spite of the large and tough laboratory, three teams out of five were successful in the integration of C, N, and P components. We observed many scenes where shouts for joy arose from around upon succeeding in the CNP integration. This

is considered to be another proof that CNP laboratory is successful in giving the students a sense of accomplishment.

Some students in the first period, however, complained that they did not understand well what other subgroups were doing. This suggests a need of some devices for students to be more aware of other subgroups.

From the suggestion as well as our experiences on the first period of the CNP laboratory, we introduced *progress check sheet*, a sheet to record the progress of each member in a team. Three columns of the sheet list steps of P, C, and N sub-laboratories in time sequence. A row of the sheet shows the current progress of the members in a team. They put the date when they have finished a step. All the students in the same team share a sheet and can see what other subgroups are currently working on. We expected that by sharing the sheet they will feel a sense of cooperations, stimulating more active communications. We also intended by the sheet that students obtain a cross-cutting view of the TinyJ system.

Students in the second period answered that the progress check sheet helped them be aware of other subgroups and understand what they are doing. They also answered that they were able to collaborate smoothly with other subgroups thanks to the check sheet. Probably due to the boosted collaboration, all of the five teams were successful in the CNP integration in the second period, which is one of the distinguished improvements

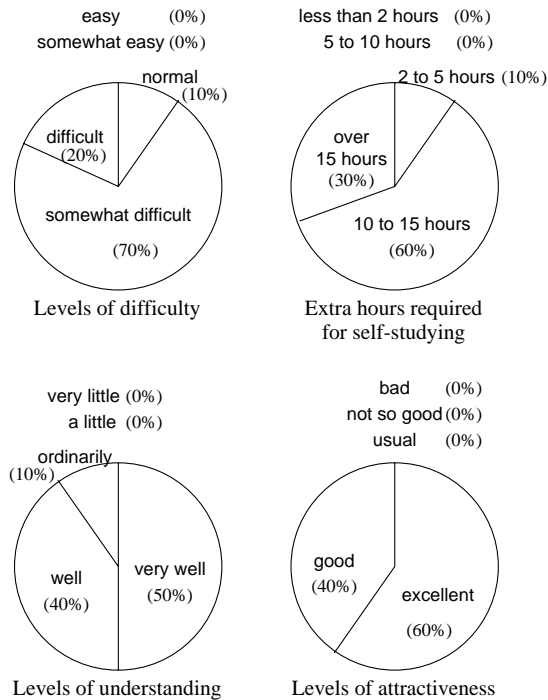


Figure 6: Summary of the questionnaire.

from the first period.

5 Conclusions

An integrated laboratory dealing with computer networks, compiler design, and computer organization has been developed. In the laboratory, students understood the assigned components and their interfaces with other components. After discussing and adjusting their specifications, they designed and implemented these components, and integrated them cooperatively into a system. The goals of the integrated laboratory have been proven to be fulfilled from the response of students who performed the laboratory, approving our intention of the laboratory and verifying its effectiveness.

Several improvements have been made to encourage students' cooperations. But we are aware of a lack of testing methodology. Two approaches are considered: 1) giving them a set of test suites; 2) teaching the way of testing. Both approaches are to be brought into the laboratory, which belong to future works.

Acknowledgements

The authors are grateful to Prof. Tan Watanabe at UEC, the original TinyC inventor, who has been supporting our work with many respects. Mr. Masato Naraoka at UEC

contributed to maintaining laboratory equipments. We also thank to members of Abe lab. for developing many peripherals for Tiny J. Special thanks are due to students who challenged the laboratory with great interests and contributed to many improvements.

References

- [1] R. B. Brown, R. J. Lomax, G. Carichner, and A. J. Drake. Microprocessor design project in an introductory VLSI course. *IEEE Trans. of Education*, 43(3):353–361, 2000.
- [2] D. Kassabian and A. Albicki. A protocol test system for the study of sliding window protocols on networked UNIX computers. *IEEE Trans. Education*, 38(4):328–334, 1995.
- [3] T. Katsu, D. Oosuga, M. Tsuruta, and K. Abe. Design and implementation of a 32 bit RISC processor MinIPS. *Bull. of the Univ. of Electro-Comm.*, 10(2):71–78, 1997.
- [4] K. Morita and K. Abe. Implementation of UDP/IP protocol stack on FPGA and its performance evaluation. In *Proc. IPSJ General Conf. Special5*, pages 157–158.
- [5] The Joint Task Force on Computing Curricula IEEE-CS and ACM. *Computing Curricula - Final Draft*. <http://www.computer.org/education/cc2001/final/index.htm>, December 2001.
- [6] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface, Second Edition*. Morgan Kaufmann Pub., 1998.
- [7] J. Postel. Echo protocol. RFC 862, May 1983.
- [8] T. Watanabe. *Composing a compiler*. Asakura Pub., 1998.

A lab course of Computer Organization

J. Real, J. Sahuquillo, A. Pont, L. Lemus and A. Robles

{jorge, jsahuqui, apont, lemus, arobles}@disca.upv.es

Computer Science School

Department of Computer Engineering

Technical University of Valencia (Spain)

Abstract

Lecture topics in Computer Organization courses offered by different Universities around the world do not differ significantly. This is because, in general, lecturers use the same textbooks and are inspired by common curriculum sources. However, lab courses and project assignments require more and more expensive resources (computers, assemblers or assembler simulators, logic circuit simulators, ...) This fact, together with the rapid advance of these tools, causes lab courses to widely differ among universities.

