Workshop on Computer Architecture Education

Saturday, June 19, 2004

The Changing Role of Computer Architecture Education within CS Curricula

(invited presentation)

Reiner Hartenstein
TU Kaiserslautern
http://hartenstein.de

More than 98% of all microprocessors are found within embedded systems. The British Department of Trade and Industry predicts, that by the year 2010 more than 90% of all program code will be implemented for embedded applications. But the qualification of our „typical“ CS (and CIT) graduates is torpedoed by deficits of our current computer architecture education limiting the horizon to procedural programming in the time domain. However, meanwhile the fundamental machine model is no more just the „von Neumann“ paradigm merely supporting an instruction-stream-based mind set. For embedded systems the basic model is a symbiosis between CPU and primarily data-stream-based accelerator co-processors. Implementing applications for embedded systems also requires hardware / software partitioning decisions. Since meanwhile morphware [1] [2] and Reconfigurable Computing (RC) has become mainstream, also the accelerators are programmable by loading configware code downto their hidden RAM [3]. What is urgently needed is the qualification for programming in time (programming software) and programming in space (programming configware). But currently the software for the CPU is mainly implemented by software people, whereas the accelerators are implemented by EEs or other hardware people.

Communication problems between these two groups of experts having different backgrounds are the reason for the deep chasm between RC and the way, how "classical" CS people look at parallelism [4]. The situation is comparable to the well-known hardware / software chasm. In education until recently RC has been mainly the subject of embedded systems or SoC design within EE departments, whereas most classical CS departments have ignored the enormous additional speed-up opportunities which can be obtained from this field. Only a few departments provide special courses mostly attended by a small percentage of graduate students. Conferences like ISCA have stubbornly refused to include RC and related areas in their scope. Also many major players in the IT market have mainly ignored this area.

More recently this situation is beginning to change. An increasing number of colleagues from the area of computer architecture as well as from classical parallel computing or supercomputing communities is going to be ready to discuss fundamental issues with us [5] [6] [7]. Last year, Intel Research at Hillsboro, Oregon, held a major internal workshop on RC. It has been told, that also Microsoft has held an internal workshop on this area. Other major players already joined this movement, like Hewlett Packard, IBM, infineon, Motorola, Sony, ST microelectronics, Texas Instruments, Toshiba, and many others. Accordingly a major break-through also in CS education and CAE is overdue. All scientific know-how ingredients needed are available - ready to be integrated in CS curricula: software / configware co-compilation [8] [9], software to configware migration [10] [11], mapping applications onto morphware [10] [11] [12] [13], architectural resources for data-stream-based anti machines [14] [15], and many others. Not only FPGAs, but also coarse grain data path array platforms are available commercially, along with application development tools [16].

It is time to take these promising opportunities to upgrade our CS curricula by converting programming and software engineering into a duality of software engineering and configware engineering, based on the co-existence of two machine paradigms, the classical instruction-stream-centered model of the CPU, and, the data-stream-based anti machine model, being the direct counterpart of von Neumann. Because of this duality of basic models the configware / software chasm can be easily bridged without requiring hardware and circuit expertise from our CS graduates.

This new road map is based on the duality of an instruction-stream-based mind set, and a data-stream-based mind set [1] [4] [15] [17]. Not only the HPC community urgently needs to benefit from a curricular revision, but also the rapidly increasing percentage of programmers implementing code for embedded systems. However, most CS graduates are not qualified for this changing labour market. With their procedural-only mind set they cannot cope with hardware /

configware / software partitioning. To avoid a disaster for future CS graduates looking for their first job, CS departments have to wake up. Here we have a good chance to become successful trailblazers by forming a RC old boys' network together with colleagues from "classical CS", organized like the Mead & Conway movement more than 20 years ago [18].

References

[1] R. Hartenstein (invited chapter): Morphware; in: A. Zomaya (editor): Handbook of Innovative Computing; LNCS series, Springer Verlag Heidelberg/New York, 2004

[2] http://morphware.net/

[3] http://configware.org/

[4] R. Hartenstein (invited paper): The Digital Divide of Computing; Proc. 2004 ACM Int'l Conf. on Computing Frontiers (CF04); Ischia, Italy, April 2004

[5] R. Hartenstein (keynote): Software or Configware? About the Digital Divide of Computing; IPDPS 2004, Santa Fe, NM, 2004

[6] R. Hartenstein (opening keynote): Reconfigurable HPC: torpedoed by Deficits in Education? 1st Workshop on Reconfigurable High Performance Computing (RHPC); 7th Int'l Conf. on High Performance Computing and Grid in Asia Pacific Region (HPC Asia 2004), Omiya Sonic City, Japan July 20-22, 2004

[7] R. Hartenstein (invited paper): Data-Stream-based Computing and Morphware; Joint 33rd Speedup and 19th PARS Workshop (Speedup / PARS 2003), Basel, Switzerland, March 2003

[8] K. Schmidt et al.: Combining Structural and Procedural Programming by Parallelizing Compilation; Proc. 1995 ACM Symp. on Applied computing, Nashville, Tenn., Feb 1995

[9] J. Becker et al.: Parallelization in Co-Compilation for Configurable Accelerators; Proc. ASP-DAC'98, Yokohama, Japan, Febr 1998

[10] R. Kress et al.: A Data Path Synthesis System for the reconfigurable Data Path Architecture; Proc. ASP-DAC 1995, Chiba, Japan, August 1995

[11] U. Nageldinger et al: KressArray Xplorer: a new CAD Environment to optimize Reconfigurable Data Path Arrays; Proc. ASP-DAC 2000, Yokohama, Japan, Jan 2000

[12] R. Hartenstein (embedded tutorial): A decade of Reconfigurable Computing: a Visionary Retrospective; Proc. DATE 2002, Munich, Germany, March 2002

[13] R. Hartenstein (embedded tutorial): Coarse Grain Reconfigurable Architectures; Proc. ASP-DAC 2001, Yokohama, Japan, Jan 2001

[14] A. Hirschbiel et al.: A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware; Proc. InfoJapan'90, Tokyo, Japan, 1990 -- Invited reprint in: Future Generation Computer Systems 7 91/92, p. 181-198, North Holland

[15] M. Herz et al. (invited paper): Memory Organization for Data-Stream-based Reconfigurable Computing; Proc. IEEE ICECS 2002, Dubrovnik, Croatia, Sept 2002

[16] http://pactcorp.com

[17] J. Becker et al. (solicited paper): Configware and morphware going mainstream; Journal of Systems Architecture, vol. 49, Issue 4-6 (Sept 2003)

[18] R. Hartenstein (opening keynote): Are we ready for the Breakthrough?; 10th Reconfigurable Architectures Workshop 2003 (RAW 2003), Nice, France, April 2003

A Computer Architecture Education Curriculum through the Design and Implementation of Original Processors using FPGAs

Yutaka Sugawara, Kei Hiraki

Department of Computer Science
University of Tokyo
Bunkyo-ku, Tokyo, Japan
{sugawara, hiraki}@is.s.u-tokyo.ac.jp

Abstract

In this paper, we present the new curriculum of the processor laboratory of the Department of Computer Science at the University of Tokyo. This laboratory is a part of the computer architecture education curriculum. In this laboratory, students design and implement their own processors using field-programmable gate arrays (FPGAs), and write the necessary software. In 2003, the curriculum of the laboratory was changed, the main change being that the FPGA was changed to a large one to increase the range of design trade-offs. As a result, students have been enabled to implement the techniques used in modern processors such as FPU, cache, branch prediction, and superscalar architecture. In this paper, we detail the new curriculum and note the educational results of the year following the changes. Especially, we focus on the educational advantages of the large FPGA size.

1 Introduction

In architecture education, understanding existing architectures and acquiring skills to design new architectures are important goals for the students. Typical architecture education curriculums include both lectures and laboratories to achieve the goals. Concerning the lectures, architecture educations based on quantitative approaches are established using popular text books such as one written by Hennessy and Patterson [1]. However, students cannot acquire skills to design new architectures from just lectures. Laboratories are necessary to train students how to design new architectures.

For that reason, the processor laboratory [2] was introduced in the Department of Computer Science at the University of Tokyo. The laboratory started in 1992 as a part of the computer architecture education curriculum. In the laboratory, junior students design and implement processor systems using FPGAs. They build entire computer systems including processor architecture and software. The laboratory's main goals for the students are:

1. To precisely and concretely learn the internal structure and behavior of processors
2. To acquire a sense of trade-offs in processor architecture design
3. To experience the trade-offs involved in an entire system including software and hardware

The first goal is important because being familiar with basic processor structure is important for a good understanding of architecture techniques. The second goal is important because selecting an optimal architecture under a given condition is the most important topic in designing computer architecture. The third goal is important for understanding how the performance of a computer system is affected by each of its various parts.

The first goal cannot be completely achieved from just lectures because omitted structures of processors are explained in typical cases. Even when all the signals in a processor are shown, it is hard to explain changes of signals when each instruction is executed. On the other hand, when students implement their own processors in the laboratory, they can more precisely and concretely understand the behavior of all the internal signals of processors.

In lectures, a limited part of the second goal is achieved when the targets of trade-offs are detailed by the lectures. Furthermore, some parts of the trade-off conditions are often ignored to simplify the problem. However, when the students design their own architectures in the laboratory, they can experience a wide range of real trade-offs.

The third goal is not achieved in lectures because it is hard to precisely model the trade-offs of an entire system. On the other hand, when students build whole systems of their own including software and hardware, they experience real trade-offs in respect to entire systems. Especially, they can learn how to divide functions between hardware and software.

Because FPGAs are used in the laboratory, students can immediately run the processors they have designed.

Therefore, students can try many design alternatives to experience trade-offs of the architecture and of the entire system.

The curriculum of the processor laboratory was changed in 2003, the main purpose being to increase the range of architecture design trade-offs students can experience. Therefore, we changed the FPGA used in the laboratory to a large one.

In the previous curriculum, 5K-gates FPGAs were used. Therefore, though students learnt many architecture techniques in lectures, most of the techniques could not be implemented in the laboratory because of the FPGA size limitation. For example, it was impossible to implement techniques used in modern processors such as FPU, cache, branch prediction and superscalar architecture. 1M-gates FPGAs are used in the new curriculum; therefore, students can implement most of the techniques learnt in lectures as long as they have enough development power. In this paper, we present the details of the new processor laboratory curriculum and the educational results of year following the introduction of the changes. Especially, we focus on the educational advantages of the large FPGA size.

In Section 2, we explain both the previous and the new curriculum in the processor laboratory of our department. Section 3 shows the educational results of the past year in the laboratory. In Section 4, we present related works, and Section 5 concludes the paper.

2 The Processor Laboratory Curriculum

2.1 Previous Curriculum

Until 2002, Xilinx XC4005, a 5K-gate FPGA, was used in the processor laboratory. Students were divided into groups of five or six members. The goal of the laboratory was to run a ray tracer on students' original processor systems as fast as possible.

Each group builds a processor system board, an example is shown in Figure 1. Wrapping wires are used for connecting the components. The processor was implemented on an XC4005 FPGA. Total of 256KB SRAMs and 128KB ROMs were available to implement the memory system. The processor board could communicate with workstations via uPD71051 serial I/F. This serial I/F was used for scene data input and image data output of the ray tracer running on the processor board. The students were entirely responsible for the design of the architecture of their processor, the memory system, and the I/O system on the board.

The software tools were also made by the students themselves. Each group developed a runtime library, a cross assembler, a simulator, and a cross compiler for the processor system. The runtime library included communication routines of the serial I/F and floating point calculation primitives.

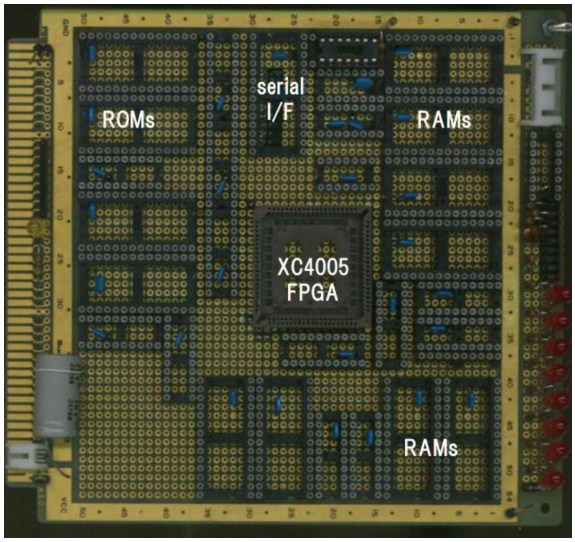


Figure 1: Example of a processor board used in the previous curriculum

The main advantages of the previous curriculum were (1) students could build and understand the entire system, and (2) they could learn what functions should be implemented using hardware when hardware size is limited. However, because of the FPGA size limitation, students could not implement most of the architecture techniques they learnt in lectures.

2.2 New Curriculum

The curriculum of the processor laboratory was changed in 2003; the main purpose being to enable students to implement most of the architectures they learnt in the lectures, something that was not achieved in the previous curriculum. The main change was to increase the FPGA size; thus, the Xilinx XC2V1000, a 1M-gates FPGA was introduced.

However, the change of the FPGA meant that building "a whole system" was no longer possible. This is because dedicated system boards are used for the laboratory and students are not required to wire components. When the components are wired using wrapping wires, the resulting circuit cannot operate at enough clock speed for the new FPGA. Furthermore, pin pitches of modern chips are too fine to be wired by hand. Therefore, we gave up making the students wire the boards by themselves.

The new system board is shown in Figure 2. The FPGA board includes an XC2V1000 FPGA, total 4MB of synchronous SRAMs, 128MB PC100 SDRAM, and USB I/F. An extension board is used for implementing additional I/O circuits by hand. In the laboratory last year, most groups implemented 7-segment LED arrays on the extension boards for debugging.

In a contest at the end of the semester, students present the processors they have made in the laboratory, and the performances of the processors are evalu-

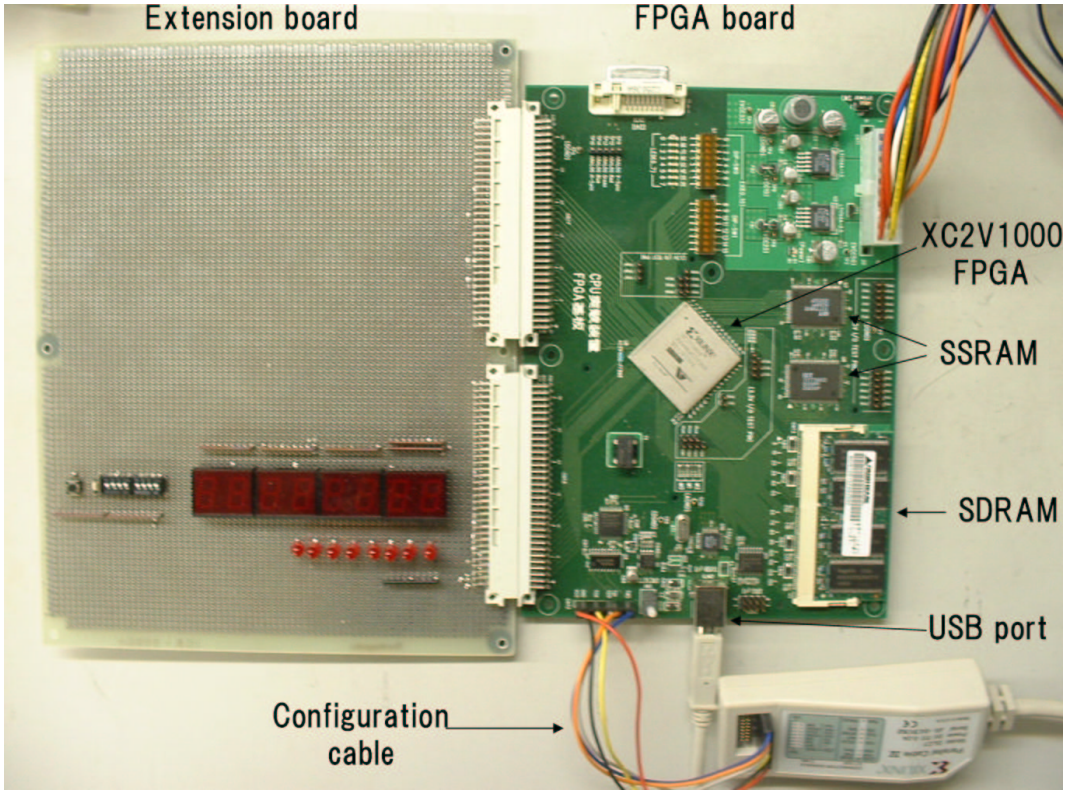


Figure 2: The new curriculum processor board

Figure 3: Output image of the new ray tracer

ated using a benchmark program. The benchmark program is an extended version of the ray tracer used in the previous curriculum. The output image of the new ray tracer is shown in Figure 3. Xilinx ISE6.1i tools are used for processor design. Table 1 shows comparisons between the previous and new curriculums.

	previous	new
FPGA	XC4005 (5K gates)	XC2V1000 (1M gates)
memory	SRAM (256KB, 100ns), ROM (128KB, 100ns)	SSRAM (4MB, 100MHz), SDRAM (128MB, PC100)
I/O	uPD71051	FTDI245(USB), etc.

Table 1: Comparisons between the previous and new curriculums

3 Educational Results

3.1 Design Result

In 2003 in the processor laboratory, the first year of the new curriculum, 6 groups designed processor systems. Table 2 shows the results of each group. The score is the execution time of the ray tracer measured in the end of semester contest.

As shown in Table 2, all groups implemented floating point units, and two groups implemented caches, important for the performance of modern processors. Implementing these techniques would have been impossible in the previous curriculum. Figure 4 shows a block diagram of the processor designed by group 1 [3] in Table 2.

group No.	Clock (MHz)	score (sec.)	features
1	50	35	FPU, pipeline I-cache, D-cache
2	50	45	FPU, pipeline I-cache, D-cache
3	40	178	FPU
4	12.5	548	FPU
5	50	641	FPU
6	50	N/A	FPU, pipeline

Table 2: Contest results of each group

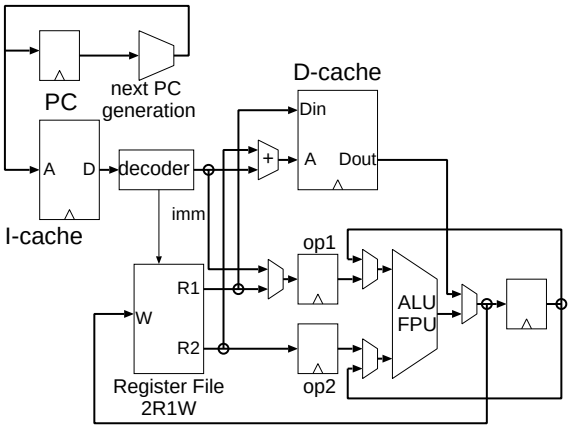


Figure 4: Block diagram of group 1's processor

In addition, 2 senior students voluntarily designed processors during the past year. One student implemented a processor with dynamic instruction scheduling using Tomasulo's algorithm. The other student designed a 2-way chip multiprocessor architecture. Furthermore, another two students are now voluntarily designing 4-way superscalar, and SIMD architectures, respectively, by extending processors they designed in the laboratory. Since these processors require many resources, they could not have been implemented using the FPGA of the previous curriculum.

3.2 Achievement of Educational Goals

Understanding Processor Structure In both the previous and the new curriculums, most of the students succeeded in implementing complete processors. This result shows that they gained an understanding of the structure of operational processors. Therefore, both curriculums successfully achieved this goal.

Acquiring a Sense of Architecture Trade-offs As detailed in Table 2, students successfully implemented FPUs and caches under the new curriculum. Further, the large FPGA enabled some eager students to implement more challenging techniques such as Tomasulo's algorithm, chip multiprocessor architecture, superscalar architecture, and SIMD architecture. None of these techniques could have been implemented using the FPGA used in the previous curriculum. Therefore, the new curriculum enabled students to experience wider ranges of trade-offs than in the regime of the previous curriculum.

Learning Trade-offs of Entire System As described in Section 2, board wiring is unnecessary in the new curriculum because a dedicated system board is used. Therefore, in the new curriculum, some of the trade-off conditions of the processor system are fixed; whereas in the previous curriculum, the processor board was fully designed and wired by students. In this respect, the previous curriculum was better than the new one.

4 Related Works

To the best of our knowledge, the processor laboratory of our department [2] has instituted the first curriculum in which students design and implement their own processors using FPGAs. Though there are many ideas using FPGAs for computer architecture education, most curriculums fully or partially specify the architecture that the students learn [4][5][6]. Our curriculum, though, allows students themselves to decide the architecture they will implement. Because all the necessary software is also made by the students themselves, instruction sets and execution models are not restricted. Therefore, students can experience a wider range of design trade-offs than usually possible.

In an idea described by Gray [7], students learn architecture trade-offs through modifying a given processor to improve the performance. However, because the baseline architecture is specified in this situation, the range of trade-offs is limited.

The CITY-1 framework [8] is similar to our curriculum in that students design their own processor architecture. However, some reference implementations are presented to induce students' ones. On the other hand, our curriculum encourages students to construct their own architecture without any guiding model.

5 Concluding Remarks

In this paper, we presented our department's newly introduced processor laboratory curriculum. The laboratory is a part of students' computer architecture education; whereby in the processor laboratory they design and implement their own processors using FPGAs. Students also write the software necessary for the processors.

The main purpose of the curriculum change was to enable students to implement most of the architecture techniques they learned in lectures; thus the increased FPGA size. In the new curriculum, 1M-gates FPGAs are now used, compared to the 5K-gates FPGAs used

previously. The result is that students can now experience a wide range of trade-offs. In the laboratory, students really implemented modern techniques such as caches and FPUs which could not have been implemented under the previous curriculum. The large FPGA size is certainly useful for teaching architecture design trade-offs.

The main drawback in the new curriculum is that students cannot design the whole system because the system board has been pre-wired. In this respect, the previous curriculum was better; though to improve this situation, from this year, we will have students construct serial I/O circuits.

References

[1] J. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition.* 1995.

[2] T. Matsumoto and K. Hiraki, "Laboratory of Designing Original Processors using FPGAs," in *Proceedings of the 2nd Japanese FPGA/PLD Design Conference & Exhibit(in Japanese)*, pp. 289–302, June 1994.

[3] http://www.nvaca.com/cpu/.

[4] Murray Pearson, Dean Armstrong and Tony McGregor, "Using Custom Hardware and Simulation to Support Computer Systems Teaching," *Workshop on Computer Architecture Education*, 2002.

[5] Daniel Ellard, David Holland, Nicholas Murphy, Margo Seltzer, "On the Design of a New CPU Architecture for Pedagogical Purposes," *Workshop on Computer Architecture Education*, 2002.

[6] Ross Brennan and Michael Manzke, "On the Introduction of Reconfigurable Hardware into Computer Architecture Education," *Workshop on Computer Architecture Education*, 2003.

[7] J. Gray, "Hands-on Computer Architecture - Teaching Processor and Integrated Systems Design with FPGAs," *Workshop on Computer Architecture Education*, June 2000.

[8] Ryuichi Takahashi, Noriyoshi Yoshida, "Diagonal Examples for Design Space Exploration in an Educational Environment CITY-1," in *Proc. of IEEE International Conference on Microelectronic Systems Education*, pp. 71–73, July 1999.

Teaching Embedded Systems with FPGAs Throughout a Computer Science Course

Vanderlei Bonato[1] Ricardo Menotti[1]
Eduardo Simões[1]
[1]Universidade de São Paulo
ICMC-Dep. Computação
São Carlos-SP-Brazil
emarques@icmc.usp.br

Marcio M. Fernandes[2] Eduardo Marques[1]

[2]Universidade Metodista de Piracicaba
FCMNTI-Ciência da Computação
Piracicaba-SP-Brazil
mmfernan@unimep.br

Abstract

Although embedded systems have been around for quite a long time, just in recent years they have attracted major industry and academic interest. There is a perception that a computing paradigm shift is taking place, and so the need to provide computer science students with the required expertise in the field. In this paper we describe our experience of using a reconfigurable computing platform throughout a number of courses. By doing so we allow students to get acquired to embedded systems concepts and practices under different contexts in the normal curriculum. The application of this strategy have allowed considerable gains for students taking embedded system courses, research projects in the field, and also professional activities.

1 Introduction

In recent years embedded computing has emerged as the new paradigm for the design and implementation of modern computer systems. They consist in the fastest growing market share for computing products, already accounting for the largest number of systems being deployed [15]. In terms of total revenue, they should also overtake desktop-based systems in just a few years. Typical examples of embedded systems include digital cameras, mobile phones, automotive control devices, and medical equipment, among others.

As it happens during any technology shift period, *skills shortage* can be a problem as the current curriculum may not address the whole set of issues involved. The range of skills required for embedded systems design encompass knowledge about hardware devices, computer architecture, microprocessor, and high-level language programming, among others [17]. Although these topics are adequately taught in Computer Science and Engineering courses, students tend to see them as isolated units, with little relation to embedded systems. We believe that reconfigurable computing [6] can be used as a platform for teaching all of those subjects, and also to expose students to some of the main concepts and practices in that field.

In this paper we describe our experience on how teaching computer architecture and related courses using reconfigurable computing allows students for a better understanding of key concepts involved in embedded systems design and implementation. The remaining sections discuss related technologies and concepts (2), the platform and tools employed by our courses (3), and how key concepts related to embedded systems are inferred from those disciplines (4). That section also describes how this strategy paves the way for research projects based on embedded systems. Finally, the last section (5) brings the conclusions of our experience and future directions.

2 Embedded Systems and Reconfigurable Computing

An *embedded system* can be described in general terms as an application specific system implemented using a programmable processor, usually integrated with other hardware devices such as sensors and actuators [4]. As opposed to desktop-based systems, embedded systems are designed to deliver the expected functionality and performance for one (or just a few) task, usually being part of a larger system. Key characteristics often required in such systems include real-time constraints, low-power consumption, and low-production cost. An important design decision is the so called hardware-software partitioning, defining the tasks that will be executed by an ASIC (Application Specific Integrated Circuit), or a software programmable microprocessor.

The latter approach can be done using off-the-shelf devices, typically a microcontroller, or developing a SoC (System-on-Chip), with reusable IP (Intellectual Property) components. It should be noticed that the semiconductor industry is shifitting towards SoCs, as pointed out by the latest version of the International Technology Roadmap for Semiconductors [9].

An alternative to those approaches is to use *reconfigurable computing* [6], a technology based on reprogrammable integrated circuits, nowadays commonly known as FPGAs (Field Programmable Gate Arrays). FPGA based systems allow performance levels comparable to those based on ASICs, but with advantages of on-site reprogrammabilty, and a shorter development cycle. Those are key attributes to deal with changing requirements, and time-to-market pressures, respectively.

In the addition, there are some evidences that the transistor count for FPGAs may be experiencing a higher growth rate than microprocessors, being comparable to the one observed for memory logic. As a result, at some point it may be possible to build FPGA based systems that are more complex than microprocessors. An evidence of this trend has been recently produced by Xilinx, with the announcement that in the near future it will make available devices topping the one billion transistor mark [19]. It should also be noticed, however, that FPGAs are still considerably slower and consume more power than ASICs, which still prevent their use in some scenarios.

3 Teaching Platform

.

As already said, key concepts and practices for embedded systems design are often taught in a number of courses with little relation, at least from the student's perspective. For this reason, we have adopted an *FPGA based platform* as a core component for those disciplines, in a attempt to encourage a "think embedded" attitude among students. The use of an FPGA platform to teach computer architecture and related disciplines has been adopted by many courses, sometimes employing supporting tools specially designed for that (e.g. [16]). However, to the best of our knowledge, it has not been adopted with the specific goal of enabling students to practice embedded systems concepts *within* other courses.

The choice for a reconfigurable computing platform, as opposed to other alternatives, was mainly due to the following reasons:

- It does not constrain the student to a particular microprocessor architecture or simulation tool;

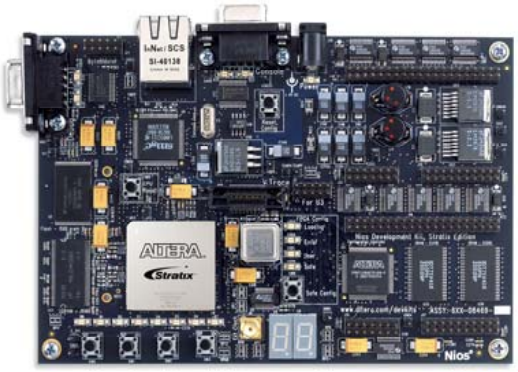


Figure 1: Kit Nios Stratix Edition

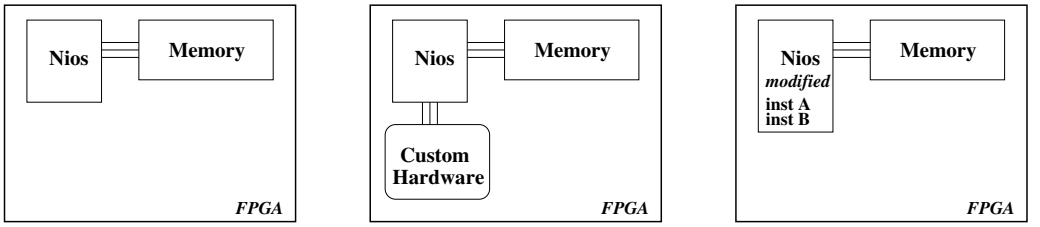
- It accommodates a wide range of complexity levels, from simple logic blocks to a complete SoC implementation;

- It allows students to get acquired with EDA (Electronic Design Automation) and high level programming language tools.

The adopted teaching platform is based on Altera FPGA development boards, in particular the **UP2** and **Nios Stratix Edition**. The UP2 [3] is a low cost board composed of reconfigurable hardware, with a capacity of 70K gates. It also includes I/O devices and output monitoring capabilities.

The Nios Stratix Edition [1] is more sophisticated, consisting of a FPGA with 1M gates, RAM and Flash memory modules, and support for Ethernet and RS232 communication (Figure 1). The board also comes with Nios, a 32-bit RISC *softcore* processor, which can be easily synthesized into the FPGA. This processor has has a five-stage pipeline, with independent buses for data and instructions, respectively. It also allows for the implementation of *custom instructions*, a desirable feature to improve the performance of time critical sections of code. It is also possible to design logic to gain access to external resources such as memory and I/O devices. All these features have proven to be valuable in the teaching process.

The **Quartus-II** EDA tool is used during all phases of an FPGA based project, either with the UP2 or the Stratix board. In these phases are included design, compilation, timing analysis, simulation, and chip configuration. Projects are organized as modules, which facilitates reusing them. Modules can be defined by a schematic design, or using hardware definition languages. The languages currently supported are Verilog, VHDL, and AHDL. Programming the Nios RISC processor is carried out with an integrated tools chain, which includes the GnuPro C compiler, an assembler, and a debugger.



a) CPU+ Memory b) CPU + Custom HW + Memory c) CPU w/ custom instructions + Memory

Figure 2: Embedded system architecture scenarios for laboratory practices.

For the students, working on the design of systems including *both* hardware and software implementations have shown to be a key element to tackle embedded system concepts. The teaching projects they are exposed to are previously designed, implemented and tested, which allows for the generation of template files to be used by students. By doing so, they are prevented from spending time on repetitive tasks, common to most practices, and can concentrate on the important aspects of the work.

4 Main Courses and Activities

In this section we describe the main courses of the computer science curriculum that have adopted the reconfigurable computing platform presented in Section 3. As seen in Figure 2, during those courses students can work with embedded systems under three basic architecture scenarios: a) the simplest one, consisting of a CPU and memory modules, b) using custom hardware blocks to improve performance of key sections of the application, and c) modifying the softcore CPU to include custom instructions.

Considering that the basic contents of those courses are homogeneous and well know among academic staff, we concentrate only on those details that are relevant to teach embedded systems concepts and practices. Please also notice that, for the sake of generality, the actual name of those courses may differ from the ones listed in the next section. A particular feature of some of those courses is the availability of *on-line* exams, using the EDA tool they became acquired to. Students are given a set of requirements and asked to design a circuit to meet the specification. As an example, they are asked to design a circuit to generate the signals corresponding to a given vector signal. The solutions are then sent electronically to tutors, in order to be marked. A poll among approximately one hundred students have shown that over 80% of them thinks the methodology is better than traditional ones.

4.1 Digital Systems

This topic is taught in two courses, introducing students to basic elements of digital system abstractions, such as gates, flip-flops, building blocks, binary arithmetic, multiplexing circuits, and so on. The Quartus-II EDA tool presented in Section 3 is used to implement simple lab practices to gain insight on the actual structure, behaviour, and interaction of those components. More complex assignments include the design and implementation of an ULA (Logic-Arithmetic Unit), and memory units. All projects are simulated, debugged and written to an FPGA, making possible for the student to understand timing and synchronization issues.

As an example, we have an assignement consisting in the design of *bus-based* communication interface to be implemented as an FPGA SoC. In this project students are asked to create a finite state machine for the implementation of a communication protocol. This has an appropriate complextiy level for a *second course* on digital logic, allowing students to work on structures such as fifo queues, memory, registers, etc., in order to build the design shown in Figure 3. Issues related to communication networks such as bus contention, priorities, and transmission delay are also introduced, paving the way for more elaborated projects in computer network courses (Section 4.6).

4.2 Computer Organization

Computer Organization refers to the main units that compose a Von-Neumann machine, i.e., ALU and control unit, memory, register file, and buses [14]. The students can build a basic working system using Verilog or VHDL, starting from a simple instruction set architecture, and then expanding the ISA with other instructions. This is a particularly useful experience as the use of custom blocks (or instructions, when possible) is an effective way to execute performance sensitive tasks. Student designs are implemented and tested into the FPGA platform, which in practice gives them a feeling of a running embedded system. By doing so we allow them to practice some key concepts of embedded system design without the overhead of a new

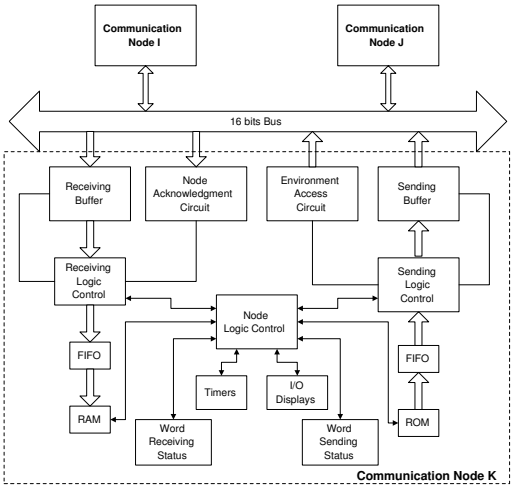


Figure 3: A bus-based communication interface.

introductory course on the subject.

4.3 Computer Architecture

The theory part of this course concentrates on the usual topics such as microprocessor and pipeline organization, memory hierarchy, interconnection to I/O devices, etc. During the lab activities, students are asked to expand the CPU design of the Computer Organization course, using pipelining techniques to enhance performance. This helps to give them a better understanding on the differences that can be found between CPUs, even when they implement the same instruction set. This knowledge can be used to better evaluate functionality, performance, and power consumption when choosing a CPU for a given embedded application [7].

In this course students have their first experience with the Nios softcore processor. Simple applications are implemented in C language, to be executed in the FPGA CPU implementation (Figure 2a). Then, key sections of code are implemented with custom logic, along with the additional code to switch processing and communicate data between the Nios CPU and the FPGA hardware blocks (Figure 2b). Comparing the performance results from both implementations is a good way for students to learn why hardware-software partitioning is such an important step of embedded systems design.

4.4 Compilers

The typical one-semester compiler course tend to spend more time on front-end concepts, as little time is left for back-end design and implementation. However, on a second course (usually at the graduate level) code generation and optimization techniques are the main focus, bringing compilers closer to computer architecture. We use the rich set of microprocessor architectures for domain-specific embedded applications to illustrate how code generation techniques can take advantage of that to improve performance dramatically [11]. We also use the FPGA platform for some practices aiming to create custom instructions for the Nios processor (Figure 2c), and then modify the GnuPro compiler to target those new instructions. The experience also helps students to have real evidence of a real-world aspect: that useful architecture feature may be difficult to unlock for the application code. Evaluating the quality of the *tools chain* can be as important as doing so for the architecture itself.

4.5 Operating Systems

Introductory operating systems courses tend to be general, not biased towards domain specific concepts. However, we do try to relate aspects such as interrupts, concurrency, scheduling, I/O and the device drivers to the lab platform students are getting used to. Some simple practices are designed to show the effects of not having an embedded operating systems running on the background, which is usually taken for granted on the desktop environment they are more familiar with. This experience may help students to understand what to look in the multitude of embedded operating systems available [10]. Those practices also help to introduce a new concept to them: code size, which can grow very quickly with the addition of their custom "operating system". We are currently working on a Nios port for *eCos* (embedded Configurable operating system), an open-source, configurable O.S. for embedded systems [8]. Once it is done we will be able to work with more elaborated practices, and also research projects.

4.6 Computer Networks

Computer network courses comes in all shapes and flavours depending on the course orientation. Some of them concentrates on high level abstractions, being the Internet an ubiquitous example. Others concentrate on fundamental aspects, such as the OSI model or wireless protocols. Laboratory practices varies according to the chosen emphasis, and whenever possible we offer a choice of practices on an embedded system scenario. One example is a project aiming to implement a network connection between two FPGA based SoCs, letting for the student the use and customization of the required protocols. That can be made on top of existing implementations, such as an Ethernet core described in VHDL. A related research project under development refers to the integration of ethMac [13], an Ethernet

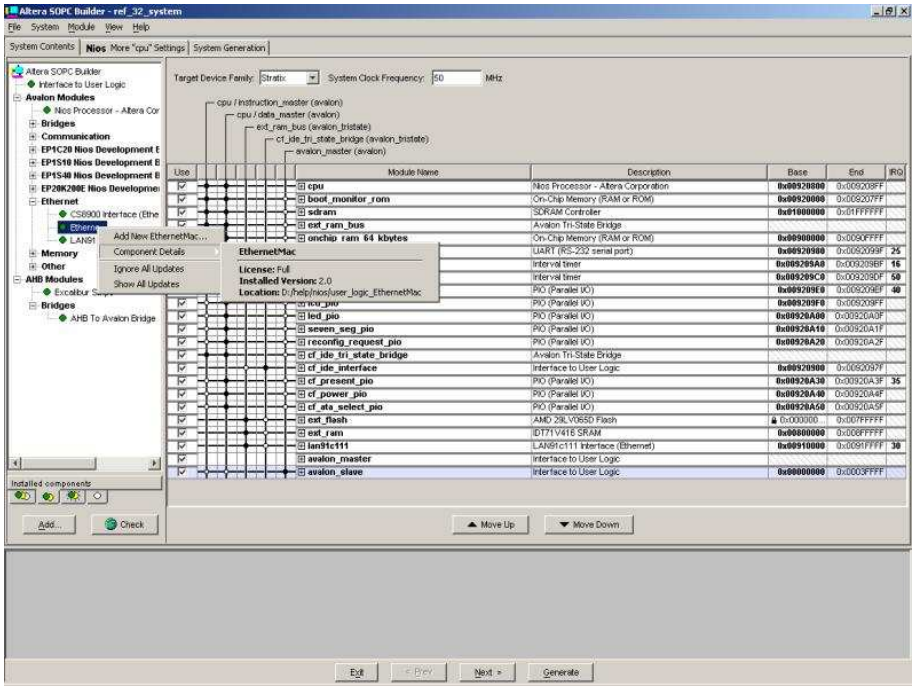


Figure 4: SoPC Builder: A tool for SoC design and implementation.

MAC (Media Access Control) core designed for implementation of CSMA/CD LAN in accordance with the IEEE 802.3 standards. The ethMac Verilog code can be integrated with a Nios CPU and other softcore devices, to create a complete SoC with the required functionality.

4.7 Embedded Systems Design

We offer specefic courses on embedded systems design at both, undergraduate and graduate level. Students enrolling in the introductory undergraduate course clearly benefit from the previous experience of developing several small projects on a embedded platform. By doing so we can concentrate on the real issues of embedded systems design such as CPU architecture, and coding more sophisticated applications using high level languages [4]. It should be noticed that some analysts estimate that the *software* of embedded systems account for 80% of the *total cost* of development [9]. We give special attention to the later as there is a clear shift in embedded system design from low-level assembly implementations to the use of C or C++ language. That is not only due to productivity reasons, but also due to the emergence of complex architectures, such as VLIW [18], that can only be fully exploited by using optimizing compilers specifically targeted to them [11].

The introductory course uses the FPGA platform described in Section 3, and also DSP microcontrollers, such as the Motorola DSP56800 family. The graduate courses concentrate on SoC design, using the Altera SoPC Builder (Figure 4), a tool specially aimed at the design and implementation of SoCs on programmable chips [2]. Projects are defined according to the application areas of our research programs, which includes robotics, computer architecture, control and automation, among others.

5 Interaction with Research

As already stated, the increasing capabilities of the hardware and software currently employed for embedded systems design also results in a growing interest from academic research initiatives. That can be the case in either basic research, or applications. In our department we have projects in both areas. As an example of research on basic aspects of embedded systems, we have a project called Architect-R. Its aim is to build a tool for automatic generation of hardware and software components to implement systems for robotics [12]. Research on specific applications include implementations of multimodal interfaces, such as voice and gestures recognition systems. These can be used in a number of domains such as robotics, virtual reality, etc. As an example, the system shown in Figure 5 consists of a CMOS camera connected to an FPGA, which implements a RAM-based neural network to recognize hand gestures [5]. This embedded

system shown to be robust, is able to meet real-time constraints (processing rate of 30 frames per second), and has a high efficiency in the recognition process. In addition, it also has on-chip training capabilities, a desirable functionality enabled by the reconfigurable hardware.

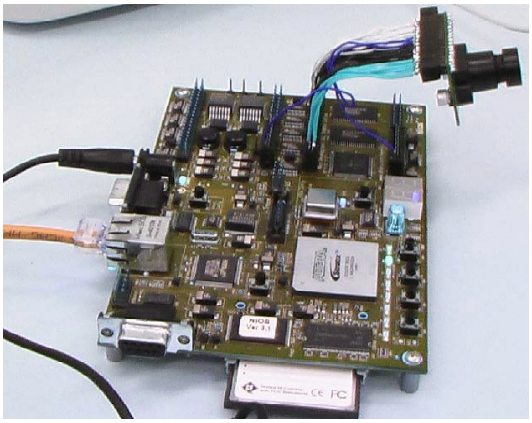


Figure 5: An embedded gesture recognition system

We have been following a number embedded systems research projects (Master's level) in our institution. Some analysis allow us to conclude that students that have been through those courses described in Section 4 are considerably more comfortable to work in this area than those that have not (typically coming from other institutions). Obviously a successful research project depend on other factors as well, but the learning curve to tackle basic concepts and practices of embedded systems is clearly reduced by using the approach described in this paper. We believe that graduates going to industry are also better equipped to start solving problems in the field. That has been confirmed to us when receiving some informal feedback from former students.

6 Conclusions

We have described our experience of teaching embedded systems to undergraduate and graduate students using a reconfigurable computing platform (FPGAs). Our goal was to devise a strategy to shift (or at least balance) the emphasis from desktop based to embedded computing, but without overloading the current curriculum. After some years of developing and applying the methodology, we understand that satisfactory results have been achieved. That is based on the increased interest from students to follow research projects related to embedded systems, the improved performance of graduates in this domain, and also some feedback from former students working in the industry. We are still working to improve practice assignments, and also the coordination with the theoretical contents of those courses, trying to accomodate teaching priorities.

References

[1] Altera Corp. Nios Development Kit, Stratix Edition, May 2004. http://www.altera.com/products/devkits/altera/kit-nios_1S10.html.

[2] Altera Corp. SOPC Builder, May 2004. http://www.altera.com/products/software/system-/products/sopc/.

[3] Altera Corp. UP2 Design Laboratory Kit, May 2004. http://www.altera.com/education/univ/kits/unv-kits.html.

[4] A. Berger. *Embedded Systems Design: An Introduction to Process, Tools, and Techniques*. CPM Books, USA, 2002.

[5] V. Bonato, E. Simes, M. M. Fernandes, and E. Marques. A gesture recognition system for mobile robots. In *Proceedings of ICINCO-1st International Conference on Informatics Automation, Control, and Robotics*, Lisbon, Portugal, 2004 (to appear).

[6] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[7] S. Cotofana, S. Wong, and S. Vassiliadis. Embedded processors: Characteristics and trends. In *Proceedings of the 2001 ASCI Conference*, Netherlands, 2001.

[8] eCos. eCos Home Page, May 2004. http://eCos.sourceware.org/.

[9] D. Edenfeld, A. B. Kahng, M. Rodgers, and Y. Zorian. 2003 technology roadmap for semiconductors. *Computer*, 37(1):47–56, Jan. 2004.

[10] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. H. Jr. A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro*, 21(3):54–68, May/June 2001.

[11] J. Glossner, J. Moreno, M. Moudgill, J. Derby, E. Hokenek, D. Meltzer, U. Shvadron, and M. Ware. Trends in compilable dsp architecture. In *Proceedings of The 2000 IEEE Workshop on Signal Processing Systems*, USA, 2000.

[12] R. Gonçalves, J. Cardoso, M. Fernandes, E. Marques, et al. Architect-R: A system for reconfigurable robots design. In *Proceedings of SAC-2003 -The 18th Annual ACM Symposium on Applied Computing*, USA, 2003.

[13] Opencores.org. Ethernet mac 10/100 mbps:overview, May 2004. http://www.opencores.org/projects.cgi/web/-ethmac/overview/.

[14] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., USA, 1997.

[15] B. R. Rau and M. Schlansker. Embedded computing: New directions in architecture and automation. Technical Report HPL-2000-115, Hewlett Packard Laboratories, Sept. 29 2000.

[16] C. Teuscher, J. O. Haenni, F. J. Gomez, H. F. Restrepo, and E. Sanchez. A tool for teaching and research on computer architecture and reconfigurable systems. In *Proceedings of the 25th Euromicro Conference*, volume 1, pages 343–350, Milan, Italy, September 8–10 1999. IEEE Computer Society, Los Alamitos, CA.

[17] W. Wolf and J. Madsen. Embedded systems education for the future. *Proceedings of the IEEE*, 88(1):23–30, Jan. 2000.

[18] K. Wong and N. Topham. OneDSP: A unifying DSP architecture for systems-on-a-chip. In *Proceedings of ICASSP-2002 - International Conference on Acoustics Speech and Signal Processing*, USA, 2002.

[19] Xilinx, Inc. Press release no. 03131, May 2004. http://www.xilinx.com/prs_rls/silicon_vir/-03131_nextgen.htm.

Extending FPGA based Teaching Boards into the area of Distributed Memory Multiprocessors

Michael Manzke and Ross Brennan
Trinity College Dublin
Ireland
michael.manzke@cs.tcd.ie,ross.brennan@cs.tcd.ie

Abstract

Reconfigurable hardware, in conjunction with soft-CPUs, *has increasingly established itself in computer architecture education. In this paper we expand this approach into the area of distributed memory multiprocessor systems.*

Arguments that supported the introduction of reconfigurable hardware as a substitute for commodity CPUs on educational computer architecture boards are equally applicable to teaching hardware that facilitates the construction and configuration of multiprocessor systems.

The IEEE Standard for the Scalable Coherent Interface (SCI) *was chosen as the interconnect technology because it enables the demonstration of the most important architecture concepts in this context. This interconnect exhibits high bandwidth and low latencies and not only specifies a hardware* Distributed Shared Memory (DSM) *architecture, but also defines cache coherence protocols. Consequently an implementation of this standard allows the design of* Non-Uniform Memory Access (NUMA) *and* cache-coherent NUMA (ccNUMA) *multiprocessor systems.*

1 Introduction

The initial design objective for the next generation of computer architecture teaching boards was driven by the desire to design a system that would provide *reconfigurable hardware* resources for a range of *soft-CPUs*. So far the following three *soft-CPUs* are available for the system:

- Open-source SPARC LEON P-1754 [1]
- Op-code compatible MC68008 [2]
- Teaching Instruction Set Processor [3]

Current final year projects will provide a *Java Virtual Machine* and an *Advanced Teaching Instruction Set Processor*. The latter processor will include *Instruction Level Parallelism (ILP)* concepts.

The design of these boards promises a number of advantages over the use of commodity CPUs. Most importantly it enables undergraduate students to test their synthesisable hardware description language (HDL) models on real hardware. The current generation of teaching boards is based on the Motorola 68008 processors. Second year undergraduate student are asked to construct a complete microprocessor system by wire-wrapping integrated circuits (ICs) such as memory and input-output (IO) devices. The next generation of hardware will also allow the students to experiment with a microcoded multi-cycle instruction set processor that they design in VHDL in the second semester of the course. This is one example that demonstrates how the introduction of *reconfigurable hardware* has significantly widened the scope of the boards to now cover several subjects throughout the Computer Science and Computer Engineering undergraduate degree courses at TCD, ranging from the simple design of an instruction set processor to the design of high performance floating point pipelines.

Our successful prototype, in conjunction with three *soft-CPUs* that were either adapted to the board's architecture or newly developed, has motivated us to take the design one step further by integrating hardware that allows for the construction of closely coupled multiprocessor systems through the interconnection of these boards. We chose the *Scalable Coherent Interface (SCI)* as the most suitable interconnect technology [4]. The interconnect architecture is implemented through a hybrid design of commodity *LC3 Link Controllers* from Dolphin Interconnect Solutions Inc.[1] and Field Programmable Gate Arrays (FPGA) [5]. The *LC3 Link Controller* implements link-level aspects of the SCI standard whereby higher level parts of the SCI specification,

[1] This work is supported by Dolphin Interconnect Solutions Inc.

such as the cache-coherence protocol, are implemented through *reconfigurable hardware* to guarantee access to these functions.

This system architecture enables the configuration of *Non-Uniform Memory Access (NUMA)* and *cache-coherent NUMA (ccNUMA)* multiprocessor systems. Furthermore, SCI switches may be used to demonstrate a direct interconnection network. As an alternative, indirect networks can be constructed through multidimensional meshes. Implicit communication is provided by the shared address space of the *NUMA* or *ccNUMA* architecture and explicit communication may be implemented on top of shared address space architectures.

2 The Interconnect

The *Scalable Coherent Interface (SCI)* as defined in the IEEE standard 1596-1992 provides bus-like services by replacing a system-bus with 16 bit parallel point-to-point unidirectional links. This approach overcomes electrical limitations of system buses that are characteristic for *Symmetric Multiprocessor (SMP)* systems. Multiprocessor systems can be constructed with this technology that scale up to several thousand nodes.

2.1 SCI in Commodity Systems

Defined in 1992, SCI is a well established technology and many high performance cluster implementations employ this interconnect (e.g, PC2 University of Paderborn Germany, University Of Delaware - The Bartol Research Institute USA and National Supercomputer Centre in Sweden) [6]. Subsets of the SCI standards have been implemented and are available as commodity components. In particular, Dolphin [7] have implemented PCI cards that bridge PCI bus transactions to SCI transactions. Compute nodes with PCI slots may be interconnected through PCI-SCI bridges together with a suitable SCI fabric topology, thus bridging their PCI buses. References made by one of these nodes into its own PCI address space are translated into a SCI transaction and transported to the correct remote node. The remote node translates this transaction into a memory access, thus providing a hardware DSM implementation. Programmed IO (PIO) and Direct Memory Access (DMA) may be performed without the need for system calls.

In a commodity SCI card a PCI-SCI bridge translates between PCI transactions and SCI transactions and forwards them onto the PCI bus or the BLink bus. The SCI BLink bus interconnects the PCI-SCI bridge with up to seven SCI Link Controllers (LC) or alternative components. SCI cards with two SCI Link Controllers attached to the BLink are consequently suitable for the construction of a 2-dimensional torus. Systems with more than one LC can route packets over the BLink to the correct LC according to a routing table. This enables distributed routing of SCI packets between individual SCI rings without an expensive central SCI switch. Routing is configured during SCI fabric initialisation. Every LC has an input and output port and the output port of one LC component is connected via a cable to the input port of another LC component. These links are 16 bit parallel and unidirectional with a bandwidth of 667Mbytes/s.

2.2 SCI and Cache Coherence

The connection of SCI link controllers to the IO bus via a bridge was not intended during initial specification of the SCI standard but it allows commodity component manufacturers to offer SCI subsystems that may be attached to a diverse set of computer architectures as long as they provide a standard IO bus. This therefore enables the construction of Non-Uniform Memory Access (NUMA) machines with commodity PCs. This approach prohibits the implementation of cache coherence as defined in the SCI IEEE 1596-1992 standard [4].

2.3 SCI and Real Time Constrains

One possible application of the teaching board is in the context of embedded distributed control systems that must meet real-time constrains. The SCI technology with its low latencies and deterministic behaviour would assist such architecture to meet these constraints. SCI is already being used in mission critical real-time applications. For example, Thales Airborne System employs SCI for backplane communication in their EMTI unit (Data Processing Modular Equipment). This scalable unit is integrated into the Mirage F1, 2000 and Rafale combat aircraft, NH-90 helicopters, Leclerc tanks, submarines, the Charles de Gaulle aircraft carrier and strategic missiles [8]. The application of SCI technology in airborne systems provides ample evidence concerning its suitability for real time applications. Some research on the SCI fabric's suitability for real time application was conducted at Trinity College Dublin [9, 10].

3 The Teaching Board

The design of the teaching board avoids the PCI-bus as the interface to the SCI interconnect. Two commodity *LC3 Link Controllers* from Dolphin Interconnect Solutions Inc. [5] are directly connected to a *Memory Bridge Field Programmable Gate Array (FPGA)*. This interconnection is implemented through a *64 bit Backside Link (B-Link)* bus [11] . A *soft-CPU FPGA* is also connected to the *Memory Bridge FPGA* via the CPU's system-bus. The *Memory Bridge FPGA* implements the *SCI upper level protocol management* and bridges bus transactions between the *soft-CPU's* system-bus and the

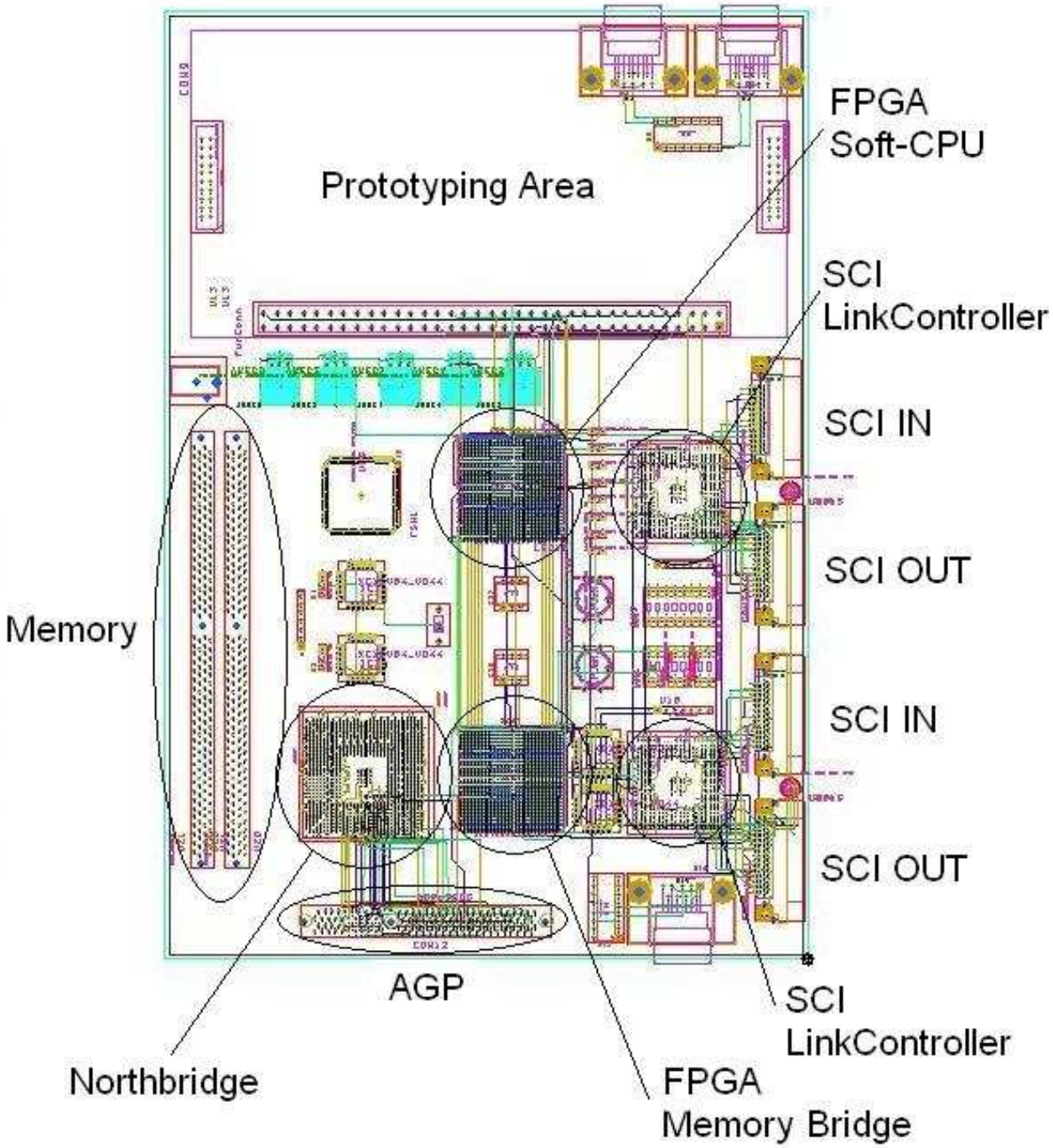


Figure 1: Printed Circuit Board

SCI *B-Link*. Figure 1 shows the layout of the *Printed Circuit Board* and annotates the main components of the system. The *Memory Bridge FPGA* is also connected to a *Northbridge* to enable access to the local memory of the system. The function of the *Memory Bridge FPGA* is to route the *soft-CPU's* memory references on its system-bus to either the local memory, via the *Northbridge*, or to remote memory by translating the memory reference into a SCI transaction. These SCI transactions are routed via the SCI interconnect fabric to the correct remote node. The SCI transactions will then be converted to a local memory reference by the remote *Memory Bridge FPGA*, therefore implementing a hardware *Non-Uniform Memory Access (NUMA)* multiprocessor systems. This approach does not require *Operating System Calls* to execute load and store transactions on remote memory but it does require software intervention to implement cache coherence.

The SCI IEEE 1596-1992 standard [4] defines *Cache-coherence Protocols* as an optional implementation. This directory based protocol enables processors to cache data from remote memory locations while maintaining the coherence of the multiple copies. There are also commercial products that implement the SCI *Cache-coherence Protocols* in hardware. A good example of a *cache-coherent NUMA (ccNUMA)* machine that implements SCI including the *Cache-coherence Protocols* is the *NUMA-Q* [12]. Figure 2 assumes that the soft-CPU FPGA holds an Open-source SPARC LEON P-1754 [1] and the *Memory Bridge FPGA* implements in addition to the SCI upper level protocol management a SCI Cache Coherence Protocol. This figure highlights the components of the teaching board that may be used by the students to implemented their design solutions through VHDL. This area is labeled *Reconfigurable Hardware* or *Playground*. The remaining components on the board are commodity ICs that can only be configured by the students through the modification of registers.

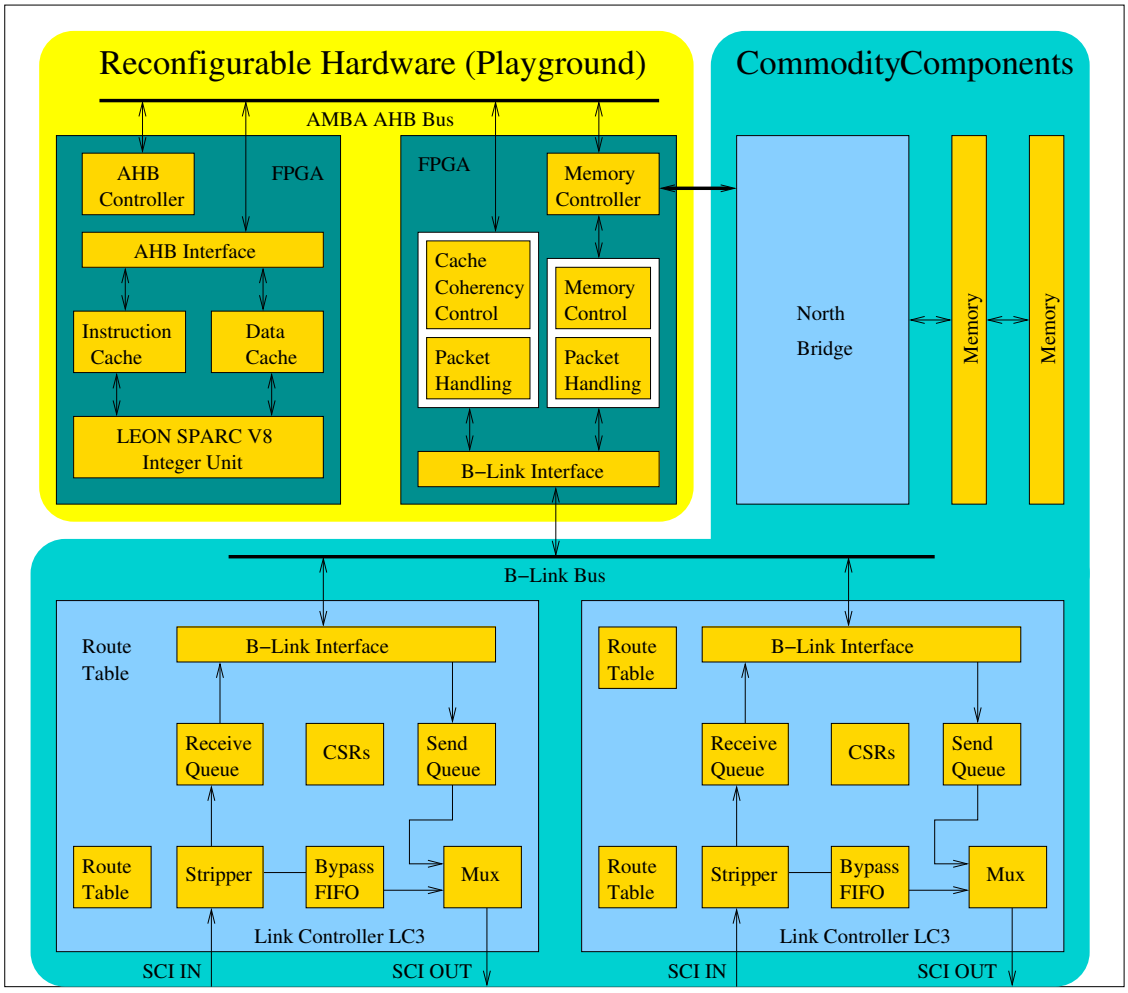


Figure 2: System contiguration that implements a LEON CPU with Cache Coherency Protocol

For example the two *Route Tables* in the *LC3 Link Controllers* can be manipulated by the students to implement the desired routing on the *64 bit Backside Link (B-Link)* bus [11] or the SCI fabric.

Figure 3 shows how individual boards could be interconnected as 2D torus to build a *cache-coherent NUMA (ccNUMA)* multiprocessor system. Again assuming that the soft-CPU FPGA holds a Open-source SPARC LEON P-1754 [1] and the FPGA Memory Bridge implements in addition to the SCI upper level protocol management a SCI *Cache-coherence Protocols*.

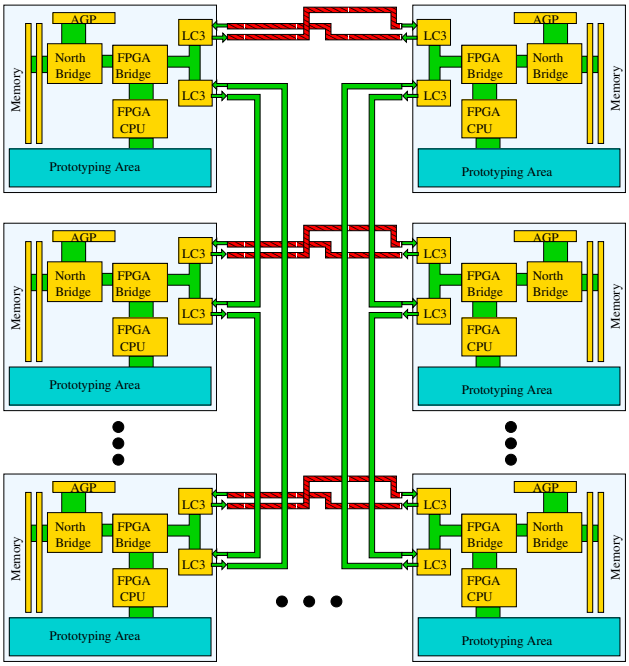


Figure 3: 2D Torus Multiprocessor

The *Memory Bridge FPGA* is responsible for acting as a bridge between the different commodity components on the board and so interfaces with both the *soft-CPU FPGA* and the *Northbridge*, an Intel Graphics and Memory Controller (GMCH) [13], in addition to the two SCI *LC3 Link Controllers*.

The board is designed to be compatible with any of the XC2V-FF896 range of FPGAs [14]. This gives the option to install different density FPGAs on the board. An XC2V2000 FPGA is required for the *Memory Bridge FPGA*. The *soft-CPU FPGA* can be either an XC2V1000/1500/2000 chip depending on board requirements. The XC2V2000 FPGA would be suitibable for a SMP solution. This option of components has direct consequences on the final cost of the boards as the FPGAs are the most expensive parts of the system and their pricing is determined by their memory density and speed grades.

3.1 Hard and Soft Hardware

The *Northbridge* is included in the design as it provides access to DDRRAM. It connects the *Memory Bridge FPGA* via a *Front-side Bus (FSB)* interface and also provides the possibility to attach a gigabit ethernet communications link to the system using the dedicated CSA bus on the *Northbridge*.

The bus connecting the *Memory Bridge FPGA* to the *soft-CPU FPGA* is a combination of the *AMBA Advanced Highspeed Bus (AHB)* and *Advanced Peripheral Bus (APB)* [15], allowing direct memory mapping between modules in both FPGAs. Finally, there is also a dedicated bus interface between the *soft-CPU FPGA* and the *Prototyping Area* on the board, which enables students to connect components (such as ROM and RAM) directly to the soft-CPU core in the FPGA, bypassing the *Memory Bridge FPGA*.

Although there are several soft-CPU core options available for the board, the LEON P-1754 SPARCv8 certified processor HDL model [16, 17] should be used if an open source operating systems with a supporting tool-chain is required. The CPU offers many features including a configurable cache, dedicated debugging link, floating point and memory management units, which may be individually configured or disabled. The main internal bus for the Leon2 core is the AMBA bus, which will be modified to communicate directly with the AMBA bus present in the *Memory Bridge FPGA*. When the board is configured with the full version of the Leon2 core, it will be able to communicate with all of the peripheral devices on the board. In order to reduce the complexity of the system, where necessary for teaching, a second version of the Leon2 core, with most of the internal components disabled, can be uploaded onto the board or alternatively one of the other soft-CPU cores, which have been modified for the board, can be used.

3.2 Operating Systems and Sofware

As the Leon2 core is fully SPARCv8 complient, there is a choice of several different operating systems for the board. RTEMS will be used where it is neccessary to have a hard-real-time operating system and Linux will be used as a more general purpose operating system. RTEMS is an open source real-time-operating-system (RTOS) designed for embedded systems [18]. It has multitasking capabilities and is POSIX 1003.1b complient. It also has support for TCP/IP and the GNU toolset chain (includ-

ing ISO/ANSI C and ISO/ANSI C++). It supports several different filesystems including FAT32/FAT16 and NFS and can connect to GDB (Gnu Debug) over ethernet or serial port.

Several different ports of Linux were made for the Leon2 core and these are now being merged into the main linux kernel tree [19]. As such, it is possible to configure and compile a linux kernel suitable for use on the board. It also has full support of the GNU tool-chain.

3.3 Educational Scope

The teaching boards provide students with three *Playgrounds* for their experimental work:

- Prototyping Area - For wire-wrapping IC components to interface logic in the *soft-CPU FPGA*.
- Reconfigurable Hardware - For soft-CPUs, Logic, Memory Management, Cache-coherence Protocols and much more.
- Interconnecting Boards with SCI - Different SCI fabric configurations in conjunction with logic in the Reconfigurable Hardware.

In an effort to increase the synergy of various hardware related subjects of the Computer Science and Computer Engineering syllabi we incorporated as many feature as necessary to enable students to incrementally build on previous experimental experience. It remains to be seen to what extend the academic staff in the departments will adapt the features provided by teaching boards in their courses. The first set of prototype PCBs will be manufactured and populated with integrated circuits in October 2004. The subsequent academic year will be used to debug the hardware and develop VHDL models for the reconfigurable hardware. The academic year 2005/2006 will introduce the board to undergraduate student in the second year Computer Architecture course.

The following list gives an example of the potential applications of the board in the Computer Science undergraduate degree:

- Introduction to Computing
- Digital Logic Design
- Systems Programming
- Computer Architecture I - Microprocessor Systems
- Computer Architecture I - Computer Architecture
- Computer Architecture II - Workstations
- Computer Engineering
- Systems Software - Operating Systems
- Compiler Design II

4 Conclusions

This paper argues for a single multi-purpose FPGA based lab-board which provides features suitable for the entire hardware related subjects of Computer Science and Computer Engineering undergraduate and postgraduate students. It was demonstrated that reconfigurable hardware not only allows the system to operate under a range of soft-CPUs, it also provides the means to let students experiment with their own synthesised HDL models. Furthermore the hardware implementation of the link level part of the *Scalable Coherent Interface (SCI)* standard through commodity components in conjunction with an FPGA implementation of higher level protocol management and cache coherency protocols enables students to build and experiment with *Non-Uniform Memory Access (NUMA)* and *cache-coherent NUMA (ccNUMA)* multiprocessor systems. The Leon soft-CPU allows the system to execute Linux and RTEMS, therefore providing a Unix like general purpose operating system and a hard-real-time operating system. All these three components Leon, Linux and RTEMS are open source which reduces the cost of the system significantly and more importantly gives students access to the hard and soft source code. Last but not least students may wire-wrap a microprocessor system on the prototyping area and operate the IC components with a soft-CPU of their choice by switching the soft-CPUs system bus interface from the Bridge-FPGA to the prototyping area. It should be emphasised that many of the functions provided by the board have been initially developed or evaluated through final year project. This is to point out that the complexity of the various teaching objectives is well suitable for Computer Science and Computer Engineering undergraduate students.

References

[1] R. Brennan and M. Manzke, "On the introduction of reconfigurable hardware into computer architecture education," in *Workshop on*

Computer Architecture Education WCAE 2003 (E. F.Gehringer, ed.), pp. 96–103, June 2003.

[2] D. Lynch, "A motorola 68008 opcode compatible vhdl cpu," April 2004. http://www.cs.tcd.ie/Michael.Manzke/fyp2003-2004/DavidLynch.pdf.

[3] L. Redmond, "Design of a teaching instruction set processor in vhdl," April 2004. http://www.cs.tcd.ie/Michael.Manzke/fyp2003-2004/LauraRedmond.pdf.

[4] P. M. Kelty, *IEEE Standard for Scalable Coherent Interface*. IEEE, ieee std 1596-1992 ed., March 1992.

[5] Dolphin Interconnect Solutions Inc., *LC3- SCI Link Controller for System Area Networks*. http://www.dolphinics.com/products/hardware/lc3.html.

[6] "Clusters @ top500," May 2004. http://clusters.top500.org/.

[7] "Dolphin interconnect solutions inc.," May 2004. http://www.dolphinics.com.

[8] "Dolphin interconnect solutions inc.," June 2000. http://www.dolphinics.com/news/2000/june020-2000.html.

[9] M. Manzke and B. Coghlan, "Non-intrusive deep tracing of sci interconnect traffic," in *SCI-Europe* (W. Karl and G. Horn, eds.), pp. 53–58, September 1999.

[10] B. C. O. L. Michael Manzke, Stuard Kenny, "Tuning and verification of simulation models for high speed interconnection," in *PDPTA* (H. Arabnia, ed.), pp. 1087–1093, June 2001.

[11] Dolphin Interconnect Solutions Inc., *A backside link (B-Link) for scalable coherent interface (SCI nodes)*, May 1996.

[12] H. Hellwagner, *SCI: Scalable Coherent Interface*, vol. 1734 of *Lecture Notes in Computer Science*, ch. 1. The SCI Standard and Applications of SCI, pp. 26–30. Springer, 1999.

[13] Intel, *Intel 865G/865GV/865PE/865P Chipset*. Intel, March 2004. http://developer.intel.com/design/chipsets/.

[14] Xilinx, *Introduction to the VirtexII Product Family*. Xilinx Inc, December 2001. http://www.xilinx.com.

[15] ARM, *AMBA Specification V2.0*. ARM Limited, May 1999. http://www.arm.com.

[16] J. Gaisler, *Leon2-1.0.10 Users Guide*. Gaisler Research, December 2002. http://www.gaisler.com.

[17] SPARC, *SPARC V8 Manual*. SPARC International Inc, January 1992. http://www.sparc.org.

[18] "Rtems is the real-time operating system for multiprocessor systems," May 2004. http://www.rtems.com/.

[19] "Linux for leon2 processor," May 2004. http://www.gaisler.com/linux.html.

Teaching Computer Architecture Using an Architecture Description Language

Sandro Rigo, Marcio Juliato, Rodolfo Azevedo, Guido Araújo, Paulo Centoducatte

Computer Systems Laboratory - Institute of Computing, University of Campinas
Cidade Universitária Zeferino Vaz, Po. Box 6176, Campinas-SP, Brazil
{srigo, marcio.juliato, rodolfo, guido, ducatte}@ic.unicamp.br

Abstract

This paper presents the use of the ArchC Architecture Description Language (ADL) as a support tool for computer architecture courses. ArchC enables students to perform several experiments using its automatically generated SystemC simulators, covering topics from simple single-cycle (functional) models to pipeline and memory hierarchy simulation. We show how instructive may be the process of modeling a processor using an ADL and suggest several possible exercises, following the course development structure presented in the classical Hennessy and Patterson's computer architecture didactical book. Moreover, we report how the experience of assigning students to study and to model modern embedded architectures has provided good results on an undergraduate computer architecture course at IC-UNICAMP. The simplicity and flexibility of the ADL, along with its simulation features, proved to be an useful tool not only for research, but also for computer architecture education.

1 Introduction

Architecture description languages (ADL) have been introduced to help designers face the development challenges that have arisen in the past few years, due to the increasing complexity of modern architectures. These difficulties have forced hardware architects and software engineers to reconsider how designs are specified, partitioned and verified. As a consequence, designers are starting to move from hardware description languages (VHDL, Verilog) and also beyond the RTL level of abstraction toward the so called *system level design*, where a tool for evaluating a new designed instruction set architecture, which automatically generates a software toolkit composed of assemblers, simulators, etc is mandatory. Such tools are commonly based on an architecture description language.

Besides their application and well known suitability for designing and experimenting with new architectures in the industry, architecture description languages can be very useful for academic purposes, like teaching/researching computer architecture at undergraduate and graduate level. On one hand, at the undergraduate level, models of well known architectures are appropriate to learn how a pipelined architecture works, including interlocking, hazard detection and register forwarding. If allowed by the ADL, this model can be plugged to different memory hierarchies in order to illustrate how the performance of a given application can vary, depending on the choice made for cache size, policy, associativity, etc. On the other hand, at the graduate level, researchers can use ADLs to model modern architectures and experiment with their ISA and structure with all the flexibility demanded in research projects. This paper is focused on the application of an ADL in a computer architecture course.

A common structure of an introductory computer architecture course is presented in the classical computer architecture book by Hennessy and Patterson [5]. The course starts with the instruction set architecture (ISA), i.e., presenting different instruction formats and how the processor manage to decode each instruction during execution. Some knowledge of assembly language programming is exercised at this moment. After understanding how an ISA is built, the student is ready to learn how instructions are really executed, how the data and computation flow inside the processor. The truth is: it is difficult to students to realize how all these features are implemented, and how they really work together inside a micro-processor without a tool to experiment with. This is the point were a software toolkit based on an ADL becomes very useful. Students can grab the knowledge about ISAs and pipelines from books and classes, and then fix it through the implementation of a processor model using an ADL, and get a simulator to experiment with and really *see* the whole thing running.

ArchC [6, 9] is an open-source ADL that fits very well in this context. It is being used as a support tool for computer architecture courses in the Institute of Computing, at University of Campinas, Brazil. Since the language, documentation, its parser, and simulator generator tools are all in public domain on the Internet, it is easy to students to get and to start using ArchC. This paper shows how to use ArchC MIPS models to illustrate computer architecture courses following exactly the structure presented in [5]. Moreover, we present how modeling modern computer architectures may be a good exercise for students.

The remaining of this paper uses the Hennessy and Patterson's book as a guide for presenting architecture concepts were ArchC may be a useful tool for illustration and experimentation. The text is organized as follows: Section 2 mentions some related work, Section 3 contains a brief introduction to the ArchC language, Section 4 shows how the ADL may be a useful tool on a computer architecture course, covering instruction set introduction, single-cycle, multi-cycle, pipelined, and memory hierarchy examples, Section 5 shows that an ADL enables teachers to introduce modern architectures even in an introductory course. Finally, we present our conclusions in Section 6.

2 Related Work

Considering automatic generation of a software toolkit for architecture exploration, one can find several ADLs on the literature, like: nML [2], ISDL [3], EXPRESSION [4], and LISA [11]. But no work has been published reporting and/or exploring the didactical capabilities of these languages. In fact, they have a serious drawback considering their application on computer architecture education, since none of these languages has all its tools and models published on public domain. All ArchC tools mentioned in this work, along with several architecture models can be freely obtained from [6].

Architecture simulators like SPIM [7] or SimpleScalar [1] may be used for didactical purposes. SPIM is a MIPS assembly simulator, compatible with the R2000/R3000 processors. It reads and executes assembly language code, but is not capable of executing binary files. SimpleScalar offers a MIPS like ISA, called PISA, for didactical purposes, along with a GCC port to this target. MASE [10] is a graphical simulation environment built on top of SimpleScalar. RaVi [8] comprises a set of multimedia MIPS based-modules for dynamic visualization of hardware behavior. These approaches do not provide automatic retargetability of their simulators and do not offer the flexibility of describing architecture behaviors in several levels of abstraction, or the easiness to model new architectures as provided by C++ based ADLs.

3 The ArchC Architecture Description Language

ArchC is an architecture description language initially conceived for processor architecture description, aiming to facilitate and accelerate processor description, combined with enough expression power to model several classes of architectures (RISC, CISC, DSPs, etc). ArchC allows users to fast explore a new ISA by automatically generating software tools, like SystemC simulators. Nowadays, ArchC is capable of describing processors as well as a memory subsystem. Memory hierarchies can be declared, containing several levels of memories and caches. Caches can be configured to simulate different set associativities, write polices, replacement strategies, and line sizes.

A processor architecture description in ArchC is divided in two parts, making clear the necessity of both behavioral and structural information. The `Instruction Set Architecture (AC_ISA)` description is where the designer provides details about instruction formats, size and names combined with all information necessary to decoding and the behavior of each instruction. In the `Architecture resources (AC_ARCH)` description, he/she informs ArchC about storage devices, pipeline structure, memory hierarchies, etc. Based on these two descriptions, ArchC will generate a behavioral simulator written in SystemC for the architecture, that may be purely functional or cycle-accurate, depending on the abstraction level used for instruction behavior descriptions. One important characteristic is that instruction behaviors, which are the largest part of the code in a processor model in ArchC, are described in pure C++ code. There is no restrictions, so model designers are capable of declaring their own methods and variables. C/C++ are largely used and it becomes very easy to students to start using ArchC. Moreover, there are complete GCC ports for MIPS and SPARC, including libraries to generate binary elf files ready to be loaded on ArchC simulators. This enables users to experiment with their own programs using ArchC simulators, and to execute real-world applications, including system calls emulation, like JPEG and MPEG coders.

We are going to use several pieces of ArchC code to illustrate our examples in this text, explaining some characteristics of the language as necessary, but readers should refer to the *Archc Language Reference Manual* [9] for a complete description of the ArchC's syntax and tools.

4 ArchC as a Support Tool for Teaching Computer Architecture

This section describes how ArchC can be a useful tool for developing projects and exercises, on a computer architecture course based on the classical didactical book from Hennessy and Patterson: "Computer Organization & Design" [5]. As this reference will be frequently mentioned throughout this text, for the sake of simplicity, we are going to refer to this book as *COD* from this point on.

```
AC_ISA(mips){

  ac_format Type_R = "%op:6 %rs:5 %rt:5 %rd:5 0x00:5 %func:6";
  ac_format Type_I  = "%op:6 %rs:5 %rt:5 %imm:16:s";
  ac_format Type_J  = "%op:6 %addr:26";

  ac_instr<Type_R> add, addu, subu, multu, divu, sltu;
  ac_instr<Type_I> lw, sw, beq, bne;
  ac_instr<Type_I> addi, andi, ori, lui, slti;
  ac_instr<Type_J> j, jal;

  ISA_CTOR(mips){

    load.set_asm("lw %rt, %imm(%rs)");
    load.set_decoder(op=0x23);

    store.set_asm("sw %rt, %imm(%rs)");
    store.set_decoder(op=0x2B);

    add.set_asm("add %rd, %rs, %rt");
    add.set_decoder(op=0x00, func=0x20);

    addu.set_asm("addu %rd, %rs, %rt");
    addu.set_decoder(op=0x00, func=0x21);

                                ...
  };
};
```

Figure 1: MIPS ISA Description in ArchC

4.1 Instruction Types, Assembly Mnemonics and Decoding

The use of ArchC in a computer architecture course can start as early as in the third chapter of COD, where the authors introduce the instruction representation inside a computer: the Instruction Set Architecture (ISA).

First, the MIPS assembly language is introduced, followed by information on how to translate it to the MIPS machine language. In order to do this translation students must learn about MIPS instruction formats and binary encoding, and finally how machine code is decoded by the processor. This is exactly the information contained inside an `AC_ISA` description, as illustrated by Figure 1. Students can do an `AC_ISA` implementation using the knowledge they are gathering from the book on instructions, assembly syntax, formats, fields and decodification, and also do some experiments with the decoder generated by ArchC, issuing some instructions in binary format to see if they supplied enough decoding information for each instruction in the ISA.

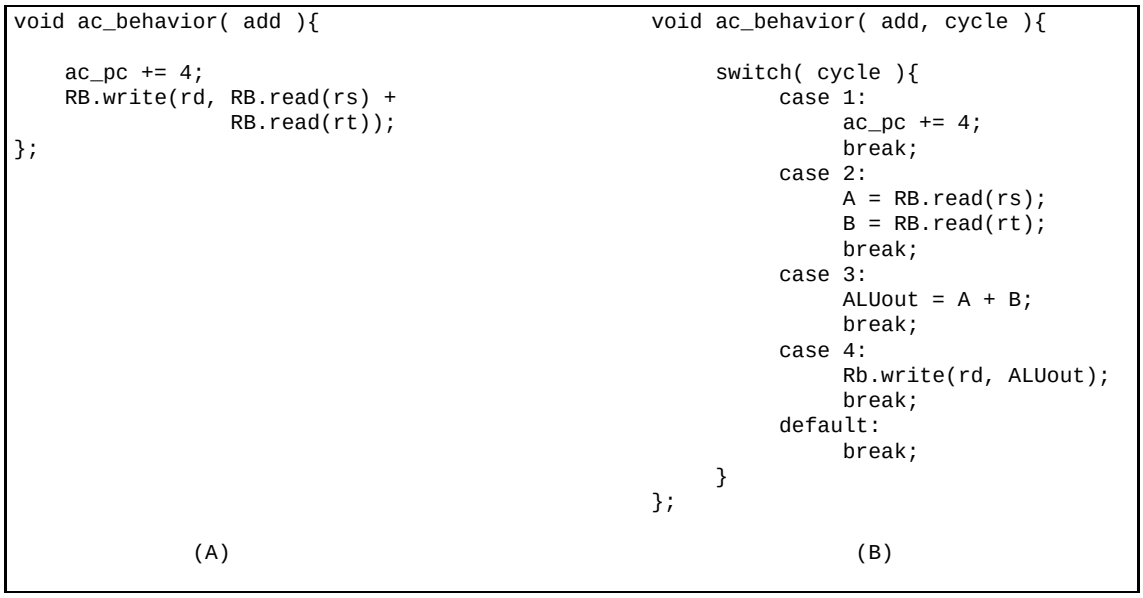
4.2 The Single-cycle and Multi-cycle Datapaths

By doing the simple exercises related in the previous section, the students can have their first contact with instruction set definitions and with the ArchC tools. This experience is important to the following tasks.

We call a functional model in ArchC a model that does not have any timing information, i.e., a model that executes one instruction per cycle. That is exactly the first example of a datapath construction presented in the book. Of course, the high abstraction level of ArchC models does not comprise functional units and signals, but the exercise of modeling the behavior of each instruction in C++ and trying to figure out which functional units and signals would be necessary to build a single-cycle datapath capable of executing such a behavior may be very instructive. COD authors suggest exercises like: write a functional simulator for the single-cycle and the multi-cycle versions of the datapath presented in the book using a hardware description language, like Verilog or VHDL. Authors predicted that students would take a week to develop each one of these simulators. Both of them can be easily coded in ArchC, for such a short and simple instruction set. Figure 2 (A) shows the functional version of the MIPS `add` instruction behavior, and Figure 2 (B) shows its multi-cycle version, according to the description given in COD pages 385–388. We estimate that a functional model of a fifteen or twenty-instruction of a MIPS-like ISA could be developed in three or four hours of work, after going through the theory presented in the book. Remember that instruction behaviors are written in C++, which is a language that most of the students are very familiar with. Another three or four hours of work would be enough to refine this functional model to a multi-cycle model, which is exactly the process of re-writing instruction behaviors to make them look like the example in Figure 2 (B).

4.3 The Multi-cycle Datapath with Pipeline

After the single and multi-cycle datapath concepts are sedimented, it is natural to introduce the concept of



```
void ac_behavior( add ){

    ac_pc += 4;
    RB.write(rd, RB.read(rs) +
                 RB.read(rt));
};
```

(A)

```
void ac_behavior( add, cycle ){

     switch( cycle ){
          case 1:
               ac_pc += 4;
               break;
          case 2:
               A = RB.read(rs);
               B = RB.read(rt);
               break;
          case 3:
               ALUout = A + B;
               break;
          case 4:
               Rb.write(rd, ALUout);
               break;
          default:
               break;
     }
};
```

(B)

Figure 2: Single-cycle and Multi-cycle Behavior Description

pipelining, where multiple instructions are overlapped in execution. The ArchC language contemplates this approach by supporting pipelining, in which we evolve from a functional model to a cycle accurate model, differing from the first in the timing precision. While functional models execute all instructions in one clock cycle, a cycle accurate model has instruction behavior descriptions reflecting the real number of clock cycles taken by the instructions. The great benefit brought by the use of an ADL like ArchC is that students can take advantage of his previous developed functional model to, gradually refine it to a new pipelined implementation, and get it running as a software simulator for the target architecture.

ArchC provides the necessary constructions for pipeline simulation but, in order to get a complete model of the MIPS architecture with a pipeline, students will have to consider its mechanisms, like register forwarding or data hazard detection, inside their behavior description. The first step is to insert information regarding the pipeline registers and pipeline stages into the functional model, as shown in Figure 3. It is important to emphasize that the same instructions continue to exist and the Instruction Set Architecture (ISA) remains unchanged for the pipelined model, i.e., no modification on the `AC_ISA` description file, showed in Figure 1, is required.

The pipelined model divides the instructions in minor execution units to be executed in the several pipeline stages, which are declared using the `ac_pipe` keyword. However, the stages communicate to each other through the pipeline registers, which have their particular structures, i.e, their fields. The students, while modeling the pipeline, have to declare the pipeline register structures, and thus, they have inevitably to have a complete understanding about how the pipeline works, what fields are necessary in each pipeline register, and what are their functions. Pipeline registers are declared by the combination of the `ac_format` and the `ac_reg` keywords. Let us take the ID/EX pipeline register of a MIPS processor, like described in COD, as an example. Such processor has a 5-stage pipeline (Fetch, Identification, Execution, Memory Access and Write-Back), and four pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB). The student should declare a format (field structure) for each register and give a name for it, as it is done for registers in Figure 3.

From this point on, all the necessary structural information is already inserted in the model. But before running this new model, it is necessary to take the second step, which is the refinement of the instruction behaviors. This is necessary because in a pipelined model the instructions are split into several parts, and each of these parts are executed in a different pipeline stage. It is important to notice that, in despite of the higher abstraction level of ArchC models if compared with the datapaths presented in the book, student must have consolidated the concepts of pipelining and its physical structure in order to be able to model it correctly.

ArchC automatically fetches the instruction pointed by its program counter (`ac_pc`), i.e., the student does not have to worry about the instruction fetch, but do need to take care of the PC increment. ArchC also generate a decoder for the architecture, based on the information provided in the `AC_ISA` description. But in a pipelined model there are other tasks that must be performed at the ID stage. Let us take the instruction `add` as an example. Still in the ID stage, the contents of the ID/EX register must be filled-up, from where the execution stage will access the correct values for the

```
AC_ARCH(mips){
  ...
  ac_format F_IF_ID  = "%npc:32";
  ac_format F_ID_EX  = "%npc:32 %data1:32 %data2:32 %imm:32:s rs:5 %rt:5 %rd:5
                        %regwrite:1 %memread:1 %memwrite:1";
  ac_format F_EX_MEM = "%alures:32 %wdata:32 %rdest:5 %regwrite:1 %memread:1 %memwrite:1";
  ac_format F_MEM_WB = "%wbdata:32 %rdest:5 %regwrite:1";

  ac_reg<F_IF_ID>  IF_ID;
  ac_reg<F_ID_EX>  ID_EX;
  ac_reg<F_EX_MEM> EX_MEM;
  ac_reg<F_MEM_WB> MEM_WB;

  ac_pipe   PIPE = {IF,ID,EX,MEM,WB};

  ARCH_CTOR(mips) {
    ...
```

Figure 3: Inserting Structural Information Regarding the Pipeline and its Registers into the MIPS I ArchC Description

operand registers and the program counter of the current instruction, along with some *signals* to control execution into further stages. Figure 4 shows an excerpt of a possible `add` behavior description. After that, the `add` instruction goes to the execution stage (EX), where the actual computation of the sum takes place, followed by setting the fields in the next pipeline register. In the case of the `add` instruction, the memory access stage (MEM) has just to copy the EX_MEM register contents to the MEM_WB register and, finally the instruction reaches the write back stage (WB), where the result of the sum is stored into the correct destination register.

```
  ...
case ID:
  ID_EX.regwrite = 1;
  ID_EX.memread = 0;
  ID_EX.memwrite = 0;
  ID_EX.npc = IF_ID.npc;
  ID_EX.data1 = RB.read(rs);
  ID_EX.data2 = RB.read(rt);
      ...
case EX:
  EX_MEM.alures= ID_EX.data1 + ID_EX.data2;
  EX_MEM.regwrite= ID_EX.regwrite;
  EX_MEM.rdest   = ID_EX.rd;
         ...
  break;
   ...
```

Figure 4. Modeling Instruction Behaviors Considering the Pipeline

The main point of this example is that, assuming that students are familiar with the basics of C/C++ programming languages, this operations are quite simple to be implemented, because they are nothing else than simple C++ statements. One very important thing when applying ArchC to computer architecture classes, is that the simplicity of the language brings the focus of the work to the architecture being described, and do not add an extra burden to the learning process due to syntax details of the language. Another advantage of using ArchC is its flexibility, since students are able to call their own functions inside behavior description methods, in order to debug the simulation. This facilitates the visualization of the pipeline internals, i.e., the student is free to watch whatever he wants by printing such data on screen, while running the simulator.