This paper summarizes the lab course on Computer Organization offered this year at the Technical University of Valencia, Spain. The course is composed by several experiences and jobs, each one aimed at working on one specific topic. Our goal is not only to introduce the tackled topics, but also to discuss some characteristics of the tools. All the tools used are freely available, which is a must for the students to be more motivated and to be able to extend their work using their own computers at home.

1. Introduction and motivation

The Technical University of Valencia offers a three-year Bachelor degree course in Computer Engineering. A modification of the curriculum has recently been undertaken to adapt it to the new trends and professional outlines. The recommendations from the IEEE/ACM Computing Curriculum 2001, as well as curricula from some relevant Spanish and foreign universities have influenced the new design. The course includes 60 lab hours (25% of the total), distributed along two core courses in the first and

second year. Each course is attended by more than 800 students, which strongly impacts on the lab organization and the type of experiments. Up to 40 students attend each lab session, working in groups of two people. Theoretical lectures are attended by up to 120 students.

To properly design the lab course it is necessary to consider the contents of the theoretical courses, the academic year when they are given and, specially in our context, the high number of students, which this is not a trivial task. One of the main problems is to choose appropriate tools for the lab experiences. An excessive use of abstract simulators is a risk because some of them (specially those very *didactic*) are quite far from the real world. On the other hand, the contents of the Computer Organization subjects are very difficult to implement in a practical way without additional technical knowledge. Finally, the tools and equipment needed for the lab sessions tend to be expensive.

Some universities propose lab courses based only on a part of the subject (generally the part whose contents are easier to practice in the lab) and they do not cover, in a practical way, the whole theoretical contents. The main reason is usually the lack of appropriate tools to do it.

The structure of this paper is the following: section 3 briefly describes the theoretical course of Computer Organization; section 4 details the lab course, both describing the experiences and the needed tools; section 5 presents the time schedule of the theoretical and lab courses. Finally, section 6 summarizes our conclusions.

2. Computer Organization theoretical course

The Computer Organization course is a core subject of the Computer Engineering degree. This course is given along the first and second year of the degree, having assigned up to 180 lecture hours in all (90 lecture hours each year). Evaluation is performed in an annual basis.

The main goal of this course is to introduce the students to the organization of computer systems through the study of each one of the functional units that compose them. Topics include data representation, digital logic, assembly language, simple processors, memory unit, input/output unit, arithmetic-logic unit, basic pipelining, and advanced processors.

Tables 1 and 2 show the themes into which each topic is broken down and the number of hours assigned to them. This information corresponds to the syllabus of the first and second year courses, respectively.

Topic	Themes	Hours
Introduction	1. Introduction to computer systems	2
Data representation	2. Data representation	9
Digital logic	3. Basic concepts of digital systems	12
	4. Combinational systems	10
	5. Sequential systems: Flip-flops	4
	6. Sequential systems: Registers and counters	8
Assembly language	7. Introduction to assembly language	10
	8. Assembly programming	6
	9. Procedures	6
Simple processors	10. Datapath	10
	11. Control unit: Hardwired realization	8
	12. Control unit: Microprogrammed realization	5
Total hours		90

Table 1. Syllabus of the first year course on Computer Organization.

Topic	Themes	Hours
Memory unit	13. Memory system	3
	14. Memory system design	10
	15. Memory hierarchy	10
Input/Output unit	16. Input/output devices	9
	17. Input/Output management	12
	18. Buses	4
Arithmetic-Logic unit	19. Integer arithmetic unit: Adders and subtractors	6
	20. Integer arithmetic unit: Multiplication and division	8
	21. Floating-point arithmetic unit	4
Basic pipelining	22. Introduction to the pipelining	6
	23. Pipelined processor	12
Advanced processors	24. Examples of contemporary processors	4
	25. Introduction to multiprocessor systems	2
Total hours		90

Table 2. Syllabus of the second year course on Computer Organization.

3. The lab course

We propose a selection of experiences on Computer Organization, aimed at covering the classical computer functional units: processor, memory, and input/output system. The lab course goals complement those of the classroom course. We have designed and selected some experiences, trying to balance the course time among the mentioned functional units according to their importance. The aim is to acquire an elementary but complete knowledge about Computer Organization as well as its basic working principles and underlying design aspects. We also discuss the selection of a set of free software tools that allow those students requiring additional time, or those who show further interest, to continue their work at home.

The described experiences are organized in lab sessions, each taking two hours of work.

3.1 Experiences

Experience 1: Assembler

Three lab sessions are dedicated to implement simple assembly language programs. The topics are assembly instructions (bare machine) and pseudoinstructions, instruction coding, data representation, and functions in assembly

language, exercising the MIPS register usage convention.

The first session is an introduction to the PCSpim interpreter [spim02] that simulates how the assembler works for the MIPS architecture. The session lab is addressed to give the students practice with several features of the tool, and to strengthen some topics studied at the classroom, like character, integer and floating-point representation, as well as memory data alignment.