Notice that the code presented in Figure 4 does not consider the possibility of data hazards. With such a model in hands, a teacher could give some small examples of MIPS machine code where, for example, an instruction needs to use a register, but this register is being used by another instruction inside the pipeline, i.e, it is still not written in the register bank. Asking students to identify the problem and to add, for example, a register forwarding mechanism to their model can be a very useful exercise, which would help to solidify some important pipelining concepts. When modeling a processor with ArchC, a student can implement data forwarding in a way that is very similar to the described by didactical books. Lets take the *if statement* showed in COD, page 480, as a didactical example on how to insert data forwarding to our pipeline. In Figure 5 (A) we see how register forwarding is shown in the book, and in Figure 5 (B), how it is modeled in ArchC.

An ADL that generates C++ based simulators is also a well suitable tool for exercises like those suggested at the end of Chapter 6 in COD, mainly the last two exercises. The first asks students to collect statistics on data hazards for a C program and write a subroutine to model the five-stage pipeline presented in the book. Authors are asking for statistics like: number of instructions executed, number of data hazards, etc. This could be accomplished by implementing a

```
            BOOK:                                    ArchC MODEL:

if(EX/MEM.RegWrite and                          if((EX_MEM.regwrite == 1) &&
   (EX/MEM.RegisterRd != 0) and                    (EX_MEM.rdest != 0) &&
   (EX/MEM.RegisterRd = ID/EX.RegiterRs))          (EX_MEM.rdest != ID_EX.rs))
   ForwardA = 10;                                  operand1 = MEM_WB.wbdata;
               ...                                            ...

               (A)                                            (B)
```

Figure 5: Register Forwarding for the rs Register in Didactical Books and in an ArchC Implementation

pipelined model in ArchC. The last exercise asks for students to elaborate a model of the single-cycle datapath in a HDL like Verilog, and then refine it toward a pipelined model. A single-cycle or multi-cycle model implemented in ArchC, like those described in Section 4.2, can be refined toward the pipelined implementation required for this exercise. By using an ADL, this task certainly would be accomplished faster than by a HDL. A interesting approach is to divide the class in groups of students, and assign some of them to use an HDL like Verilog or VHDL, some to use SystemC, and some to write models using ArchC. At the end, students could share their experiences, pointing out the advantages and disadvantages of each approach. As suggested by the authors, this would be a project that would take students at least a month to be done. ArchC simulators are also capable of executing pipeline stalls and flushes so, there are several other possibilities of experiments that can be suggested to students, while teaching pipelining in a computer architecture course.

4.4 Memory Hierarchy

Continuing with our course based on COD, the next topic would be memory hierarchies. ArchC is capable of describing hierarchies composed of caches and memories distributed at different levels. Caches can be customized by the user by choosing parameters for associativity, number of lines, words per line, replacement strategy, and write policies. This is illustrated in the cache declarations contained in the example in Figure 6. The user creates the hierarchy by describing the connections among these devices, through the method `bindsTo`, as illustrated in the last two lines of the example.

So, by adding such a memory hierarchy description to our functional model, students can experiment different cache and hierarchy configurations. A possible exercise would be to choose a particular application, or a small set of applications, and let students experiment with cache parameters for a given hierarchy. They would be able to analyze the simulation results, to compare miss rates, and determine the best configuration for each application. ArchC simulators automatically keep track of all access to storage devices, so reporting miss rates and total number of accesses to each device declared in the AC_ARCH description.

```
AC_ARCH(mips){

  ac_cache   icache("dm", 128, "wt", "war");
  ac_cache   dcache("2w", 64, 4, "lru", "wt", "war");
  ac_mem   MEM:256K;

  ac_regbank RB:34;
  ac_wordsize 32;

  ARCH_CTOR(mips){
    ac_isa("mips_isa.ac");

    icache.bindsTo( MEM );  //Memory hierarchy
    dcache.bindsTo( MEM );  //construction
  };
};
```

Figure 6. Memory Hierarchy Declaration in ArchC.

After going through all the theory in COD's 7th chapter, students have a bunch of exercises to work on, in order to fix the concepts presented in the book. In addition to the theoretical exercises, students may use ArchC simulators to experiment with memory hierarchies. For example, one of the exercises proposed by the authors ask students to analyze a trace produced by GCC for different cache organizations. For MIPS and SPARC architectures, there is a GCC port available for generating code to be run in ArchC simulators [6]. So, students can perform such an experiment using real-world applications, like JPEG or MPEG coders or some cryptography algorithm, using more than one processor and a number of different cache organizations. Moreover, they are able to compile their own programs, or examples provided by teachers, in order to do this kind of analysis. The simulation statistics provided by ArchC simulators combined with some pre-defined miss and/or hit penalty may be used to compute the performance numbers for each configuration tested. The most important part, they would actually see that the memory hierarchy may have a strong impact on the performance, and get a felling of how hard may be to tune a processor + memory system to a given real-world multi-media application, a common task for embedded systems designers.

5 Modeling Modern Architectures

An alternative approach for teaching computer architecture would be to use the MIPS architecture, following the COD book, in classes and to adopt different projects to be developed by the students. In one of the computer architecture courses at IC-UNICAMP, students were divided into several groups, and each one of those groups was assigned to a different project. The project was basically to develop a new functional model of a real-world architecture, most of them largely used in the industry as part of SoCs and embedded systems. For example, in this first semester of 2004, there are groups developing models of Intel XScale, IBM/Motorola PowerPC, Altera Nios, Infineon TriCore 2, OpenCores OR1K, and Motorola 68k/ColdFire ISAs as part of the computer architecture course. These functional models are heavily based on the instruction set, not concerning specific pipeline or cycle-accurate details of these complex architectures. They all execute one instruction per cycle. But the experience of getting into contact with different ISAs, more complex and modern than the simple RISC MIPS-I that is usually used in this kind of course, is instructive and attractive for students. They have the opportunity of learning details about architectures that are state-of-the-art in the industry, which is an extra motivation for the course. Moreover, students are getting some experience on how to build cross-compilers for GCC, in order to be able to create binary files to be loaded in their simulators.

6 Conclusions

ArchC is an architecture description language recently developed by the Computer Systems Laboratory (LSC), at IC-UNICAMP. Its based on C++ and automatically generates SystemC simulators from processor descriptions. ArchC is also capable of describing memory hierarchies. Its simulators have several capabilities to help simulation debugging and statistics collection that become useful in computer architecture education.

The language has been used for the last two semesters in computer architecture courses at IC-UNICAMP, both at the undergraduate and graduate levels, and the feedback received from the students was very positive. Among other projects, students developed functional models for real-world architectures like PowerPC and XScale. They got very motivated while developing their projects, and some of them even contributed with improvements on the ArchC tools that ended up as new features adopted by the ArchC Team in the official distribution, resulting in a kind of integration between education and research. The experience of developing architecture models gave students a deeper understanding of the concepts recently learned from the book and classes.

7 Acknowledgments

We would like to thank FAPESP (Grants 00/14376-2, 03/11674-0, and 2000/15083-9) and CNPq (ChameLeon Project) for the financial support to this work. We are also very grateful to all the students and teachers that are using ArchC, for education and/or research, whose feedback has been extremely valuable to the continuous improvement in ArchC tools.

References

[1] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin. Computer Sciencies Department., 1996.

[2] Markus Freericks. The nML Machine Description Formalism. Technical report, Technische Universitt Berlin, Fachbereich Informatiky, July 1993. Updated and Revised Version 1.5(Draft).

[3] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302, 1997.

[4] A. Halambi, P.Grun, V.Ganesh, A.Khare, N.Dutt, and A.Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *in Proc. European Conference on Design, Automation and Test(DATE)*, March 1999.

[5] J.L. Hennessy and D.A. Patterson. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 1998.

[6] http://www.archc.org. The ArchC Resource Center.

[7] http://www.cs.wisc.edu/ larus/spim.html. SPIM MIPS R2000/R3000 Simulator Homepage.

[8] Peter Marwedel and Birgit Sirocic. Multimedia Components for the Visualization of Dynamic Behavior in Computer Architectures. In *Proceedings of the Workshop on Computer Architecture Education*, 2003.

[9] The ArchC Team. *The ArchC Architecture Description Language Reference Manual*. Computer Systems Laboratory (LSC) - Institute of Computing, University of Campinas, http://www.archc.org, 2004.

[10] C. T. Weaver, E. Larson, and T. Austin. Effective Support of Simulation on Computer Architecture Instruction. In *Proceedings of the Workshop on Computer Architecture Education*, 2002.

[11] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa - machine description language and generic machine model for hw/sw co-design. In *Proceedings of the IEEE Workshop on VLSI Signal Processing, San Francisco*, 1996.

RTeasy
An Algorithmic Design Environment on Register Transfer Level

Hagen Schendel, Carsten Albrecht, and Erik Maehle
Institute of Computer Engineering
University of Lübeck
{schendel, albrecht, maehle}@iti.uni-luebeck.de

Abstract

Current developer tools and HDLs for system design are powerful instruments and support a variety of abstraction levels but they are too complex for didactic purposes. This paper describes the RTeasy IDE, an algorithmic design environment on register transfer level that has been developed to provide a simple system-design tool for didactic purposes to be used e.g. in introductory courses in computer engineering and digital design. The RTeasy tool suite includes an HDL, a simulator and further design features. As an example, it is applied to the design flow of a shift-multiplier.

1 Introduction

Nowadays, the design of complex systems and the implementation of new architectures demands high-level tool support. Different abstraction levels such as the gate, the register, and the processor levels are used to define and describe the structure and behavior of new designs. Especially the gate level is well-supported. Current hardware description languages (HDL) such as VHDL [2], Verilog, and ABEL [3] are utilized. Their integrated development environments (IDE) provide programming support, gate-level simulation, and download opportunities to suitable devices. Unfortunately, these languages and tools require a high-level knowledge and experience of system design. For educational purpose and due to the tremendous number of functionalities most of the common tools are too difficult for beginners. In the introductory course on computer engineering [4] and its following lab course at our university, the register transfer notation (RTN) of John P. Hayes [1] is applied. Here, second-year students of computer science are taught the principles of digital systems and system design. Hayes' RTN allows them to create own hardware modules based on an algorithmic description. Especially in a didactic view this way of description is well chosen because of the similarity of high-level programming and the RTN. The IDE RTeasy backs the design flow with editor, parser and simulator for a variant of Hayes' RTN.

In the following the RTeasy tool suite is presented. Section 2 introduces the RTeasy HDL. Then, the RTeasy IDE is described in Section 3 and applied in a small design example in Section 4. Section 5 concludes the presented work.

2 RTeasy HDL

The RTeasy HDL is a register transfer language based on RTN. In the following section the basic modeling constructs of the RTeasy HDL are introduced. Their usage is demonstrated by arbitrary chosen parts of RTeasy HDL code. An RTeasy program consists of two parts: declarations of components and the program body containing a description of the algorithm. Registers, busses and memories are the provided components and can be declared as follows:

Registers and Busses are declared by an identifier in uppercase letters and the number and order of bits in brackets. In the brackets the left number of the colon stands for the most significant bit (MSB) and the right one stands for the least significant bit (LSB). The numbers represent the indices used in the program.

Example:

declare register A(7:0), STATUS(1:5), RDY
declare bus INTERNAL_BUS(7:0)

A is an eight-bit register where the MSB has index 7 and the LSB has index 0, STATUS is a five-bit register with MSB index 1 and LSB index 5, and RDY is a single-bit register which does not need any indices. INTERNAL_BUS is an eight-bit bus similar to register A. In contrast to registers, busses hold their data for only one clock cycle followed by a reset to 0. In hardware they are e.g. realized as signal lines with tristate drivers.

Memories are declared by an identifier followed by address and data registers enclosed in brackets. These registers must be declared beforehand.

Example:

declare memory MEM(AR,DR)

Here, AR is the address register and DR is the data register. The memory dimension depends on the size of the address register determining its address space and the size of the data register determining the memory-word width. Data transfer between data register and memory is triggered by the commands **write** MEM and **read** MEM.

The program body generally describes a finite state machine where each state includes some concurrent hardware operations given by data transfers between registers or combined registers performed directly or via busses. A simple timing model is used with each statement executed in exactly one clock cycle.

Each state or concurrent-command sequence is separated by a semicolon and has the following form:

[*label* :] *concurrent operations* ;

Note that a label can be left out. The semicolon can be interpreted as a state transition and indicates the end of a clock cycle. In general, the next state is the one after the semicolon. For the modification of the succeeding state an absolute branch command is provided:

goto *label*

It can be combined with the if-statement described in the following to get a conditional branch.

The concurrently executed RT operations, see below, per state are given by a comma-separated sequence.

Example:

BUS ← A + B, RESULT ← BUS

Note that only busses instantly take new values that are written on it in the *same* clock cycle, registers do not take their new values until the *end* of the clock cycle. The example above shows concurrently executed RT operations and the use of busses. The first RT operation writes the sum of A and B on the bus BUS. The second RT operation switches the bus signals to the input ports of the register RESULT. A trace of the input values of RESULT shows after a half cycle that first all bits are zero, then some hazards follow, and finally the result of A + B appears. In fact the observed values are the same as the output of the adder circuit belonging to the expression A + B, delayed by the bus delay.

In the following all statements which can build up a concurrently executed statement sequence are shown:

Conditional Statements have the form:

if *expression* **then** *RT operations*
[**else** *RT operations*] **fi**

The conditions are evaluated one half cycle ahead of the RT operations they account for. So, in general the evaluation bases on the global state of all registers and busses at the end of the preceded clock cycle.

The **if**-statements can be used wherever an RT operation may occur. They may be nested to any arbitrary level as this is only a conjunction of conditions easily realizable by AND gates.

Expressions are used to model computations in a similar way to high-level programming languages. Here, the operands are registers, busses and bit-word constants instead of variables and constants. Parts of registers or buses and single bits can be combined to *combined bit-words* using the dot operator.

Example:

A ← A(6:0).A(7)

The example shows a left rotation by one bit of the 8-bit register A. Combined bit-words are handled like normal registers or busses because they are only arbitrary combinations of signals in hardware. Bit-word constants are either positive decimal, binary led by % or hexadecimal led by $.
The set of operators includes the binary operators $+, -, <, <=, >, >=, =, <>$, and, nand, or, nor, xor and unary operators $-$ (sign) and not with usual precedences. The arithmetic operators + and − generate a carry bit, thus, the result is one bit wider than the widest operand. All other operators deliver a result being as wide as the widest operand. Missing bits of the smallest operand are extended by leading zeros. For arithmetic operations registers, busses and combined bit-words are processed right-aligned. Logical operators operate bitwise on bit-words.

Example:

```
A <> B # 1 if A and B not equal
%1 + 7  # %1000 or 8
not B   # B is bitwise negated
```

Comments can be inserted wherever it is necessary by # as a line comment.

Register Transfer Operations are written as

combined bit-word ← *expression*

It is not allowed to transfer data from bus to bus so that a bus can only be used on the left or on the right side in an RT operation. This is due to the fact that busses are realized as signal lines. The triggering of an RT operation results in signal propagation from the circuit belonging to the expression to the input ports of the combined bit-words elements. When the triggering edge of the operation units occurs the registers read their input signals and save them for the next cycle. When an RT operation writes on a bus, the signal would be propagated through other circuits representing expressions using the bus.

On many points in the execution of algorithms further computations depend on values evaluated shortly before, i.e. in the same clock cycle.[1] The application of RTeasy constructs defined above does not allow to use the results of any RT operation in the conditional expression of a conditional statement in the same concurrent statement sequence. The conditional expressions are evaluated utilizing the global state of the *preceding* clock cycle. So, there is no chance to realize a conditional branch by a goto-statement embedded in an if-statement if the conditions need the values of the *same* clock cycle. The general solution is a bulky conditional branch in the next state where the first operation of the branch is included. In the example below, the first program code consumes two clock cycles because the conditional branch is performed in a separate cycle although it would be possible to avoid it. This problem occurs frequently so that the introduction of a handy notation for these conditional branches is worthwhile. That is why RTeasy provides an additional separator: the pipe symbol. Now, a state can be of the form:

[*label* :] *concurrent operations* | *conditional branch* ;.

The *conditional branch* which is an **if**-statement only including **goto**-statements may use the results of the RT operations on the left side of the pipe operator symbol. So, the pipe operator saves one clock cycle per each step of the loop, see the program example. For internal processing or hardware implementations, the pipe symbol is expanded to equal but bulky statements.

Example:

The RTeasy program

```
LOOP: COUNTER ← COUNTER + 1;
      if COUNTER < 20 then
        goto LOOP
      else
        # do something
      fi;
      # go on
```

can be refactored by

```
LOOP: COUNTER ← COUNTER + 1
      | if COUNTER < 20 then goto LOOP;
      # do something
      # go on
```

The command **nop** must be used to indicate that no RT operations should be triggered. It can be used to describe a state with no signal output.

[1] This has also been taken into account during the design of the C programming language regarding the `++i` construct.

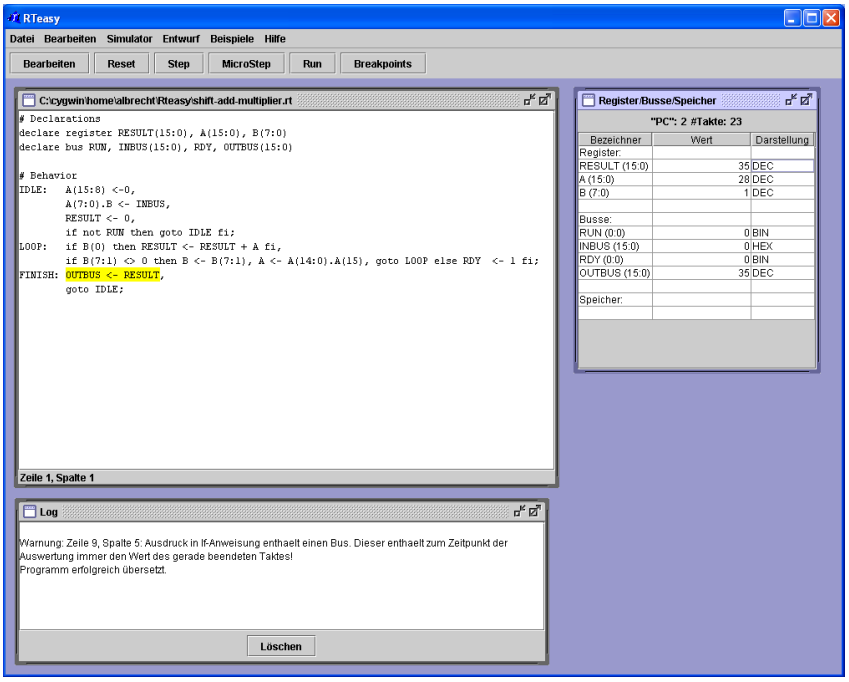


Figure 1: Stepwise Simulation in RTeasy

3 RTeasy IDE & Simulator

The design concept of the IDE of RTeasy resembles an assembler or embedded systems IDE. Actually, the only thing still missing is an opportunity to download developed designs to target devices. It provides a text editor with usual functionalities and a basic help system. The IDE has two working modes, editing and simulation. After launching RTeasy, the system is always set in editing mode where programs can be written, loaded, or saved. The simulation mode is entered by clicking the 'Simulate' button. Then the system performs syntactic and semantic analysis on the contents of the editor window which is the upper window on the left side in Figure 1. In case of a successful analysis, the simulation status window which is the upper one on the right side in Figure 1, pops up. It contains all declared registers, busses and memories in the sequence of their declaration. The full memory contents can be shown in a separate window. Each register and bus is depicted and attributed with its dimensions and current value. The contents of registers, busses, and memory cells may be shown in one of five modes: binary, decimal, signed two's complement decimal, hexadecimal and signed two's complement hexadecimal. All values of registers, busses, or memory cells can be interactively changed by the user. The simple concept avoids complex layout problems as they would occur by graphical representations such as RT-level block diagrams and nevertheless displays all relevant information.

Simulation Capabilities

The user controls the simulator by some buttons well known from other simulation environments. There are 'Reset', 'Step', and 'Run/Stop' with the expected functionality and the 'MicroStep' button explained below.

One 'Step' begins at each simulated clock cycle with the triggering edge of the control unit. Note that our RT designs can be split up into two units, the control unit representing the algorithmic behavior and the operational unit including all hardware components such as registers, busses, memories and arithmetic and logic units. The simulated state is marked in the editor window, the upper left one in Figure 1, by a colored background and the contents of the status window shows the values the registers, busses and memories have taken at the end of this clock cycle. The values shown on

busses are reset at the end of each clock cycle because the control unit emits other control signals at the beginning of the next cycle.
The 'Run' button launches a continuous execution of the 'Step' simulation and changes its caption to 'Stop' so that the next click aborts the infinite simulation. Simulation may also end when the last state is executed without a **goto**-statement or the program quits because of the **goto end**-statement. 'MicroStep' provides a detailed view to the concurrent execution of RT operations. Although they are executed concurrently this feature allows the traversal through the states on an operation-by-operation base. During the traversal the currently executed RT operations are marked yellow whereas conditions not met are marked with magenta background. The contents of the status window is consecutively updated as well ignoring their concurrency. If the operations would be executed in the order of appearance in the program 'MicroStep' and 'Step' simulation would differ and even the first one would not hold because busses might not reach their final values. Thus, all bus writes are simulated first.
In addition the IDE provides breakpoints which are useful for debugging purposes.

Design Tools

Beyond modeling and simulating features RTeasy IDE provides design tools for further system development. These features are gathered in the 'Design' menu. They include extraction of control and conditional signals and model expansion. The extraction of control and conditional signals defines the input and output of the control unit. RT operations are simply assigned with numbers representing their triggering control signal lines. Equal RT operations will be assigned to the same number. Conditional signals are extracted from **if**-statements. Boolean expressions that contain expressions of other types such as arithmetic ones are splitted, nested **if**-statements are flattened by combining the boolean expressions. This flattening is necessary to build up the state transition table of the control unit. Furthermore, conditional signals that only occur together in state transition tables can be merged by optimizations. Model expansion unfolds the right side of each pipe symbol and merges nested **if**-statements to approach the description of a synthesizable finite state machine.

Currently, the opportunities of the 'Design' menu will be enlarged by student research project. The goal of this work is a function that generates VHDL code for control and operation unit. The exported code can be used and simulated by other tools such as Mentor Graphics' FPGA Advantage [5].

4 RTeasy Example

The example shown in Figure 2 is a simple shift/add multiplier. Its interface consists of two 16-bit busses, an incoming (INBUS) and an outgoing one (OUTBUS), and two signals (1-bit busses): RUN (in) and RDY (out). Furthermore, the model makes use of two 16-bit registers A, and RESULT and one 8-bit register B. The underlying algorithm is quite simple:

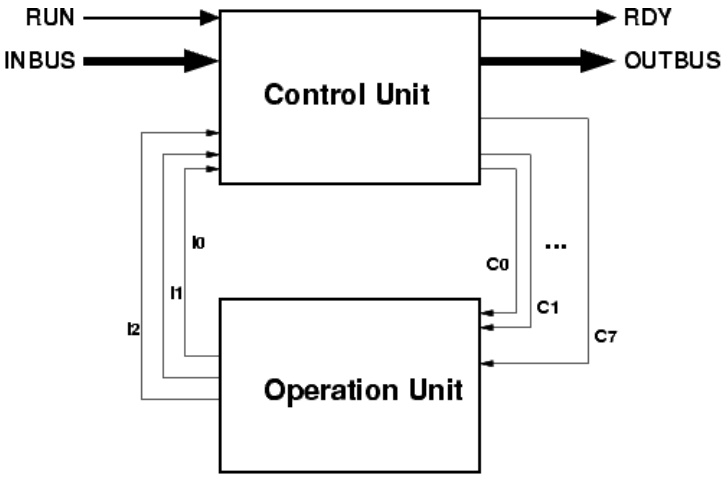
1. As long as RUN is not set, read both factors from INBUS. The first factor, which takes the higher 8 bits, is transferred to the lower part of register A. The second one, stored in the lower 8 bits, is put to register B. The higher 8 bits of A and RESULT are initialized by zero.

2. For each bit k in B, beginning at the LSB with index 0, add $2^k \cdot$A to RESULT if $B_k = 1$.

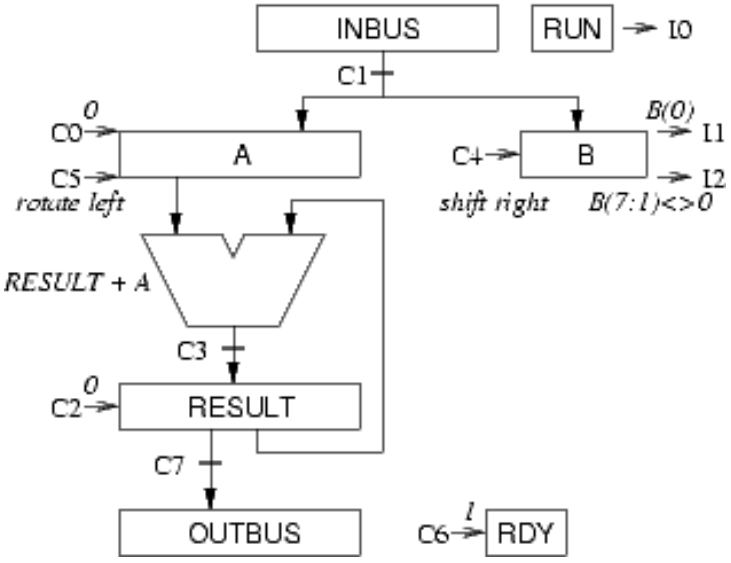
3. During the last iteration RDY is set to 1 and in the next cycle the contents of RESULT is written on OUTBUS.

The addition of $2^k \cdot$A is realized by left-shifting of A. The test $B_k = 1$ is realized by right-shifting B and testing the LSB. The loop is aborted if the remaining part of B does not contain any bit with value 1.
The screenshot in Figure 1 shows that RTeasy identifies 7 unique RT operations and 3 conditional expressions. The conditional expressions are mapped to input signals for the control unit and the RT operations to output signals to affect the behavior of the operation unit. The block diagram of Figure 2(b) and the listing of Figure 2(c) are attributed with these signals. Figures 2(a) shows the composition and interconnection of the two units.

This example design contains three kinds of RT operations with different effects shown in the block diagram. Simple data transfer operations such as C1 and C7 that only transfer data from one entity to another. Then there are operations on registers such as shift, rotate, and set/reset operations (C0,C2,C4,C5,C6). The RT operation triggered



(a) System diagram.



(b) Block diagram of the operation unit.

 # Declarations

declare register RESULT(15:0), A(15:0), B(7:0)
declare bus RUN, INBUS(15:0), RDY, OUTBUS(15:0)

 # Behavior

IDLE:	A(15:8) ← 0, A(7:0).B ← INBUS, RESULT ← 0,		C0,C1,C2
	if not RUN **then goto** IDLE **fi**;	I0	
LOOP:	**if** B(0) **then** RESULT ← RESULT + A **fi**,	I1	C3
	if B(7:1) <> 0 **then** B ← B(7:1), A ← A(14:0).A(15), **goto** LOOP	I2	C4,C5
	else RDY ← 1 **fi**;	I2	C6
FINISH:	OUTBUS ← RESULT, **goto** IDLE;		C7

(c) RTeasy program code describing algorithmic behavior and usage of components.

Figure 2: Design example of a shift/add multiplier.

by C3 is a special one. It involves an adder circuit which is represented by the V-shaped symbol in the block diagram.

The extracted information is used to generate a Moore or Mealy state machine for the control unit. These state machines can be minimized by well-known techniques and implemented utilizing their optimized switching functions taken from the state transition table, one-shot circuits, or a modulo sequencer.

5 Conclusion

In this paper, an algorithmic design environment on register transfer level is presented. The tool bases on a modified version of the RTN introduced by John P. Hayes [1]. The IDE is implemented using Java and works fine on different platforms such as Solaris, Linux and Windows 2000/XP. In the last winter term, its usability was tested and proven by the application in the introductory course of computer engineering. Second-year students of computer science quickly accepted the tool and easily applied the IDE at home and at the university to solve exercises and to deepen their acquired knowledge of system design. Moreover, they have supported the development of RTeasy with critical remarks and useful proposals for improvement.

In contrast to previous paper designs it was know possible for them to test and debug their algorithms with the simulator. Furthermore, the simulation proved to be very helpful in introducing basic RT algorithms in the lectures replacing "hand simulations" on the blackboard.

Beside the generation of VHDL code, the future work includes the extension of simulation features as well as more design tools. It is planned to include the generation and viewing of VCD (Value Change Dump) traces of register and bus values into the IDE. Additionally, a capability of unit tests should be implemented to support the proofreading of student exercises and automatic testing of complex designs.

References

[1] John P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, 3rd Edition, 1998.

[2] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 1995.

[3] Lattice Semiconductor Corporation. *ABEL-HDL Reference Manual*. Reference Manual, DSNEXP-ABL-RM Rev 8.0.2, 2003

[4] Erik Maehle. *Technische Grundlagen der Informatik*. Course script, Institute of Computer Engineering, University of Lübeck, Germany, 2003.

[5] Mentor Graphics Corporation. *FPGA-Advantage*. Datasheet, 2003

CREATING SHARABLE LEARNING OBJECTS FROM EXISTING DIGITAL COURSE CONTENT

Rajendra G. Singh rajsingh@tstt.net.tt
Margaret Bernard mbernard@fsa.uwi.tt
Ross Gardler rgardler@saafe.org

Department of Mathematics and Computer Science
Faculty of Science and Agriculture
The University of the West Indies
St. Augustine, Trinidad, W.I.

ABSTRACT

Our research is targeting Instructors that have course material as a collection of various digital documents (raw content) and whose objective is to re-structure this raw content into a standards-based format in order to support a higher degree of content reuse, sharing and easier maintenance.

In previous work, we differentiated a Reusable Learning Object (RLO) from a Sharable Learning Object (SLO) and developed a model which can be applied to convert RLOs into SLOs [4]. In this paper, we present an iterative five-step method to re-structure selected raw content into RLOs. The model from the previous work is then applied to convert the RLOs into SLOs.

Thus far, we have used raw content from one Instructor's Computer Architecture course and found that conversion of the raw content can successfully result in a subset of the raw content residing in SLOs, a form which is more conducive to reuse, sharing and content maintenance.

In ongoing work, we are applying the methodology to additional raw content from several other Instructors (Computer Science courses) with a view to refining and automating the process where possible.

1. INTRODUCTION

Many instructors have developed or have access to digital content for their respective courses. It is usually the case that most of the content (which we will call *raw content*) is a collection of documents in various formats such as PDF, HTML, XML, other documents created with Word Processor applications, and an abundance of slide-documents created with presentation applications, and in recent years, multimedia (audio/video) documents.

While Instructors have, in the past, successfully used such content in the physical classroom environment, they are now challenged to adapt their material to support eLearning goals. The term eLearning is defined to be *learning in an environment where the classroom is no longer a physical entity in the learning process, but transcends space and time with the use of computers and computer-related technologies* [1]. The classroom is thus a digital classroom. Having a digital classroom means, that the content (digital content) placed in standards-based eLearning systems receives more opportunities for reuse and sharing. It is therefore desirable that the content be structured to allow both maintaining the relevance of the content over time, as well as pedagogically defined to be contained in a vessel to promote reuse and sharing among different courses.

Some Instructors are discovering that transporting existing raw content into eLearning systems using Content Packaging standards does not necessarily produce reusable and sharable content, because there is no pedagogical framework involved. For example, the SCORM Content Aggregation Model (CAM) [2] facilitates the storing, aggregating, sequencing and packaging of raw content into course modules and courses, but the content itself can exist in forms (raw content) that are difficult to maintain, reuse and share.

Gehringer [3] reported that it was difficult to achieve sharing of Computer Architecture course documents (raw content) among Instructors. One reason given by the Instructors was that they had obsolete and unpolished content within their collection of raw content.

With issues such as these in mind, our aim was to develop a method that will select existing raw content and re-structure it to facilitate reuse and sharing among standards-based eLearning systems. The standards-based format is a Sharable Learning Object (SLO), which we can concisely define as *a content vessel with pedagogical constraints*.

The full definition of a SLO is given and explained in Section 2. Section 2 also describes the method for re-structuring selected raw content into SLOs. In addition, some comments are included as part of the methodology since we have applied it to select raw content from one instructor's Computer Architecture course (CS21E).

We review in Section 3 the infrastructure required for using SLOs, and identify how far we have reached with the design and implementation of the infrastructure.

Section 4 gives the conclusion, some ongoing and future work, followed by the list of references.

2.0 CREATING SHARABLE LEARNING OBJECTS

The method that we have developed takes selected raw content and re-structures it into Sharable Learning Objects (SLOs). The method would not necessarily produce a complete set of SLOs that can be aggregated and sequenced to reproduce what the raw content may have successfully been able to achieve. This is due to the pedagogical nature of our SLO, where content must be appropriately matched to a Single Learning Objective for inclusion. Selected content is placed in what we call a Reusable Learning Object (RLO). The distinction made between a RLO and SLO is given next.

2.1 Distinction between a Reusable Learning Object and a Sharable Learning Object

In previous work, Singh and Bernard [4] defined and differentiated a Reusable Learning Object (RLO) from a Sharable Learning Object (SLO). The definition which follows was originally adapted from Cuthbert and Himes [5], and Figure 2.1 is a conceptual representation of the SLO as implemented:

*A **RLO** is a reusable chunk of content with the following **two fundamental properties**:*

- *instructionally sound content with a focused learning objective*
- *facility that allows the learner to practice, learn and receive assessment*

*and, a **SLO is a RLO** with the following additional **Interoperability property**:*

- *metadata or keywords that describe the object's attributes and mechanisms for communicating with any eLearning System.*

Note that the latter part of the Interoperability property is what determines if a RLO is sharable. We used the Sharable Content Object Reference Model (SCORM) Runtime Environment (RTE) [2] to implement this part of the property. The first part of the Interoperability property ensures that a specific SLO can be discovered among others in a repository of SLOs. In implementing this first part, we applied metadata to the SLO as specified by the SCORM Content Aggregation Model (CAM) [2].

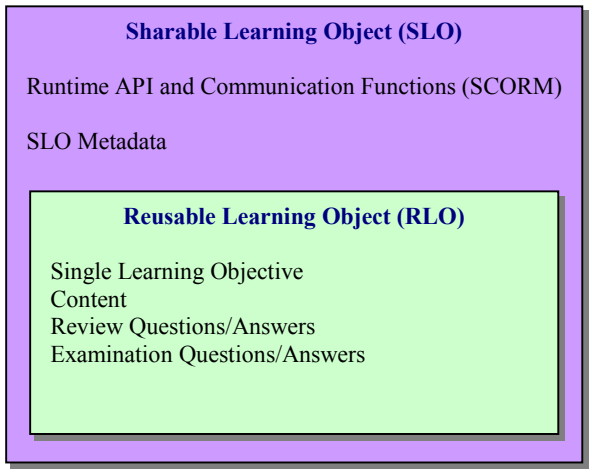


Figure 2.1 – Conceptual representation of a SLO

2.2 Methodology used to extract existing digital content into Reusable Learning Objects

The aim of this methodology is to select and extract as much of the existing raw content into Reusable Learning Objects.

The approach is an iterative five-step process to select appropriate content for the RLO with opportunities to refine and re-structure as the extraction is taking place. The steps are as follows, along with some experiences we had with the Computer Architecture course (CS21E):

1. ***Start with a high-level syllabus, and create a detailed Table of Contents (TOC) for the course where each learning topic and subtopic can be easily identified.***

   In most cases, a document should already exist with this information. The CS21E course had such a document, and we simply refined it to produce the TOC as required. It is quite reasonable to expect that the refinement continue while the other steps are taking place. Figure 2.2 is a partial listing of the TOC we used for the CS21E course.

2. ***Review each topic/subtopic and list as many Learning Objectives as possible as non-conjunctive sentences so that each sentence deals with one and only one Learning Objective.***

   Learning Objectives are statements of what the learner will be able to do after studying the content [6]. An example of a Learning Objective statement is "*The learner would be able to decompose the Instruction Cycle into subtasks*". Content must only relate to the single Learning Objective, hence the reason why we require non-conjunctive statements.

Figure 2.3 lists a set of Learning Objectives for some of the topics/subtopics shown in Figure 2.2.

Pipelining:
- Introduction
- Instruction Cycle
- Throughput
- **Hazards:**
  - Introduction
  - Structural Hazard and Solution
  - **Control Hazard and Solutions:**
    - Introduction
    - Multiple Streams
    - Pre-fetch Branch Target
    - Loop Buffer
    - Branch Prediction
    - Delayed Branch
  - Data Hazard and Solution
- **Review Questions:**

Micro-Operations:
- Introduction
- Fetch Cycle
- Interrupt Cycle
- Execute Cycle
- Micro-Operations Flowchart
- **Review Questions:**

Input/Output:
- Introduction
- I/O Module Function
- I/O Module Structure
- **I/O Techniques:**
  - Programmed I/O
  - Interrupt Driven I/O
  - Direct Memory Access
- **Review Questions:**

Figure 2.2 – Partial detailed Table of Contents for the CS21E course

Pipelining:

Introduction

- ✓ *Be able to decompose the Instruction Cycle into subtasks*

  <Content …Instruction Cycle… />
- ✓ *Be able to identify how pipelining improves throughput*

  <Content …Throughput… />

Hazards:

- ✓ *Be able to identify when a hazard can occur*

  <Content …Introduction… />
- ✓ *Be able to solve the problem of a structural hazard*

  <Content …Structural Hazard… />

Control Hazard and Solutions:

Introduction

- ✓ *Be able to solve the problem of a Control Hazard using Multiple Streams*

  <Content …Multiple Streams… />
- ✓ *Be able to solve the problem of a Control Hazard using Pre-fetch Branch Target*

  <Content …Pre-fetch Branch Target… />
- ✓ *Be able to solve the problem of a Control Hazard using Loop Buffer*

  <Content …Loop Buffer… />
- ✓ *Be able to solve the problem of a Control Hazard using Branch Prediction*

  <Content …Branch Prediction… />
- ✓ *Be able to solve the problem of a Control Hazard using Delayed Branch*

  <Content …Delayed Branch… />
- ✓ *Be able to solve the problem of a Data Hazard*

  <Content …Data Hazard… />

Review Questions:

FIGURE 2.3 – Learning Objectives for some of the Topics/Sub-topics shown in Figure 2.2

3. Select the associated raw content for achieving each Learning Objective identified.

In this step, select, separate, edit and refine the appropriate content for each Learning Objective. This process will involve the copying and pasting of content from the original documents to form new documents (called Assets) with only the selected content. It is important to ensure that navigational links are maintained when multiple Assets are created so that the flow of content is as intended. A main Asset should be identified, which is the one with the starting content. At this point, we can set up a RLO document to have the Learning Objective, and reference the main Asset. The navigational links will include the related Assets. Figure 2.4 (a) and 2.4 (b) depicts a scenario with multiple Assets.

4. Include the Review Questions/Answers.

In other definitions of Learning Objects [7] this part is not necessarily integrated, however we believe that it should be part of the RLO since the Questions/Answers we are identifying for inclusion should be directly

related to the Learning Objective and content. In this way, both the learning content and the formative assessment activities contribute to achieving the learning objective and are stored together as an integrated whole.

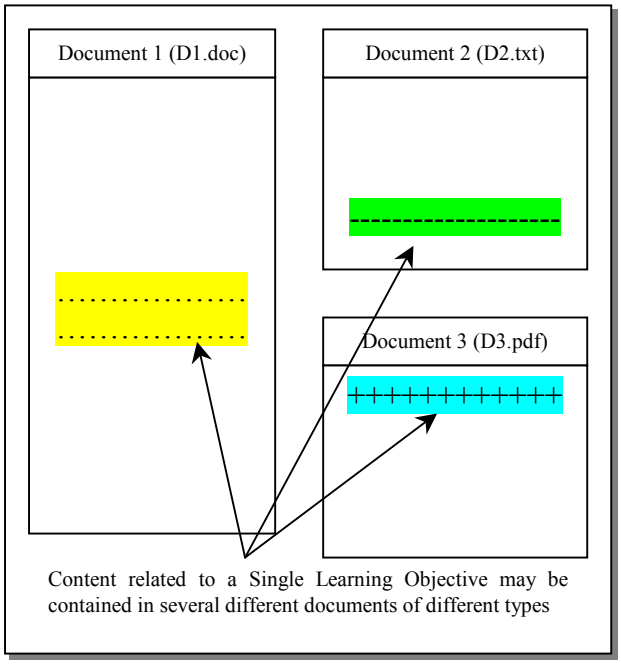


Figure 2.4 (a) – Multiple documents with content

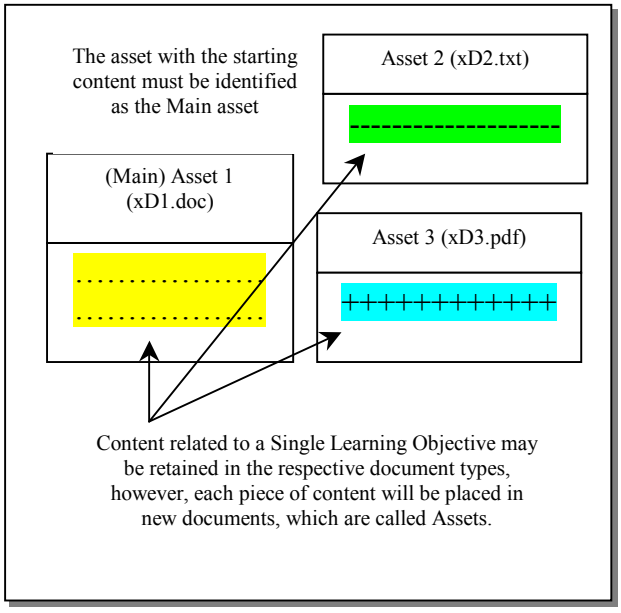


Figure 2.4 (b) – Multiple assets for a single RLO

5. Include the Examination Questions /Answers.

The Examination Questions/Answers is a preferred set of Questions/Answers that can be used by instructors for examinations purposes for summative assessment. Review Questions /Answers can be referenced here as well. Again, the Questions/Answers should be directly related to the Learning Objective and related content.

2.3 Transforming Reusable Learning Objects into Sharable Learning Objects

In order to transform a RLO into a SLO we must add metadata and include communication functions. The metadata will describe its properties which can be used to determine, to some extent, how an eLearning system (Learning Management System – LMS) should interact with it. The metadata is also used to facilitate discovery of the SLO when it is placed in a Digital Repository (see Section 3). Communication functions will allow a LMS to know what interactions are taking place between the Learner and the SLO so that the LMS can interact intelligently in the delivery of the course material.

The SCORM metadata standard for Content Objects [2] was applied to the SLOs. This metadata standard is based on the IMS [8] and LOM LTSC [9] metadata standard. We chose this standard because the metadata for each SLO can reside as a standalone XML document, thus lending support to the SLO structure depicted previously in Figure 2.1. The specifications are given in the SCORM metadata XML Binding [2].

To create different metadata documents for each SLO we used a template-metadata document with all the generic data filled-in, then replicated the document for each SLO and filled-in the unique data.

A model developed in previous work [4] allows the functions to be automatically attached to a RLO when converting to a SLO. For the communication functions we chose the SCORM RTE [2], and implemented the minimal set which includes a function to locate the LMS Application Programming Interface (API) when the SLO is first launched, then a call to LMSInitialize() to initialize the SLO and inform the LMS that it has been launched successfully or unsuccessfully. When the Learner completes reviewing the content presented by the SLO, a LMSFinish() function is triggered to let the LMS know, so that the next SLO sequenced can be launched or the course ended.

Adding the metadata and including the functions identified above satisfies the Interoperability property for transforming the RLO into a SLO.

3. INFRASTRUCTURE REQUIRED FOR USING SHARABLE LEARNING OBJECTS

This section gives a brief insight into the standards-based environment where SLOs can reside, and how to use existing standards to aggregate, sequence and package SLOs into modules and courses. It also identifies our current research and implementation status with respect to each aspect of providing an overall infrastructure for standards-based eLearning.

Digital Repository

SLOs are usually placed in a Digital Repository (DR) where facilities exist for their discovery. That is, search and selection based on the metadata, which describes the SLOs properties. In addition, by including a single Learning Objective for each RLO we will be able to perform searches in the DR based on Learning Objectives, which will allow for a focused discovery of more relevant SLOs.

Currently the SLOs that we have created for CS21E are held in directories, which are manually maintained. The design of a Digital Repository (DR) is still under consideration, however, our choice may be to use the IMS Digital Repository Interoperability (DRI) standards [10] to maintain the DR, since the goal is to have a standards-based DR.

Aggregating and Sequencing

In version 1.3 of the SCORM, standards were introduced to specify aggregation and sequencing of SLOs [2]. Having selected relevant SLOs, we can then aggregate and sequence into a module or course where the navigation among the SLOs is specified as part of the sequencing. We were unable to reproduce the entire CS21E course using our pedagogically defined SLOs, since we had an incomplete set of SLOs. The set is incomplete because some content is related to multiple Learning Objectives and was therefore not selected to be in a SLO. We continue to review possible solutions to this problem. In the interim, we can allow a deviation from the SLO definition for the problem cases, and allow multiple learning objectives and content. This approach would produce SLOs of a higher granularity, which has less opportunity for reuse, sharing, and are more difficult to maintain, but can be included in the aggregation and sequencing to reproduce the original module or course.