The second session has three types of exercises. The first one deals with the instruction coding. Students must codify some assembly language instructions and check if their results match to those given by the tool. The second one is addressed to check the results of some instructions that use predefined target registers (e.g., LO and HI for integer division and multiplication instructions). The last one is addressed to running a program that performs the scalar product of two vectors. Students must run the program and answer some questions: i) to determine which function it performs, ii) to identify the pseudoinstructions of the program, and iii) to explain why the assembler not always codifies a given pseudoinstruction by using the same machine instructions (e.g., the load address instruction).

In the last session, the students must break down the scalar product program in two parts: main program and procedure. The programs must be implemented by using the *callee-saved* as the *procedure call convention*.

Experience 2: The Processor

Three lab sessions are dedicated to the study of the central processing unit (CPU). The main goal of these sessions is to develop a simple CPU (no pipelining) that executes a reduced instruction set - a subset of the MIPS architecture [Patterson97]. The different CPU elements are interconnected by means of busses. The instructions include several arithmetic and logic operations, load and store, and different types of branch instructions, including unconditional, conditional and jumps to subprograms. These instructions permit to implement simple, though fully operating sample programs that can be traced during their execution, allowing the student to follow their steps in the datapath and the activation of the relevant control

signals. We use the Xilinx schematic editor and functional simulation tools to implement and test the resulting circuitry [Xilinx01].

The first session is an introduction to the tool itself, as this is the first time it is used. During this session, a register file is implemented and tested. It takes a long time to develop the whole register file, therefore an almost complete version is supplied for the students to complete and test it, according to a set of predefined experiments. The second session deals with a complete datapath, including a Program Counter, Arithmetic and Logic Unit, the memory interface and several auxiliary registers and very simple operators like fixed shifters and a sign extender. Most of these units are supplied in advance and the work to do consists in interconnecting units and testing the resulting datapath by executing isolated instructions. The third session completes the CPU implementation with a Control Unit (CU). It is based on a phase counter and the needed combinational logic to generate the 24 control signals required by the datapath. The students are required to complete the design of the CU by implementing a couple of control signals and then put it together with the datapath. The memory circuit contains a simple program with a loop that has to be tested.

Experience 3: Memory Design

This experience is organized in three sessions. The common goal of all is them is to understand how the memory system in a computer is designed, from the basic cell to the construction of memory modules based on smaller elements and including the decoding and selection system. For this purpose we use the simulation environment Xilinx as tool.

This first lab session deals with the internal structure of memory circuits. The students must design a small memory unit (16x1 bit). We propose this small size for practical reasons: the memory structure designed is also valid for larger memories; the only difference is the number of elementary cells and the size of the decoding circuits.

In the second session, we give the students a predesigned 32Kbytes RAM element, in order to build a 256 Kbytes memory module. The students

must pay special attention to access different types of data (bytes or 16 bits words). For checking purposes we supply a module that acts like a CPU, generating addresses and byte selection lines.

In this session, we supply a circuit that simulates a memory system composed by 4 different modules and a checking element that acts as an address generator. With all these circuits the students must implement different memory maps.

Experience 4: Cache Design

The goal of this session lab is to understand why cache memories are the basic and ineludible mechanism that computers incorporate to reduce memory accesses latency.

We give the students a small testing program written in C language (in similar manner to D. Patterson [Patterson01]), to experimentally determine the parameters of the computer's caches.

To perform the experiments the program defines an array of 1 mega integer elements size, and different scenarios are modeled. Each scenario is determined both by the amount of elements that are accessed (1K elements, 2K elements, ...) and by the stride (1, 2, 4, ..., 512K). The program has a main loop that runs repeatedly many times in which the elements of the scenario are accessed to measure the data access time. Then, all the resulting times are averaged. The loop execution time is relatively long (approximately 1 second) in order to get precision in the measure process.

From the results, the students must firstly notice the number of cache levels. Then, for each cache level they must determine: i) the block size, ii) the set associativity, iii) the cache size, iv) approximately how fast the cache hit is, and v) approximately how fast the cache miss is. Some other parameters about the memory hierarchy like the page size and the page fault penalty are also determined.

Experience 5: The input/output system

The main objective of this experience is to practice the basic methods of synchronization: status checking (polling) and interrupts. To achieve this, the students develop simple interactive programs by using the input/output available facilities.

In the first session, we present a hypothetical case of communication between a MIPS R2000 processor and two basic I/O devices: the keyboard and the printer. A simulator acts like these two devices mapped in memory positions. Both are character-oriented devices. The PC keyboard is used as the input device while data output is displayed in a window that simulates the printer. The students must write a small program in MIPS R2000 assembly language to read characters from the keyboard and print them in the printer. The program must use polling for synchronization and program-controlled for data transfer.

In the second part the students have the opportunity to practice interrupt handling in a real computer (PC compatible). They also can access the PC memory and I/O maps. We propose them two typical problems to solve: first, students must modify some of the system interrupts (clock and keyboard are the proposed ones) writing the appropriate routines to handle them. In a second step, they must extend the service given by an existing interrupt handler by linking the system routine with their own handler.