Packaging

As initially mentioned in Section 1, the SCORM CAM specifies how to package SLOs that have been aggregated and sequenced into a module or course for transporting between different SCORM conformant eLearning Systems. Such a package is called a Content Package [2]. We produced our Content Packages, inclusive of aggregating and sequencing using an application available from an opensource project [11].

Publishing

Content Packages are created for publishing or presenting to the Learner. Our publishing application is being developed as part of a larger opensource project [12], and is close to first-alpha completion (by August 2004). Notice in Figure 2.4 (b) that we have no restriction on document types for Assets. This is because our publishing application accepts multiple input formats and present the content in one user selected output format.

4. CONCLUSIONS, ONGOING AND FUTURE WORK

The paper presents an iterative five-step method to re-structure selected raw content into Reusable Learning Objects (RLOs). A previously proposed model is then applied to convert the RLOs into Sharable Learning Objects (SLOs).

We found that re-structuring of the raw content was a time-consuming event, which was due mainly to the enforced pedagogical structured definition we had for a RLO. Our approach has however resulted in a set of SLOs which we believe are of superior structural quality than the original raw content since they are more conducive to maintenance, reuse and sharing among standards-based eLearning systems.

It is important that a guided process, as we have described, exist for instructors to transform existing raw content to eLearning environments, since it will promote a more open approach for content reuse and sharing, which can in turn facilitate continuous refinement on content quality and relevance.

As ongoing work, we are continuing conversion of some other (Computer Science) courses for which Instructors have supplied raw content, and are considering ways of automating the process as much as possible in order to reduce the conversion time.

Further, the intention is to eventually design and create SLOs for most of the Computer Science courses offered by the Department of Mathematics and Computer Science at the University of the West Indies. The aim is to support initially blended-learning, fully online degree offerings, and then progressively extending both forms to other

departments and faculties at the University, which will extend our online reach to Caribbean students. Currently, some courses are being offered using WebCT [13], however, the course material used are all raw content. If more courses are heading towards full online degree offerings then it is best to have the content in the form of SLOs with supporting infrastructure as described.

One additional and very important area that we are looking at is the issue of content quality. The approach that is being considered is to have the SLOs in an internationally available Digital Repository (DR) for discovery and reuse in courses that may be offered at other tertiary-level institutions, and to facilitate peer-review and rating based on quality and relevance of content within each SLO.

REFERENCES

1. Singh, R. G. (2003). Dynamic Internal Access to Coarse Sharable Learning Objects: An approach for reducing re-authoring efforts, *M.Sc. thesis, The University of the West Indies, Trinidad.*

2. Sharable Content Object Reference Model (SCORM) [online]. http://adlnet.org

3. Gehringer, E. F. (2002). A Survey of Web Resources for Teaching Computer Architecture, *Proceedings WCAE 2002, Workshop on Computer Architecture Education,* Ancorage, Alaska, May 26, 2002. Workshop Proceedings, pp. 126-131. Available: http://www4.ncsu.edu/~efg/wcaes.html

4. Singh, R. G., Bernard, M. (2004). A Model for Maintaining Interoperability of Coarse XML Sharable Learning Objects after Re-Authoring in a Standards-Based Editor. *Proceedings of the Winter International Symposium on Information and Communication Technologies - A Volume of the ACM International Conference Proceedings Series.* pp: 309-314.

5. Cuthbert, A., Himes, F. (2002). Creating Learning Objects with Macromedia Dreamweaver MX [online]. http://www.macromedia.com/resources/elearning /objects

6. Dee Fink, L. (2003). Creating Significant Learning Experiences: An Integrated Approach to Designing College Courses, Jossey Bass.

7. Wiley, D. A. (2000). Connecting learning objects to instructional design theory: A definition, a metaphor, and a taxonomy. In D.Wiley (Ed.), *The Instructional Use of Learning Objects* [online]. http://reusability.org/read/chapters/wiley.doc

8. IMS. (2000). *Learning Resource Meta-data Specification Version 1.1* [online]. Available: http://www.imsproject.org/specifications.cfm

9. IEEE/LTSC. (2001). *Information Technology - Learning Technology - Learning Objects Metadata LOM: Working Draft 6.1 (P1484.12)* [online]. Available: http://ltsc.ieee.org/doc/ index.html

10. IMS Digital Repositories Specification (2003). [online]. Available: http://www.imsglobal.org/digitalrepositories/index.cfm

11. The RELOAD Project. Website: http://www.reload.ac.uk/

12. The Burrokeet Project. Website: http://www.burrokeet.org/

13. WebCT portal at The University of the West Indies. WebSite: http://courses.sta.uwi.edu/webct/public/home.pl

Introduction to Formal Processor Verification at Logic Level : A case Study.

Paul Amblard, TIMA-CMP, 46 av. Félix Viallet, 38031 GRENOBLE Cedex, France
Fabienne Lagnier, Vérimag, Equation, 2 avenue de Vignate, 38610 GIERES, France
Michel Levy, LSR-IMAG, B.P. 72, 38042 St MARTIN D'HERES Cedex, France
email : Paul.Amblard, Fabienne.Lagnier, Michel.Levy@imag.fr
Université Joseph Fourier, Grenoble, France.

Abstract—
This paper presents the case study proposed to 3^{rd} year students in our department of computer science. It is a practical activity in the first "Computer Architecture" Unit of the curriculum. This practical activity has several aims : 1) understanding a subtle mechanism in processor architecture, 2) experimenting the relations between logic level and RTL level descriptions and 3) practicing formal methods of verification. The main original point is the use of extraction (and minimization) of the full description of an automaton from the logic schema based on flip-flops and gates. In a certain way, the reverse of classic "automaton synthesis".

I. Introduction

Computer Science is an important component in Grenoble University due to environment : Grenoble region is surrounded by mountains and semiconductor plants. It is sometimes described as a kind of Silicon Valley. 2003 serious estimations[1] give the following data : 17700 people work in computer science, 12300 in electronics, 2300 computer scientists work for electronic components sector and 900 others in the field of electronic CAD.

Our activity is motivated by 2 common sense ideas :

- Not all the computer scientists are processor architects but every computer scientist must know architecture principles. ([12])
- Not all the computer scientists use formal methods in their work, but every computer scientist must know formal methods principles.

One of our tasks is to give all the students a good basic knowledge in processor architecture. Some of them will further specialize in this field. Many others will not and will choose image synthesis, embedded systems, networks engineering or one of many other fields. We want also them to have practiced formal methods.

In this paper we shall give the main ideas founding this activity, then, in section 3 we shall describe the context of education. The techniques used and the example itself are described in further sections 4 and 5.

[1] Reported by "Electronique International Hebdo, 25 Mars 2004, p 38

II. Ideas in processor architecture and verification

The main ideas we want our students to acquire are organized around three topics : 1) processor architecture, 2) abstraction levels in digital circuits description and 3) validation of a digital design.

A. Basic concepts in processor architecture

Let us give a list of basic principles that our students must master at the end of our Unit :

- There is a border between hardware and software. Machine language is the lowest level of software and the highest level of hardware. At the lowest level, interpretation of machine language instructions does not result from execution of a program but from circuits activity.
- Execution of one instruction costs several time steps. One of the reasons is that one instruction could require several memory accesses to fetch the instruction code and the operands.
- One such basic step consits in a state change in a "data-path" automaton. A "data-path" automaton is a Finite State Machine (FSM) but we prefer to describe its *implementation* by registers and ALUs rather than its *function* by a list of states and transition arrows.
- Controlling the flow of steps is done by a controller taking into account Instruction Register and Interruption Requests. The controller behaviour can be described by an FSM with actions associated to states. Microprogramming was the basic technique to implement such controllers.
- Pipelining is a common technique used to implement processors. In this case it is not convenient to describe the controller by a "centralized" function.
- Pipelining introduces a problem for conditional control transfer instructions.

B. Basic concepts in digital circuits description

Before introduction to processor (and computer) architecture, the students already know the basics of digital design. They can synthesize and describe *small* circuits at the logic level. Processor architecture will make obvious the need for Register Transfer Level (RTL) description for *big* circuits.
The 4 kinds of circuits of figure 1 and associated descriptive formalisms are used. Students progress in understanding these categories. Obviously the border between small and big circuits is partially a matter of convention. As a consequence the border between logic level and RT Level is such that some "intermediate complexity" circuits can be described at both levels. The example presented in this paper was designed to be such a circuit. The proof itself relies on logic description.

C. Basic concepts in validation

The main technique used to check correct functionality in computer science is to put the object (hardware or software) at work (truly or by simulation).

Another technique is to list some properties the object must verify and to check them. A formal model must exist allowing to describe these properties and the implementation of the object. A formal checker can be used. This second technique is now commonly used for combinational circuits where the model is Binary Decision Diagram representation of boolean functions. ([1])

A similar approach, based on Model Checking, is used for Finite State Machines. In this case the problem of combinatorial explosion of the number of states cannot be avoided.

Theorem provers are often used in processor verification at a research level. (See for example conferences such as CAV, DATE, DAC, FMCAD, CHARME). M. Velev has very well described the introduction of such a technique in advanced education ([10]).

Our case study uses a variant of Model Checking and simulation.

III. Context of education

For the reader to understand WHERE our case study occurs in the curriculum, let us briefly present it. Our Unit is in a curriculum of computer science where formal methods and techniques are important. This Unit is the first one in computer architecture. The companion book (in french) is [2]. Archives of exams are available at [17].

In this section, we describe the place of architecture and formal methods in the curriculum of our department.

Before introduction to computer architecture, basic digital design techniques are already known (Karnaugh maps, FSM synthesis). Similarly, basic programming in C and in assembly language are already known. Assembly language is based on Sparc[2]. Concurrently with Computer Architecture, introduction to compilation and advanced programming in assembly language occur : How to program functions, environment and frame pointer management in stacks ? How to read code produced by a standard C compiler ?

This Unit of Computer Architecture is the first encounter between students and the Hardware/Software interface. A home-made simulator allows to discover the principles of instructions execution. It is based on a microprogrammed organization. After that introduction, simulation of a pipelined machine is made. The students must introduce the "by-pass" mechanism in the RTL description of the given processor. In the Unit, memory hierarchy is "independant" from instructions interpretation. We do not deal with the interactions between caching and executing. In this Unit, nothing is said about advanced techniques [9].

The practical activity presented hereafter illustrates (a part of) the pipeline organization of a SPARC.

For many reasons, not described in this paper, our department strongly focusses on formal methods. For instance in digital circuits design introduction [3], some circuits proofs are done. Students simulate the circuits at the logic level, then they are also invited to check equivalence of two combinational circuits, then they also verify equivalence between two synchronous implementations of Finite States Machines. Similarly programming techniques education are taught in relation with formal descriptions and verification of programs ([7]).

IV. Environment and Proof techniques

This section presents the tools and techniques used for this activity. The main originality is to use a kind of "reverse" synthesis of Finite State Machines.

A. How do we describe ?

All the descriptions are given in the language LUSTRE ([4] and [5]). In circuits description LUSTRE is close to Lola, the language used by N. Wirth in his book. ([11])

Description may be of different types :

- Circuits described as a set of nodes : the nodes contain logic gates and edge-triggered D-type flip-

[2]For technical reasons, we are currently moving from Sparc to another processor

Type	of circuits	implementation objects	description formalisms	level
small	combinational circuits	gates (pass transistors)	boolean algebra	logic
big	combinational circuits	adders, coders, MUXes	arithmetic and composition	RTL
small	sequential (synchronous)	gates, flip-flops	FSM bubbles and arrows	logic
big	sequential circuits	registers, ALUs, busses,..	FSM with actions	RTL

Fig. 1. The different kinds of circuits

```
node mux1bit (i, t, e: bool) returns (s:bool);
let
  s = (i and t) or (not i and e);
tel;

node add1bit (a,b, ret_in :bool) returns (som, ret_out: bool);
let
  som     = a xor b xor ret_in ;
  ret_out = a and b or a and ret_in or b and ret_in;
tel;

node addNbits (const N: int; a,b: bool^N) returns (r: bool^N);
var carry : bool^(N+1) ;
let
  carry[0] = false ;
  (r[0..N-1],carry[1..N]) = add1bit(a[0..N-1], b[0..N-1], carry[0..N-1]);
tel;
```

Fig. 2. A flavour of Lustre descriptions

flops. The only data type is boolean. (Cf node `mux1bit` or `add1bit`) hereafter.

• Generic circuits of size N, dealing with boolean vectors of size N. Registers have N flip-flops. Adder can be N bits wide. (Cf node `addNbits` hereafter). Notice the "implicit" repetition of add1bit in addNbits. N must be instanciated before effective use. This allows us to have a same description for any N-bits circuits, we only need to change one constant. The same feature exists in VHDL.

• Circuits described as a hierarchical or compositional set of nodes. The nodes can be different (cooperating) automata. The language is such that, basically, all the automata share the same clock. Due to this feature, LUSTRE is often refered to as a *synchronous* language. ([6])

This Lustre example in figure 2 is given for illustration.

B. What do we obtain ?

From the logic description, it is possible to simulate the circuit. The gate or flip-flops delays are not taken into account. Timing diagrams can be drawn. This step is done by the students.

Another use, more orignal can be made : We *compile* the circuit description. Let us examine the meaning of this compilation.

Given a circuit description of the FSM logic implementation by gates and flip-flops, the Lustre compiler [4], [16] can *compute* the automaton as a set of states and a full description of the two functions : transition function and output function. Obviously this is the reverse task compared to the very common synthesis tools available in all standard CAD packages.

If the input description contained several automata, the compiler computes the product automaton. The description of the result automaton is given either in an internal textual form, or in C language, ready for compilation, or in a graphical form. It could as well be given in VHDL or an other Hardware Description Language. The execution of the C version gives the same results than a simulator. We use the textual form with the students.

A complementary tool gives the minimal automaton equivalent to the proposed one.

This essential facility is used in the introduction to digital design to formally check equivalence between 2 circuits. ([3])

V. THE CASE STUDY

Our circuit is a pipelined processor. We got a VHDL description of a SPARC architecture from the European Space Agency site (Leon version [14]). To make the example as simple as possible, we made drastic simplifications and limited ourselves to a simple mechanism : delayed control transfer instructions. The example is organized around the part

	code	possible behaviours
	Instr1	sequence if `cond` is true at I2
I2	Brcond `label`	Instr1, Brcond label, Instr3, Instr5,..
	Instr3	
	Instr4	
	...	
`label`	Instr5	sequence if `cond` is false at I2
	...	Instr1, Brcond label, Instr3, Instr4, ...

Fig. 3. Delayed branch mechanism : a small SPARC program and the two possible behaviours given by the sequences of instructions

address	label	instr		sequences (values of PC)
0	`zz`	instr0		0 1 2 3 4 (not at 2)
1		instr1		0 1 2 3 7 (yes at 2)
2		brcond	`ss`	
3	`tt`	instr3		3 4 5 6 7 . (not at 5) (not at 6)
4		instr4		3 4 5 6 0 1 (yes at 5) (not at 6)
5		brcond	`zz`	3 4 5 6 7 3 (not at 5) (yes at 6)
6		brcond	`tt`	3 4 5 6 0 3 (yes at 5) (yes at 6)
7	`ss`	instr7		

Fig. 4. A complex short SPARC program and the possible behaviours. If the condition tested in line 2 is true, the sequence of values of the program counter is 0, 1, 2, 3, 7 ; if this condition is false the sequence is 0, 1, 2, 3, 4. Similarly if the condition tested in line 5 is false and condition tested in line 6 is false, the sequence of instructions is 3, 4, 5, 6, 7.

computing the next value of the Program Counter (PC). We study the so-called *delayed branch* mechanism.

A. How does progress a SPARC *Program Counter ?*

The system of SPARC is different from the standard one and is well known ([15], [13]). To make this paper self-contained, we recall it. There are Control Transfer Instructions (CTI). Different CTI exist : Jump and Link, Conditional Branch and Call. The instruction written immediately after a CTI is executed first, then the transfer of control occurs. This mechanism is known as *Delayed Branch*. The instruction inserted is said to be in the *Delay Slot*. We shall simplify here by considering only conditional branch instructions. We do not use the mechanism of *annul bit* in the frame of this paper.

In the small program of figure 3 two sequences of instructions may occur (assuming that Instr1, Instr3, Instr4, Instr5 are not CTI) :

- if the condition is **true** when it is examined in instruction I2 the sequence of instructions is
`[Instr1, Brcond label, Instr3, Instr5]`
- if the condition is **false** when it is examined in instruction I2 the sequence of instructions is
`[Instr1, Brcond label, Instr3, Instr4]`

This behaviour is made possible by the existence of a (classical) register Program Counter (PC) and of another information named Next Program Counter (nPC) in the documentation. The immediate question is obviously : *What occurs when two CTI are written consecutively ?* (However the standard practice of a programmer is not to write programs with such features [8].) The complete documentation ([13], [15]) explains the different possible behaviours in this case. We take here a simplified version.

We shall present such a situation in figure 4 : the program contains two consecutive conditional branches. They appear in lines 5 and 6.

Let us examine this small program. The expected behaviours depends upon the values of the condition during execution of instruction 5 and 6. For instance if the condition tested in instruction at address 5 and the condition tested in instruction at address 6 are both true, the sequence of values of the Program Counter is 3, 4, 5, 6, 0, 3. The others sequences are given on the figure itself.

B. Our experiment with this Very Reduced Computer

For this experiment, we use only :

- the Program Counter (PC)
- the Next Program Counter value (nPC),
- the combinational incrementer associated with these registers,
- the Instruction Register (RInst) containing the current instruction. It has two fields : opcode and displacement.
- the adder used to add a displacement (depl) to ob-

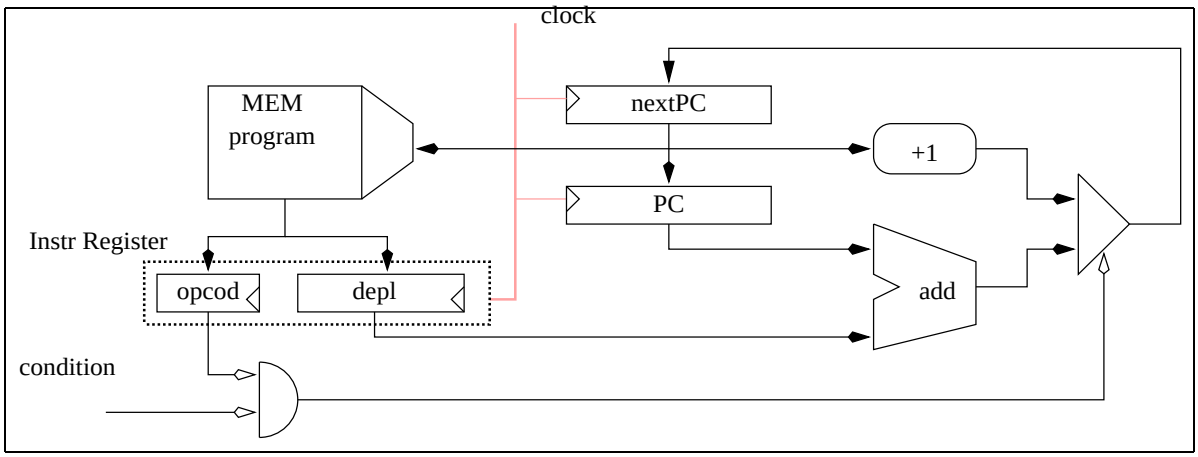


Fig. 5. Organization of the Program Counter updating in reduced SPARC p rocessor

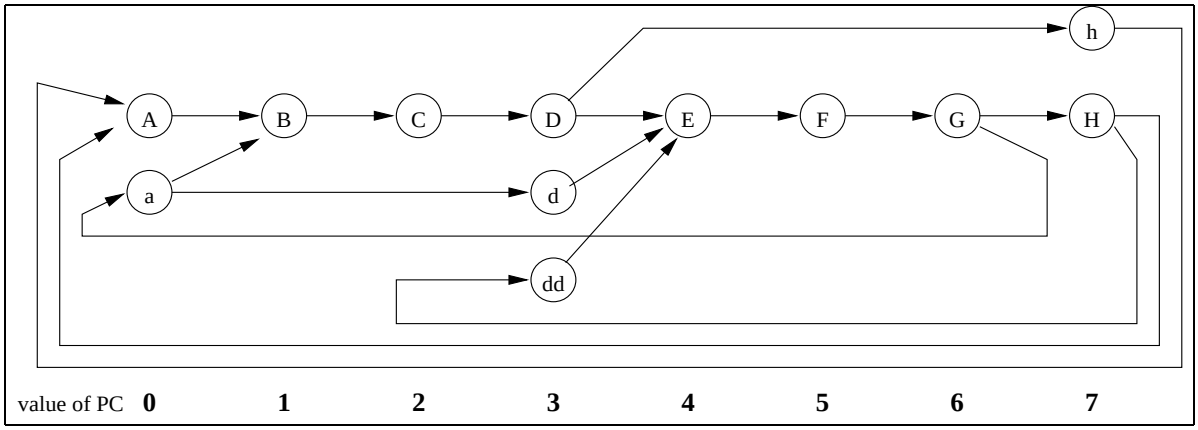


Fig. 6. The automaton obtained by the Lustre compiler. The corresponding values of the Program Counter are indicated. When a state has two "next states", the first one correspond to cond = true, the other one to cond = false.

tain the branch target address

The Register Transfer Level description of the system is given graphically in figure 5.

Our circuit is composed of this restricted SPARC and of a memory containing the aforementionned program. This memory can be a ROM because we do not use any STORE instructions.

Our experiment is based on a description of this processor+memory at a logic level. In this example we restricted the data path to 3 address bits and to 4 data bits. The ROM contains 8 4-bits words as in figure 4. The Op-Code has only one bit (true for a BRCOND, false for a NOP) and the displacement is coded on 3 bits. It is enough for our experiment as will be shown.

To put focus on the role of the condition, we consider it to be an external input. The logic description is simple : 3 bits adder, 3 bits incrementer,... The only input of this circuit is `cond`.

The students are invited to compile the Lustre description of this logic description, obtain an automaton, and minimize it.

They obtain the automaton described by figure 6.

C. Results and comments

Figure 6 gives the states obtained from the compiler. How do we understand this automaton ? We name the states A, B, C, D, E, F, H and a, d, dd and h. A is the initial state. In regards to the states we added the corresponding values of the Program Counter. For instance in states D, d and dd, the PC value is 3. Let us comment a transition in the automaton :
- Arrow D $\rightarrow$ h (PC = 3 $\rightarrow$ PC = 7) correspond to the instruction Brcond at address 2 and a condition True. As we have already seen, in this case the sequence of values of PC contains 2, 3, 7.

All the possible behaviours given in figure 4 correspond to a path in this automaton. The sequence of values of the PC 3, 4, 5, 6, 0, 1 (condition true in

instruction line 5 and condition false in instruction at line 6) correspond to the sequence of sates D, E, F, G, a, B.

Detailed exploration of this automaton gave us confidence that our PC computation mechanism is correct with respect to the specification of the processor with delayed branch. All the PC sequences of figure 4 are present on the automaton. We could also observe that our simplified model has introduced an artefact : *cond* seems to be tested one clock cycle too late.

The main activity of students is this task of understanding. They have to enter in the SPARC documentation and they must relate the results of this lab sketch to the "true world"

VI. Conclusion

We have presented an introductory activity for three fields : computer architecture, digital logic design of processors and formal verification. Obviously no generalisation of this technique can be made for a true size processor. We make the students aware of this point.

This case study also shows the relations between RTL and logic levels. It seems to us necessary to explore some logic implementations of RTlevel tricks such as pipeline, branch delayed instructions, ..

Introduction to read papers about formal proofs of processors would be a further step, but it needs complementary knowledge for the students. Thanks to M. Velev ([10]) we have a rich list of references as a starting point.

References

[1] **S. B. Akers :** Binary Decision Diagrams, IEEE Transactions on Computers, Vol C-27, June 1978.

[2] **P. Amblard, J.C. Fernandez, F. Lagnier, F. Maraninchi, P. Sicard et P. Waille :** *Architectures Logicielles et Matérielles*, Dunod, 2000 (in french).

[3] **P. Amblard, F. Lagnier and M. Levy :** Introducing Digital Circuits Design and Verification Concurrently, Proceedings of the 3^{rd} European Workshop on Microelectronics Education, Aix en Provence, 18-19 May 2000, Kluwer, pp 261-264.

[4] **N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud :** The Synchronous Data-flow Programming Language Lustre, Proceedings of the IEEE, pp 1305-1320, September 1991.

[5] **N. Halbwachs, F. Lagnier and C. Ratel :** Programming and Verifying Real-time Systems by Means of the Synchronous Data-flow Programming Language Lustre, IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, September 1992, pp 785-793.

[6] **N. Halbwachs :** *Synchronous programming of reactive systems*, Kluwer Academic Pub., 1993

[7] **M. Levy, L. Trilling :** A PVS-Based Approach for Teaching Constructing Correct Iterations FM'99 (Formal Methods), LNCS 1709, pp 1859-1860

[8] **R. Paul :** *SPARC Architecture Assembly Language Programming, and C*, Prentice-Hall, Inc., 1994

[9] **Sima D., Fountain and Kacsuk** *Advanced Computer Architectures*, 1997, Addison-Wesley

[10] **M. Velev :** Integrating Formal Verification into an Advanced Computer Architecture Course, ASEE 2003 Annual Conference, `http://www.ece.cmu.edu/~mvelev/ASEE03.pdf`

[11] **N. Wirth :** *Digital Circuit Design*, Springer-Verlag, 1995.

[12] I.E.E.E Micro, may-june 2000, Computer Architecture Education.

[13] *The SPARC Architecture Manual*, version 8, Prentice-Hall, Inc., 1992.

[14] `http://www.estec.esa.nl/wsmwww/leon/leon.html`

[15] `http://www.sparc.com/standards/V8.pdf`

[16] `http://www-verimag.imag.fr/SYNCHRONE`

[17] `http://tima-cmp.imag.fr/~amblard/EXAMS/exams.html`

The Case for Broader Computer Architecture Education

William J. Dally
Computer Systems Laboratory
Stanford University

Keynote address

Most introductory computer architecture courses – at both the graduate and undergraduate level – are primarily courses on CPU architecture. They tend to cover instruction set architecture, processor microarchitecture, and perhaps caches. While this is all useful content, our students would be better served by an introductory course that paints a more complete picture of computer architecture and one that better places computer architecture in context with the related fields of digital design and compilers.

Most computers today are not the desktop, laptop, or server boxes that we have historically associated with computing, but rather are embedded computing devices that control the engines in our cars, perform the modem functions in our cell phones, process the images in our cameras, TVs, and printers, or process packets traversing networks. The computing in these devices is typically performed by a combination of CPUs and special purpose hardware. The hard problems solved by architects in these systems do not involve the CPU, but rather the system-level organization of the device – the division of the problem over computing resources and the interconnect, memory organization, and I/O organization of the system. The CPU(s) is (are) typically not a major contributor to either cost (a RISC CPU with I and D caches is less than 1mm^2 today – and most of that is cache) or performance (most of the heavy lifting is done by special-purpose devices).

Even for PCs and servers, where CPUs do have a large impact on cost and performance, the CPU is not where the architect spends the bulk of their time. System-level interconnect, memory bandwidth, and I/O bandwidth tend to dominate.

To better reflect the challenges faced by actual architects, our introductory courses should broaden their coverage of architecture by treating system level issues and considering the architecture of embedded computing devices – not just traditional "computers." This will better serve both the students who plan to specialize in architecture and those for whom the introductory course will be their only exposure. For the specialists, a system-oriented course will expose them to the type of architecture they are more likely to be practicing. Very few people architect CPUs. Many people architect systems using CPUs – and other computing devices. For the non-specialists a system-oriented course will give them a better overview of computer architecture as a field than a narrow treatment of CPUs.

To make room for the systems content in an introductory course, much of the detailed treatment of CPUs must be dropped from such a course. Such material rightly belongs in an advanced course on CPU architecture – much as detailed treatment of interconnection networks is deferred for an advanced course.

Many of the problems faced by architects cannot be solved entirely within the domain of architecture. Digital design and compiler technology are critical to solving many architectural problems. Yet many architects are not proficient in these areas.

Choosing between alternative organizations typically requires estimating the delay, power, and area of memories, interconnect, and logic. Performing such estimates is remarkably easy. However, many computer architects do not have this skill. Instead they rely on a separate "design group" to give them estimates, or use "canned" programs that perform estimates for a particular structure (e.g. Cacti for caches). Neither of these solutions really works because the architect does not develop an intuition about the alternatives that comes from understanding how they work at the next level down – and hence cannot use this intuition to arrive at the "right" alternative – which is almost never one of the initial alternatives. For example, I am constantly astounded by the large number of practicing architects that do not have a good feel for the speed/power/area tradeoffs of memories and hence believe that DRAMs are inherently slower than SRAMs. Also, even though power is a critical issue, few architects know the energy required by a particular operation (add, read an 8K RAM, lookup in a 32-entry CAM, clock a word into a pipeline register, transfer a word 10mm across a chip). This makes it nearly impossible for them to accurately estimate the power of proposed architectures.

A detailed understanding of digital design takes years of experience; however, simple models of area, power, and delay can be taught in a week or two. The use of such models drastically changes the nature of microarchitecture exploration. No longer is the task to develop the system that increases performance at any expense (and without regard for impact on clock rate) but the task becomes one of achieving specified performance – including the impact on clock rate - while staying within area and power constraints. Such simple models of delay, area, and power should ideally be included in an introductory course.

Many problems faced by an architect are better solved at compile time than run time – or even by JIT compilers that are invoked at run-time. Statically scheduling a sequence of instructions is far more efficient (and results in a better schedule – if all the information is available) than scheduling them dynamically. Similarly, specializing a piece of code given the data type or value of a variable using partial evaluation is far more efficient than "prediction" of various types. In a system with many computing "elements" (some CPUs, others specialized), a compiler plays a key role in "mapping" the problem to the elements. The best solution to most architecture problems is usually a combination of compile-time software and run-time hardware.

Many (not all) architects, unfortunately, view the compiler as a given. They see the architecture problem as one of running existing binaries, or compilers as someone else's problem. This may be appropriate for a CPU architect tasked with developing the next generation x86, where they really do have to run old binaries. However, it is not an attitude that we should cultivate in our students. Such an attitude is extremely limiting, ruling out entire classes of solutions to problems.

By including a small amount of back-end compiler technology in an introductory architecture course – preferably with an exercise that illustrates the advantages of solving a problem with a combination of hardware and software – we enable these students to view compiler technology as another tool in their arsenal and open up a range of solutions not accessible to those who view a compiler as a black box.

Our architecture students would benefit greatly from a broader introduction to computer architecture – one that focuses on system (rather than CPU) architecture and considers a broad class of computing systems (not just traditional "computers"). At the same time, we need to enable our students by giving them a broad range of tools to apply to architecture problems. Two key tools are back-end compiler technology and simple models of delay, area, and power.

Visualizing the MMIX Superscalar Pipeline — Not Only for Teaching Purposes

Axel Böttcher

Munich University of Applied Sciences, Department of Computer Science/Mathematics
ab@cs.fhm.edu

Abstract

In this paper, we introduce an environment to visualize the internal activities of superscalar processors. The visualization environment is dedicated to the MMIX-processor. It uses its pipeline simulator but is implemented as a Plug-In to the Eclipse platform.

This environment helps to teach Computer Architecture on a less abstract level. It will be seen that a lot about hardware can be learnt by using a piece of software.

Additionally, this visualization environment would introduce a state of the art IDE and familiarize the students with it. This is a nice and important side effect for general educational purposes.

1 Introduction

Understanding the behavior of modern superscalar processors is becoming more and more difficult as their complexity increases. The ability of processors to simultaneously fetch several instructions and dispatch them to many parallel execution units, as well as elaborated branch prediction schemes and multilevel caches have an almost incomprehensible impact on the execution. Although the execution is nevertheless deterministic. This makes it somewhat difficult to teach these topics without being too abstract.

In this paper we present a visualization environment for the pipeline simulator of Donald Knuth's MMIX-processor. This virtual processor has mainly been designed for educational purposes and will be used as a reference in forthcoming issues of "The Art of Computer Programming". It is a state-of-the-art RISC machine providing many features implemented modern processors. During tha last few years, this processor has proven to be very useful for teaching undergraduate courses on computers and IT-Systems.

Donald Knuth provides a complete environment to write and simulate programs for this machine [1]. The summit is a simulator that simulates completely pipelined versions of the processor, clock cycle by clock cycle. This simulator is highly configurable and thus can be used to simulate realistic models of existing machines. Due to its unlimited configurability it is called Meta-MMIX, short mmmix. The simulator is a very valuable high complex, although well debugged program. Its output, however, is limited to a lot of textual information that can be generated to describe the machine's state during each clock cycle.

We developed a visualization environment to generate graphical representations based that textual information. This tool is used to demonstrate internal behavior of superscalar processors. It is not limited to demonstration purposes but can be used to support experiments and also for other courses than computer architecture. For example it could be used in a course on compilers to study the effects of register coloring or register spilling on program behavior and performance.

We will present this visualization environment and then briefly discuss some case studies of how it can be used and what can be expected to gain from it.

2 The Visualization Environment

During execution of a program, the pipeline simulator can display a large amount of information as text output. Depending on the information requested, this can amount to many kilobytes per clock cycle. So there is a need for a post-processor to get the most use out of that information. We used the open source Eclipse platform [2] to implement a visualization environment. Eclipse is a lean development platform written in Java. It can be extended by own contributions in an extremely flexible manner. A side effect of using this tool is that the students are familiarized with a state-of-the-art IDE.

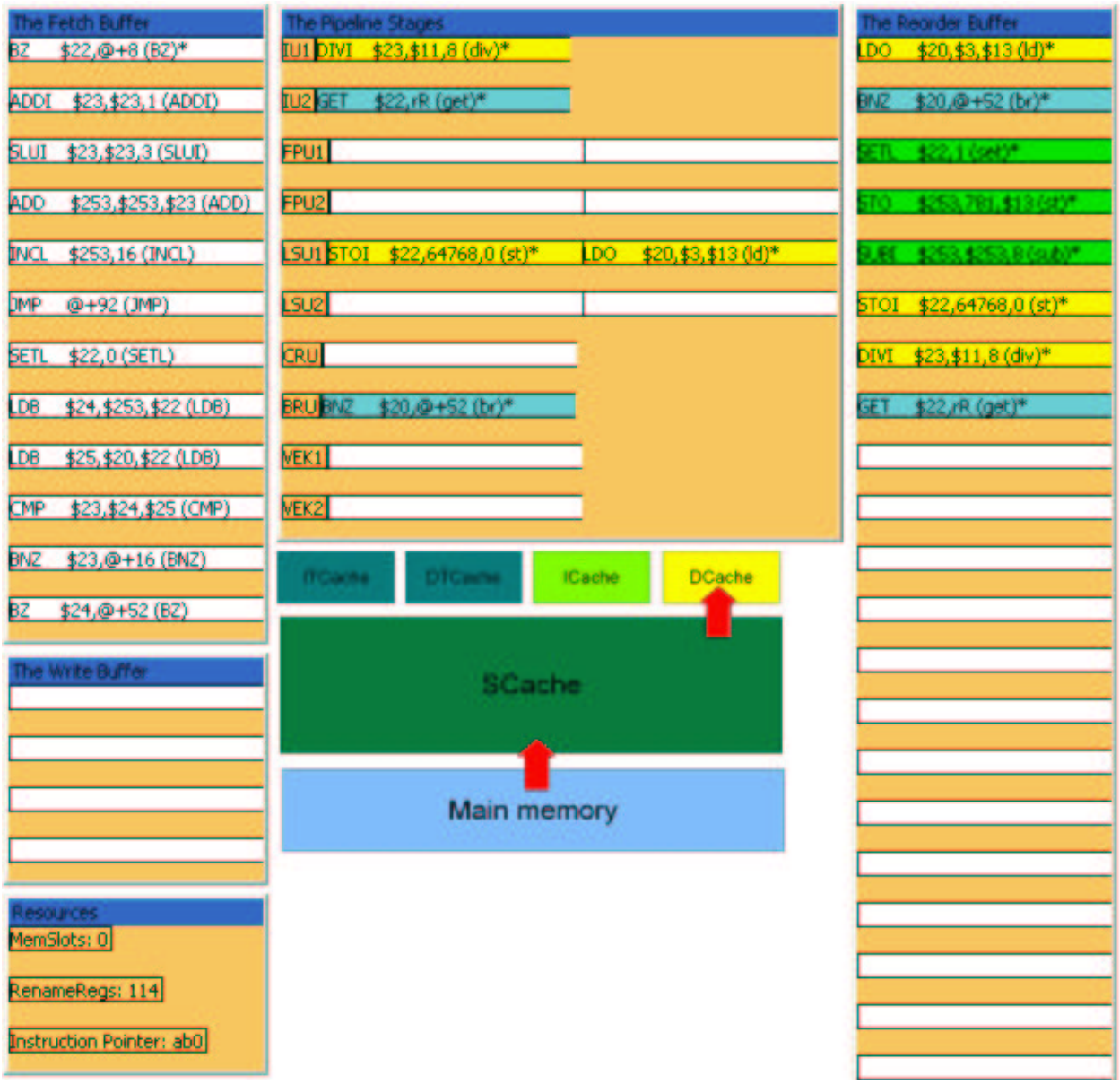


Figure 1: Visualization of the processor configuration during one single clock cycle. Instructions will proceed from left to right.

At the moment the pipeline visualization consists of two main views: First, an *overview* of the configuration – see Figure 1 – showing in detail from left to right:

- The fetch buffer content, i.e. those instructions that have already been fetched (loaded) but are not yet being executed.
- The execution units (titled *The Pipeline Stages* in Figure 1) with all instructions being currently executed. Instructions are taken from the fetch buffer in strict program order and scheduled (dispatched) to the units. However, the next instruction can only be scheduled when there is a unit available that is able to execute this type of instruction. Execution itself will start as soon as all the operands are available.

  Those units for loading/storing (LSU) and for floating point operations (FPU) are themselves pipelined [1]. Thus a new instruction can be scheduled as soon as the previous one has entered the second stage.
- The reorder buffer (ROB) containing all instructions that are waiting for operands, being executed, or having finished execution but have not yet been committed. Execution of instructions can end out of order, because – once on a unit – some may stall due to long execution times, or unavailability of operands e.g. during memory accesses – a so called Read-After-Write hazards.

  In Figure 1 we see three completed instructions in the ROB (number 3 to 5; those with green background), three in execution (the second, sixth and seventh; yellow background), and two instructions that are stalled (second and last; blue background). Instructions will be committed and thus are leaving the ROB in strict program order.
- The state of important resources like number of available rename registers, write buffer entries, or the program counter.
- Overview of memory activities indicated by arrows between the components of memory hierarchy. In the example we see that the D-Cache is filled from main memory through the S-cache.

Furthermore an *activity view* window gives an overview of the reorder buffer's content and memory activity versus time. Each pixel in the diagram of Figure 2 corresponds to one clock cycle. So the figure shows about 650 cycles (corresponding to 650ns when we assume a clock speed of 1GHz!). The bar graph in the upper half just represents the number of instructions in the ROB. Coloring is as above: blue for stalled instructions, green for finished instructions, and yellow for still executing instructions. Sometimes we observe some finished instructions in red color that will be discarded due to mispredicted branches.

The involved execution units can not yet be determined from this view; a double-click on the required cycle shows the details in the other window. The lower part of the activity view shows activity of the memory interface which is the main reason for stalls in the pipeline. We omit the details here concerning by which line can be concluded for what reason the memory interface is busy (e.g. filling of data/code/secondary caches or write back).

In more detail, Figure 2 shows the following main steps:

1. Although the pipeline is stalled due to a load operation, the memory interface is busy loading new instructions. This has just been started before the load was issued.
2. The requested data are supplied from the cache. The instructions just loaded are scheduled, some are committed and again new instructions have to be fetched from main memory.
3. A few new instructions arrive but the fetch of further instructions immediately stalls the pipeline again.
4. Some new instructions have been fetched. But same as above: new instructions are being fetched just before the issue of a load operation.

The visualization environment is implemented as a Plug-In to the Open Source development platform eclipse [2]. The programming language for this exercise is Java. This has several advantages: All facilities Eclipse is offering can be used, e.g. handling of dirty editors, project management, or progress indication.

The `mmmix`-simulator is run in a separate process and just its output is parsed, so there is no need to modify that program. An adapter class wraps the process and thus allows integration into a Model-View-Controller architecture.

3 Usage in Class

The most obvious possibility for usage of the environment are demonstrations done by the teacher. The students may speculate which effects changes

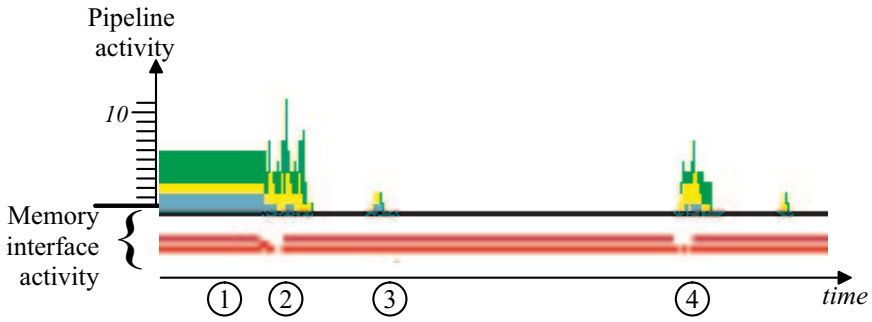


Figure 2: Overview of the activity on the execution units and on the memory interface. The memory interface shows (top to bottom) three types of activity in this example: filling of instruction cache, filling of secondary cache, and filling of data cache.

to the configuration or changes to the program will have during execution (most of the time also educated guesses come out to be wrong).

Secondly there are infinite possibilities for experiments that can be done by the students themselves: Trying to find all parameters of a real existing processor and letting them simulate `MMIX` using this set of parameters.

Furthermore the students can be asked to extend the visualization itself. Eclipse is a highly modular Java based system. We tried to follow the Model-View-Controller paradigm with the implementation of the Plug-In. As a framework we present interfaces where relevant information can be retrieved and thus own views to `MMIX`'s pipeline interna can be programmed by students. So we also have a basis for complex software engineering student projects.

Finally, scientific investigations considering the interworking of architectural parameters and software can be done [3]. The visualisation helps to get a quick overview of the system behavior.

4 Some Case Studies

4.1 Test configuration

We have configured the simulator to behave as far as possible like the PowerPC 970 processor [4]. Since there are no timing characteristics for access to main memory available, we assumed 20 cycles to address memory and to read/write data. Details can be found in table 1; please note that each of the parameters can be changed nearly arbitrarily.

parameter	value
ROB entries	120
max. ops dispatched/cycle	8
max. ops committed/cycle	5
Execution units	9 (see Fig. 1)
data-cache (L1)	64KB
instruction cache (L1)	64 KB
L2-Cache	512 KB
memory address time	20
memory read time	20
memory write time	20

Table 1: Used configuration values to adopt PowerPC970 [4].

4.2 A very simple example

As a very first and quite silly example we will show a drastic effect of a mispredicted branch which, however, we constructed artificially. The following code snippet shows a long running operation `FDIV` (floating point division – takes 40 clock cycles) and a branch depending on its outcome.

```
1          LOC    #100
2 Main     FDIV   $2,$1,$0
3          BNZ    $2,1F
4          ADD    $2,$2,1
5          ADD    $3,$3,1
6          ADD    $4,$3,1
7          ADD    $5,$3,1
8          ADD    $6,$3,1
...        ...    ..
```

The subsequent fast additions will overtake (except that one in line 4) So the instruction window (reorder buffer) gets filled with finished but uncommitted instructions. Finally the branch will execute and turns

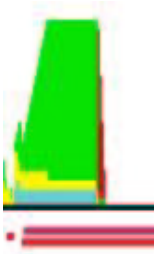


Figure 3: Effect of a mispredicted branch: many finished instructions have to be discarded in this particular case (red bars).

out to have been mispredicted. So all speculatively executed instructions have to be discarded from the reorder buffer. The activity view for this situation can be seen in Figure 3. In practical applications up to now we have not observed this extreme behavior.

4.3 Performance of Quicksort

To give a further rough overview of the possibilities offered, we take a closer look at the execution of a reference implementation of the quicksort algorithm. Figure 4 shows the pipeline activity for 500 cycles, each taken from three different sections. In part a.) the execution during a partionioning step of the array is shown with memory accesses to a cold data cache. Long stalls due to pending load instructions can be seen. Part b.) shows partinioning with memory access to warm cache. Thirdly, part c.) shows execution during the end game with insertion sort and also a warm cache. Each part has its own characteristics. Parts b.) and c.) show a steadily running pipeline, whereas in part a.) we have lots of instructions in the pipeline, most of them stalled. Here we see a starting point for further program optimization.

5 Limitations

Finally we have to comment on the limitations of the pipeline simulator:

- The number of pipeline stages is fixed. For better comparability with existing processors it would sometimes be helpful, to insert some additional stages (e.g. for register renaming).
- The memory/chipset interface is relatively inflexible. The only parameters to configure are a memory address time and read/write times.
- There are no reservation stations. Thus issue (schedule) of instructions to execution units is always in order and stalls as soon as no unit for the next instruction is available.

6 Conclusions and Further Work

Simple examples of `MMIX`-programs for a superscalar configuration have been demonstrated. Understanding the details of the execution on a clock cycle basis is quite a tricky task and all but straightforward. The visualization environment greatly supports the understanding and analysis. Trouble spots are easier to detect giving greater focus to investigations.

The visualization environment shall be extended to cover cache details and to display more information in the activity diagrams (e.g. types of instructions that are currently being executed) as well as the state of more resources.

From `mmix-plugin.sourceforge.net`, the plugin to visualize the `MMIX`-pipeline for the eclipse platform can be downloaded. Eclipse itself is located at `eclipse.org`.