Experience 6: Circuits to Support Integer Arithmetic

The main objective of this experience is to design simple integer arithmetic circuits and to modify them to achieve better performance by using pipelining techniques. This experience is organized in three sessions. In the first one, the students must implement a 16 bit adder/subtractor for integer numbers by using 4 bit carry lookahead adders (CLAs). The basic circuits (half and full adders) that form the CLA must also be implemented. Next, they develop a fast multiplier for two 6 bit unsigned numbers by using a Wallace tree. For this purpose, they build and interconnect carry save adders. The last stage of the Wallace tree is built by using the already implemented CLAs. To complete the fast multiplier, the students must build a partial product generation circuit that takes the two integer operands as inputs and generates the six partial products to feed the Wallace tree. Finally, they have to split this multiplier circuit into pipeline stages. For this aim, the students must identify the pipeline stages and establish the suitable clock period to improve the circuit speedup. The students must simulate

and measure the response time. Moreover, they must calculate the speedup the pipeline achieves.

Experience 7: Pipelined Processor

The goals pursued in this lab session are to understand the concept of pipelining, identify hazards, realize how hazards affect performance, and to know how the different solutions for conflict solving are implemented.

A program that simulates the behavior of a pipelined DLX processor [DLXide] is used. The DLX processor [DLX02] exhibits a similar architecture to that of MIPS. In the simulator, instruction execution can be tracked in a time diagram, cycle by cycle, therefore it allows to follow their walk through the different stages. The simulator permits also to define a particular technique for hazard solving, including bubble insertions, forwarding, predict-not-taken branches and delayed branches. The datapath (shown by the simulator) appears modified according to the technique applied. Control signals, memory and register contents and some statistics are also made available by the simulator, which permits to extract some conclusions based on quantitative data.

A simple but illustrative assembler program is supplied for the students to trace its execution in the pipelined datapath. First, they must solve dependencies by inserting bubbles and then counting the resulting CPI. Secondly, more effective techniques such as forwarding and branch prediction are exercised, allowing to observe how these techniques work and to compare results with the previous experiments.

4.2 The tools

For the experiences described in the previous subsection, we are currently using different tools. Below, we briefly describe how we use them and how they allow us to reach the goals of the lab experiences.

1. **Logical board.** It is basically a circuit board with some logic gates and flip-flops that can be interconnected by means of wires and connectors. The board also allows for commercial integrated circuits to be added, thus increasing the number of different exercises that can be tackled. By

using real circuits and wires, the student realizes the difficulties in implementing real circuits (bad connections, collision of outputs, etc.) which are more difficult to detected when logical simulators are used. The logical board is used for the most basic circuits, leaving the complex ones to be simulated.

2. **MIPS simulator PCSpim.** For assembly language experiences, we use this free MIPS simulator to implement and trace simple programs. The simulator is complete enough for the intended purposes and makes it straightforward to work in assembly language without having to deal with particularities of the platform. On the other hand, it represents an important economical saving, as PC's are available in all of our labs, differently to MIPS-based computers.
3. **Xilinx schematic editor and simulation tools.** The Xilinx Foundation is an application framework for programming logical devices with logical functions of different levels of complexity, from very simple combinational functions to virtually any larger project with both combinational and sequential components, allowing for tristate devices as well as conventional ones. The tool is complex, if used as a whole, but for the purposes of the course, we only need to be able to specify a circuit and to simulate it. The Xilinx tool offers several ways of specifying a circuit, namely a Hardware Description Language, a Finite State Machine and a Schematic Editor. The last one is the most appropriate for our students, since this is the common way of describing circuits in the classroom as well. On the other hand, the simulator is a powerful tool that allows us to track the behavior of the specified circuit in connection with the schematic editor. Despite the complexity of the whole application, our students quickly learn where to click to carry on their work, since the working platform is well bounded from the very beginning of the corresponding lab exercises. This tool has

proven to be very suitable for implementing our simplified RISC datapath and for the control unit as well. It is also used in the exercises related with memory modules.

4. **DLXide** is a simulation tool of the DLX computer. This simulation tool has been developed by lecturers from the Computer Engineering department of the Technical University of Valencia with the aim of providing a suitable environment for performing pipelining experiences. The simulator is able to simulate the pipelining execution unit of the DLX computer in a cycle-by-cycle basis, also showing how the instructions progress through the pipelining stages. For simplicity, it only supports the integer instructions of the DLX architecture. The tool permits to edit, assemble, and execute a DLX assembly program. There exist separate cache memories for instructions and data. User can initialize and modify both machine registers and data memory contents, which are displayed in two separate windows. Moreover, it is possible to display the instruction memory contents and the instruction addressed by the program counter. Through the configuration window, the user can establish the mechanism used for hazard solving among the following techniques: stalls, predict-not taken, delay-slot 1, and delay-slot 3 for solving control hazards, and stalls and forwarding for solving data dependencies. Step-by-step simulation shows how each mechanism solves the hazards. The simulator runs on MS Windows and Linux operating systems.

5. 2. Coordinating theoretical and lab courses

The first and the second year theoretical courses are 30 weeks long organized in two weekly sessions 1.5 hours long, where both theory and problems aspects are lectured. Sessions take place in classrooms of 160 students capacity.

The lab courses have the same duration as the theoretical and their timing must be synchronized.