7 Acknowledgements

The author wishes to thank Donald E. Knuth for the nice `MMIX`-processor and for having motivated this work, and Martin Ruckert for many helpful discussions

References

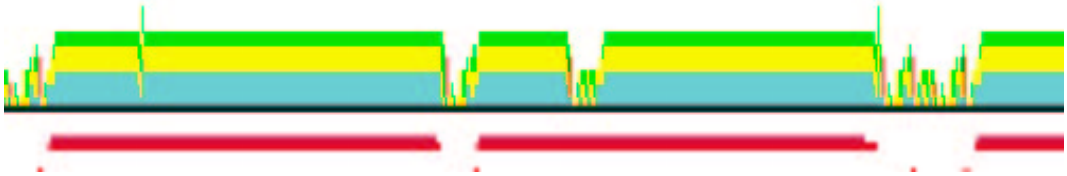
[1] D. E. Knuth, *MMIXware: A RISC Computer for the Third Millennium.* Berlin, Heidelberg: Springer-Verlag, 1 ed., 1999.

[2] E. Gamma and K. Beck, *Contributing to eclipse – Principles, Patterns, and Plug-Ins.* Addison-Wesley, 1 ed., 2004.

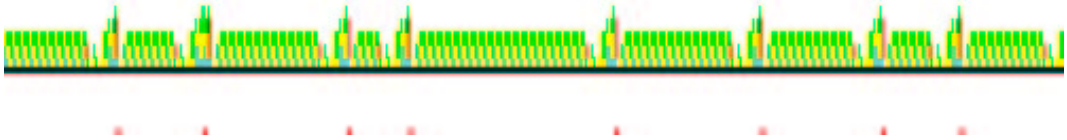
[3] A. Böttcher, "A visualization environment for superscalar machines," *Facta Universitatis (to appear)*, vol. 16, 2004.

[4] P. Sandon, "Powerpc 970: First in a new family of 64-bit high performance powerpc processors," *IBM technical note*, 2002.

a)



b)



c)

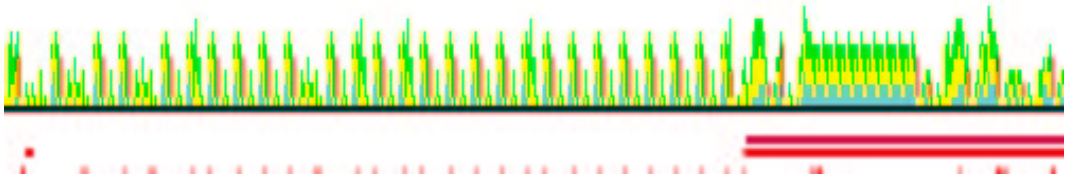


Figure 4: Experiments with quicksort: a) cold cache, b) warm cache c) insertion sort.

Bridges to computer architecture education

Peter Marwedel, Birgit Sirocic
Dept. of Computer Science,
University of Dortmund,
44221 Dortmund, Germany
{peter.marwedel,birgit.sirocic}@udo.edu

Abstract

Bridging the gap between the student's current knowledge and the technical details of computer systems is frequently required for the education of undergraduate students or students with a lack of previous technical knowledge. In this paper we will describe how to build bridges by the combination of introducing Flash-animations[1] and the educational units of the RaVi system[2]. In a first step the Flash-based animations make the students familiar with the underlaying principles. After that the student could jump to the more technical context by using the educational unit of RaVi. We have developed two bridges, one for explaining the principles of cache coherency protocols and the other for showing the concept of processor pipelines.

1 Introduction

Nowadays, many students are looking at computer science from an application perspective. Learning how to use the computer and to design new applications is very attractive. Learning why and how computers are functioning seems to be a lot less appealing for many students. Moreover, it is our experience that some kind of bridge has to be provided for students to learn how computers are working. It is not sufficient to start right away with technical terms and issues. There seem to be two reasons for this: missing prerequisites and lack of motivation. This problem is amplified in classes with students with very heterogeneous backgrounds.

In the context of this situation, it is unacceptable to just give up and accept that students just focus on the application perspective. A serious lack of knowledge of fundamental principles would be the result and students would not really be able to understand the principles of the equipment they are using in their professional career. This could very easily lead to misconceptions and damages.

So, the issue is: how to motivate the students to learn about the internals of computers and how to provide the bridges that are required for the students to enter areas such as computer architecture?

One of the approaches for removing barriers that prevent students from entering certain areas is to write very motivating books. However, this approach does not really fully utilize the opportunities that exist due to modern media. Modern media can provide a very motivating path from the student's current knowledge to what the student should know about operating principles of computer systems.

In this paper, we will describe how modern media can be used to motivate students to study the operating principles of modern computer systems and how to use modern media to bridge the gap between the student's current knowledge and the technical details of computer systems.

This paper is structured as follows: in section 2, we describe related work. In section 3, we present our "bridge" for understanding the MESI protocol for multiprocessor caches. In the following section, we show how processor pipelines can be introduced to an audience that has limited previous knowledge about processors and a limited motivation. A short description of the didactical background you will find in section 5. Experiences and conclusions for our future work are explained in section 6. The final section provides a summary of the current paper.

[1] We acknowledge the contribution of Alexandra Nolte who created the artwork of Flash-based animations with a great level of dedication and artistic professionalism.

[2] We gratefully acknowledge the funding of the RaVi-project (which is a subproject of the SIMBA-project) by the German ministry of research and development (BMBF). The RaVi system is a set of interactive training components for the teaching of the computer architecture. Each component visualizes an aspect of the dynamic behaviour that is found in computer architectures.

2 Related work

The fact that bridges need to be provided for the students to look at the details of computer systems with some level of dedication has already been realized by a number of educators.

For example, this view is dominating for the book ”Computer systems - A programmer’s perspective” by Bryant and Hallaron [1]. In order to motivate students to look at caches, for example, the topic is introduced using the effect of caches on the performance to different versions of the same algorithm. In this context, it is remarkable that the author’s University (Carnegie Mellon University) managed to significantly increase the percentage of female students (if compared to male students). Female students are said to focus more on applications of computers and less so on their internals. Unfortunately, the approach of Bryant et al. is totally restricted to the use of classical media, such as books.

One of the inherent limitations of books is their inability to visualize system dynamics. Moreover, they hardly provide any interactivity apart from adding marks to the pages. In order to remove these limitations, simulation of computer systems has been used. In order to be useful in a classroom, simulation-based educational units have to designed for this purpose. Otherwise, complex menus and high graphical resolution requirements would make it too difficult to understand the units quickly. Simulators in this class include HADES [4], Ptolemy II [3] and the RaVi simulator [7]. The focus of RaVi is on the visualization of complex system behavior which cannot be explained with the animation capabilities of Powerpoint and similar tools. RaVi includes educational units for visualizing the dynamics of microprogrammed MIPS-machines, pipelined MIPS-machines, dynamic out-of-order instruction scheduling techniques and cache protocols. According to our experience, RaVi meets its goal. There is a corresponding large number of downloads of RaVi every month and according to our statistics, RaVi is used throughout the world. An inherent limitation of RaVi is that RaVi does not really provide bridges that bridge the gap between the student’s knowledge and the details of computer architectures.

The goal of this paper is to describe how the two approaches just mentioned can be combined.

3 A bridge for the MESI multiprocessor cache coherency protocol

The first bridge described in this paper addresses problems in explaining the principles of cache coherency protocols with RaVi educational units. The units provide a detailed simulation of four caches and processors connected to one ”main” memory. According to our experience, this is already a too detailed view for many classes of attendees, including colleagues from other areas within computer science. We found it useful to avoid this problem by relating the technical details to real life. For the multiprocessor cache, we choose a banking system as an example. In this example, there is a bank that has a number of branches (which will later correspond to processors). Customer requests at the branches include obtaining statements from the bank, withdrawals, and deposits. Initially, local branches are not allowed to have local copies of the account information. Accordingly, each of the customer service requests results in sending a messenger to the central bank in order to check and update account information. Obviously, this approach is very slow and it makes sense to allow local copies of account information. With such local copies (later called caches) customers using only one branch are serviced in a short amount of time. For customers moving to different branches, messages still need to be exchanged. Invalidate and well as update messages are required. For each of the accounts, four different states can be distinguished: modified, shared, exclusive and invalid. The meaning of these four states can typically be understood even by the non-specialists. Problems which exist for multiprocessor caches can be introduced very easily. For example, the problem of quickly invalidating local copies of account information in the local branches in order to avoid people getting too much money is very obvious. Also, the problem of identifying the branch that has the up-to-date account information while being in the exclusive state can also be demonstrated easily.

This example has been implemented in a Flash-based animation. The level of interactivity in these animations is sufficient for the ”bridges”. It would be less appropriate for the MESI educational units, as the behavior of the protocol is quite complex and difficult to implement in Flash. Fig. 1 shows a screenshot of the corresponding Flash animation.

In the actual presentation, the correspondence between the central bank and the main memory as well as the correspondence between the local branches and the local processor/cache systems are explained next. This is quite easy to understand even for the non-specialist.

4 A bridge for processor pipelines

For processor pipelines, it is possible to use a similar case from real life. The corresponding Flash-animation is based on a production pipeline and uses pipe stages similar to those that are found in MIPS pipelines. The first stage is the dispatch stage. Instructions for fabri-

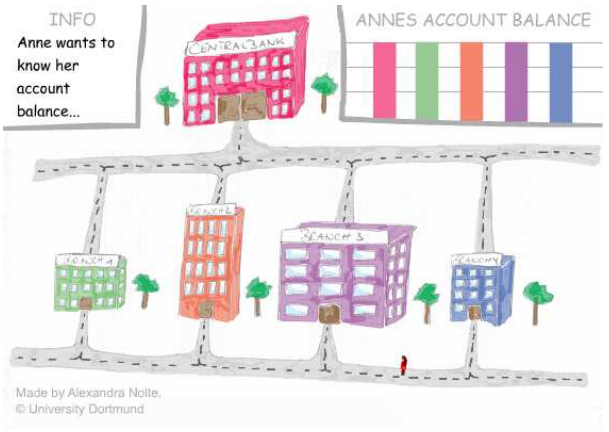


Figure 1. Flash-animation for the cache coherency protocols (segment of a screenshot)

cating hammers are issued in this stage. Components required for this fabrication are fetched in the second stage. The third stage is the execution stage. This is where the hammers are actually made. The fourth stage is idle as long as only hammers are made. The fifth stage is the storage stage. Hammers are stored in this stage. It is important to show that these stages operate concurrently. Fig. 2 shows a screenshot of this Flash-animation.

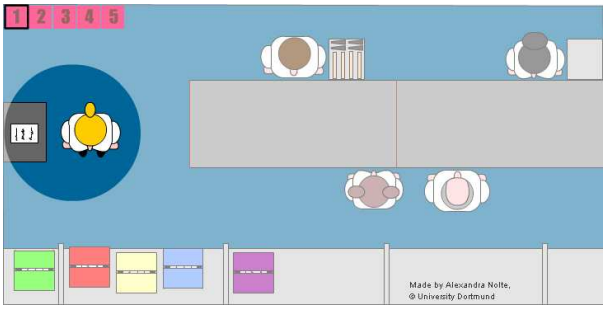


Figure 2. Flash-animation for processor pipelines (segment of a screenshot)

In the actual presentation, the correspondence between the pipe stages and MIPS pipe stages are explained next. Again, it is quite easy to understand even for the non-specialist.

5 Didactical Backgound

The underlaying didactical method which we are realising by the Flash-based animations, is a kind of anchored learning [2]. The method of the anchored learning is based on the idea, that the learner can find a link from his already existing understanding of the real-world to the new information and concepts by providing them a so called anchor. The learners who find the link, can construct their own new knowledge base and the understanding of the new main principles will be easier.

Our Flash-animations are providing a kind of an anchor by highlighting the analogy of the every-day example and the technical example. That means, the students who know the main principle of a production pipeline can find the link to the main principle of the processor pipelines. For those students who do not know how a production pipeline works, the anchor does not archive its aim.

A problem for implementing the anchored learning is that an adequate anchor has to be provided. That means, the analogy of both examples have to match every aspect of the examples. There is a high risk for misunderstandings or missing a key point, if one or more aspects do not match.

6 Experiences and future work

Both bridges described above have been used in presentations. The pipeline was used in a presentation to sophomore students having no previous knowledge of computer architecture. Feedback was limited, but it is interesting to note that without this ”bridge” we would not even have dared to discuss such technical details in this audience. A second presentation was done at Seoul National University. The audience was quite mixed, including colleagues, graduate and undergraduate students. A first result was the observation that the Flash animations helped to maintained a good level of interest in the audience, despite the heterogeneity of it. A second observation was that switching between Flash-, Powerpoint- and RaVi-animations was challenging for the presenter.

This problem was avoided in a follow-up presentation at KAIST in Taejon, Korea. RaVi- as well as Flash-animations were linked to Powerpoint slides using the macro facility of Powerpoint. This improved the situation for the presenter significantly, as he could better focus on his speech and did not have to care about switching between applications. An interesting result is the proposal to have a Korean version of the MESI-bridge. Unfortunately, there is only very poor support for generating localized versions of Flash-animations.

We believe that the approach described in this paper has been very successful so far. Next, we will be working on a similar approach for embedded system design. The

topics to be covered are presented in our recently published book [5]. Slides corresponding to that book are also available for download [6]. We observed that the unavailability of tools for the visualization of the dynamic behavior of embedded systems is a serious bottleneck. We have therefore decided to extend the approach described above to embedded systems. Candidate topics include models of computation, real-time scheduling and communication protocols.

There are still some limitations of the approach described in this paper. As can be seen from the above examples, currently available technologies for dynamic visualization are far from optimal. The separation between simulation-based visualization and Flash is certainly not ideal. Powerpoint-macros are frequently disabled due to security issues. Unfortunately, none of the two tools can actually replace the other.

7 Summary

In this paper, we have described an approach for bridging the gap between the knowledge and motivation of undergraduate students and the details that must be discussed when the internal operations of computer are explained. It is based on using an intuitive entry into the world of computer architectures. Flash animations turned out to be sufficient for this purpose. These animations were appreciated by the audience, but it must be kept in mind that they are just bridges. The area reached via those bridges must also be presented, preferably using animation and interactivity. Currently, a combination of Powerpoint and simulation-based presentation is used. The combination of presentation techniques provided the required bridges to computer architecture to a very heterogeneous audience, including freshman and software-oriented people. However, it would be nice to have a portable and serviceable tighter integration between the different presentation techniques.

References

[1] R. Bryant, E. Randal, and O'Hallaron. *Computer Systems A Programmer's Perspective.* Prentice-Hall, 2003.

[2] Cognition and Technology Group at Vanderbilt (CTGV). Anchored instruction and its relationship to situated cognition. *Educational Researcher, 19 (6)*, pages 2–10, 1990.

[3] U. B. EECS. The Ptolemy II homepage. *http://ptolemy.eecs.berkeley.edu*, 2004.

[4] N. Hendrich. A Java-based framework for simulation and teaching. *Proceedings of the 3rd European Workshop on Microelectronics Education*, pages 285–288, 2000.

[5] P. Marwedel. *Embedded System Design.* Kluwer Academic Publishers, 2003.

[6] P. Marwedel. The homepage for the book Embedded System Design. *http://ls12-www.cs.uni-dortmund.de/ marwedel/kluwer-es-book/*, 2004.

[7] P. Marwedel and B. Sirocic. Multimedia components for the visualization of bynamic behavior in computer architectures. *23rd Design Automation Conference*, pages 378–384, 2003.

Improving Instruction Set Architecture Learning Results

José M. Claver, María I. Castillo, Rafael Mayo
Dept. of Computer Science and Engineering
University Jaume I
12080 Castellón (Spain)
E-mail: {claver|castillo|mayo}@uji.es

Abstract

In this article, we put forward a new methodology and strategy for teaching the Instruction Set Architecture in a "Computer Organization" unit. This unit belongs to the second year of the undergraduate program in Computer Science at our University. In particular, we have centered our effort on the development of laboratory sessions, focused on an adequate introduction of assembler language of a general purpose processor. Our methodology has taken into account, among other aspects, the choosing of processor and tools, the pace with which concepts are introduced and the responsibility of learning. Thus, we obtain a less traumatic approach for our students to one of the most important subjects in the background of our future computer engineers and scientists.

Results obtained show students response is positive. This effect is reflected in student's interest and the ease with which they are been able to solve the exercises we set them to do. Because of that, an improvement on learning in this subject and this aspect is reflected in the subsequent evaluation of students.

1. Introduction

Generally, Computer Architecture units taught in Computer and Computer Engineering undergraduate programs begin during the first year, and are followed in subsequent years, with varying intensity and depth depending specialization. The contents of these units include many aspects that are fundamental for the training of future computer scientists and engineers. It is lively that these aspects will retain their importance in the syllabus for the next 10 to 20 years [2], since the development of applications and their performance requirement have an important impact on the architecture, structure and organization of computers [9, 10, 12]. From another point of view, we do not foresee over the next years any important modifications to the way computers work and are constructed. Nowadays, in the architecture of current processors we are using some ideas that appeared more than 30 years ago. However, we are expecting new developments that will make possible very different approaches to the widely used computational models, like quantum and molecular computing [3].

There is a generalized consensus about the contents that must be taught in the first computer architecture courses [1, 2], but there are different alternatives with respect to the kind of processors and tools used for describing and studying how they work [5]. In this sense, it is particularly interesting to study processor organization, instruction set architecture (mainly assembler programming), memory use and input/output control. These contents are, for this reason, the core of first units on Computer

Architecture. Nevertheless, the high sophistication and configuration variety of current computers means that these must be studied in varying depth according to the final course goals [6]. An important complement of computer architecture contents is the study of the technological aspects of computer design, but we do not consider these aspects in our article.

As we will see in this paper, the choice of processor and the tools used to teach this subject are very important, but the pace of the course and responsibility have also a decisive influence on the learning of the concepts and techniques we want our students to learn. These two elements determine whether or not students can achieve the planned goals. If care is not taken to adequately adjust the pace of the course and encourage a responsible attitude on the part of the students, many of them will lose the thread of studies and finally drop out.

The rest of this paper is organized as follows. In section 2, we review some more outstanding aspects related to teaching in introductory Computer Architecture subjects. In section 3, we compare the experience of teaching this subject at other Universities. In section 4, we show the more important aspects of our teaching methodology. In the last section, we explain the principal conclusions derived from our experience.

2. Introductory teaching in Computer Architecture

Introductory Computer Architecture units include, among other digital aspects, the study of following lecture topics:

- Processor: Structure and organization, data path, instruction set and machine language, assembler language and their relation to high level languages.
- Memory: Organization and management, instructions and data storing, and cache.
- Input/Output: Asynchronous control by status register and exceptions, and protocols.
- Performance: Analysis and comparatives.

These contents must be transmitted to students both in theoretical lectures in the classroom and in laboratory work. In this way, they can acquire the knowledge and abilities set out in the unit syllabus.

The most important conceptual aspects must be presented in the lectures and then extended and elaborated in laboratory work where a practical application of these concepts are given. In this context, the pace and the order of laboratory work must be adjusted to theoretical lectures. Furthermore, in laboratory work students acquire the ability to perform analysis and synthesis. For that, it is often to enhance assembler language study (exoarchitecture) by using a real machine or simulator of an architecture strongly related to the model explained in previous theoretical lectures (associated to endoarchitecture and microarchitecture [7]), preferably the same.

There is a time in the design of an academic project in which we must adopt, among various alternatives, an example processor and a tool to teach this subject.

2.1. Processor

It seems clear that it is preferable to choose a real processor rather than a hypothetical one as this will mean that it can be used as a part of a real device such a commercial computer or a development system designed for laboratory. It is impossible to do this with a hypothetical processor. Nevertheless, real processors introduce some special features that cannot be extended to other processors, since its design is the result of practical and economic considerations. These special

features can introduce, during the first years, additional complexities in the learning process. Furthermore, the perspective acquired by students may be not adjusted to more extended processor architecture. So, it is important to choose an suitable processor.

On the other hand, hypothetical processors can be better adapted to academic needs at each moment, in function of student knowledge level. Thus, it is easier for them to assimilate and apply the concepts and techniques involved. These processors are often designed by choosing different abstraction levels of some more extended general-purpose processors.

2.2. Tools

The tool chosen to teach this subject may be a commercial computer, a development system, or a simulator. A commercial computer can be directly used if a real processor is chosen. In this case, the assembler and software development used must be adapted to the characteristics of a particular processor, as well as its possibilities and restrictions. In this situation, processor analysis is indirect and limited by the organization of a particular machine (memory, cache, bus, etc.).

When a simulator is used, it can be designed for a real or hypothetical processor (original or as abstraction of a real processor). Simulators of hypothetical processors could be better adapted to the needs of a particular unit. Furthermore, complexity can be increased in subsequent courses by extending and modifying initial abstractions. Thus, it is possible to introduce more advanced concepts one at a time without students needing to learn about new tools or assembler languages.

In both the above cases, it is possible to use computer resources (directly or no), although these resources are always more limited if we use simulators.

3. Experience in other Universities

In recent years, teaching experience on introductory computer architecture subjects has featured by following characteristics:

- Choose, for the first year, a simple hypothetical processor instead of a real 8 bit processor (i8085, Z80, MC6800, R6500, etc.). The main goal is to reduce the gap between student knowledge and introductory concepts in the first years of computer architecture.
- In the second year, it is typical to opt for one of the well-known real 16 bit CISC processors produced in the 80's (Intel i8086, Motorola MC68000, etc.). In some cases, a 32 bit RISC processor is selected (MIPS R2000, ARM, MC88010, etc.), in order to reach a better approach to the complexity of current commercial processors [5].
- Simulators, alone or combined with development systems and market processors, are used to show the relationship between assembly/machine languages and architecture, to appreciate the challenge of producing efficient and correct programs, and to develop applications with real hardware.

4. Our proposal

In the first year of our Computer Science program, the "Introduction to Computers" unit presents elementary concepts about how computers work, and the basic switching logic. In the second year, "Computer Organization" course presents a more detailed study of computer architecture and organization. Important parts of this unit are the instruction set architecture and the assembler programming.

Until last year, the processor we were using (since 1992) to teach "Computer Organization" was the Motorola MC68000, one of the most elegant exponents of CISC architecture. There are a lot of reasons why MC68000 is the most used processor to teach Computer Architecture (its streamlined architecture combined a powerful instruction set with moderately easy-to-learn assembly language). Nevertheless, its architecture is far from the seminal processors of current superscalar architectures.

In laboratory classes, students used a MC68000 based development system with a complex environment which cannot be used outside laboratory. Each laboratory session was organized as follows: it began with a lecture by the teacher, in which many new concepts were presented and the goal of each session was fixed. These goals were well specified, but very ambitious, and close attention by students was required. Furthermore, before students began to work, they need a meticulous study of its paper description, in which was included an abstract of the principal concepts explained by teacher. These two tasks take up the greater part of the laboratory sessions. Because of this, students had little time to develop the proposed exercises.

Laboratory sessions were designed in increasing order of complexity, as students were to analyse and design assembler programs, of variable complexity, from the first session. These programs included data declaration, different kinds of instructions, and a great variety of addressing modes. This situation is habitual in the assembler language teaching of analyzed universities.

All the above circumstances encourage us to plan an alternative in order to make it easier and more comfortable for our students to reach the goals we set them in these laboratory sessions. We base this change on the following initial goals:

1. Simple processor architecture.
2. Abstraction of a more advanced real processor.
3. Simulator easy to use.
4. Laboratory work can be continued at home.
5. Self-learning.
6. No need to attend with fixed timetable.
7. Personalized rate of learning (asynchronous learning)

In order to obtain the two first conditions we opted for a non-segmented abstraction of the MIPS R2000 processor, which has easier structure and is easier to program than a real R2000 processor [12]. This is a RISC processor; therefore, it has a simple instruction set and reduced addressing modes. Furthermore, in more advanced units we can use the same or similar processor (as the hypothetical, but realistic, DLX processor), without current abstractions [8, 10].

Conditions 3 and 4 are obtained by using the SPIM simulator. Concretely its graphic version called XSPIM, developed by James R. Larus from the Wisconsin University, which works under Linux and DOS/Windows operating systems [11]. This is an integrated simulator, where all information about processor and memory can be shown, and it is easy to use. The latest version (from the 6.3) of this simulator can show original R2000 programming difficulties, activating delayed load and delayed branch functionalities. Furthermore, as this simulator is a freeware software and multiplatform, students can use it at home.

The last three conditions are reached thanks to the planning and development of an adequate laboratory textbook. Each part of this textbook is associated with one or several laboratory sessions and is self-contained. In each chapter new concepts are presented, the more advanced the unit, the more complex they become and it is supposed that only previous session concepts

are known. Figure 1 shows a concentric vision of contents corresponding to laboratory sessions.

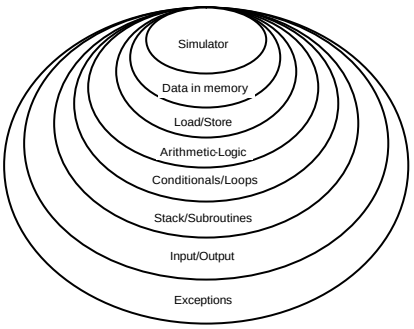


Figure 1. Contents of corresponding laboratory sessions.

We have taken special care with students learning rate when designing the contents of each session. We don't forget the maxim that says: first analyze and after synthesize. For that reason, all sessions begin with a little introduction and several example programs that students must analyze and understand the behavior of. In the following step, we propose some changes to previous example programs that students must analyze at another time. Thus, students increase their participation and make some simple guided synthesis. Finally, synthesis problems, such as short development projects, are proposed in order to test the correct understanding of the techniques and concepts introduced. In the next chapter/session, students begin with other example programs that they must analyze, and follow the same steps again. These steps are shown in Figure 2.

Thus, laboratory work is structured as a consecutive set of questions which require from students: to analyze an example of assembler program, to modify them, and complete a design based on the concepts and techniques they have learned (see Figure 3 for an example of this structure work). A textbook containing laboratory work is also electronically obtainable from the unit website [4].

When students have doubts about some concept or technique, they only have to review earlier sessions. Thus, there is no need for a teacher to be near the student at all times, and students work, systematically, solving questions and learning actively, because there is no obligatory attendance at laboratory sessions. Logically, students work at their own learning pace. Students needing more time on a particular session know that they have to work outside laboratory programmed sessions, either in free time access or at home, to finish all laboratory sessions. Only in this way, do students have any guarantee of passing the evaluation of assembler programming.

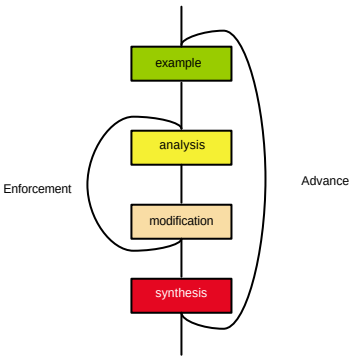


Figure 2. Learning flow of laboratory sessions.

Since simulator use is not an end goal of this laboratory, students don't have to show their ability to use it in the evaluation tests. Otherwise, in the laboratory they analyze, modify and design little programs by using concepts and techniques shown in theoretical lectures, and these subjects are the core of evaluation tests.

Figure 4 shows the number of students who have passed (results great than or equal to 5) and the overall of results obtained at course 00/01, in which the new methodology is introduced, have increased respect to early years (98/99 and 99/00). Tendency lines of last two years highlight this behaviour.

The number of students attending exams rose 20%, as dropouts are reduced (these are greater in the first year this unit is coursed). However, these are very poor test results (between 0 and 3), which constitute at least 16% of students presented. However, this percentage is less than 23% compared to the course 99/00.

Example	Edit a file with the following code: … Description: …
Analysis	**Question 1.** What code does …? ¿What is the value of...? **Question 2.** Show which instructions do...? **Question 3**. If datum ... has a 5 what happens...?
Modification	**Question 4.** Modify the code to ....
Synthesis	**Question 5.** Develop a program ....

Figure 3. General structure of laboratory work.

5. Concluding Remarks

Our new methodology for the laboratory has shown that it is good to choose an RISC processor abstraction and a simulator for introductory computer architecture units. Although, it is also important that students follow their own pace and are actively responsible in this process. Sometimes it does not happen because students don't know how to do it. Our textbook [4] is an auxiliary tool to guide them in this way. Furthermore, in this new laboratory methodology, students advance more quickly, and see more contents in this unit, as input/output control and treatment of exceptions.

This methodology, as well as our textbook, has been adopted as a guide to other computer science programs at our University and at other Spanish universities. Evaluation results show that this is a good methodology improving instruction set architecture and assembler programming learning.

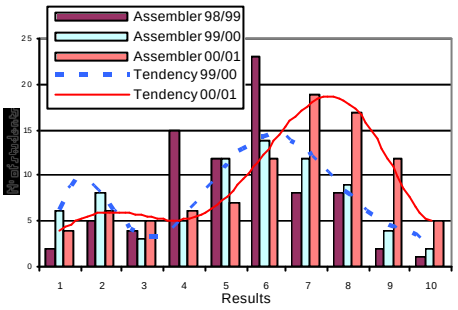


Figure 4. Histogram of the last three years assembler evaluation results.

References

[1] ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 1991*. http://computer.or/education/cc1991.

[2] ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 2001*. http://computer.or/education/cc2001.

[3] A. Barenco, A. Ekert, A. Sanpera, C. Machiavello. *Un saut d'échelle pour les calculateurs*. La Recherche, Nov 1996, http://www.qubit.org/intros/comp/comp.htm

[4] M.I. Castillo, J.M. Claver. *Prácticas guiadas para el Ensamblador del MIPS*

R2000 (in spanish). Dep. of Computer Science and Engineering. University Jaume I Editions. 2001, http://yan.act.uji.es/E38.
[5] A. Clements. *Selecting a Processor for Teaching Computer Architecture*. Microprocessors and Microsystems, may 1999.
[6] A. Clements. *The Undergraduate Curriculum in Computer Architecture*. IEEE Micro, pp. 13-22, may - june 2000.
[7] S. Dasgupta. *Computer Architecture - A Modern Synthesis*. John Wiley & Sons, 1989.
[8] E. Farquhar, P.J. Bunce. *The MIPS Progra-mmer's Handbook*. Morgan Kaufmann, 1993.
[9] J.P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, 1998.
[10] J.L. Hennessy, D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3th edition, 2003.
[11] J.R. Larus. *MIPS. A R2000/3000 Simulator.* University of Wisconsin, http://www.cs.wisc.edu/~larus/spim.html.
[12] D.A. Patterson, J.L. Hennessy. *Computer Organization and Design. The Hardware/ Software Interface*. Morgan Kaufmann, 2nd edition, 1997.

Integrating Research and e-learning in Advanced Computer Architecture Courses

Mouna Nakkar
Dr. Mouna Nakkar is with the University Of Sharjah

ABSTRACT

This paper presents novel methods in teaching advanced computer architecture courses. These methods include presenting fundamental computer architecture issues using e-learning; employing visual aids to teach fundamentals concepts like Caches, pipelining and scheduling. In addition, this paper discuss integrating research into the course is beneficial to the students pursing a PhD career.

1. INTRODUCTION

Advanced Computer Architecture courses are usually taught in the senior year of undergraduate curriculum or first year graduate curriculum. This is a fundamental course for microprocessor designers and computer architectures. Hence, establishing a good understanding for this course is a must for graduating engineer.

The concepts taught in this courses includes: measuring performance, Instruction Set Design, Memory Hierarchy and Caches, Pipelining and its Hazards, Instruction Level Parallelism, I/O storage, and latest contemporary computer architecture issues. The course usually combines the software and hardware approaches that increase performance of the microprocessor design. The course introduces these concepts and presents quantitative approaches to measure the feasibility of these approaches on performance emphasizing on the differences between hardware and software approaches.

There are several computer architecture books [1], [2], and [3] available to the Computer Architecture Instructor. However, only reference [1] gives a quantitative approach to computer architecture concepts. Also, Hennessy & Patterson's gives a comprehensive documentation on most of computer architecture topics.

This paper will first discuss some computer architecture concept set for e-learning like Cache Associativity, superscalar microprocessors, and dynamic scheduling algorithms. Then discuss integrating research topics in the course will be presented.

2. CACHE SET ASSOCIATIVTY

This section proposes changing the presentation of the set associativity concept. The concept of cache set associativity is presented in [1] could be presented in a better way. This paper simplifies the presentation of associativity concept to make it better for students to visualize. Figure 1 & 2 show the set associativity explained according to [1]. This approach presents the cache to be split into number of sets and each set has equal number of lines. For example, a 2-way set associative cache having 8 lines will have 4 sets and each set has two lines.

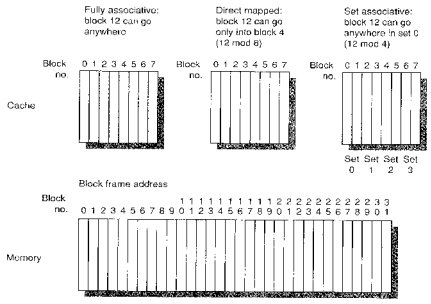


Fig. 1. Hennessy & Patterson's Memory Diagram [1].

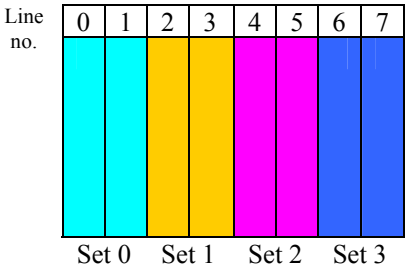


Fig. 2. Hennessy & Patterson's 2-way set associative cache.

This paper proposes set associativity to be dividing the cache into n sets; each set has number of lines. For example, the same 2-way set associative cache with 8 lines will be divided into two sets and each set has four lines. Figure 3 shows a 2-way associative cache with four lines in each set. The hardware will be the same whether the concepts is explained according to [1] or this approach, but this novel way is better to visualize the concept of associaitivity.

3. COMPLICATED CONCEPTS MADE EASY USING VISUAL AIDS

Computers and microprocessors are rich with concepts that seem complicated for new students. These concepts are easily explained with visual aids. Concepts like Pipelining and its hazards, Superscalar design, Instruction Level Parallelism, and Dynamic Scheduling. This section shows these concepts explained using visual aids like PowerPoint animations.

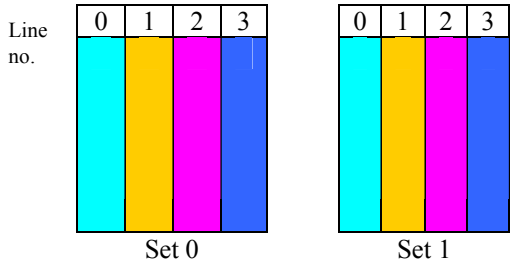


Fig. 3. Novel approach for explaining the concept of cache associativity.

3.1. Pipelining and Hazards

The concept of pipelining is simple to understand, but pipelining hazards can get complicated. Figure 4 shows a slide of a PowerPoint presentation showing each pipeline stage with respect to its instruction. This if for a DLX processor of five pipeline stages (Instruction Fetch, Instruction Decode, Instruction Execute, Instruction Memory write, and Instruction Write Back). In PowerPoint presentations, each instruction will appear at a given time such as shown in Figure 5. Figure 6 will be display the next instruction to be issued in the pipeline stage and so forth.

The pipeline could be also shown in terms of cycles, meaning display the events at each clock cycle as shown in Figure 5 for instruction issued.

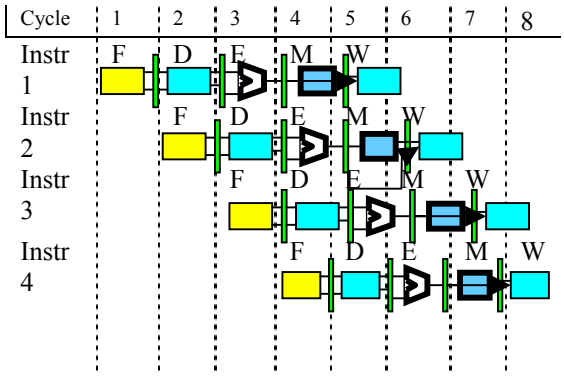


Fig. 4. DLX pipeline stage.

For pipeline hazards, the visual aid could show bubbles inserted in the pipeline. Figure 7 shows bubbles and data forwarded using arrows.

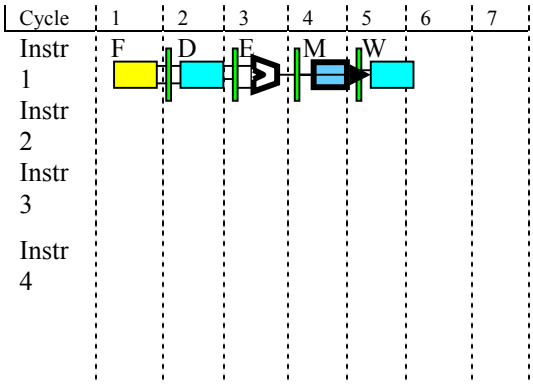


Fig. 5. DLX pipelining the first stage.

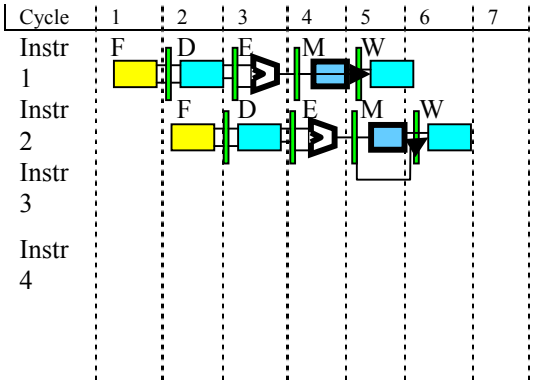


Fig. 6. DLX pipelining the second instruction.

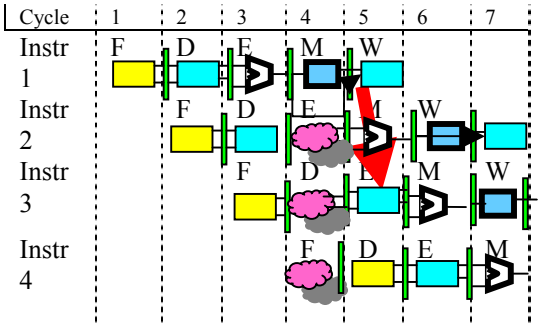


Fig. 7. DLX pipeline stage.

3.2. Superscalar and multi-issue machines

The concept of superscalars can also be explained with the visual aids. For example, Figure 8 shows a 2-way issues for a DLX supersacalar machine where one pipeline is assigned for integer and the other for floating-point operations. Note that floating-point operation takes 3 cycles to execute. Again this could

be presented to the students as a motion animation where two instructions are issued at a given time.

Instruction type	Pipeline Stages						
Integer	F	D	E	M	W		
FP	F	D	E	E	E	M	W
Integer		F	D	E	M	W	
FP		F	D	E	E	E	M
Integer			F	D	E	M	W
FP			F	D	E	E	E

Fig. 8. DLX Superscalar issue of Integer and FP pipeline.

3.2. Instruction Level Parallelism

Instruction Level Parallelism and Dynamic Scheduling is made easy with visual aids. For example, Tomasula's algorithm for dynamic scheduling can be easily understood using animations. Figure 9 shows the window of Dynamic scheduling at cycle =0 for five instructions scheduled on this window.

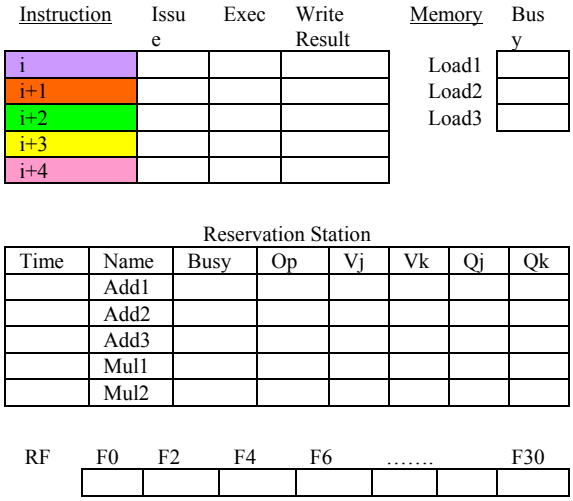


Fig. 9. Tomasula's algorithm

The engineering student can then fill in the blank in each cycle and the given time for it. For example the student can be writing each cycle result as the lecture taking place. Note that these examples are obtained mostly form [1]. However, the idea to involve the student in the process of learning and solving the problem as it been presented for the very first time is novel.

4. INTEGRATING RESEARCH TOPICS

Advanced Computer Architecture is rich with new topics that are in the research stage. The student must be aware of these topics before completing any advanced computer architecture course. This could be integrated in the course project where the students are asked to provide a quantitative measure for these new topics, or it could be in the form of exercise and small end of the course homework. The effect of this research on there understanding is tremendous.

The best types of research papers to provide for the students are papers that provide the original concept. Also, paper that compares different microprocessors and their implementations [4] [5] would be good challenge to the students. However, for advanced courses and computer architects the best approach is to challenge the students with advanced topics [6]. This will give more bases for creativity when pursing their engineering career.

5. CONCLUDING REMARKS

Advanced Computer Architecture is rich with advanced topics. Some of the universities nowadays offer two levels of Computer Architecture courses for graduate level engineers. This paper offers a better ways to present some of the typical concepts of computer architecture such as pipelining, pipelining hazards, cache associativity, and Instruction Level Parallelism, and Dynamic Scheduling. There are more creative ways to present computer architecture concepts that are not presented here.

The most advanced way of learning is through visual aids and e-learning. Future trends in teaching Computer Architecture may lead to e-learning at a distance. This could be explored in future papers.

6. REFERENCES

[1] J.L. Hennessy & D.A. Patterson, "Computer Architecture A Quantitative Approach 3rd edition," Morgan Kaufmann Publishers, USA, 2003.

[2] Barry Wilkinson, "Computer Architecture Design and Performance 2nd edition," Prentice Hall Europe, 1996.

[3] M. Morris Mano, "Computer System Architecture 3rd edition," Prentice Hall, Inc. USA, 1993.

[4] Bruce Jacob and Trevor Mudge, "Virtual Memory in Contemporary Microprocessors," IEEE Micro, vol. 18, no 4, pp. 60-75, July 1998.

[5] M. Mittal, A. Peleg and U. Weiser "MMX technology Architecture Overview," Intel Technology Journal, pp. 1-12, Q3, 1997.

[6] Heald et. all, "Implementation of the third generation SPARC V9 64-b microprocessor," ISSCC Digest of Technical Papers, pp 412-413, 2000.

Bridging Undergraduate Learning and Research in Software and Hardware

Liang Cheng[1] and **Dale Parson**[2]
[1]Laboratory Of Networking Group (LONGLAB, http://long.cse.lehigh.edu)
Department of Computer Science and Engineering, Lehigh University
19 Memorial Drive West, Bethlehem, PA 18015, USA
cheng@cse.lehigh.edu
[2]Agere Systems, Inc.
1110 American Parkway NE
Allentown, PA 18109, USA

Abstract

Embedded processing, where computers are used to monitor and control dedicated hardware, is a growing presence within mainstream computer science and engineering. Network processing, where embedded processors monitor and control communication networks, is a premier example of embedded processing. This paper presents contents of a *Network Systems Design* course used to introduce undergraduate students to understanding software-hardware co-design concepts and acquiring practical experience in embedded processing. The achieved goals of the course include: (*i*) carrying out lab-based introduction to embedded processing in its application area of network processing, and (*ii*) strengthening ties between academic study of network processing and industrial practice in the field, given the fact that most advances in network processor architectures to date have been made in industry. Responses from students approved our intention of the "hands-on" lab-based introduction using a modular network processing laboratory and verified the effectiveness of integrating academic study with industrial experience.

1 Introduction

Embedded processing, where computers are used to monitor and control dedicated hardware, is a growing presence within mainstream computer science and engineering [1-3]. Network processing, where embedded processors monitor and control communication networks, is a premier example of embedded processing.

There is a growing need for undergraduate students to understand software-hardware co-design concepts and to acquire practical experience in embedded processing [4]. Network processing and network processor architecture provide an ideal context to teach software-hardware co-design at the advanced undergraduate level in computer science and engineering. In fact, network processor architecture is undergoing rapid evolution, making it a dynamic area for observation and contribution. The network systems in which network processors are deployed are also growing and evolving. These systems include substantial hardware and software components.

This paper describes an advanced undergraduate course that was designed and developed about network processing, integrated with a modular network processing laboratory, to bridge undergraduate learning and research in both software and hardware. Over the last several years there have been a number of graduate-level courses developed on network processors (e.g., [5]). We have adapted the graduate-level courses on network processors into an undergraduate-level course on network processing and processors. The course materials, including course notes and laboratory exercises have been developed and are freely available on the Internet to academic institutions teaching similar software-hardware co-design courses. A researcher from industry has co-taught the course, which adds valuable industrial experience in these fields to the course.

The modular network processing laboratory has been designed and utilized to teach undergraduate students in a "hands-on" manner the operation of a network processor as well as elements of network devices. There exist a number of papers that are useful references for designing lab sessions of this course. For example, [6-8] have discussed education of networking concepts via hands-on experiments or practical experience. We have observed that their course design can be improved by offering *safety-net* characteristics and industrial experience components. Safety-net means that students who fail to complete a particular assignment are still able to move forward to the next assignments and eventually get the incomplete part done. Experience with such a software-hardware combined environment will

benefit students in the scientific, mathematical, and engineering disciplines.

2 Course Information

2.1 Components and schedule

The semester long *Network Systems Design* course consisted of four components: *Introduction*, *Traditional Network Systems*, *Network Processor Technology*, and *Example Network Processor*. They were divided into two categories: lectures and lab sessions. The course schedule is shown in Table I. The grade weights were assigned as follows: homework: 20%; midterm: 20%; lab projects: 30%; and final exam: 30%. There was no prerequisite on introductory computer network course and thus the first three weeks were used to introduce basic concepts of computer networks in a nutshell.