These courses are organized in two types of weeks (A and B), so that the type of the week alternatively changes from A to B and vice-versa. Students must attend to the lab sessions in those weeks they are registered in. Sessions are two hours long every two weeks and take place in labs of 40 students capacity. This has proven to be more suitable than having weekly sessions of 1 hour.

Table 3 shows the planning of the theoretical and the lab course of the first year. Numbers on the top row refers to the week number. Central row shows the planning (theoretical and problem sessions) of the themes. The first 15 weeks focus on the study of both the data representation and the digital logic topics (T3, T4, T5 and T6) mentioned above. Next, 7 weeks and a half are dedicated to the study of both machine and assembly languages. The remaining weeks are addressed to implement a simple datapath and its control unit (both hardwired and microprogrammed). The bottom row refers to the lab sessions. As it can be seen, lab sessions begin at the same time as classroom sessions. Some times; e.g., when studying simple data paths, the lab session starts a little bit before the theoretical topic is studied at classroom. This does not cause any inconvenient, because that time is devoted to study how the tool (Xilinx in this case) works.

Table 4 shows the temporal planning for the second year course detailed above. In this case, no overlap appears between the theoretical and the lab course.

WEEK NUMBER														
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			T2	T3			T4			T5	T6			
P 1				P 2							P 3			

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
T7			T8	T9	T10				T11	T12				
P 4							P 5							

Table 3. Planning of classroom and lab sessions of the first year course. Legend: P refers to practical experience and T to topic.

WEEK NUMBER															
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	T2		T3		T4			T5		T6					
P1					P2			P3							

17	18	19	20	21	22	23	24	25	26	27	28	29	30
T7	T8		T9	T10	T11			T12	3				
P3	P4							P5					

Table 4. Planning of classroom and lab sessions of the second year course. Legend: P refers to practical experience and T to topic.

6. Conclusions

In this paper we have presented a lab course on computer organization, and we conclude that a complete course needs the following requirements:

1. A set of tools of a very different nature (assembler, logical circuit simulator, pipeline simulator) to cover the whole theoretical course.
2. It is important that the tools be as close as possible to a professional tool (e.g. we are currently using the educational version of a professional tool.)
3. It is necessary to devote an important amount of time to learn how the tools work, therefore it is important to chose tools also used in other subjects (e. g. the Xilinx framework is used in Logical Design courses too.)

7. References

- [Patterson01] D.A. Patterson, Course CS61C1C: Machine Structures, UC Berkeley, <http://inst.eecs.berkeley.edu/~cs61c/fa01/calendar/week13/lab10/>, Fall 2001
- [Spim02] J. Larus, SPIM: a MIPS R2000/R3000 simulator, <http://www.cs.wisc.edu/larus/spim.html>, 2002.
- [Xilinx01] Jan Van der Spiegel. Xilinx Web page. <http://www.prenhall.com/xilinx/>, 2001.
- [DLX02] Computer Systems Laboratory, FTP Site for Interesting Software, <http://max.stanford.edu/max/pub/hennessy-patterson.software/max-pub-hennessypatterson.software.html>
- [DLXide] P. López. DLXide web page. <http://www.gap.upv.es/people/plopez/english.html>
- [Patterson97] D. A. Patterson and J. L. Hennessy, Computer organization and design: the hardware/software Interface, Morgan Kaufmann publishers, 2nd edition, 1997.

A Survey of Web Resources for Teaching Computer Architecture

William Yurcik
Illinois State University
wjyurci@ilstu.edu

Edward F. Gehringer
North Carolina State University
efg@ncsu.edu

Abstract

The use of Web resources is becoming a core part of teaching computer architecture. In this paper we identify five notable Web sites that specialize in teaching tools for computer architecture instructors and discuss the role they can play in facilitating learning. While these Web sites contain a wide range of valuable resources, there remain gaps in what is available online. Community support appears meager for making tools and resources available. We conclude that the computer-architecture community faces challenges both in the content of Web-based materials (accurate and appropriate information) and the process (making information known and available to academic community).

1.0 Introduction

Computer architecture is a difficult subject both to teach and learn for a plethora of reasons including—

- the dynamic nature of the subject, the lifecycle of current computer technology is arguably less than three years and decreasing rapidly
- the ever-expanding amount of relevant material, as new techniques are being developed continuously to build upon existing techniques
- the need for to understand disparate subjects, from electronic circuits to digital logic to assembly-language programming to system design, as well as higher level programming and discrete math and performance analysis and ...
- its lab component, requiring the design and execution of both hardware and software experiments, and
- increasingly higher levels of abstraction hiding more and more lower-level details.

Of course, computer architecture is not the only course facing these challenges, but it may be the one course that faces all of them simultaneously. One academic study of this situation found that even experienced computer architecture instructors found they are not confident or current in some topics considered core to the course [2]. Novice instructors and instructors teaching outside of their specialty area are in a worse situation.

Collectively, however, the computer architecture community possesses an impressive array of knowledge, experience, and tools for teaching the subject. In recent years, many of these resources have been migrating to the Web.