Table I. Lecture and Lab Schedule

Component	Lecture	Lab
Introduction (6 hours)	Course introduction, network architecture, layering & protocols, OSI and Internet architecture; Encapsulation, hardware building blocks, encoding, framing; Error detection, Ethernet (802.3), FDDI, switching and forwarding, circuit switching; Packet switching, IP, service model, socket, routing and forwarding; UDP and TCP.	Traffic monitoring and throughput measurement
Traditional Network Systems (12 hours)	Computer architecture; Packet processing algorithms and functions; Protocol software, socket; Hardware architecture for packet processing; Classifi-cation and forwarding; Switching fabrics.	Basic router configuration; Firewall, ethereal, switch vs hub
Network Processor Technology (6 hours)	Network processor introduction; Complexity of network processor design; Network processor architectures; Scaling a network processor; Design tradeoffs and consequences.	SystemC models and simulation
Example Network Processor (10 hours)	Overview of Agere network processor and FPL classification language; System architecture and modeling; Stateful network processor applications; Policing, buffer management and traffic shaping; Agere site visit; Network processing trends.	Network processor bridge; Fragment-ation and Encapsulation Stateful FPL application

2.2 Achieved Goals

The course offered in Fall 2003 consisted of both software and hardware components. The students were exposed to a variety of important software-hardware co-design concepts. They learned to program algorithms for network processing, use tools to design network processors, and construct network devices of complex network processing systems in a well-structured, hierarchical way.

We have created a project-based introduction to embedded processing in its application area of network processing, where there is increasing demand for skills and for which we anticipate substantial advances in technology. Students have gained hands-on experience in both the general area of embedded processing and in the specific area of network processing.

In addition, we have successfully strengthened ties between academic study of network processing and industrial practice in the field, given the fact that most advances in network processor architectures to date have been made in industry. Agere researchers have participated in the course development in terms of co-teaching lectures, developing laboratory sessions, conducting Agere site visit, and supervising internship.

Moreover, the development of the advanced undergraduate course in network processing has leveraged existing educational resources, including: (*i*) Classic texts and laboratory exercises in network processing before the advent of network processors, particularly Internet-oriented materials; and (*ii*) review feedback to *Network Systems Design Using Network Processors* (Agere Version of [9]), a new text by Professor Douglas Comer of Purdue University.

3 Network Processing Laboratory

3.1 Overview

The purpose of the network processing laboratory projects or assignments is for students to develop a thorough understanding of network processing concepts, architectures, algorithms and techniques by implementing them. “Learning through doing” forces the students to digest the information presented in classes to the point where they can instruct the computer how to apply it. Active learning such as this has a higher chance of having a lasting effect on students than if the students passively listen to lectures without reinforcement.

The architecture of the laboratory projects/assignments breaks the task of implementing

network devices into smaller, more manageable chunks. They incrementally build on top of each other to incrementally create a complete hardware-software solution to a sophisticated network processing system.

A *safety net* was provided for students who fail to complete a particular assignment. We also offered the benefit in the laboratory that students have an opportunity to work with many different partners throughout the semester.

The overall approach of the laboratory sequence is to start with high-level, application-oriented networking concepts with which students are already familiar, such as Internet communications and the World Wide Web, and work our way down networking protocol layers in the examination of underlying protocols and their processing in software and dedicated hardware. Once we have explored underlying mechanisms, labs reverse their direction, examining how network processor architectures are evolving to handle higher-level protocol layers at full speed. Thus the laboratory sequence consists of an analysis stage leading to underlying mechanisms, followed by a synthesis stage that reveals the forces behind current trends in network processing evolution.

3.2 Lab Projects

Below is a list of the six incremental lab projects associated with the laboratory practice sessions.

(1) Traffic monitoring and throughput measurement (step 1 of analysis):

Initial exercises use ethereal/tcpdump and similar network traffic monitors to capture and observe live packets created by real applications such as web browsers and email. Concepts include generation and observation of structured traffic, a central activity in professional network processing. Students use traffic monitors and generators learned in this step in all subsequent steps.

(2) Basic router configuration and raw socket (step 2 of analysis):

The router configuration lab helps students to understand more of network protocols by configuring Cisco routers to support various network topologies of the local area network and architecture such as VLAN. Also a homework-oriented assignment of raw socket concentrates on conventional network programming interfaces used by protocols and applications.

(3) Firewall, ethereal, switch vs. hub (step 3 of analysis):

Projects place network interface cards (NICs) on conventional computers into promiscuous mode and control packet receipt and transmission directly. This lab session includes three parts: configuring firewalls using `iptables` in Linux; using `ethereal` to capture network packets and observe the packets in various layers; and comparing the difference between a switch and a hub.

(4) SPA network processor simulator (step 1 of synthesis):

At this stage we begin reworking the mechanisms used in the previous stages into a form supported by dedicated network processing instruction sets and multiprocessor topologies. Exercises begin with an examination of fast path (a.k.a. wire speed or hard real-time) processing as contrasted with slow path (a.k.a. control path or non-real-time) processing, using both high-level functional simulation environments and actual network processor development environments, including tools from Agere Systems, Inc.

(5) SystemC models and simulation (step 2 of synthesis):

This stage uses high-level, functional simulation to explore hardware building blocks such as pattern matchers that are part of network processors. Students complete design of a hardware block and simulate its interactions with other network processor components. In a complete system design, a system designer can simulate execution of network processing code such as routing on a simulated processor written in SystemC. Students exercise this two-tiered simulation of hardware and software called co-simulation.

(6) Stateful FPL application (step 3 of synthesis):

Exercises focus on a representative sample of programs illustrating how network processors are currently used. Examples include bridges, routers, network address translators, and firewalls. For example, using Agere FPL (Functional Programming Language) to deploy a hash table to implement a learning Ethernet bridge.

4 Student Response

In a survey question asking students' comments about the course, 40% of the students mentioned that they liked the hands-on labs, and 10% of these students stated that the later labs tied everything together. In addition, 40% students found the subject matter to be relevant to today's network field. They felt that the material was interesting and presented well, and they learned a lot of new material. Other

students appreciated that the professors were well-qualified and treated the students with respect.

Twenty percent of the students felt that the course could be split over two semesters, with the first semester introducing the basics of network system design and the second semester introducing more advanced topics in greater detail. Other suggestions by individual students included having more labs like the first two, and providing more real work and fewer simulations.

5 Conclusions and Future Work

This paper presents contents of the *Network Systems Design* course used to introduce undergraduate students to understanding software-hardware co-design concepts and acquiring practical experience in embedded processing. The goals achieved include: (*i*) carrying out lab-based introduction to embedded processing in its application area of network processing, and (*ii*) strengthening ties between academic study of network processing and industrial practice in the field.

The next time this class is taught, a prerequisite of an introductory undergraduate course on computer networks should be imposed and the number of lectures and labs on the introduction of networking concepts would probably be increased. In addition, the student presentations on "what I learned" would be reserved for the second half of the semester.

More Agere's software will be adapted to our undergraduate course. Currently there is a production-quality network processor simulator (System Performance Analyzer – SPA) that Agere has donated for use in the course. There are also two prototype software tools that teaching assistants could enhance for use in the course. One is a network processor emulator (SAUNA) that translates network processor code into C code that can run on a PC containing two network interface cards. This emulator will allow students to design and test network processor algorithms on inexpensive PC hardware; code runs at PC speeds rather than at faster network processor speeds, but algorithms work identically. Having the emulator in addition to actual network processor hardware supports more lab stations at low expense, and it scales readily to inexpensive reuse at other colleges and universities. The other prototype software tool is an open source embedded system debugger from Agere (RTEEM) that a teaching assistant will enhance for debugging and algorithm visualization of network processing programs running on the emulator.

Acknowledgment

This project has been partly financed by a grant from the Commonwealth of Pennsylvania, Department of Community and Economic Development, through the Pennsylvania Infrastructure Technology Alliance (PITA), and it has also been supported by National Science Foundation DUE CCLI Award #0310745 and donations from Agere Systems, Inc.

References

[1] W. Bux, W.E. Denzel, T. Engbersen, A. Herkersdorf, and R.P. Luijten, "Technologies and building blocks for fast packet forwarding," *IEEE Communications Magazine*, Vol. 39, No. 1, Jan. 2001, pp. 70–77.

[2] T. Wolf and J.S. Turner, "Design issues for high-performance active routers," *IEEE Journal on Selected Areas in Communications*, Vol. 19, No. 3, March 2001, pp. 404–409.

[3] Y. Coady, S. O. Joon, and M.J. Feeley, "Using embedded network processors to implement global memory management in a workstation cluster," *Proceedings of the Eighth International Symposium on High Performance Distributed Computing*, 1999, pp. 319–328.

[4] P. Paulin, F. Karim and P. Bromley, "Network processors: a perspective on market requirements, processor architectures and embedded S/W tools," *Proceedings of the DATE* 2001 *on Design, Automation and Test in Europe*, 2001, pp. 420–429.

[5] Network processor homepage at Purdue University, http://www.cs.purdue.edu/np/.

[6] M. McDonald, J. Rickman, G. McDonald, P. Heeler, and D. Hawley, "Practical experiences for undergraduate computer networking students," *Journal of Computing in Small Colleges*, Volume 16, Issue 3, March 2001.

[7] K. M. Sivalingam and V. Rajaravivarma, "Education of wireless and ATM networking concepts using hands-on laboratory experience," *ACM SIGCSE Bulletin*, *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, Volume 31, Issue 1, March 1999.

[8] P. Steenkiste, "Networks: a network project course based on network processors," *Proceedings of the* 34*th Technical Symposium on Computer Science Education*, February 2003.

[9] D. Comer, *Network Systems Design Using Network Processor*, Prentice Hall, Upper Saddle River, New Jersey, USA, 2003.

Teaching Basics of Instruction Pipelining with HDLDLX

Miloš Bečvář
Department of Computer Science and Engineering,
Faculty of Electrical Engineering,
Czech Technical University in Prague
Karlovo nám. 13, Prague 2, Czech Republic
becvarm@fel.cvut.cz

Abstract: *HDLDLX is a graphically described VHDL model of 5-stage integer pipeline of well known DLX processor. It can be used as a platform explaining logic-level implementation of pipelined processor as a complement to SW functional simulators. Students can interact with model by implementing hazard resolution logic or modifying the pipeline structure. Even though that the model is internally represented in VHDL, the previous knowledge of this language is not required. HDLDLX can be used in conjunction with HDL Designer and Modelsim tools from Mentor Graphics corporation. Article also discusses pros and cons of using commercial EDA tools in undergraduate computer architecture course.*

1. INTRODUCTION

Good understanding of instruction pipelining is essential for current computer architecture students. RISC processors DLX [1] or MIPS64 [2] are used as examples explaining the main principles. It is a common practice to reinforce understanding of this topic by practical assignments done by students. One way to help understand of pipelining is making students to optimize programs for these processors. Cycle accurate simulators [3] with visualization capabilities such as DLXView [4], WinDLX [5] or WinMIPS64 [6] or MIPSIt [7] are used for program verification. Although these simulators give a good view of pipelined instruction execution, actual implementation of pipeline and associated hazard detection logic is usually hidden. Moreover the pipeline implementation in these simulators is mainly fixed although some parameters could be changed (e.g. functional unit latencies). Main question is how the students could experiment with actual pipeline structure on the logic design level. Some teachers presented dedicated tools which requires the students to specify the pipeline structure using specialized Hardware Description Language. These tools usually generates a complete VHDL or Verilog netlist of the processor specified. Example of such tool is ASIP Meister [8].

It is obvious that development and maintenance of similar tool is relatively complex task. We present another approach which leverages commercial EDA tools as teaching aids in undergraduate computer architecture course.

We were looking for a tool which allows design of simple pipelined processor and its simulation without knowledge of Verilog or VHDL. This requirement is a result of the fact that VHDL is introduced only to hardware oriented students after the undergraduate computer architecture course.

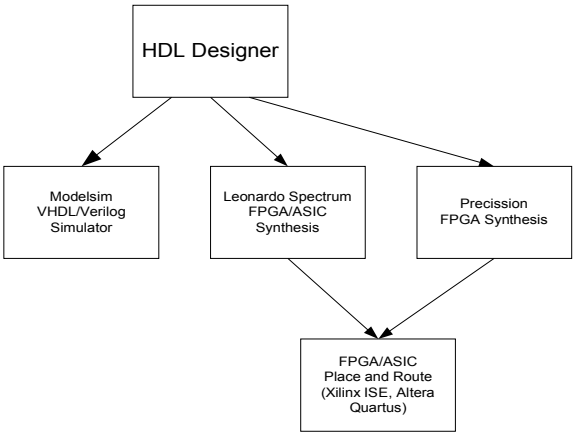
Finally we decided to use a graphical VHDL entry tool – HDL Designer from Mentor Graphics corporation for our experiments. In this article HDLDLX – a graphical VHDL model of well known integer DLX pipeline is presented. This model can be used together with HDL Designer and Modelsim for simple experiments with instruction pipeline.

The rest of the paper is organized as follows – section 2 presents an overview of HDL Designer and its use in computer architecture course. Section 3 outlines the developed HDLDLX model. Section 4 describes the use of this model in undergraduate computer architecture course. Section 5 presents conclusions and future work.

2. HDL DESIGNER OVERVIEW

HDL Designer is a professional EDA tool intended to be a "designers cockpit". It offers a graphical VHDL entry and integrates several downstream design tools in a single GUI – namely simulator Modelsim, synthesis tool Leonardo Spectrum, Precession and others (see fig. 1) Out intention was to build a flexible graphical model of integer DLX pipeline and simulate it using a common VHDL simulator Modelsim. The reason why we decided to use this tool was in fact that we used it in specialized design courses and it was possible to extend the number of licenses without increase of maintenance fee (offered in Mentor Graphics High Education Program).

Figure 1: HDL Designer flow



One of important question was how the processor model will be represented assuming that students do not know the VHDL or Verilog. HDL Designer offers several different types of graphical VHDL entry :

- **Block Diagram**
  It was obvious that top-level representation of the processor should be in block diagram. This diagram should be the same or similar to block diagrams used during lecture to simplify orientation of students and save time.
- **Truth Table**
  This view is useful for defining combinatorial components of the pipeline. It is also well known abstraction and easily understandable by students.
- **State Diagram**
  This abstraction is very useful for Finite State Machine specification. However, our integer pipeline does not use any state machines and this abstraction currently is not used.
- **Flow Chart**
  Flow Chart is a graphical equivalent of VHDL process. It has been used for sequential elements in design such as memories and registers

From all these components, the tool generates VHDL files which are submitted to Modelsim for simulation. Thanks to tight integration of Modelsim and HDL Designer, it is possible to cross-probe between Modelsim waveforms and graphical representation in HDL Designer. Namely, it is possible to observe a values of signals directly in the block diagram.

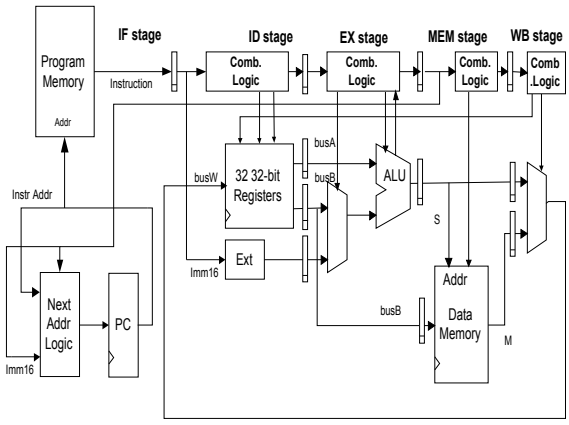
Although the creation of the model was relatively easy task for us, the big question was whether students are able to learn how to use it. Even when we use only limited functionality of the tool the overhead to learn how to use it can overweight actual benefits. For this reason high effort was spent in preparation of documentation and step by step user guide.

We return to this issue later in section 4. Another disadvantage of using commercial EDA tool is that it runs only with connection to licensing server. It limits the ability of students to run this tool from their home computer if they do not have an access to the Internet.

3. HDLDLX MODEL

Although the latest edition of Hennessy-Patterson book [2] switched to MIPS64 processor, we decided to implement 32-bit DLX processor because we use WinDLX and DLXV simulators in rest of the semester and DLX ISA during lectures. However, resulting model can be relatively easy modified to 64-bit MIPS due to similarity of pipeline structure.

Figure 2: HDLDLX pipeline



3.1 HDLDLX Pipeline Overview

DLX is well known 32-bit RISC processor used in computer architecture courses and many different simulators of this processor exist. These tools differs in the variant of DLX implementation because it evolves throughout the book.

Our main intention was to create the same variant of DLX as the one used during our lectures.
In the first pipelining lecture, the evolution of integer DLX processor datapath and controller from single-cycle non pipelined processor through multicycle processor to 5-stage integer pipeline.
The implemented model corresponds to this simple 5-stage integer DLX pipeline (Instruction Fetch – Instruction Decode – Execute – Memory – Write Back).

Following section outlines the implementation of DLX pipeline components.

3.2 HDLDLX Pipeline Components

HDLDLX consists of pipelined datapath and controller. Datapath is created by PC, program memory, register-file, ALU, data memory, multiplexers and pipeline registers. Controller consists of combinatorial logic in every stage and pipeline registers.

- **Program Counter (PC)** is fed by the value from Next Address Logic. Depending on control signal, the PC is either incremented by 4 or jump to branch target address.

Important control signal of PC is *IF_stall* which can prevent the PC from changing its value. This signal is very useful in implementation of pipeline stalls.

- **Program Memory** is implemented as a ROM. Its content can be preloaded from the file (containing instructions encoded in hexadecimal form). The size of Program Memory is limited to 16K x 4B (higher bits of PC are ignored).

- **Register file** implements 32 32-bit registers. It can be described as 3-ported synchronous RAM with R0 hardwired to zero. To be conform with book, the register file contains internal bypassing – data from write port could be *forwarded* directly to one of read ports in the same clock cycle (in exception of write to register R0 which is never forwarded). Register file ignores any attempt to write to register R0 which can be used to simplify the pipeline control.

- **ALU** is implemented using Truth-Table abstraction as a black box. It supports only limited set of binary and unary operations (see section 3.3). However number of operations can be expanded by increasing the size of ALU control bus and expanding the truth-table. Besides the result of operation, it also produces zero indication which can be used in branch evaluation.

- **Data Memory** is implemented as ideal synchronous RAM. Memory model pre-loads initial data from the text file and dumps its content into file after the simulation is finished. The size of Data Memory is currently limited to 64KB and higher address bits are ignored.

- **Controller.** As could be seen from fig. 2, the controller is implemented as a sequence of combinatorial logic sliced by pipeline registers. In real implementation, the instruction is decoded in ID stage into internal representation which flows through the pipeline stages. However, it would be very difficult for students to observe the pipeline behavior in this case. For this reason we let the complete 32-bit instruction code to flow through the whole pipeline. In every stage, the necessary control signals are decoded from this 32-bit instruction code. Although, this is slightly redundant implementation, it allows to understand the pipeline much easily**.** Control combinatorial logic in every pipeline stage is described as truth-table and can be easily modified.

- **Pipeline Registers** are implemented as rising-edge triggered D flip-flops. Two types of flip-flops is used in the pipeline - normal D-flip-flops in the datapath and specialized in the controller. Specialized D-flip-flops in the controller have two control signals – *stall* and *clear*. There are dedicated *stall* and *clear* signals for every pipeline stage (e.g. *ID_stall, ID_clear, EX_stall, EX_clear* etc.) If the *stall* signal is asserted, the pipeline register retains its value, if *clear* signal is asserted, the pipeline register is synchronously cleared. Clearing pipeline register in controller efficiently means that NOP is inserted to this pipeline stage.

As could be seen from fig.2 , the pipeline evaluates branch instruction in the MEM stage and no forwarding and no hazard detection is implemented. These features has to be added by students.
Main methodology to implement these changes is defining of *stall* and *clear* signals behavior either by drawing schematics or writing VHDL equation. We would discuss this in detail in the section 4.

3.3 HDLDLX Instruction Subset

Only limited instruction subset is implemented in HDLDLX. However, all types of instruction are supported (Register-Register, Register-Immediate, Load, Store and Branch).
Number of supported instructions can be expanded by changing combinatorial logic in controller and ALU.

4. EXPERIENCE WITH HDLDLX

We currently use the HDLDLX in our undergraduate computer architecture course. This course is obligatory for all computer science and engineering students and current capacity is around 300 students. The course has a single 90-minute lecture and single 90-minute laboratory seminar per week. HDLDLX is used in laboratory seminars during 4 sessions as could be seen in table 1. Experiments with HDLDLX were done every second week and interleaved with simulations on WinDLX. It means that students can compare two models of the same architecture. WinDLX also helps in understanding what must be implemented in HDLDLX. The ultimate goal is that both simulators process the same integer program equivalently.

Following subsections outline the actual use of HDLDLX during the course.

Table 1 HDLDLX in undergraduate CA Course.

Week	***Lab Overview***
1	Introduction to HDL Designer and HDLDLX model Simulation of HDLDLX with pipeline data hazards
2	Implementation of RAW hazard resolution logic (pipeline interlock)
3	Adding of control hazard resolution logic into DLX pipeline with stalls. Implementation of forwarding
4	Finalizing of forwarding

4.1 Introductory Session

As could be seen from table 1, a first session is spent in introduction to the tool and model. An ultimate goal of the first session is a brief explaining of HDL Designer and Modelsim and more detailed description of HDLDLX model.

Although only limited functionality of HDL Designer is used, some time must be spent in setting up the tool and explanation of necessary steps in using this tool in various situations. Tool setup was relatively easy task accomplished only by downloading and expanding of HDL Designer library from a web into a user directory. All HDLDLX model components were stored in shared library and students had read-only access to this library. It means that only top-level part of HDLDLX was stored in student's libraries and only this part can be modified. This restriction saves a lot of time possibly spent in tracking of peculiar bugs unintentionally introduced by students modifying of model components.
After setting-up the library in the user directory, the tool use was relatively simple and consists mainly from sequence of mouse clicking. In the beginning, students must open a project, open a library within this project and finally open structural view of HDLDLX. Luckily, the majority of these steps is performed only during a first HDL Designer run and later the tool opens the library automatically. Although it was not completely necessary, we briefly explained the concept of projects, libraries and blocks and their different views to students. The rest was not difficult to understand and students considered HDL Designer just as any other schematic editor. Generation of VHDL model, compilation and invoking of Modelsim was automated by a single clicking on Modelsim icon in HDL Designer. Overall the tool setup and necessary explanation took around 30 minutes.

Problem of explaining HDLDLX pipeline is more demanding. However, it was simplified by the fact that the same (simplified) DLX pipeline is explained in the same week during lecture. In the first lab, the effort is spent mainly in explaining how each type of instruction flow through pipeline and purpose of various parts of DLX datapath. This is illustrated by running of simulation of sample program.
Next, the concept of inserting stalls into pipeline using *stall* and *clear* signals is explained. Initial model of HDLDLX does not contain any hazard detection and resolution logic and students may actually observe the effect of RAW hazards in the pipeline. A good teaching aid is the ability of cross-probing signals between HDL Designer schematic diagram and Modelsim.

4.2 Experiments performed with HDLDLX

A list of experiments possible with HDLDLX is presented in table 1. Simple experiments are suitable for undergraduate course where students have limited access to the tool. More complex experiments can be performed in graduate course as a half semester assignments assuming that graduate students will have more knowledge of the HDL Designer and better access to the tool.

All simple experiments lead to specification of some form of combinatorial logic into pipeline. Typically, equations for *stall* and *clear* signals must be specified for implementation of pipeline interlocks and branch instruction. Forwarding is implemented by adding of multiplexers into datapath and specifying control of these multiplexers.

Table 2 Experiments with HDLDLX

Simple experiments possible with HDLDLX
• Implementation of data hazard detection logic and stalling of the pipeline • Implementation of pipeline flushing after branch instruction • Implementation of data forwarding • Moving branch evaluation into ID stage and implementation of delayed branches
More complex experiments with HDLDLX
• Implementation of Program and Data Caches and pipeline stall due to “Cache miss” • Implementation of multicycle operations (e.g. multiplication) and associated WAW, structural hazard resolution logic • Implementation of exceptions

During the first run of the course, we proposed use of schematic diagram or VHDL subset for specifying this combinatorial logic. It was expected that students would prefer schematic diagrams over learning of subset of a new language. However, majority of students decided to directly write VHDL parallel signal assignments. We thought that students preferred a text description because it is faster and it reminds them software programming.

Overhead of learning subset of VHDL syntax was not high. Students received a one-page simplified description of the VHDL parallel statements and some of them were even able to write these statements before the end of the first session with HDLDLX. Parallel assignments have smallest learning overhead in VHDL . Their another advantage is in fact that they introduce a dataflow way of thinking.

The major difficulty encountered by students was distinguishing between *std_logic* and *boolean* statements which use the same overloaded operators (e.g. AND, OR, NOT). It suggests that using Verilog can be even more straightforward in this application.

4.3 Experience from First Run on HDLDLX

After first introductory session, students work in groups of two autonomously and tried to complete

assigned tasks. A role of teaching assistant during these sessions was in helping students to overcome difficulties with VHDL and trying to push them on a way to find solution. Although the fact that students can work on the simulator only in the school complicated their task, it also limited the possibility of cheating by copying solution of other groups.

Experience shows that around 50 % of groups completed the assignments during the expected time and obtained a full number of points. The rest of students required more time but majority of them were finally able to complete it also. It was an important role of teaching assistant to check that students understand "their" solution to limit the possibility of cheating.

5. CONCLUSIONS AND FUTURE WORK

Current experience shows that HDLDLX is a good teaching aid in explaining basics of instruction pipelining. The use of commercial EDA tool allows relatively fast model development comparing to building custom simulator. Although the model is based on VHDL, the students were able to use it without previous knowledge of this language. Students of undergraduate computer architecture course were able to learn a limited subset of this language relatively easy and preferred using it over schematic diagrams.

Students who completed the assigned tasks get better understanding of complexity of hazard detection logic implementation. Moreover, they were also introduced to contemporary tools and language used in design of digital circuits. Students mostly stated that their task was relatively difficult but very interesting. A fact that a commercial tool is used was also positively appreciated and some students were interested to use this tool in the future courses.

A good experience we had with HDLDLX confirmed our decision to introduce Hardware Description Languages in early courses of logic design. It means that in the future, students will come to undergraduate computer architecture course with basic knowledge of VHDL which will offer new possibilities.

HDLDLX model will be soon available on the internet - http://service.felk.cvut.cz/hdl_dlx.html

REFERENCES

[1] Patterson, D., Hennessy, J.,Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers 1996, 2nd edition

[2] Patterson, D., Hennessy, J.,Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers 2002, 3rd edition

[3] Yurcik, W., Wolffe, G., Holliday, M. , A Survey of Simulators Used in Computer Organization/Architecture Courses**,** In: *Proc. of the 2001 Summer Computer Simulation Conference* (SCS 2001), Orlando, USA

[4] Zhang, Y., Adams, G.B.: An Interactive Visual Simulator for DLX pipeline, *Newsletter of IEEE Computer Society Technical Cometee on ComputerArchitecture,* September 1997

[5] Gruenbacher, H., Khosravipour, M.,WinDLX and MIPSim Pipeline Simulators for Teaching Computer Architecture, In: Proc. of *IEEE Symposium and Workshop on Engineering of Computer Based Systems* (ECBS'96) Friedrichshafen, 1996, GERMANY

[6] http://www.computing.dcu.ie/~mike/ winmips64.html

[7] Brorson, M.., MipsIT – a Simulator and Development Environment using Animation for Computer Architecture Education, In: *Proc. of Workshop of Computer Architecture Education*, Anchorage, USA , 2002

[8] ASIP Meister. http://www.eda-meister.org

Software Implementations of Division and Square Root Operations for Intel® Itanium® Processors

Marius Cornea
Intel Corporation

Abstract

Division and square root are basic operations defined by the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [1], and are implemented in hardware in most modern processors. In recent years however, software implementations of these operations have become competitive. The first IEEE-correct implementations in software of the division and square root operations in a mainstream processor appeared in the 1980s [2]. Since then, several major processor architectures adopted similar solutions for division and square root algorithms, including the Intel® Itanium® Processor Family (IPF). Since the first software algorithms for division and square root were designed and used, improved algorithms were found and complete correctness proofs were carried out. It is maybe possible to improve these algorithms even further.

The present paper gives an overview of the IEEE-correct division and square root algorithms for Itanium processors. As examples, a few algorithms for single precision are presented and properties used in proving their IEEE correctness are stated. Non-IEEE variants, less accurate but faster, of the division, square root and also reciprocal and reciprocal square root operations are discussed. Finally, accuracy and performance numbers are given. The algorithms presented here are inlined by the Intel and other compilers for IPF, whenever division and square root operations are performed.

Introduction

One of the design goals for the Intel® Itanium® architecture, finalized in the late 1990s, was to achieve world-class performance in floating-point computations. For this reason, the floating-point architecture included many novel features for Intel processors: available 82-bit floating-point format (1-bit sign, 17-bit exponent, and 64-bit significand), 128 floating-point registers, rotating registers and other support for software pipelining, multiple status fields, flexible computation modes, and a floating-point multiply-add instruction with only one rounding error in the addition step [3][4][5]. Today it is a known fact that this goal was achieved: presently, 16 of the 17 top positions (1 through 7 and 9 through 17) in the SpecFP 2000 ranking list for speed of single processor systems are held by machines based on Itanium processors.

The floating-point multiply-add instruction `fma` was at the basis of efficient software implementations of the floating-point division and square root operations. An important application of this instruction is in the calculation of exact remainders. For example for a division a/b, where a and b are floating-point numbers, a sequence of increasingly better approximations q_0, q_1, … q_{i-1}, q_i of the quotient a/b can be calculated using the Newton-Raphson or another equivalent method. A final approximation q_i can be obtained that can be rounded correctly as specified by the IEEE Standard 754-1985, provided a correction term (remainder) r_{i-1} can be calculated exactly based on the penultimate approximation q_{i-1}:

$$r_{i-1} = a - b \cdot q_{i-1}$$

If the approximation q_{i-1} is good enough[1], it can be shown that r_{i-1} calculated with an `fma` instruction can be represented always exactly as a floating-point number. The floating-point multiply-add operation, which is not defined by the current IEEE Standard for Binary Floating-Point Arithmetic, is thus essential in calculating IEEE-correct results in software for division and square root in the three most widely used formats defined by the standard: single precision, double precision, and double-extended precision. A brief review of some of the IEEE floating-point formats available in the Itanium architecture is included here for reference.

In general, floating-point numbers are represented as a concatenation of a sign bit, an M-bit exponent field containing a biased exponent, and an N-bit significand field (in this context N = 24, 53, or 64). Mathematically:

$$f = \sigma \cdot s \cdot 2^e$$

where $\sigma = \pm 1$, $s \in [1,2)$, $e \in [e_{min}, e_{max}] \cap \mathbf{Z}^2$, $s = 1 + k/2^{N-1}$, $k \in \{0, 1, 2, \ldots, 2^{N-1}-1\}$, $e_{min} = -2^{M-1} + 2$, and $e_{max} = 2^{M-1} - 1$. Let $\mathbf{F}_N$ be the set of floating-point numbers with N-bit significands and unlimited exponent range (no special values such as zeros, infinities, or NaNs[3] are included). The main

[1] It suffices for q_{i-1} to be accurate to one *unit-in-the-last-place* (ulp). A unit-in-the-last-place represents the weight of the least significant digit of a floating-point number. For a floating-point number f with N bits in the significand, $f = b_0.b_1b_2\ldots b_{N-1} \cdot 2^e$, the value of one ulp is $1\ ulp(f) = 2^{e-N+1}$.

[2] $\mathbf{Z}$ is the set of integer numbers.

[3] NaN stands for not-a-number. NaNs are symbolic values encoded in floating-point format, used most often to cause or be the result of invalid operations.

parameters of the formats used in the software implementations discussed in the paper are shown in Table 1.

Table 1. Floating-Point Formats Available in the Itanium Architecture (subset)

Format	Precision (N)	Exponent Bits (M)	Exponent Range
Single	24	8	$-126 \le e \le 127$
Double	53	11	$-1022 \le e \le 1023$
Double extended	64	15	$-16382 \le e \le 16383$
Register single	24	17	$-65534 \le e \le 65535$
Register double	53	17	$-65534 \le e \le 65535$
Register	64	17	$-65534 \le e \le 65535$

The division and square root operations discussed here have in general two different implementations available for every format: one that minimizes latency, and one that maximizes throughput. The latency-optimized versions minimize the number of clock cycles elapsed from the beginning of the computation until the result is available. In most cases this is easy to determine, because the majority of floating-point instructions have a latency of 4 clock cycles on the Itanium 2 processor. The throughput-optimized versions minimize the number of clock cycles elapsed between the moments when two consecutive floating-point results are generated. The latter are intended for use in software-pipelined loops, and the resulting throughput depends on the number of functional units available. For example, the throughput-optimized single precision division algorithm uses 7 floating-point instructions, and possibly three memory access instructions. The limiting factor in this case is the number of floating-point instructions. On the Itanium 2 processor, which has two floating-point units available, it will take on average 7/2 = 3.5 clock cycles to generate a result with the throughput-optimized algorithm (but only if the loop is unrolled once, otherwise the throughput will be of 4 clock cycles/result).

IEEE-Correct Floating-Point Division

Division operations that comply with the IEEE Standard 754-1985 have a clearly defined result. In general (exceptions are the cases of underflow or overflow) this is the exact result rounded to the destination precision, using the IEEE rounding mode currently in effect (rounding to nearest, toward zero, toward positive infinity, or toward negative infinity). Division for Itanium processors is implemented based on iterative algorithms, starting with an 11-bit approximation y_0 of the denominator's reciprocal. This value is provided by a special instruction performing a table lookup, `frcpa`, and has a relative error of at most $2^{-8.886}$:

$$y_0 = 1/b \cdot (1+\varepsilon_0),\ |\varepsilon_0| < 2^{-8.886}$$

Multiplying this value by a, a first approximation of the quotient is obtained and its relative error e_0 can be calculated. The symbol *rn* denotes the IEEE round-to-nearest mode, and *rnd* represents any IEEE rounding mode.

$$q_0 = (a \cdot y_0)_{rn} = a/b \cdot (1+\varepsilon_0)$$

$$e_0 = (1 - b \cdot y_0)_{rn} = -\varepsilon_0$$

This approximation can be further improved if the value of q_0 is multiplied by the polynomial $1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1}$, derived from the identity

$$(1 + \varepsilon_0) \cdot (1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1}) = 1 - (-\varepsilon_0)^k$$

The result (ignoring for now the rounding errors) will be:

$$q \approx a/b \cdot (1 - (-\varepsilon_0)^k)$$

In addition, an optimal way of calculating the product of this polynomial by $1+\varepsilon_0$ has to be determined for each division algorithm: with the lowest latency for latency-optimized operations, and with the lowest number of floating-point instructions for throughput-optimized operations.

Consider as a first example the latency-optimized single precision division algorithm.

Single precision division, optimized for latency

The following algorithm calculates the single precision value $q'_3 = (a/b)_{rnd}$, where a and b are single precision numbers. All the other intermediate results are 82-bit floating-point register format numbers. The precision used for each step is shown too. An approximate value of the result is also shown, calculated assuming that the rounding errors are negligible.

(1) $y_0 = 1/b \cdot (1+\varepsilon_0),\ |\varepsilon_0| < 2^{-8.886}$
table lookup

(2) $q_0 = (a \cdot y_0)_{rn} = a/b \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(3) $e_0 = (1 - b \cdot y_0)_{rn} = -\varepsilon_0$
82-bit floating-point register format

(4) $q_1 = (q_0 + e_0 \cdot q_0)_{rn} \approx a/b \cdot (1-\varepsilon_0^2)$
82-bit floating-point register format

(5) $e_1 = (e_0 \cdot e_0)_{rn} \approx \varepsilon_0^2$
82-bit floating-point register format

(6) $q_2 = (q_1 + e_1 \cdot q_1)_{rn} \approx a/b \cdot (1-\varepsilon_0^4)$
82-bit floating-point register format
(7) $e_2 = (e_1 \cdot e_1)_{rn} \approx \varepsilon_0^4$
82-bit floating-point register format
(8) $q_3 = (q_2 + e_2 \cdot q_2)_{rn} \approx a/b \cdot (1-\varepsilon_0^8)$
17-bit exponent, 53-bit significand
(9) $q'_3 = (q_3)_{rnd} \approx a/b \cdot (1-\varepsilon_0^8)$
single precision

This shows that the intermediate approximations q_0, q_1, q_2, and q_3 are getting increasingly closer to a/b. The last step is needed to reduce the precision of the result to 24 bits, for the single precision format. As steps (2) and (3), (4) and (5), and (6) and (7) respectively can be executed in parallel, the total latency on the Itanium 2 processor will be of 6 x 4 = 24 clock cycles. In software-pipelined form, this algorithm could generate on average a result every 9/2=4.5 clock cycles. However, an algorithm can be found that has better throughput characteristics.

Single precision division, optimized for throughput

The first idea was to modify the latency-optimized algorithm so that the first five steps generate increasingly better approximations y_1 and y_2 of 1/b, rather than q_1 and q_2. The subsequent steps would be to calculate

$$q_0 = (a \cdot y_2)_{rn}$$

then an exact remainder

$$r_0 = (a - b \cdot q_0)_{rn}$$

in the penultimate step, and the correctly rounded result

$$q_1 = (q_0 + r_0 \cdot y_2)_{rnd} \approx a/b \cdot (1-\varepsilon_0^8)$$

in the last step. This would result in a latency of 7 x 4 = 28 clock cycles, which is worse than that of the previous algorithm, but a better throughput of 8/2 = 4 clock cycles/result. However, an even better algorithm could be found after noticing that $\varepsilon_0^8 < 2^{-71.088}$ leads to a value q_1 before rounding that is more accurate than needed for an IEEE-correct single precision result. The relative error incurred when rounding a real number to single precision is less than 2^{-24}, and about twice as much accuracy should be enough (as shall be seen in the subsection on Correctness Proofs). It suffices for example to calculate $q \approx a/b \cdot (1-\varepsilon_0^6)$ where $\varepsilon_0^6 < 2^{-53.316}$. The best throughput-optimized algorithm is thus:

(1) $y_0 = 1/b \cdot (1+\varepsilon_0)$, $|\varepsilon_0| < 2^{-8.886}$
table lookup
(2) $e_0 = (1 - b \cdot y_0)_{rn} = -\varepsilon_0$
82-bit floating-point register format
(3) $e_1 = (e_0 + e_0 \cdot e_0)_{rn} \approx -\varepsilon_0+\varepsilon_0^2$
82-bit floating-point register format
(4) $y_1 = (y_0 + e_1 \cdot y_0)_{rn} \approx 1/b \cdot (1+\varepsilon_0^3)$
82-bit floating-point register format
(5) $q_1 = (a \cdot y_1)_{rn} \approx a/b \cdot (1+\varepsilon_0^3)$
17-bit exponent, 24-bit significand
(6) $r_1 = (a - b \cdot q_1)_{rn} = -a \cdot \varepsilon_0^3$
82-bit floating-point register format
(7) $q = (q_1 + r_1 \cdot y_1)_{rnd} \approx a/b \cdot (1-\varepsilon_0^6)$
single precision

In software-pipelined form, this algorithm can generate on average one result every 3.5 clock cycles. However, for this the loop would have to be unrolled once, so that it will contain an even number of floating-point instructions. Then two results will be generated on average every 14/2=7 clock cycles.

Similar algorithms were designed for double and double-extended precision division operations. In each case, the optimal sequence was selected that would still afford sufficient accuracy in the final result $q \approx a/b \cdot (1 - (-\varepsilon_0)^k)$ to allow for correct IEEE rounding in all cases. Of all possible sequences, the one that minimized the number of clock cycles was chosen for latency-optimized algorithms, and the one with the lowest number of floating-point instructions for throughput-optimized algorithms. The complete set of IEEE-correct algorithms for the division operation can be found in [6], where source code for all the IPF division algorithms can also be obtained.

Correctness Proofs

Proofs were developed to show that the results of the division algorithms proposed for single, double, and double-extended computations are IEEE-correct for any combination of operands and for any of the four IEEE rounding modes. This included showing also that the floating-point exception status flags are always set correctly, and that unmasked exceptions trap as specified in the IEEE Standard (using the user status field sf0 only in the first and last computation steps and the reserved status field sf1 in the intermediate steps helps ensure correct IEEE exception behavior; note that Itanium processors have four status fields available). To prove that the results are always numerically correct, three properties were used [7]. (The values N of concern in this context are N = 24, N = 53, and N = 64.)

Theorem 1. Let $a, b \in \mathbf{F}_N$, such that $a/b \notin \mathbf{F}_N$, $q^* \in \mathbf{R}$, and $N_1 \in \mathbf{N}$[4], $N_1 \geq 2 \cdot N + 1$.

If q^* is within 1 ulp of a/b in $\mathbf{F}_{N1}$, then

$$(q^*)_{rnd} = (a/b)_{rnd}.$$

Theorem 2. Let $b \in \mathbf{F}_N$, with the restriction that the significand of b is not 1.11…1. Let $y \in \mathbf{F}_N$ be an approximation of 1/b within 1 ulp of 1/b in $\mathbf{F}_N$. Then the computation:

$$e = (1 - b \cdot y)_{rn}$$
$$y' = (y + e \cdot y)_{rn}$$

[4] **R** is the set of real numbers, and **N** is the set of natural numbers.

yields $y' = (1/b)_{rn}$.

Theorem 3. Let $a, b \in \mathbf{F}_N$. If $y \in \mathbf{R}^*$ is within 1/2 ulp of 1/b in $\mathbf{F}_N$, $q \in \mathbf{F}_N$, and $q \cong a/b$ is within 1 ulp of a/b in $\mathbf{F}_N$, then the computation

$$r = (a - b \cdot q)_{rn}$$

$$q' = (q + r \cdot y)_{rnd}$$

yields $q' = (a/b)_{rnd}$.

Theorem 1 was applied in proving correctness of the latency-optimized single precision division algorithm. Relative error evaluations for steps (1) through (8) showed that q_3 is within 1 ulp of a/b in $\mathbf{F}_{49}$. Theorem 1 proves that in step (9), $q'_3 = (a/b)_{rnd}$ (i.e. a/b is correctly rounded, as specified by the IEEE Standard).

Theorem 3 was applied in proving correctness of the throughput-optimized single precision division algorithm. First it was shown that y_1 is within 1/2 ulp of 1/b in $\mathbf{F}_{24}$ and q_1 is within 1 ulp of a/b in $\mathbf{F}_{24}$. Theorem 3 states that steps (6) and (7):

$$(6)\ r_1 = (a - b \cdot q_1)_{rn}$$

$$(7)\ q = (q_1 + r_1 \cdot y_1)_{rnd}$$

yield $q = (a/b)_{rnd}$.

Theorem 2 was needed only for the double-extended division algorithms, where the operands and the result have the same precision as the intermediate calculations. This makes it more difficult to rely just on simple relative error evaluations to show for example that y in the last step is within 1/2 ulp of 1/b as required by Theorem 3, but Theorem 2 makes this possible. One special case had to be treated separately, when the significand of b is 1.11…1 (but for this case it could be checked directly that $y' = (1/b)_{rn}$).

The mathematical proofs of correctness were checked further using an automatic proof checker written in HOL [8].

Non-IEEE Floating-Point Division

There are applications where strict IEEE accuracy for floating-point computations may not be required, and instead faster basic operations would be of more benefit. To cover such needs, non-IEEE floating-point division algorithms were derived from the IEEE-correct versions, with relative errors not exceeding 1 ulp (the IEEE-correct operations have relative errors of at most 0.5 ulp). Non-IEEE algorithms were designed also for reciprocal operations, which are not defined by the IEEE Standard. The division operations performed by the non-IEEE algorithms are thus slightly less accurate, but faster than their equivalent IEEE-correct algorithms.

For example, the non-IEEE single precision division algorithm is:

(1) $y_0 = 1/b \cdot (1+\varepsilon_0),\ |\varepsilon_0| < 2^{-8.886}$
table lookup
(2) $q_0 = (a \cdot y_0)_{rn} = a/b \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(3) $e_0 = (1 - b \cdot y_0)_{rn} = -\varepsilon_0$
82-bit floating-point register format
(4) $e_1 = (e_0 + e_0 \cdot e_0)_{rn} \approx -\varepsilon_0 + \varepsilon_0^2$
82-bit floating-point register format
(5) $q_1 = (q_0 + e_1 \cdot q_0)_{rnd} \approx a/b \cdot (1+\varepsilon_0^3)$
single precision

The same algorithm can be used both in latency-optimized as well as throughput-optimized code.

Only a limited correctness proof is required in this case. The maximum relative error of the result has to be determined, and it has to be proved that overflow and underflow conditions occur reasonably close to those for the IEEE-correct algorithm. The exception status flag for precision is not checked in this case.

The algorithm for calculating the non-IEEE single precision reciprocal is even simpler:

(1) $y_0 = 1/b \cdot (1+\varepsilon_0),\ |\varepsilon_0| < 2^{-8.886}$
table lookup
(2) $e_0 = (1 - b \cdot y_0)_{rn} = -\varepsilon_0$
82-bit floating-point register format
(3) $e_1 = (e_0 \cdot e_0 + e_0)_{rn} \approx -\varepsilon_0 + \varepsilon_0^2$
82-bit floating-point register format
(4) $y_1 = (y_0 + e_1 \cdot y_0)_{rnd} \approx 1/b \cdot (1+\varepsilon_0^3)$
single precision

Similar algorithms for non-IEEE double precision division and reciprocal are given in [9], together with source code.

Latency, Throughput, and Accuracy for Division and Reciprocal Operations

Latency and throughput values for the single, double, and double-extended IEEE-correct and non-IEEE division and reciprocal operations on the Itanium 2 processor are given in Tables 2a and 2b.