Finding the right resource for teaching a specific topic is problematic, so this paper seeks to provide an orientation to the current state-of-the-art in computer architecture education resources on the Web. The remainder of this paper is organized as follows: Section 2 describes in some depth the five major Web sites containing computer architecture educational resources. Sections 3 and 4 focus on the contrasting resource needs of new and experienced instructors in computer architecture. Section 5 attempts to identify gaps in what is available on the Web versus the needs of instructors and Section 6 seeks to understand why this gap exists. We close with a summary and conclusions.

2.0 Computer Architecture Education Web Sites

Reference 2 highlights the fractured state of computer architecture education, but there have been several attempts to address this problem via community effort. This section describes five significant computer-architecture education sites that contain valuable resources for the community. A survey of these Web sites also reveals unexpected insights into the current state of computer architecture education.

2.1 Computer Architecture and Assembly Language (CAALE)

<http://www.sosresearch.org/caale/>

An NSF-sponsored working group on "Distributed Expertise for Teaching Computer Organization" convened at the July 2000 Innovation and Technology in Computer Science Education (ITiCSE) conference in Helsinki Finland under the direction of Lillian (Boots) Cassel of Villanova University and Deepak Kumar of Bryn Mawr College. The two tangible products of this working group are the CAALE Web site and the seminal collaborative paper that identifies both current problems and potential future solutions for facilitating better computer-architecture education [2].

The goal of CAALE is to serve as a repository for Web-accessible resources identified by the working group, such as links to courses, people, textbooks, simulators, papers, organizations, relevant news items, career information, and conferences. Currently, CAALE is unevenly developed with many links containing no content. Work continues to populate the Web site.

CAALE makes its primary contribution with its comprehensive list and categorization of textbooks and simulators. Response to the CAALE simulator list especially has been immediate, continuous, and growing. It has facilitated data-mining of simulator resources, as presented in two recent papers [5,6]. Future plans include enhancing the interactivity of the Web site using XML integrated with database processing to enable queries to the Web site for information.

2.2 WWW Computer Architecture Page

<http://www.cs.wisc.edu/~arch/www/>

A long-time fixture in the computer architecture community has been the WWW Computer Architecture Page that is hosted at the University of Wisconsin-Madison (and mirrored in India and Japan). Though focused mainly on research, it contains downloadable versions of many simulators and compilers that could be used across a range of educational levels. The extensive content on the Web site include links to architecture projects, organizations, and tools such as simulators, compilers, benchmarks, and traces. It also has links to commercial

organizations, online publications, books, and newsgroups.

The WWW Computer Architecture Page makes its primary contribution with its comprehensive list of researchers, research groups, and conferences. This site is a one-stop virtual location for learning about the state of the art in computer architecture research, especially that emanating from educational institutions.

2.3 NETCARE

<http://punch.ecn.purdue.edu/Netcare/>

NETCARE (NETwork-computer for Computer Architecture Research and Education) is a Web-accessible distributed infrastructure of software tools and computing resources developed at Purdue University. It provides a common environment for testing, sharing, and evaluating tools for teaching and research in computer architecture and programming. It allows users to actually run tools in conventional Web browsers.

NETCARE was developed to address many of the hurdles mentioned in the introduction. Instructors need to obtain access to the hardware resources that meet their requirements, and then install it. They also need to support it, by disseminating documentation and answering questions, and develop educational content, such as tutorials and homework assignments. NETCARE performs all of these functions; small classes are able to use NETCARE facilities directly, while instructors of large classes can load the NETCARE software onto their own server.

Another important feature of NETCARE is its user interfaces. Research simulators often come with text-based interfaces. NETCARE wraps these in graphical interfaces that are tailored to the needs of novice users. This has the advantage of presenting a number of tools with similar interfaces, thus facilitating the task of learning to use them.

NETCARE currently provides 16 tools for computer architecture, including the uniprocessor simulators Daisy, DLX-View, Shade, SimpleScalar, MySimpleScalar, XSpim, and 68HC12 Simulator; the multiprocessor simulators HPAM Sim, RSIM, WWT2, and WWT2H, and cache simulators CacheSim5, CACTI, and DineroIV. Accounts may be

requested by filling out a form at the NETCARE home page.

2.4 Computer Architecture Course Database < <http://cd.csc.ncsu.edu>>

In addition to simulation projects, computer architecture courses include other homework problems and, of course, exams. These materials are also potentially reusable. The Computer Architecture Course Database currently contains about 1000 problems suitable for use on homework or tests, many with solutions. The goal of the project is to encourage instructors to share materials. When an instructor grants permission, material is downloaded from the Web and semiautomatically loaded into the database, where it can be located by keyword or fulltext search. Anyone with an account on the system is granted the right to reuse the material in his or her own classes, but not to republish it.

Because it has proved to be much easier to induce instructors to use the database than to get them to contribute material, an alternative means of finding material has been provided in the form of a search engine that searches computer architecture sites at educational institutions around the world. A single request can search both the database and the Web. While material retrieved from the Web may not be freely reused, it is possible to seek permission from the copyright holder (usually the instructor who established the site). Accounts may be requested by e-mail to efg@ncsu.edu.

2.5 SIGMicro <<http://www.acis.ufl.edu/~microWeb/>>

ACM SIGMicro, the Special Interest Group on Microarchitecture, launched a Web site in 2001. Called the Computer Microarchitecture Center, it contains an education section with a listing of microarchitecture courses and course Websites. It also has links to most of the other resources mentioned in this paper. An interesting section is the new Reviews area, which is intended to contain reviews of educational tools and documents. This area is awaiting its first entry. It also contains pointers to the proceedings of several past WCAEs.

3.0 Resources for New Instructors

New instructors, and experienced instructors teaching outside of their area of expertise, desire directed teaching resources focused on getting started and survival skills in the classroom such as—

- Web syllabi of similar courses at different universities
- identification of textbooks bundled with teaching aides (slides, test banks, software)
- homework, project, and test problems with solutions
- visual and intuitive simulations of computer architecture concepts to promote active learning
- contact information for other computer architecture instructors (support group)

Current Web sites can provide many of these resources efficiently if the new instructor knows where to look.

New instructors need to learn (1) “best practices” for teaching computer-architecture topics, and (2) the resources that are available for them to use and tailor to their own teaching environment. The first goal (best practices) could be addressed by cross-referencing resources so that it is possible to see which textbooks, simulators, etc. are used by which types of courses, and which ways of teaching particular topics have become the “consensus” approach of the discipline. The second goal (breadth of resources) can be addressed by encouraging the worldwide computer architecture to place innovative resources on the Web and make them available to anyone over the Internet.

4.0 Resources for Experienced Instructors

After teaching a course for a few semesters, an instructor is likely to have a repertoire of lectures. The main challenges at this point are developing new homework assignments, labs, and exams. For homework assignments and exam questions, the Computer Architecture Course Database can be very helpful. It contains many questions on the Hennessy/Patterson texts, and microarchitecture in general, with caches

being the most widely covered topic. However, more contributions are being sought, as detailed in Section 6.

For lab projects, experienced instructors might consider the simulators available through NETCARE and CAAL. WCAE has published several papers related to simulators. Eight of these are still available on the Web. Two of these are targeted at the DLX architecture used in Hennessy and Patterson's *Computer Architecture: A Quantitative Approach* [7]; Dan Hyde's VHDL approach [8] and the DLX-view [9] simulator. Two of them use the MIPS architecture, a SimpleScalar enhancement from Manjikian [10] and MipsIt from Brorsson [11]. One targets Patt & Patel's LC-2 architecture [12]. The others are RSIM, a simulator for ILP-based shared memory multiprocessors and uniprocessors [13], SATSim, a superscalar architecture trace simulator using interactive animation [14], and esim, a design language simpler than VHDL, implemented in Tcl, in which students can build and simulate digital modules [15].

Experienced instructors also face the challenge of remaining current in the field. While some teaching resources lend themselves as a base upon which to build the future, many new tools will need to be developed from scratch. This makes tool development environments for experienced instructors an important area of investment.

5.0 What is Missing?

While the Web sites we have identified contain invaluable educational content, there are still critical voids that need to be addressed. The most glaring omissions include:

- a teaching computer architecture virtual support group
- implementation experience with the new ACM/ABET Computing Criteria 2001 for computer architecture-related courses
- a pooling of teaching resources, with Web sites being one forum but not necessarily the only forum

While progress has been made, it must be accelerated. Novice educators must be guided to

teaching resources and experienced educators can become disconnected from current mainstream teaching resources. In both instances Websites can provide a glue to maintain healthy teaching relationships and professional growth in the field.

6.0 The Tragedy of the Commons

In his classic 1968 paper, "Tragedy of the Commons" [4], Garrett Hardin illustrates that an open resource owned collectively and shared by all (a "commons") will be exploited by free-riders until depletion. Without the property rights of ownership, there is little or no incentive to contribute to care of the commons.

We apply this metaphor to Web site content for teaching computer architecture - there are few incentives beyond altruism to share teaching resources. Most instructors do not contribute and yet gain from the hard work of a select few. There is a need to either increase incentives to share resources or make it easier to do so.

In our work on the Computer Architecture Course Database, we found that only 29 of 73 instructors contacted agreed to contribute their materials in electronic format to our database [3]. Those who declined to contribute were asked why. We heard from about a dozen of them. Their concerns were divided about equally into two categories.

- 1) *Copyright concerns.* Some instructors could not contribute because their materials had borrowed heavily from copyrighted works, such as textbooks, making their course materials "derivative works." Others were writing textbooks and wanted to include their course materials, but feared that making their material available in advance would compromise the market for their books.
- 2) *Diffidence.* Many other instructors were concerned that their materials were not polished enough, either because they were teaching a course for the first time, or because they had not been able to devote enough attention to it. This concern has also been noted by Cassel [1]. Her advice is, "Get over it!" Only by access to shared materials can

we eliminate this perception of inadequacy.

To give instructors an incentive to contribute, a feature is currently being added to the Computer Architecture Course Database to track how often specific items have been downloaded. A high reuse count will indicate a problem or lecture that other instructors find quite useful. This would be one of the few quantitative measures of teaching contributions (beyond student course evaluations), and could help buttress cases for tenure and promotion.