Table 2a. Latency, Throughput, and Accuracy for IPF IEEE Division and Reciprocal Operations

Operation	Latency (clock cycles)	Throughput (clock cycles/ result)	Accuracy (ulps)
Single Precision Division	24	3.5	0.50
Double Precision Division	28	5.0	0.50
Double-Extended Precision Division	32	7.0	0.50
Single Precision Reciprocal	24	3.5	0.50
Double Precision Reciprocal	28	5.0	0.50

Theoretical error bounds for the non-IEEE operations are given in Table 2b. These are guaranteed upper bounds, but might not be reached in some cases. For this reason, maximum errors observed in testing are also included in the table.

Table 2b. Latency, Throughput, and Accuracy for IPF Non-IEEE Division and Reciprocal Operations

Operation	Latency (clock cycles)	Throughput (clock cycles/ result)	Theoretical Accuracy (ulps)	Observed Accuracy (ulps)
Single Precision Division	16	2.5	0.6585	0.6524
Double Precision Division	20	4.0	0.5018	0.5010
Double-Extended Precision Division	NA	NA	NA	NA
Single Precision Reciprocal	16	2.0	0.6585	0.6487
Double Precision Reciprocal	20	3.5	0.5010	0.5003

IEEE-Correct Floating-Point Square Root

Square root operations that comply with the IEEE Standard 754-1985 return the exact result rounded to the destination precision, using the IEEE rounding mode currently in effect. The square root operation for Itanium processors is implemented based on iterative algorithms as well, starting with an 11-bit approximation y_0 of the reciprocal square root. This value is provided by a special instruction performing a table lookup, `frsqrta`, and has a relative error of at most $2^{-8.831}$:

$$y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0), \quad |\varepsilon_0| < 2^{-8.831}$$

Multiplying by a, a first approximation of the square root is obtained and its relative error δ can be calculated:

$$S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$$
$$\delta = 1/2 \cdot (1 - S_0 \cdot y_0)_{rn} = -\varepsilon_0 - 1/2 \cdot \varepsilon_0^2$$

Note that $1 - 2\,\delta = (1+\varepsilon_0)^2$. Just as for division, the approximation S_0 can be improved further if it is multiplied by $1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1}$. The result (ignoring the rounding errors) will be:

$$S \approx \sqrt{a} \cdot (1 - (-\varepsilon_0)^k)$$

A complication in this case is the fact that the relative error δ calculated for S_0 is not equal to $-\varepsilon_0$, as it was for the division operation. In order to use the identity:

$$(1 + \varepsilon_0) \cdot (1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1}) = 1 - (-\varepsilon_0)^k$$

a polynomial in δ has to be found, that approximates sufficiently well

$$1 - \varepsilon_0 + \varepsilon_0^2 - \ldots + (-\varepsilon_0)^{k-1} + \ldots = 1/(1 + \varepsilon_0)$$

For this, the value of ε_0 is calculated from $\delta = -\varepsilon_0 - 1/2 \cdot \varepsilon_0^2$:

$$\varepsilon_0 = -1 + \sqrt{(1 - 2 \cdot \delta)}$$

The McLaurin series expansion for $1/(1 + \varepsilon_0) = 1/\sqrt{(1 - 2 \cdot \delta)}$ is:

$$1 - \varepsilon_0 + \varepsilon_0^2 - \varepsilon_0^3 + \varepsilon_0^4 - \ldots = 1 + \delta + 3/2 \cdot \delta^2 + 5/2 \cdot \delta^3 + 35/8 \cdot \delta^4 + 63/8 \cdot \delta^5 + 231/16 \cdot \delta^6 + \ldots$$

An approximation of the expansion in δ consisting of a few terms can be used to design an algorithm converging toward the square root value. The coefficients of some of the higher degree terms in this approximation can even be modified to make the calculation easier. Because of the truncation, the result will be

$$S \approx \sqrt{a} \cdot (1 + O(\varepsilon_0^k))$$

instead of

$$S \approx \sqrt{a} \cdot (1 - (-\varepsilon_0)^k)$$

where $O(\varepsilon_0^k)$ denotes a polynomial containing terms in ε_0 of degree k or higher.

In addition, an optimal way of calculating the product of this polynomial by $1+\varepsilon_0$ has to be determined for each square root algorithm: with the lowest latency for latency-optimized operations, and with the lowest number of floating-point instructions for throughput-optimized operations.

Consider as a first example the latency-optimized single precision square root algorithm.

Single precision square root, optimized for latency

The following algorithm calculates $S = (\sqrt{a})_{rnd}$ in single precision, where a is a single precision number. An approximate value of the result is also shown, calculated assuming that the rounding errors are negligible. The approximation is expressed in terms of ε_0 and/or δ, as convenient:

(1) $y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0), \quad |\varepsilon_0| < 2^{-8.831}$
table lookup

(2) $H_0 = (0.5 \cdot y_0)_{rn} = 1/(2 \cdot \sqrt{a}) \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(3) $S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(4) $d = (0.5 - S_0 \cdot H_0)_{rn} = -\varepsilon_0 + 1/2 \cdot \varepsilon_0^2 = \delta$
82-bit floating-point register format

(5) $e = (1 + 1.5 \cdot d)_{rn} \approx 1 + 3/2 \cdot \delta$
82-bit floating-point register format

(6) $T_0 = (d \cdot S_0)_{rn} \approx = \sqrt{a} \cdot \delta \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(7) $G_0 = (d \cdot H_0)_{rn} \approx 1/(2 \cdot \sqrt{a}) \cdot \delta \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(8) $S_1 = (S_0 + e \cdot T_0)_{rn} \approx$

$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2)$
17-bit exponent, 24-bit significand

(9) $H_1 = (H_0 + e \cdot G_0)_{rn} \approx$
$1/(2\sqrt{a}) \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2)$
82-bit floating-point register format

(10) $d_1 = (a - S_1 \cdot S_1)_{rn} \approx$
$a \cdot (5 \cdot \delta^3 + 15/4 \cdot \delta^4 + 9/2 \cdot \delta^5)$
82-bit floating-point register format

(11) $S = (S_1 + d_1 \cdot H_1)_{rnd} \approx$
$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2 + 5/2 \cdot \delta^3 +$
$35/8 \cdot \delta^4 + 63/8 \cdot \delta^5 + 81/16 \cdot \delta^6 + 27/8 \cdot \delta^7)$
$= \sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 - \varepsilon_0 + \varepsilon_0^2 - \varepsilon_0^3 + \varepsilon_0^4 - \varepsilon_0^5 +$
$O(\varepsilon_0^6)) = \sqrt{a} \cdot (1 + O(\varepsilon_0^6))$
single precision

This shows that approximations S_0, S_1, and S are getting increasingly closer to $\sqrt{a}$. As steps (2) and (3), then (5), (6) and (7), and also (8) and (9) can be executed in parallel, the total latency on the Itanium 2 processor will be 7 x 4 = 28 clock cycles. In software-pipelined form, this algorithm could generate a result every 11/2=5.5 clock cycles. However, an algorithm can be found that has better throughput characteristics.

Single precision square root, optimized for throughput

The following algorithm for the calculation of the single precision square root has the least number of instructions possible, and therefore is best suited for software-pipelined loops. It calculates $S = (\sqrt{a})_{rnd}$ in single precision, where a is a single precision number:

(1) $y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0), \ |\varepsilon_0| < 2^{-8.831}$
table lookup

(2) $H_0 = (0.5 \cdot y_0)_{rn} = 1/(2 \cdot \sqrt{a}) \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(3) $S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format

(4) $d = (0.5 - S_0 \cdot H_0)_{rn} = -\varepsilon_0 + 1/2 \ \varepsilon_0^2 = \delta$
82-bit floating-point register format

(5) $d' = (d + 0.5 * d)_{rn} \approx 3/2 \cdot \delta$
82-bit floating-point register format

(6) $e = (d + d * d')_{rn} \approx \delta + 3/2 \cdot \delta^2$
82-bit floating-point register format

(7) $S_1 = (S_0 + e * S_0)_{rn} \approx$
$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \ \delta^2)$
17-bit exponent, 24-bit significand

(8) $H_1 = (H_0 + e * H_0)_{rn} \approx$
$1/(2\sqrt{a}) \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2)$
82-bit floating-point register format

(9) $d_1 = (a - S_1 \cdot S_1)_{rn} \approx$
$a \cdot (5 \cdot \delta^3 + 15/4 \cdot \delta^4 + 9/2 \cdot \delta^5)$
82-bit floating-point register format

(10) $S = (S_1 + d_1 \cdot H_1)_{rnd} \approx$
$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2 + 5/2 \cdot \delta^3 +$
$35/8 \cdot \delta^4 + 63/8 \cdot \delta^5 + 81/16 \cdot \delta^6 + 27/8 \cdot \delta^7)$
$= \sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 - \varepsilon_0 + \varepsilon_0^2 - \varepsilon_0^3 + \varepsilon_0^4 - \varepsilon_0^5 +$
$O(\varepsilon_0^6)) = \sqrt{a} \cdot (1 + O(\varepsilon_0^6))$
single precision

Only steps (2) and (3), and then (7) and (8) can be executed in parallel, so the latency of 8 x 4 = 32 clock cycles is worse than that of the previous algorithm. However, its throughput of 10/2 = 5 clock cycles/result is better. It can be noticed that even though the throughput-optimized sequence differs slightly from the latency-optimized one, they both calculate practically the same result. The rounding errors might accumulate differently, but the end result was shown to be IEEE-correct in both cases.

Similar algorithms were designed for double and double-extended precision square root operations. In each instance, the optimal sequence was selected that would still afford sufficient accuracy in the final step to allow for correct IEEE rounding in all cases. Similar to the single precision square root, an optimal sequence was determined in each case, that would lead to a result in the form $S \approx \sqrt{a} \cdot (1-(-\varepsilon_0)^k)$. Of all possible sequences, the one that minimizes the number of clock cycles was chosen for latency-optimized algorithms, and the one with the lowest number of floating-point instructions for throughput-optimized algorithms. The complete set of IEEE-correct algorithms for the square root operation can be found in [6], where source code for all the IPF square root algorithms can also be obtained.

Correctness Proofs

Proofs were developed to show that the results of the square root algorithms proposed for single, double, and double-extended computations are IEEE-correct for any combination of operands and for any of the four IEEE rounding modes. This included showing that the floating-point exception status flags are always set correctly, and that unmasked exceptions trap as specified in the IEEE Standard (just as for division, using the user status field sf0 only in the first and last computation steps and the reserved status field sf1 in the intermediate steps helps ensure correct IEEE exception behavior). To prove that the results are always numerically correct, two properties were used [7]:

Theorem 4. Let $a \in \mathbf{F}_N$ and ulp ($\sqrt{a}$) one ulp of $\sqrt{a}$ in $\mathbf{F}_N$. If $\sqrt{a} \notin \mathbf{F}_N$, then for any $f \in \mathbf{F}_N$, the distance between $\sqrt{a}$ and f satisfies

$$| \sqrt{a} - f | > 2^{-N-1} \cdot \text{ulp} (\sqrt{a})$$

Theorem 5. Let $a \in \mathbf{F}_N$ and ulp ($\sqrt{a}$) one ulp of $\sqrt{a}$ in $\mathbf{F}_N$. For any $m \in \mathbf{F}_{N+1} - \mathbf{F}_N$ (midpoint between two consecutive floating-point numbers in $\mathbf{F}_N$), the distance between $\sqrt{a}$ and m satisfies

$$| \sqrt{a} - m | > 2^{-N-3} \cdot \text{ulp} (\sqrt{a})$$

These two properties show that if $\sqrt{a}$ cannot be represented as a floating-point number with an N-bit significand (which is the non-trivial case to check),

then there are exclusion zones of known minimal width around any floating-point number, as well as around any midpoint between two consecutive floating-point numbers, within which $\sqrt{a}$ cannot exist. The minimum distance between $\sqrt{a}$ and f or $\sqrt{a}$ and m can be determined analytically, as well as values of the argument a for which $\sqrt{a}$ is close to points f or m [7] (few points are 'really close' and they can be determined relatively easily). Excluding a number of these points a has the effect of increasing the widths of the exclusion zones. This can be done until the exclusion zones are more than twice wider than the maximum error of the result that approximates $\sqrt{a}$. It means that the exact result and the approximation computed by the algorithm are on the same side of any floating-point number or any midpoint, and therefore they will both round to the same floating-point value. For the relatively few cases of arguments inside the increased exclusion zones, verification was carried out directly.

Similar to the case of the division algorithms, the mathematical proofs of correctness were checked further using an automatic proof checker written in HOL [10].

Non-IEEE Floating-Point Square Root

Just as for division, non-IEEE floating-point square root algorithms that are less accurate but more efficient were derived from the IEEE-correct versions, with relative errors not exceeding 1 ulp (the IEEE-correct operations have relative errors of at most 0.5 ulp). Non-IEEE algorithms were designed also for reciprocal square root operations, which are not defined by the IEEE Standard. For example, the non-IEEE single precision square root algorithm is:

(1) $y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0)$, $|\varepsilon_0| < 2^{-8.831}$
table lookup
(2) $S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(3) $d = (1 - S_0 \cdot y_0)_{rn} = -2 \cdot \varepsilon_0 + \varepsilon_0^2 = 2 \cdot \delta$
82-bit floating-point register format
(4) $e = (0.5 + 0.375 \cdot d)_{rn} \approx 1/2 + 3/4 \cdot \delta$
82-bit floating-point register format
(5) $T_0 = (d \cdot S_0)_{rn} \approx 2 \cdot \delta \cdot \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(6) $S = (S_0 + e \cdot T_0)_{rnd} \approx$
$\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2) =$
$\sqrt{a} \cdot (1 + 5/2 \cdot \varepsilon_0^3 + 15/8 \cdot \varepsilon_0^4 + 3/8 \cdot \varepsilon_0^5) =$
$\sqrt{a} \cdot (1 + O(\varepsilon_0^3))$
single precision

The same algorithm can be used both in latency-optimized as well as throughput-optimized code.

Only a limited correctness proof is required. The maximum relative error of the result has to be determined, but the exception status flag for precision is not checked in this case.

The algorithm for calculating the non-IEEE single precision reciprocal square root is:

(1) $y_0 = 1/\sqrt{a} \cdot (1+\varepsilon_0)$, $|\varepsilon_0| < 2^{-8.831}$
table lookup
(2) $S_0 = (a \cdot y_0)_{rn} = \sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(3) $d = (1 - S_0 \cdot y_0)_{rn} = -2 \cdot \varepsilon_0 + \varepsilon_0^2 = 2 \cdot \delta$
82-bit floating-point register format
(4) $e = (0.5 + 0.375 \cdot d)_{rn} \approx 1/2 + 3/4 \cdot \delta$
82-bit floating-point register format
(5) $T_0 = (d \cdot y_0)_{rn} \approx 2 \cdot \delta \cdot 1/\sqrt{a} \cdot (1+\varepsilon_0)$
82-bit floating-point register format
(6) $S = (y_0 + e \cdot T_0)_{rnd} \approx$
$1/\sqrt{a} \cdot (1+\varepsilon_0) \cdot (1 + \delta + 3/2 \cdot \delta^2) =$
$1/\sqrt{a} \cdot (1 + 5/2 \cdot \varepsilon_0^3 + 15/8 \cdot \varepsilon_0^4 + 3/8 \cdot \varepsilon_0^5) =$
$1/\sqrt{a} \cdot (1 + O(\varepsilon_0^3))$
single precision

Similar algorithms for non-IEEE double precision square root and reciprocal square root are given in [9], together with source code.

Latency, Throughput, and Accuracy for Square Root and Reciprocal Square Root Operations

Latency and throughput numbers for the single, double, and double-extended IEEE-correct and non-IEEE square root and reciprocal square root operations on the Itanium 2 processor are given in Tables 3a and 3b.

Table 3a. Latency, Throughput, and Accuracy for IPF IEEE Square Root and Reciprocal Square Root Operations

Operation	Latency (clock cycles)	Throughput (clock cycles/ result)	Accuracy (ulps)
Single Precision Square Root	28	5.0	0.50
Double Prec. Square Root	36	6.5	0.50
Double-Ext. Precision Square Root	40	7.5	0.50
Single Precision Reciprocal Square Root	52 (sqrt+div)	8.5 (sqrt+div)	1.0 (sqrt+div)
Double Prec. Rec. Sq. Root	64 (sqrt+div)	11.5 (sqrt+div)	1.0 (sqrt+div)

Theoretical error bounds and maximum errors observed in testing are both included in Table 3b.

Table 3b. Latency, Throughput, and Accuracy for IPF Non-IEEE Square Root and Reciprocal Square Root Operations

Operation	Latency (clock cycles)	Throughput (clock cycles/ result)	Theoretical Accuracy (ulps)	Observed Accuracy (ulps)
Single Precision Square Root	20	3.0	0.9449	0.8194
Double Precision Square Root	32	5.5	0.5001	0.5000
Double-Extended Precision Square Root	NA	NA	NA	NA
Single Precision Reciprocal Square Root	20	3.5	0.9449	0.8860
Double Prec. Rec. Square Root	32	6.5	0.5031	0.5007

Conclusion

Several factors determined the implementation in software of the division and square root operations for Itanium processors. A first consideration was flexibility, as alternative algorithms can be easily substituted for the original ones, should this be needed. One example is using non-IEEE algorithms instead of IEEE-correct ones when accuracy can be relaxed for the benefit of better performance. Second, the software implementations of these operations inherit the high degree of pipelining in the basic floating-point multiply-add operations, leading to high-throughput algorithms. Third, as in typical applications division and square root are not extremely frequent, it may be that the die area on the chip that would be dedicated to hardware implementations of these operations could be better used for some other purpose.

The Itanium floating-point architecture was designed so that its high performance, accuracy, and flexibility characteristics make it ideal for technical and scientific computing. The present paper showed how software implementations of division and square root operations based on the fused floating-point multiply-add instruction support this goal. The principles used in designing algorithms for these operations were presented together with examples. Correctness proofs were outlined and underlying properties were stated. Non-IEEE algorithms were described in contrast with those that implement the division and square root operations mandated by the IEEE Standard 754-1985. Finally, performance numbers were presented.

Acknowledgements

Many people contributed to the development and verification of the algorithms described in this paper. Peter Markstein developed the original division and square root algorithms for software implementation. The author helped improve some algorithms, derived non-IEEE versions, performed mathematical correctness proofs, and determined special cases where the software algorithms are limited in their capabilities (not encountered in compiled code). John Harrison improved several algorithms (especially square root), and carried out automated proofs of correctness based on the existing mathematical proofs. Roger Golliver, Bob Norin, Cristina Iordache, and Shane Story reviewed the mathematical proofs of correctness or various related documents. Special thanks are due to Bob Norin for reviewing this paper.

References

[1] ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, NY, 1985

[2] Markstein P., Computation of Elementary functions on the IBM RISC System/6000 Processor, *IBM Journal*, 1990

[3] Intel(R) Itanium(TM) Architecture Software Developer's Manual, Vol 1-4, Intel Corp., 2003

[4] Cornea, M., Harrison, J., Tang, P, Scientific and Engineering Computation on Itanium™ Processors, Intel Press, 2002

[5] Cornea, M., Harrison, J., Tang, P, Intel Itanium™ Floating-Point Architecture, WCAE 2003, San Diego

[6] Divide, Square Root, and Remainder Algorithms for the Itanium Architecture, Intel Corporation, Dec. 2003,http://www.intel.com/software/products/opensource/libraries/numnote2.htm

[7] Cornea-Hasegan, M. and Golliver, R., Markstein, P. Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms, Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adelaide, 1999

[8] Harrison, J. Formal Verification of IA-64 Division Algorithms, Proceedings of the 13h International Conference TPHOLs 2000, Springer-Verlag, pp 234-251

[9] Non-IEEE Division, Square Root, Reciprocal, and Reciprocal Square Root Algorithms for the Intel Itanium Architecture, Intel Corporation, Dec. 2003, http://www.intel.com/software/products/opensource/libraries/numnote3.htm

[10] Harrison, J. Formal Verification of Square Root Algorithms, Formal Methods in System Design, Vol. 22, 2003, pp 143-153

Visual simulator for ILP dynamic OOO processor

Anastas Misev, Marjan Gusev
Institute of Informatics, PMF, Sts. Cyril and Methodius University
Arhimedova 5, PO Box 162, 1000 Skopje, Macedonia
anastas@ii.edu.mk, marjan@ii.edu.mk

Abstract

The purpose of this article is to provide an introduction to the SuperSim simulator for ILP processors as a teaching tool for computer architecture related courses. It presents the various aspects of the simulator, including the user interface, the instruction set, the configuration possibilities and applications. The main focus is on the educational usage of the simulator, through the experience gained in its actual application.

1. Introduction

Superscalar processors are one of the two major directions of ILP development. They issue multiple instructions per cycle, which results in complex decoding stage. This can lengthen the clock cycle or lead to multiple decoding cycles. Usually superscalar processors employ some kind of predecoding of instructions while they are fetched from memory to instruction cache. Pre-decode bits are attached to every instruction usually indicating the instruction class and the type of required resources.

Another aspect of multiple instruction issue is that can lead to higher performance, but at the same time it *amplifies* the restrictive effects of control and data dependencies on the processor performance. In order to reduce these effects, superscalar processors employ advanced techniques like register renaming, shelving and speculative branch processing.

Developing powerful microprocessors requires research in many different areas; such are electronics, algorithms, optimization, etc. Many new techniques are required for this process. To prove their efficiency, in a manner that allows grater freedom of research, simulation tools are very important.

The usage of simulators in the computer architecture courses has been proven as the best approach towards students' better understanding of the main architectural concepts. This is especially true for the visual simulators, since many internal features can be best understood through dataflow visualization.

2. Description of the SuperSim Simulator V 2.0

The basic considerations for designing the SuperSim Simulator were taken from the design space concept given by Sima et al [12], using similar experience of [2]. The previous versions of the simulator are covered in [7].

The main features of the SuperSim Simulator are:

- Running user code, written in its own pseudo assembler
- Syntax checking of the user code with error indication
- Extensive configuration
- Simulating a big range of processors, varying from simple RISC to advanced PostRISC
- Step by step execution
- Visual representation of each stage of the pipeline
- Fast, non visual mode for better performance
- Vast logging capabilities for performance analysis
- Detailed statistics

3. User Interface

The simulator has a very friendly user interface. It consists of several separate windows, including the code editor (Fig.1), runtime, configuration, statistics and other windows.

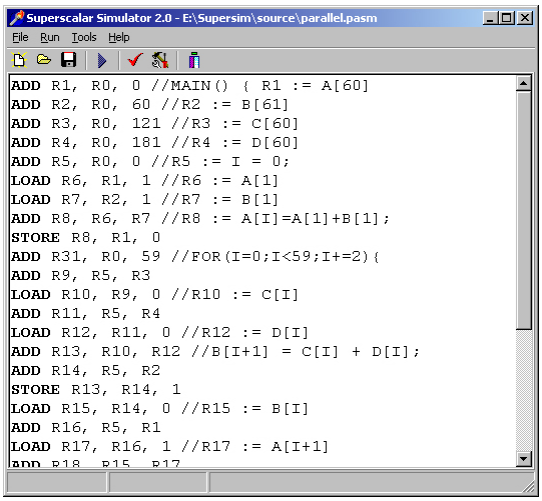


Figure 1: The code entry window

The code editor window enables the user to write its own custom code, using the pseudo assembler. The code can be saved into a file or loaded from one. Options available on this window include syntax checking with indication of possible errors and standard file management. Code can have inline

comments, separated with '//' from the instructions. Especially important is the configuration option, which defines the simulated execution environment.

4. Configuration

The configuration window consists of several major parts, each represented with a tab, as shown in fig. 2. The configuration enables choosing the number and the type of the execution units. The maximum number of execution units is 6, and the minimum is 1. Supported units are

- 1 multi cycle unit, for execution of multi cycle integer operations, like division or multiplication
- Up to 3 single cycle integer units, for execution of simple integer arithmetic
- 1 load/store unit for address calculation of the memory transfer instructions and
- 1 branch unit for calculation of the branch target addresses.

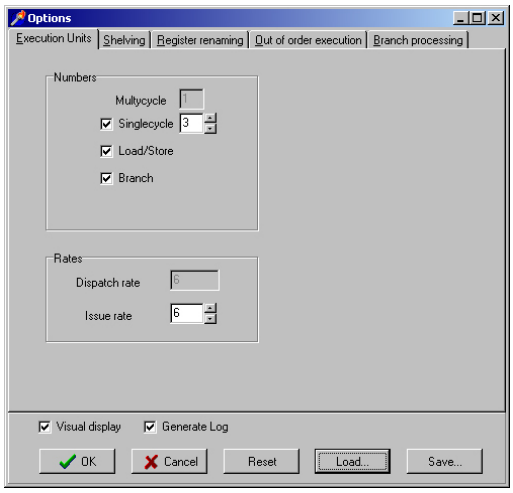


Figure 2: The options window

Only the multi cycle unit is mandatory, while the others can be added or removed. If a special unit is not used, for example the load/store unit, the multi cycle unit performs the operations.

The issue rate can also be configured on this tab, varying from 1 up to the total number of units used.

The second tab of the configuration window, shown in fig. 3, covers the use of shelving. When shelving is used, the user can select between central or dedicated reservation stations. For each station used, the number of entries can also be configured.

The next tab, fig. 4, is used for configuring the register renaming options of the simulator. If renaming is used, the number of rename buffers can be selected. Additionally, the access method for the renamed registers can be chosen from indexed or associative.

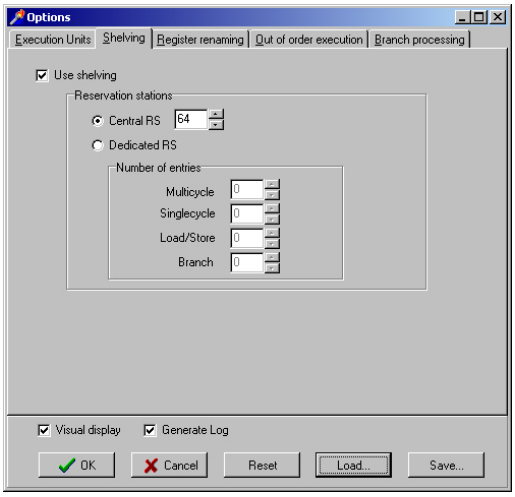


Figure 3: Shelving options

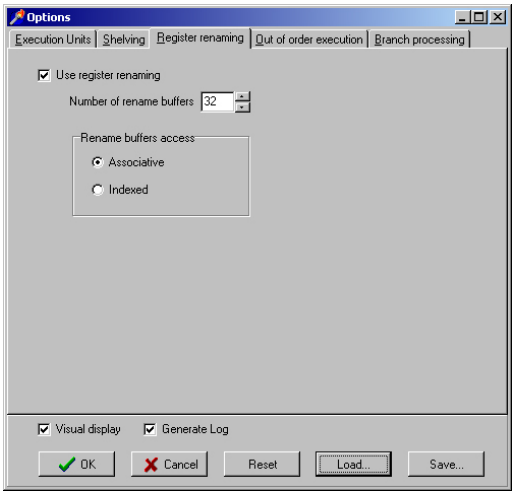


Figure 4: Register renaming options

The "Out of order" tab, fig. 5, enables the using of the out of order issue and dispatch. On the same tab, the user can adjust the number of entries in the Reorder Buffer (ROB).

The final configuration tab covers the branch processing used in the simulation, as shown in fig. 6. It can be blocking or speculative. When using speculative branch prediction, three modes are available: fixed, static and dynamic. The dynamic branch processing can be configured to use BTAC, BHT or both. It can also use global 2-bit history, for better prediction.

Other options available are turning on and off the visual simulation, which can increase performance and tuning on and off the logging option. When visualization is disabled, the number of clock cycles simulated per second is 7-10 times bigger.

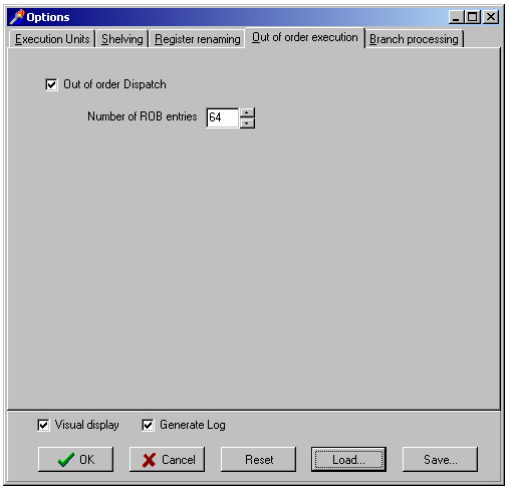


Figure 5: Out-of-order options

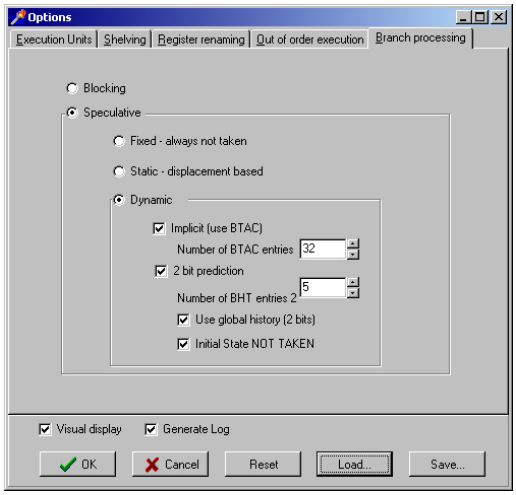


Figure 6: Branch processing options

The selected configuration can be saved into a file for later reuse, or loaded from one.

5. Runtime

The runtime environment greatly depends on the selected configuration. When full configuration is used, it looks like in fig. 7. The top part consists of some command buttons, among which are: "Close" for closing the runtime window, "Run" for running the simulation continuously, "Step" for executing cycle by cycle, "Pause" for pausing the simulation when ran in continuous mode.

Depending on the configurations some or all of the buttons in the upper right part will be enabled: "Show ROB" displays the ROB, fig. 8, "Show RF" displays the registry and rename registry file, fig. 9, "Show BT" displays the branch prediction tables window, fig 10, "Show DC" displays the data cache, fig. 11.

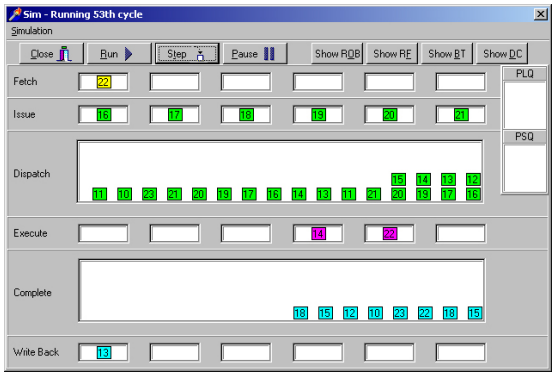


Figure 7: The runtime window

The rest of the window is divided into separate parts for each stage of the pipeline. Mandatory stages are Fetch, Issue, Execute and Write-back, while the other two, Dispatch and Complete are shown only if shelving and out-of-order execution are used, respectively. For each stage, a container represents the appropriate tables and/or buffers that hold the current instructions. In the upper left part, two separate containers represent the pending load and store queues.

The ROB window, shown in fig. 8 is used for monitoring the work of the reorder buffer. It has an entry for each instruction that has been issued and has not completed yet. Since the ROB is designed as a circular buffer, at also shows the head and the tail pointer in the buffer. Instructions are represented in different colors, depending on the stage of the pipeline they are in.

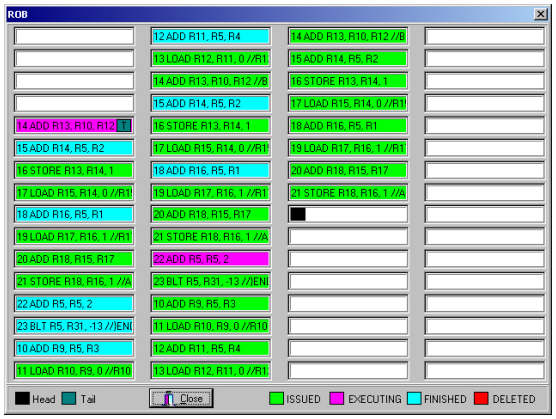


Figure 8: The ROB window

The registry file window, fig. 9, shows the state of both the architectural and the rename registers. On the left of the window, architectural registers are shown. For each rename register, there are three parameters shown: the number of the architectural register that is mapped to this rename register, the value (if calculated yet) and the latest bit.

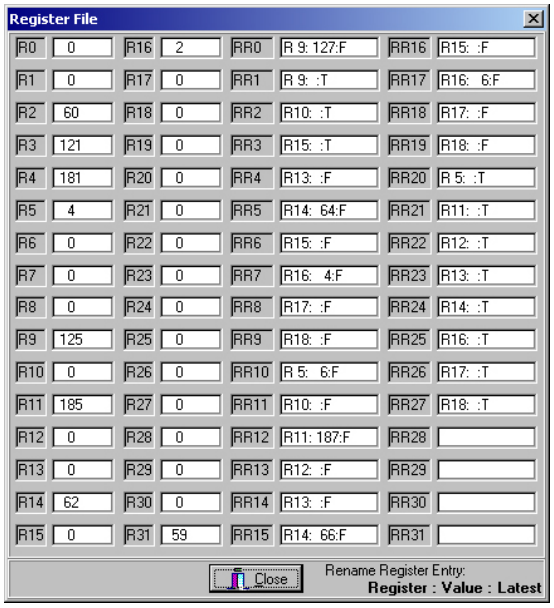


Figure 9: The Register file window

The branch tables' window, fig. 10, is used for monitoring the state of the branch prediction tables. Depending on the configuration, one or two tables are shown. They are the BHT and/or the BTAC.

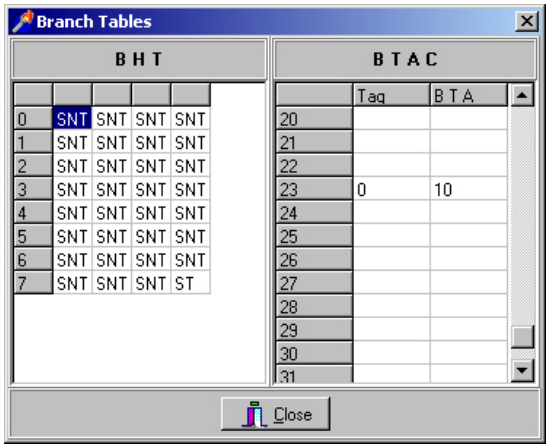


Figure 10: The Branch tables' window

The data cache window shows a map of the data memory, with each entry representing a 4-byte word, as shown in fig.11.

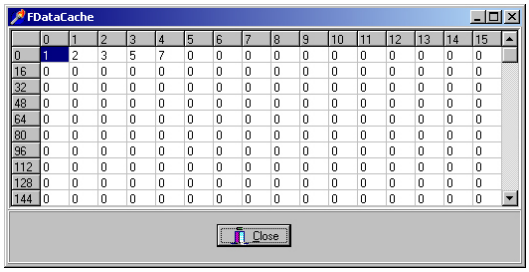


Figure 11: The Data cache window

The statistics window, shown in fig. 12, gives a detailed statistics of the simulated code and configuration. The figures include the total number of executed instruction of each type, branch statistics and prediction accuracy measures, the flow of the instruction through each stage and both memory and register data dependencies. Some advanced measures are also included, like the average number of cycles required for flushing the processor and average number of register wasted when a miss-prediction occurred.

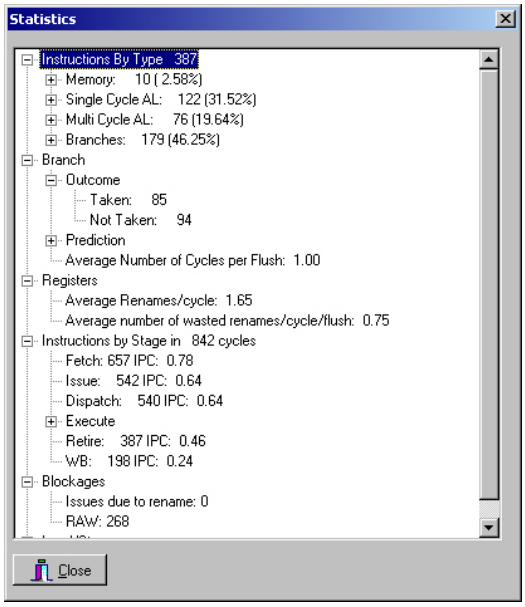


Figure 12: The Statistics window

6. Internal design

The instruction set of the simulator represents a subset of the standard modern instruction sets [6,9,11], and contains the instructions shown in table 1.

The simulator simulates a processor performing 32-bit integer operations with block diagram presented in fig.13. The floating-point part is not considered in this project. Most of the current PostRISC features [6, 9, 11] can be simulated using the SuperSim, including out-of-order issue, register renaming, shelving, branch prediction etc.

Supported memory addressing modes are displacement and indexed based [6]. While the same mnemonic is used for both modes, instruction processing is different depending on the mode. The memory is divided into instruction cache and 1024 locations of 32-bit words data cache. The memory is aligned on a word (4 bytes) boundary and all memory access instructions refer to a word address.

The maximum number of execution units is six (refer to fig. 2). Instructions that take multiple clock cycles to execute, i.e. the 'mul' instruction, are executed in the multi-cycle, which is obligatory. Optionally there can be up to three single-cycle execution units for instructions like 'add', or 'sub' that

Table 1: Instruction set

Instruction	Semantics	Comment
ADD R1, R2, R3	Regs[1] = Regs[2] + Regs[3]	The third operand can be either register, or a constant
SUB R1, R2, R3	Regs[1] = Regs[2] - Regs[3]	
AND R1, R2, R3	Regs[1] = Regs[2] & Regs[3]	
OR R1, R2, R3	Regs[1] = Regs[2] | Regs[3]	
NOT R1, R2, RX	Regs[1] = ! Regs[2]	
SHL R1, R2, R3	Regs[1] = Regs[2] SHL Regs[3]	
SHR R1, R2, R3	Regs[1] = Regs[2] SHR Regs[3]	
MOD R1, R2, R3	Regs[1] = Regs[2] Modulo Regs[3]	
DIV R1, R2, R3	Regs[1] = Regs[2] / Regs[3]	
MUL R1, R2, R3	Regs[1] = Regs[2] * Regs[3]	
LOAD R1, R2, 200	Regs[1] = Mem[Regs[2] + 200]	Reads a word from memory
STORE R1, R2, 150	Mem[Regs[2] + 150] = Regs[1]	Writes a word in memory
BEQ R1, R2, 200	if (Regs[1]=Regs[2]) IP = IP+200	The third operand can be either register, or a constant
BNE R1, R2, R3	if (Regs[1]!=Regs[2]) IP = IP+Regs[3]	
BGT R1, R2, 200	if (Regs[1]>Regs[2]) IP = IP+200	
BLT R1, R2, R3	if (Regs[1]<Regs[2]) IP = IP+Regs[3]	
BGE R1, R2, 13	if (Regs[1]>=Regs[2]) IP = IP+13	
BLE R1, R2, R3	if (Regs[1]<=Regs[2]) IP = IP+Regs[3]	

take one clock cycle to execute, one load/store unit for handling memory access, and one branch unit dedicated for branch processing. When there is no available corresponding execution unit, the instructions are executed in the multi-cycle unit, which provides the functionality of all execution units. The number of execution units determines the dispatch rate so there are no restrictions about the instructions being dispatched. Issue rate can be set up to the dispatch rate [12].

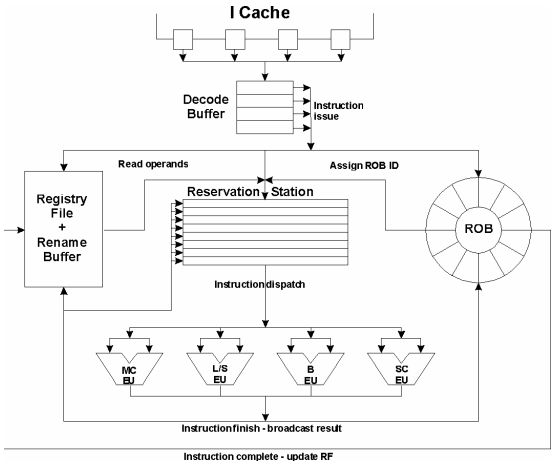


Figure 13: Block diagram of the simulator

The use of RS is optional with the possibilities shown in Fig.3. When selected, there is a choice between central or dedicated RS. Dedicated RS are placed in front of every execution unit, so the issue stage directs every instruction to the corresponding RS. In the case of central RS there must be additional logic to determine the execution unit where the instruction is dispatched. Additional requirement in the case of central RS is the number of output and input ports, which have to be larger unlike the case of dedicated RS.

Register renaming is implemented by separate register rename file (also known as rename buffer) [1,3,5,12,13,14,15,16]. The access to the rename buffer can be associative or indexed. When using associative access, there may be multiple instances of renames of one architectural register with separate notion of the last rename. In contrast only one rename per architectural register may exist with indexed access.

Out of order execution refers to whether instructions are issued out of order or dispatched out of order. When shelving is enabled instruction issuing is in order, while instruction dispatch is out of order. This design option is realized since the issue stage does not check for dependencies so there cannot be pipeline blockages due to dependency. If shelving is disabled, the only possibility is out of order instruction issue. Fig.5 shows the possible options about out of order execution [15,16].

Branch processing options are shown in Fig.6. If branch processing is speculative, predictions about branch instructions can be: fixed "always not taken", static displacement based, or dynamic with optional use of BTAC, BHT or 2 bit global history register. In the latest case BTAC is used only for recent taken branches and the use of either BTAC or BHT is obligatory if dynamic prediction is selected [17]. Additionally, when BHT is used, global BHT can be activated and the initial state can be set.

7. Implementation

The SuperSim simulator is developed using Borland Delphi and targets 32-bit Windows platforms. It has full object oriented design, with each phase in the pipeline represented by its own object. Each object has a public interface for realization of communications between the stages in the pipeline. The object architecture makes upgrading easy and intuitive.

The performance in the sense of simulated clock cycles per second varies depending on whether the visualization is on or off. When off, it simulates around 100 clock cycles per second, measured on PIII working on 650MHz. If visualization is on, this number is 7-10 times smaller.

8. Teaching ILP using the simulator

The SuperSim simulator can be equally well used in research and in education. Its visual interface helps students to understand the functionality of a RISC or PostRISC, get familiar with the basic concepts of ILP and practice their assembly language programming skills.

The simulator executables, with sample configurations and programs are available to the students through the computer architecture courses web sites. After the initial introduction of the basic simulator elements and performing some simple examples, each student is assigned a project. The project consists of writing a small assembly program (searching, sorting, prime number search, SCD, matrix operations, linked list operation, conversions etc.) and performing some analysis of the superscalar techniques on the program execution. The analysis concerned the performance impact of the key ILP factors like the number of execution units, number of register available for renaming, type of the reservation stations, ROB entries, loop unrolling and branch prediction techniques. The deliverables were the program itself and a paper explaining the results of the analysis.

The results of this method of teaching ILP were more than satisfactory. The students' interest for the course was bigger and the achieved results were better then before the introduction of the simulator [10].

9. References

[1] Austin,T., Sohi,G.S., (1992), *Dynamic Dependency Analysis of Ordinary Programs,* Proc.19th Int.Conf. on Computer Architecture, ISCA-19.

[2] Burger, D., Austin, T., (1997), *The SimpleScalar Tool Set, Version 2*, Technical report of the University of Wisconsin-Madison, Computer Science Department.

[3] Conte T.M. (1996), *Superscalar and VLIW processors*, in *Parallel and Distributed Computing Handbook*, ed. by A.Y.Zomaya, McGraw Hill.

[4] Farcas, K. et all, (1995) *Register File Design Considerations in Dynamically Scheduled Processor,* WRL Technical report 95/10, Paolo Alto, California.

[5] Gonzalez, J. and Gonzalez, A., (1995) *Identifying Contributing Factors to ILP*, Univesitat Politecnica de Catalunya, Barcelona, Spain.

[6] Gusev M. (1998), *Contemporary Computer Systems,* Medis, Skopje, Macedonia.

[7] Gusev, M., Misev, A. and Popovski, G. (1998) *Simulation of Superscalar Processor,* proc. of ITI'98, Pula, Croatia.

[8] Gusev,M, Misev.A and Popovski.G (1999), *Memory Address Dependencies,* ITI-99 pp.191-196.

[9] Hennessy J.L., Patterson D.A. (1998), *Computer Organization and Design: The Hardware Software Interface*, sec. edition, Morgan Kaufmann Publishers, San Francisco, California.

[10] Misev, A., Gusev, M., (2004), *Simulators for ILP courses*, proc. of TEMPUS CD JEP 16160 workshops, Nish, Serbia and Montenegro.

[11] Patterson, D and Hennessy, J. (1996) *Computer Architecture A Quantitative Approach,* second edition, MKP, San Francisco, California.

[12] Sima D. at al. (1997), *Advanced Computer Architectures: A Design Space Approach*, Addison Wesley Longman, Harlow, England.

[13] Smith, J. and Pleszkun, A. (1988) *Implementing Precise Interrupts in Pipelined Processors,* IEEE Transactions on computers, vol. 37, no. 5, p.562-573.

[14] Tyson,G., Austin,T. (1997), *Improving the Accuracy and Performance of Memory Communication Through Renaming*, Proc.30th Ann. Int. Symp. on Microarchitecture, MICRO-30.

[15] Wall, D. (1993) *Limits of Instruction-Level Parallelism,* WRL Research report 93/6, Paolo Alto, California.

[16] Wall, D. (1994) *Speculative Execution and Instruction-Level Parallelism,* WRL Technical Note 94/42, Paolo Alto, California.

[17] Yeh, T.Y., Patt, Y. N., (1992) *Alternative Implementations of two-level Adaptive Branch Prediction*, Proceedings of 19th Int. Symposium on Computer Architecture (ISCA) pp.124-134.