7.0 Summary

This paper reviews several computer-architecture education Web sites found valuable to both novice and experienced instructors. The goal is to provide instructors both a general educational introduction to the broad field of computer architecture as well as detailed resources for more in-depth inquiry. While valuable resources do exist, making them known and available to educators has been problematic. In addition, the field is a moving target such that new ideas and technology are being continually introduced making collective sharing of appropriate resource materials a difficult task. There is hope, however, in that the five developing Web sites noted in this paper represent a diversity of accessible teaching resources in both depth and breadth and may be complemented by additional Web sites in the future.

8.0 Acknowledgments

The CAALE Web site is supported in part by the following grant from the National Science Foundation (NSF) USA: #99-51352 from NSF-DUE-CCLI/EMD. The Computer Architecture Course Database is supported by the NSF Course, Curriculum, and Laboratory Improvement program under grant DUE #99-50318.

9.0 References

- [1] Cassel, L., SIGCSE award luncheon address, *32nd SIGCSE Technical Conference on Computer Science Education*, February 24, 2001.
<<http://lcassel.csc.villanova.edu/sigcse.ppt>>
- [2] Cassel, L., Kumar, D. et. al. "Distributed Expertise for Teaching Computer Organization and Architecture," *ACM SIGCSE Bulletin*, Vol. 33 No. 2, June 2001, pp. 111-126.
- [3] Gehringer, E., and Louca, T., "Building a Database and Search Engine For Reuse of Course Materials," *Proceedings of Frontiers in Education 2001 (ASEE/IEEE)*, Session F3C.
- [4] Hardin, G. "The Tragedy of the Commons," *Science*, Vol. 162, 1968, pp. 1243-1248.
- [5] Wolffe, G. S., Yurcik, W., Osborne, H., and M.A. Holliday. "Teaching Computer Organization/Architecture With Limited Resources Using Simulators," *SIGCSE 2002, 33rd Technical Symposium on Computer Science Education, SIGCSE Bulletin*, Vol. 34, No. 1, March 2002, pp. 176-180.
- [6] Yurcik, W., Wolffe, G.S., and M.A. Holliday. "A Survey of Simulators Used in Computer Organization/Architecture Courses," *Summer Computer Simulation Conference (SCSC)*, Society for Computer Simulation, 2001.
- [7] Hennessy, John L., and Patterson, David A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, 1997.
- [8] Hyde, Daniel C., "Using Verilog HDL to teach computer architecture concepts," *Proc. WCAE 98, Workshop on Computer Architecture Education*, June 27, 1998, Barcelona, Spain. Tools available at <http://www.eg.bucknell.edu/~cs320/Fall2001/verilog.html>
- [9] Zhang, Yiong, and Adams, George B., "An Interactive, Visual Simulator for the DLX Pipeline," *Proceedings WCAE-3, 3rd Workshop on Computer Architecture Education*, San Antonio, TX, Feb. 2, 1997. Published in *IEEE Computer Architecture Technical Committee Newsletter*, September 1997, pp. 25-31. Tool available at <http://yara.ecn.purdue.edu/~teamaaa/dlxview>
- [10] Manjikian, Nairag, "Enhancements and applications of the SimpleScalar simulator for undergraduate and graduate computer architecture education," *Proceedings WCAE 2000, Workshop on Computer Architecture Education*, Vancouver, BC, June 10, 2000. Published in *IEEE Computer Architecture Technical Committee Newsletter*, September 2000, pp. 34-41. Tool available at <http://www.cs.wisc.edu/~mscalar/simplescalar.html>
- [11] Brorsson, Mats, "MipsIt: A simulation and development environment using animation for

- computer architecture education, *Proceedings WCAE 2002, Workshop on Computer Architecture Education*, Anchorage, AK, May 26, 2002, pp. 65–72. Tool available at <http://www.embe.nu/mipsit>
- [12] Cohen, Albert, “Digital LC-2: From bits and bytes to a Little Computer,” *Proceedings WCAE 2002, Workshop on Computer Architecture Education*, Anchorage, AK, May 26, 2002, pp. 61–64. Tool available at <http://www-rocq.inria.fr/~acohen/teach/diglc2.html>
- [13] Pai, V. S., Ranganathan, P., and Adve, S. V., “RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors,” *Proceedings WCAE-3, 3rd Workshop on Computer Architecture Education*, San Antonio, TX, Feb. 2, 1997. Published in *IEEE Computer Architecture Technical Committee Newsletter*, September 1997, pp. 32–38. Tool available at <http://www-ece.rice.edu/~rsim/dist.html>
- [14] Wolff, Mark, and Wills, Linda, “SATSIm: A superscalar architecture trace simulator using interactive animation,” *Proceedings WCAE 2000, Workshop on Computer Architecture Education*, Vancouver, BC, June 10, 2000. Published in *IEEE Computer Architecture Technical Committee Newsletter*, September 2000, pp. 27–33. Tool available at <http://www.ece.gatech.edu/research/pica/SATSIm/satsim.html>
- [15] Miller, Ethan and Squire, Jon, “esim: A structural design language and simulator for computer architecture education,” *Proceedings WCAE 2000, Workshop on Computer Architecture Education*, Vancouver, BC, June 10, 2000. Published in *IEEE Computer Architecture Technical Committee Newsletter*, September 2000, pp. 42–48. Tool available at <http://www.cs.umbc.edu/~squire/esim.shtml>