WebMIPS: A New Web-Based MIPS Simulation Environment for Computer Architecture Education

Irina Branovic, Roberto Giorgi, Enrico Martinelli
University of Siena, Italy
{branovic,giorgi,enrico}@dii.unisi.it

Abstract

We have implemented a MIPS simulation environment called WebMIPS. Our simulator is accessible from the Web and has been successfully used in introductory computer architecture course at Faculty of Information Engineering in Siena, Italy. The advantages of the Web approach are immediate access to the simulator, without installation, and a possible centralized monitoring of students' activity. WebMIPS is capable of uploading and assembling the MIPS code provided by user, simulating a five-stage pipeline step by step or completely, and displaying the values of all registers, input and output data of all pipeline elements.

1. Introduction

The study of computer architecture is a challenging field because of the high complexity involved in any computer system. To ease this complexity, different tools have been developed allowing architectures to be simulated and modified. This approach is beneficial to students approaching computer architecture for the first time, because it allows them to see the execution of actual assembly programs in the architecture. One important step is that the student makes use of simulation tools to understand concepts otherwise difficult to comprehend. Our experience, started with JCachesim cache simulator [1], indicates that Web-based lab exercising is effective, sometimes even more interesting than traditional teaching to our students. We are not alone in trying to make computer architecture education more interesting to students, as can be seen in [5], where the authors used animation for this purpose.

An extensive survey of computer architecture simulators is given in [8]. For computer architecture education, especially interesting is the category of intermediate-level simulators, targeted at students that have some background in computer architecture and need a simulator that covers the principles in more detail, but are not ready for the simulator that captures all the features of the current state-of-the-art in computer research. The simulators in this category attempt to illustrate and teach two general principles: the instruction set architecture and the micro-architecture.

In many universities, MIPS architecture is studied because it is a RISC architecture that makes understanding abstract concepts of computer design easier. Another advantage of MIPS ISA is that it is used in textbooks [2], [3], which represent a reference material for teaching computer architecture in many universities, which is also the case for our faculty. There are three widely used MIPS architecture simulators: SPIM, WinDLX and MIPSim.

SPIM [4] is an assembly language simulator for the MIPS (R2000/R3000) processor that has both a simple terminal interface and a visual, window-based interface. It implements almost entire MIPS assembler-extended instruction set (detailed SPIM description can be found in [3] with more documentation available online [4]). SPIM was extensively used in our teaching, however it lacks pipeline modeling.

WinDLX [9] and MIPSim [10] are pipeline simulators developed at the Vienna Institute of Technology and were described by authors in [11]. WinDLX models the pipeline of the MIPS-like DLX architecture described in [2]. It allows for displaying and modifying all of the information relevant to the CPU (pipeline, registers, I/O, memory), enabling/disabling pipeline forwarding, changing memory size. MIPSim [10] models the MIPS architecture as in [3], with the possibility of changing memory content, but without hazard detection and forwarding units in the pipeline.

We have decided to make a five stage MIPS pipeline simulator capable of displaying the status of almost all hardware units (more than 25) in the MIPS pipeline model, as well as hazard detection and forwarding in the pipeline. Instead of improving MIPSim, which also would have been a valid alternative for our goal, we decided to create a completely new simulator that can be executed from the Web browser window. Our simulator, called WebMIPS, eases the process of learning assembly coding, mastering pipeline, control, and datapath design. However, its major advantage is the immediate accessibility to students, without any prior installing, and the possibility of monitoring their activity over the Web.

The name WebMIPS indicates that the simulator is designed for Web use, and indeed it is written in ASP language [7] and can be started by opening a simulator Web page [6]. Another advantage of the Web based service is that the user are not required to have any special operating system for accessing this software.

WebMIPS does not support the complete MIPS instruction set; the user that wants to write assembly programs on its own must consult the list of supported instructions in order to simulate the code. Since our intention was not implementing a whole assembler, the simulator supports only the basic set of instructions, which were studied during the introductory computer architecture course.

The user can load (copy/paste) MIPS assembly file or use one of the "load-and-play" (built-in) assembly examples to follow its execution in simulator. WebMIPS is not a real assembler; however, it is able to recognize if there are errors in the provided code, and to display the line with the error. The simulator is also able of displaying the program execution step-by-step or all at once. In step-by-step mode the user can follow advancing of instructions in each stage of the pipeline, and by clicking on the constituting elements of the pipeline can see the corresponding values, input and output signals in every clock cycle. WebMIPS has forwarding always enabled, resolves pipeline hazards and displays the contents of hazard detection and forwarding units in the pipeline.

2. Detailed description of WebMIPS simulator

2.1 General structure

WebMIPS is a Web application and it is executed on remote servers in multiuser mode (users can execute different code at the same time). To avoid blocking of the system in case of infinite loops, erroneous references to memory and other common programming errors, we limited the execution of each uploaded program to 1000 clock cycles. On the server, all simulation parameters can be configured.

When trying to execute unsupported assembly instructions, an error is displayed and WebMIPS indicates the corresponding line number. In standard assembly language the use of directives .text and .globl is allowed, and in this case the first instruction to be executed corresponds to the .globl label. The end of execution is not specified by syscall 10, instead in WebMIPS the execution stops at the last code line.

2.2 Loading of the code

To offer the possibility of loading proprietary code to the users, we made an ASP page section, where it is possible to program in MIPS assembly and to verify whether the code is correct. By clicking on the button "Load/Reload Program" in the upper part of the WebMIPS browser window (Figure 1), the MIPS assembler is activated.

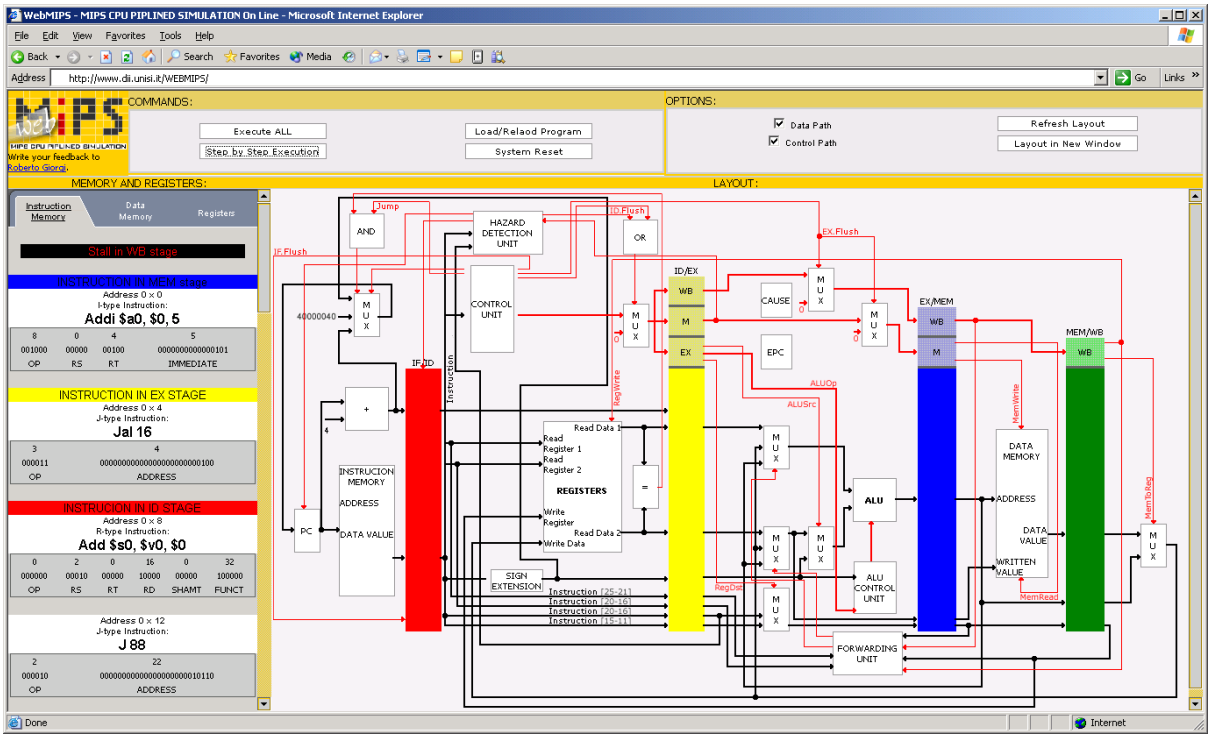


Figure 1. WebMIPS window during execution. Note the Load/Reload Program button in the upper part. The wires can be hidden to have an easier reading of the CPU units.

Not all options of the real assembly were implemented, since our goal was to demonstrate the execution of base instructions explained during our computer architecture course. However, almost all MIPS instructions can be written by combining the implemented instructions. We included a set of simple assembly programs with the scope of demonstrating its execution in MIPS pipeline.

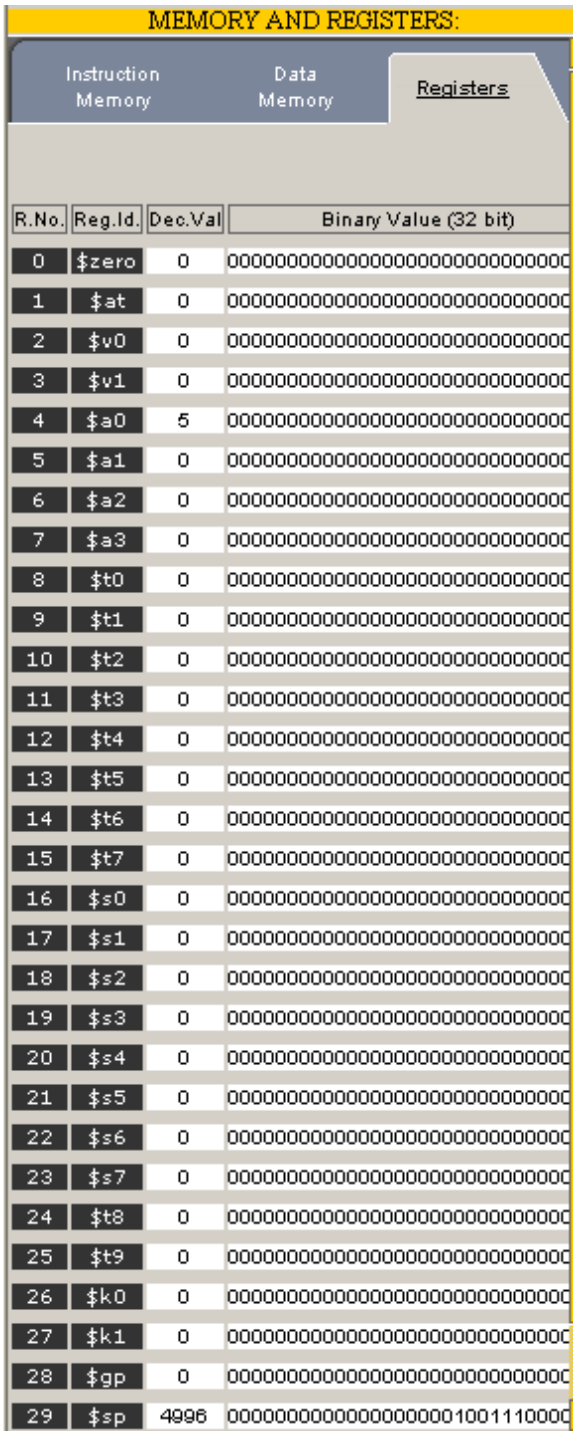


Figure 2: Registers during execution.

The functioning of our simulator can be easily understood by using some of the simple built-in (called "load-and-play") programs. The simulator keeps track of the code in execution and it can be easily modified in any moment by clicking on the "Load/Reload Program" button.

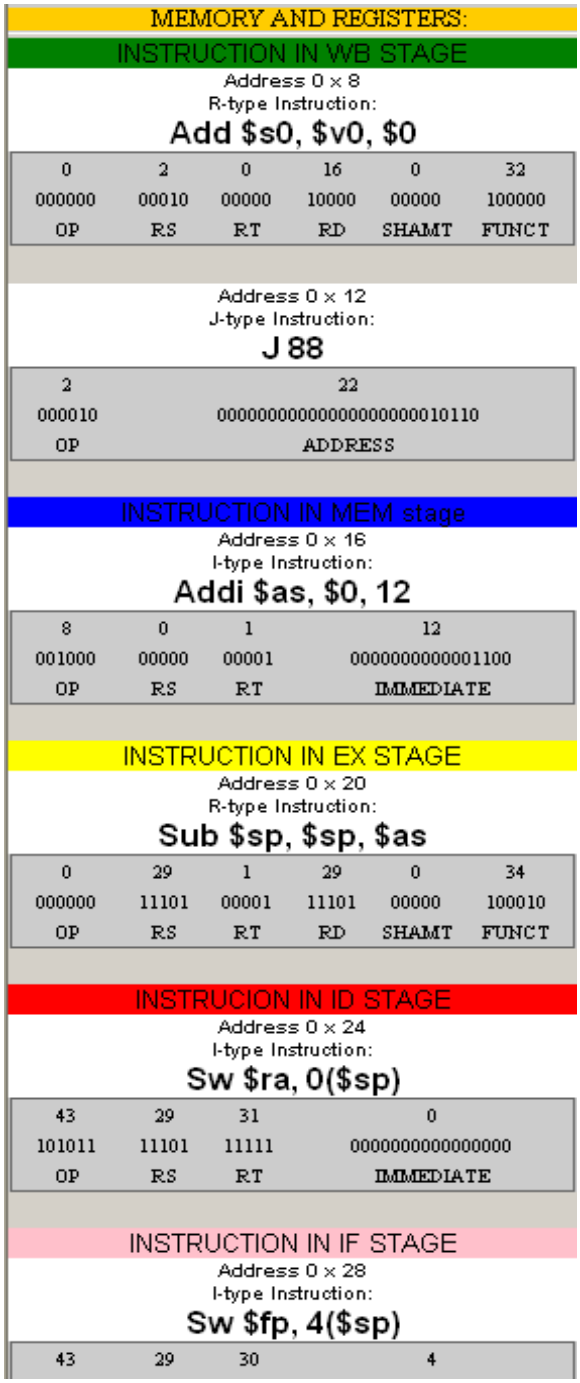


Figure 3: Instruction memory in the middle of execution.

2.2 Program execution

In order to allow users to follow program execution, the left part of the browser window is dedicated to information regarding register file, data and instruction memory. Instruction memory displays the mnemonic, memory address, type, binary translation, symbolic representation, field values and current position in the pipeline for every instruction in execution (Figures 2, 3).
The page displaying data memory can visualize single words, a word interval, or the whole memory contents. The register page demonstrates the binary content of 32 MIPS registers, which can be identified either by register number or their symbolic identifier.

The central part of the browser page is dedicated to five-stage MIPS pipeline. Since the major scope of the simulator was to facilitate understanding of pipeline principles, a user can click on any desired element of the pipeline (for example, ALU, hazard detection unit, or even a simple multiplexer) to show its input and output data. A good feature of WebMIPS is the possibility of tracking every instruction in each pipeline element by simply looking at the central graphical screen. Additionally, displaying of control/data wires can be turned on/off using a corresponding check-box.

Once loaded, a program can be executed in two modes: step-by-step or completely. In step-by-step mode, after each clicking of the "Step-by-Step Execution" the pipeline stages are updated and the user can see the changes in memory and detailed pipeline logic. After the execution has completed, the total number of clock cycles is calculated and displayed in the left-hand menu. Complete execution of the program should be is used only for verifying the correctness of the assembly code.

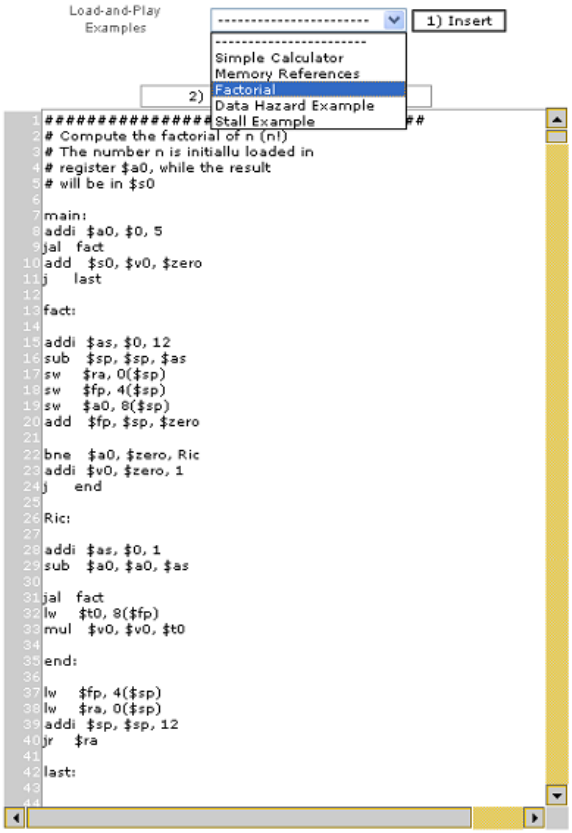


Figure 4: A simple calculator program (among built-in examples) loaded.

2.3 Analyzing pipeline data hazard and forwarding

In the implementation, branch decision is in the Decode stage of the pipeline to save one cycle. Data hazards created in this way are detected in hazard detection unit, and resolved via forwarding unit, which are shown in graphic representation of the pipeline. Among "load-and-play" programs user can find a simple four-operation calculator; the loading of this example is shown in Figure 4. We will use this simple example to illustrate the functioning of hazard detection and forwarding in the pipeline. The top of the left-hand menu lists the pipeline stages in stall during the execution of the program (Figure 5).

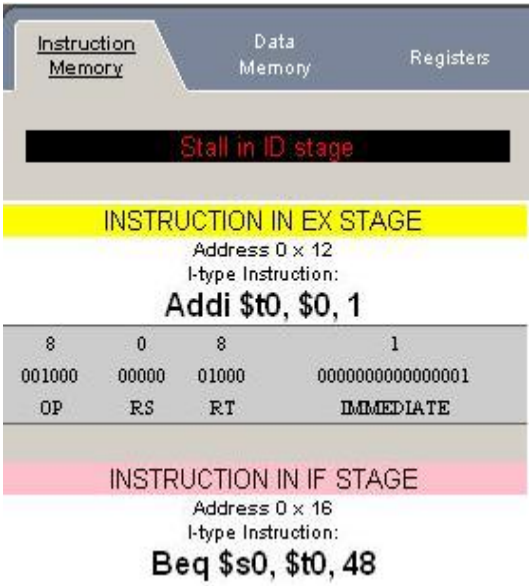


Figure 5: The top of the memory window displays a stage in stall.

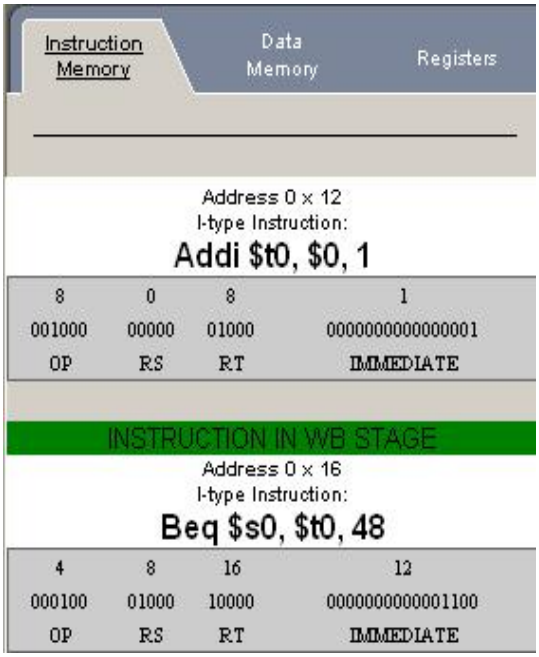


Figure 6: Stall passed through the pipeline.

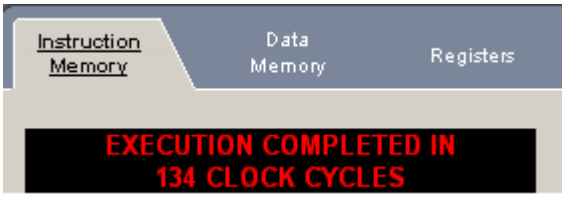


Figure 7: When the execution finishes, the top of the memory window shows the total number of clock cycles.

The pipeline with data hazard resolved is displayed on Figure 6. When the execution of the program finishes, the simulator displays total number of cycles (Figure 7). By clicking on the hazard detection and forwarding units in the pipeline a user can see the corresponding signals and follow the propagation of the stall through the pipeline (Figure 8).

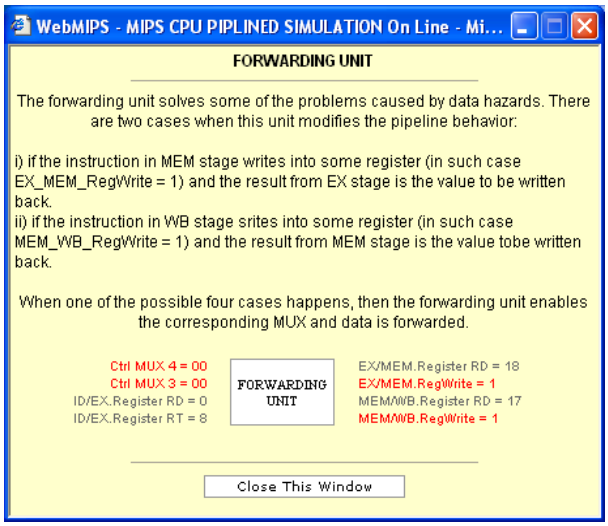
Conclusions

We have implemented a Web-based MIPS pipeline simulator called WebMIPS. Our simulator is publicly accessible and it displays execution in the Web browser window, and is capable of detecting and resolving hazards in the pipeline. The WebMIPS software was used in introductory computer architecture course at University of Siena, Italy as an auxiliary resource for explaining pipeline principles.

We received a good feedback from our students, who also appreciated its availability from any client computer (independently from the installed operating system), the possibility of executing on any Internet-enabled PC without prior installation, and its ease of use. Further plans for WebMIPS development include extending the supported instruction set to include all MIPS (R2000/R3000) instructions.

Acknowledgements

We are particularly grateful to the students: Mirko Casini, Riccardo Donati, Alem Gracic, Luca Peruzzi for the initial implementation and the testing of WebMips.



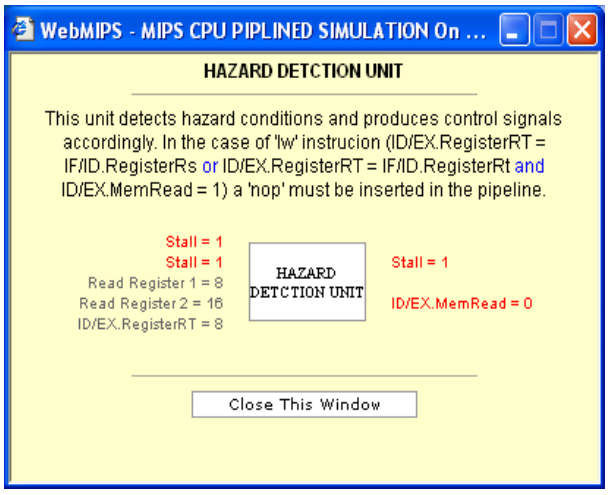


Figure 8: Details on forwarding and hazard detection in the pipeline can be seen by clicking on the corresponding unit.

References

[1] I. Branovic, R. Giorgi, A. Prete, Web-based training on computer architecture: The case of JCachesim, *Proceedings of the Workshop on Computer Architecture Education*, pp. 56-60, May 2002, Anchorage, Alaska.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach, 3rd edition*, Morgan Kaufmann Publishers, 2002.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd edition, Morgan Kaufmann Publishers, 1997.

[4] SPIM simulator Home Page, http://cs.wisc.edu/~larus/spim.html

[5] M. Brorsson, MipsIt - A Simulation and Development Environment Using Animation for Computer Architecture Education, *Proceedings of the Workshop on Computer Architecture Education*, pp. 65-72, May 2002. Anchorage, Alaska.

[6] WebMIPS Home Page, http://www.dii.unisi.it/~giorgi/WEBMIPS/

[7] C. Payne, *Teach Yourself ASP.NET in 21 days*, 2nd edition, SAMS, 2003.

[8] W. Yurcik, G. S. Wolffe, M. A. Holiday, A Survey of Simulators used in Computer Organization/Architecture Courses, *Proceedings of Summer Conference on Computer Simulation*, pp. 524-529, Orlando, Florida, July 2001.

[9] WinDLX Simulator Download Page, ftp://ftp.mkp.com/pub/dlx/

[10] MIPSim Simulator Download Page, http://mouse.vlsivie.tuwien.ac.at/lehre/rechnerarchitekturen/download/Simulatoren/

[11] H. Grünbacher, M. Khosravipour, WinDLX and MIPSim Pipeline Simulators for Teaching Computer Architecture, *Proceedings of IEEE Symposium and Workshop on Engineering of Computer Based Systems*, pp. 412-417, Friedrichshafen. Germany, March 1996.

DARC2: 2nd Generation DLX Architecture Simulator

Roger Luis Uy
De La Salle University
2401 Taft Avenue
Malate, Manila
+(632) 524-0402
uyr@ccs.dlsu.edu.ph

Marizel Bernardo
De La Salle University
2401 Taft Avenue
Malate, Manila
+(632) 524-0402
marizel@yahoo.com

Josiel Erica
De La Salle University
2401 Taft Avenue
Malate Manila
+ (632) 524-0402
Osie_22@yahoo.com

ABSTRACT
Renewed interest in computer architecture education in our university started three years ago. Since then, research framework in computer architecture has been established with emphasis on simulation of different computer architecture concepts. One of the concepts, which have generated a lot of excitement, is the topic on pipelining. Our research group had already developed a pipeline simulator based on the DLX architecture called DARC [1]. The simulator was used as a supplementary tool for both undergraduate and graduate students. It was received favorably and at the same time, they gave feedbacks and suggestions on improving the simulator. With those suggestions, DARC2, the 2nd generation pipeline simulator based on DLX architecture was developed. This paper describes the DARC2 system.

Categories and Subject Descriptors
K.3.2 [**Computer & Information Science Education**]: – *computer science education, computer architecture education, DLX Architecture.*

General Terms
Experimentation, Human Factors

Keywords
Computer Architecture, undergraduate teaching, graduate teaching, pipelining, DLX Architecture

1. INTRODUCTION
In the past, computer architecture education was given less emphasis in our university. But this has change in recent years due to the following reasons:

- Introduction of the position of Academic Area Chair. The Academic Area Chair is in charge of the development of the curriculum in their respective area. In the past, faculty members who were assigned to teach a particular subject in an academic area, developed their own syllabi and course contents. Thus, there is no continuity in the development of this area. With the academic area chair, he is task to develop a research framework, which will serve as a roadmap for the development of its area. He is also task to make sure that all appropriate textbooks and reference materials are up-to-date. Computer Architecture is classified as one academic area.
- Adoption of the computer architecture book "Computer Architecture, A Quantitative Approach" by Patterson & Hennessy [2]. The university has adopted many references but not textbook for computer architecture. Finally, in our opinion, a good book in computer architecture.

The current research framework of computer architecture in our university is focused on the development of simulation tools for the different concepts of computer architecture. In an environment where financial resources are limited, finding less expensive alternatives is always welcomed. With simulators, a quality-learning environment that is equivalent to the actual system itself is presented to students without incurring additional expenditures. Initial project is centered on the pipelining concept based from the DLX architecture. Many students are fascinated with the concept though they have a hard time visualizing them. Initially, simulation is through Microsoft® Excel file, then some students volunteered to write a module, then another. Eventually, a research group was formed to develop a "full-blown" pipeline simulator based on DLX architecture called DARC. Though there are simulation tools on this concept, but each institution is unique in their learning needs and the learning process of developing a pipeline simulation more than justify the development of our own pipeline simulation project.

2. The DARC2 ORIGIN: DARC
DARC2 is the 2nd generation of DLX Architecture Pipeline Simulator (DARC). DARC is a windows-based software system with a built-in text editor. It simulates a DLX code segment using different pipeline algorithms. We defined Pipeline 0 as the un-pipelined version of the algorithm; Pipeline 1 as the pipelined version and Pipeline 2 as the modified pipelined to minimized branch hazard. All of the algorithms are based from [2]. As an added bonus, dynamic scheduling algorithms - Scoreboarding and Tomasulo, are also provided. Users may enter as much as 1,024 instructions, with the provision for saving the program. It incorporates a compiler for identifying syntax errors, and a help

file that aids the user in correcting such errors. Pipelining results obtained are displayed through a trace of the pipeline stages, while dynamic scheduling algorithms are processed in the standard table form.

The system uses two simulation modes: one-pass and stepwise. One-pass mode allows continuous execution. Stepwise mode, on the other hand, allows instructions to be simulated one at a time. The simulator can be configured to support either shared or separate memory as option to illustrate structural hazard. Forwarding and non-forwarding are used as option to visualize data hazard. While control hazard can be resolved using pipeline freeze, predict-not-taken and pipeline 2. During simulation, hazards encountered are displayed and explained to the user.

As seen, DARC demonstrates great aid in the study of the DLX architecture. The simulator was initially offered to around 160 students in four separate classes. It was a sighed of relief for them since they could now visualize and experiment different options and situations in pipelining. They also suggested several changes, correction and enhancements. One such change is that the memory contents should be displayed as 32-bit word instead of displaying it as byte. Another suggestion is that there should be a provision for breaking out of infinite loops. The system also does not support floating-point values in IEEE standards. Students also reported some inconsistencies with the results obtained. They also suggest that besides that standard "pipeline" view, they could also visualize the flow of data to the individual components of the DLX architecture (i.e., pipeline registers, program counter) as well as the generation of the control signals. These suggestions warrant a major design of the simulation system. Thus, a new version of pipeline simulator is created – DARC2.

3. DARC2: DLX ARCHITECTURE SIMULATOR 2

DLX Architecture Pipeline Simulator 2 (DARC2) is an improved version of DARC. It is a windows-based system that utilizes graphical user interface to simulate both DLX pipelining algorithms and dynamic scheduling algorithms. Unpipelined instruction execution, Pipeline 1 and Pipeline 2 algorithms are now illustrated both through the standard tracing of the pipeline stages and through an animated diagram of the DLX data path. The animation shows the data flow through the major components of DLX architecture. DLX instructions are showed as they are processed, together with the control signals associated with them. Through the animation, the user is informed on how each internal component works and on the actual process of passing along data from each unit. To ensure consistency, all the diagrams are presented in the same manner as in [2], matching both the look and the function of the architecture. This improves the learning process and the usability of the simulator. On the other hand, results obtained through the dynamic scheduling algorithms – Scoreboarding and Tomasulo – are shown in the usual tabular form.

As with the original version, there are two modes of simulation: one-pass and step mode. Apart from this, the user also chooses among other simulation options specific of each pipelining algorithm. These options are forwarding, pipeline freeze, pipeline flush, predict-not-taken, and the use of unified or separate memory. The chosen options are greatly important since they set the conditions to be followed during the simulation process.

Moreover, users are given the freedom to choose which memory address to be assigned as base address for the instructions in the program code. This is especially useful when unified memory is utilized. The system also enables the user to change the value of the registers and the memory at any point during the simulation. With this, users can test different register values without having to construct another program code. Changes made would apply to DLX instructions that are not yet decoded in the Instruction Decode (ID) stage.

Another feature integrated to the system is the option for backtracking. DARC 2 allows values stored in the pipeline registers, as well as the other registers to be viewed at any clock cycle. From the final trace of the pipeline diagram, the user may choose the required clock cycle and the corresponding data path diagram showing data stored in the different components is displayed. During the simulation process, the system also gathers statistical data that aids in determining the effectiveness of the DLX program code used as input. The number of hazards encountered is considered and updated per clock cycle of the simulation, after which it is displayed.

A text editor, syntax checker and help system are also incorporated into the system. The text editor allows the user to key in the DLX instruction code to be simulated. The syntax checker checks the instruction code for syntax errors and notifies the user if such are encountered, providing opportunity to correct them. The help function provides topics and references related to the DLX Architecture.

4. SYSTEM STRUCTURE

The DARC 2 system is an integration of several major components that work together to achieve the requirements and specifications of the simulator. The interaction of these components - the text editor, the assembler and the function manager – is illustrated in the system's block diagram. A more detailed description of each of these components follows.

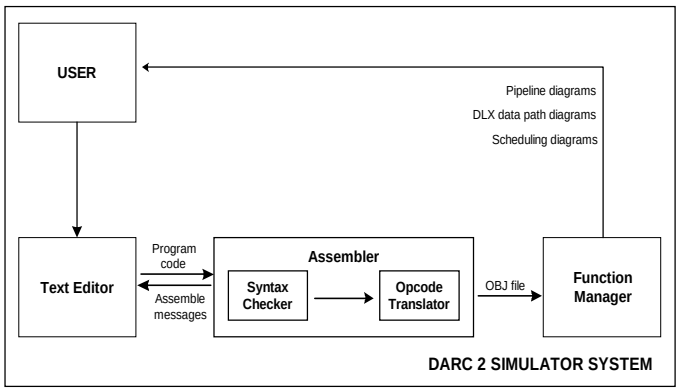


Figure 1. Block Diagram of DARC2

4.1 Text Editor

The built-in text editor of the system, as illustrated in Figure 2, accepts a DLX assembly code consisting of at most 1,024 instructions. The program code comprises of instructions written in DLX mnemonics, constructed either from scratch or chosen among the initial sample programs installed in the system. The

text editor also allows the user to save written assembly codes for later use.

The program code entered contains not only DLX instructions, but also data variable declarations as well. The user defines all variable declarations in the. DATA segment. All DLX instructions are included in the separate .CODE segment. Immediate addresses included within the instruction code itself may be in decimal, binary or hexadecimal form. For decimal, the format is *#immediate*. For binary, the format is *$immediate*. Unless written in any of these formats, the immediate is considered as hexadecimal.

After the user has keyed in the DLX instruction code through the text editor and clicking the Assemble button, illustrated in Figure 2, the assemble options window is displayed. The user inputs the base address of the instructions in the program code to be simulated. Upon which, the program code together with the value set as base address is passed on to the assembler for processing.

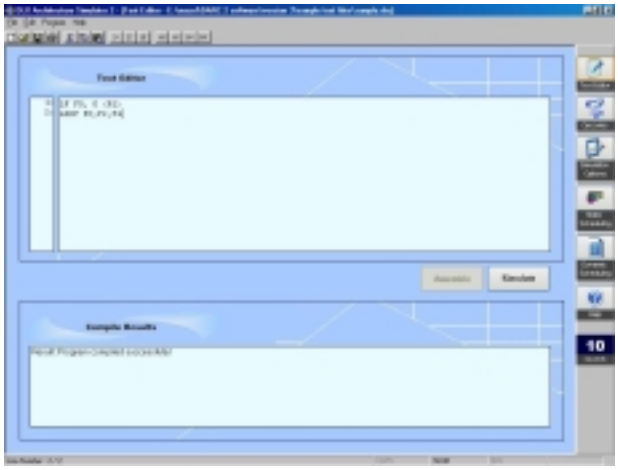


Figure 2. DARC 2 Text Editor

4.2 Assembler

The assembler module handles the processing and translation of the DLX mnemonics. The module is comprised of two sub-modules: the syntax checker and the opcode translator. After the user has created the instruction code through the text editor, the syntax checker checks the code for syntax errors. If no errors are identified, the program code is then passed on to the opcode translator. However, if errors arise, assembling is considered unsuccessful and error messages are displayed to the user. The opcode translator then gets the instruction mnemonics and translates them into opcodes, that are in turn used as input to the simulator

4.2.1 Syntax Checker

The Syntax Checker handles the instruction code written through the text editor. Each instruction line is checked for syntax errors. If there are no errors, the program code is passed on to the Opcode Translator. In the case that an error is encountered, the Syntax Checker takes note of the line number and proceeds with checking the remaining instructions, until the last instruction is reached. A message box is then displayed to inform the user of unsuccessful assembling and of the line numbers where the errors were found.

Upon receiving the program code, the Syntax Checker identifies variable declarations by checking the presence of a .DATA segment. If such exists, variable declarations are checked for correctness. Format should be *label = target address*. Label names or variables can be alphanumeric. Special characters, except for the underscore (_), are not allowed. The labels and their corresponding target addresses are stored in a temporary array for reference during simulation. In case there is no defined .DATA segment, all text input are considered DLX instructions.

Following the .DATA segment is the .CODE segment, consisting of the DLX instructions to be simulated. With each instruction, the DLX mnemonic is compared with each entry in a library of DLX instructions, called *main.lib*. The format of R-type instructions contained in the library is *mnemonic : instruction type : opcode;*. I-type and J-type instructions, on the other hand, have the format *mnemonic : instruction type : opcode : EX : MEM : WB;*. The opcode is an assigned two-digit binary code for each of the instructions. The *EX*, *MEM* and *WB* portions of the library entries pertain to the attributes of the particular instruction during simulation, specifically during the Execute (EX), Memory (MEM) and Write Back (WB) stages. In *EX*, the type of ALU operation (e.g., memory reference, register to register, register to immediate, or branch), the type of operation (e.g., basic arithmetic, logical comparison, shifting, conversion, move, or comparison) and the type of registers (e.g., immediate, floating-point or double precision) are all stated. Instructions may include additional details such as the sign of the result (e.g., signed, unsigned or floating-point) and the type of the extension (e.g., sign or zero). This is primarily due to the differences in the way instructions are executed. Thus, their parameters also vary. In the *MEM* portion, it is noted if the instruction is active during the MEM stage. For active instructions such as Load/Store instructions, the length of data to be loaded or stored (e.g., byte, halfword, word, single-precision floating point or double-precision floating point), together with the type and sign of the destination register, is indicated. Evidently, entries for other instructions inactive during this stage do not include such parameters. In *WB*, the type of operation done (e.g., register-to-register integer, register-to-register floating-operation, register to immediate, among others) is specified. All information indicated in *EX*, *MEM* and *WB* portions of the library entries are accessed later on during simulation and are used as bases in the methods that are to be performed.

Once the corresponding mnemonic is found, a library, called *type.lib*, which contains the different formats for each R-type, I-type and J-type instruction, is accessed. The library is searched for the corresponding mnemonic in order to find the correct format of that particular instruction, and compare the instruction against it. For *r-type instructions*, the format is *mnemonic <operandtype datatype>,<operandtype datatype>,<operandtype datatype>;*. *Operandtype* may be *rd*(destination register), *rs1* (source register 1) or *rs2* (source register 2). *Datatype* may be *i* (integer), *f* (single precision floating point) or *d* (double precision floating point). Operands are enclosed in brackets and are separated by commas. For *i-type instructions*, instruction formats vary for every instruction available. For instructions with memory access, the immediate is checked if it has been declared. The temporary array previously initialized is searched for a match. If the label is not found, it is then checked if it decimal, binary, or hexadecimal, by

checking the first character. # denotes a decimal immediate, while *$* denotes a binary immediate. In any case, the immediate is treated as hexadecimal.

Aside from ensuring that the instructions in the program code are of correct format, the validity of the operands used is also verified. Registers are ensured to reach until R31 only, and that R0 is not the destination field. For branch instructions, it is first determined whether the target address exists or not. For double precision floating point instructions, only even-numbered floating-point registers should be used.

In general, the Syntax Checker is not case-sensitive. The presence of commas and spaces between instruction fields are considered. Operands should always be separated by commas. Space in between the mnemonic and the first operand is important, while those in between commas and operands is negligible.

4.2.2 Opcode Translator

After the program code is checked for syntax errors, it is then passed on to the Opcode Translator. This sub-module decodes the program into codes conforming to the DLX instruction formats. For every instruction in the program code, the DLX instruction library *main.lib* is accessed. Each entry in the library contains the instruction type and opcode of different instruction mnemonics. The format for each entry is *mnemonic : instruction type : opcode*. The mnemonic of the instruction being translated is compared against each entry in the library. Once the matching mnemonic is found, its corresponding opcode is obtained. If the opcode obtained is *00*, a separated library, *special.lib*, is accessed. This library contains the mnemonics and the corresponding opcode of DLX instructions that involve general purpose registers. If the opcode obtained is *01*, the *fparith.lib* library is accessed. *Fparith.lib* contains entries of DLX instructions using floating-point registers. The entries contained in *special.lib* and *fparith.lib* follows the same format as those in the *main.lib*, each consisting of the mnemonic and a corresponding code. The difference is that, this code is for the *function* of that instruction, which constitutes the last 11 bits of the code. After obtaining the opcode, the operands are translated into their corresponding binary codes.

To illustrate further the decoding process, suppose the instruction ADD R1, R0, R2 is to be decoded into its corresponding code. This instruction is an R-type instruction and follows the format, illustrated in Figure 3.

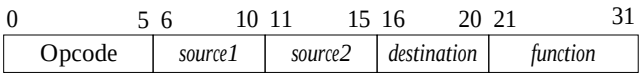


Figure 3. R-type instruction format

From *main.lib*, the entry of the ADD instruction would be *ADD : R-type : 00*. The value of *opcode* field is then 00. Then, the value of the *function* field is divided into two – the first 5 bits, which is unused and has the value of 0, and the last 6 bits, which contains the opcode from the *special.lib*. In this case, the opcode of the ADD instruction is 00. Therefore, the final code of the instruction in binary code is shown in Figure 4.

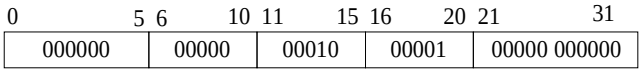


Figure 4. Sample ADD code

If errors are encountered before decoding is finished, the translator terminates without completing and error messages are displayed to the user. If there is no error, an object file is created. This file contains the corresponding binary code of each of the instructions in the program and is forwarded to the function manager to execute the necessary algorithms.

4.3 Function Manager

The function manager serves as the core of the DARC 2 system. It handles all the algorithms that the system uses, and implements the specifications defined by the user for each algorithm. It receives as input the binary codes of all the instructions in the program, as generated by the assembler.

It receives as input the object file generated by the assembler. The file contains the binary codes corresponding to each instruction of the program. The function manager analyzes the first 6 bits of each code and determines the type of instruction that will be executed and the type of operands that it will have. There are only three types of instruction: ALU operations, Load/Store operations and Branch operations. Look-up tables, containing the instructions under each type and their corresponding 6-bit code, are used in executing the different algorithms since there are different executions for different types of instructions.

The function manager implements pipelining algorithms – unpipelined, pipeline 1 and pipeline 2. After creating the codes, the count of the clock cycle starts. The memory, as well as registers and pipeline registers, are updated every clock cycle and whose values are stored in a text file.

The configurations defined by the user among the simulation options have different effects and implementations on the algorithms. These differences are reflected more in the data path diagrams than the pipeline diagrams, since it is the data flow that differs mainly with each pipeline algorithm. The data path diagram and pipeline diagrams are illustrated in Figures 5 and 6.

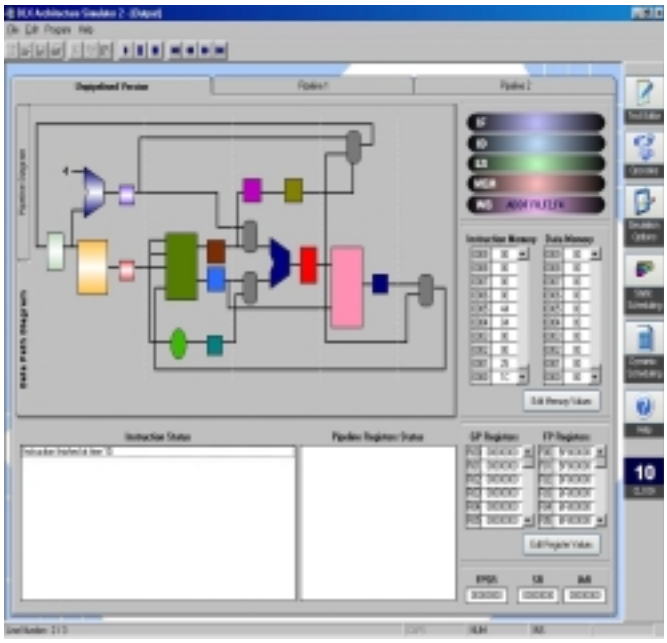


Figure 5. Output Window for Pipelining (Data Path Diagram)

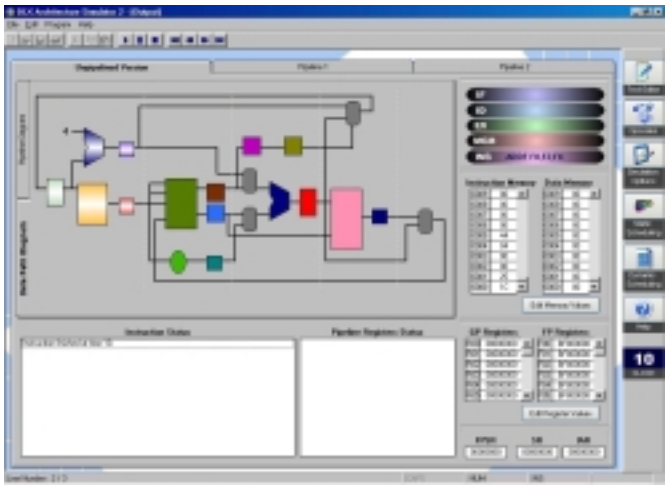


Figure 6. Pipeline Diagram of the Output Window

Unpipelined execution, Pipeline 1 and Pipeline 2 are each treated as modules. The modules contain different procedures each representing the pipeline stages. Each procedure involves only the components present within the stage. It accepts input, such as register values, from one stage, and the necessary methods are performed. The required output produced is then passed on to the next procedure. If it is pipelined execution, this does not necessarily mean the next pipeline stage. Thus, each stage is indifferent of what instruction is currently processed, and is concerned only with the methods it needs to accomplish.

Before the simulation begins, the value of the Program Counter (PC) of the last instruction is noted. This is for purposes of monitoring the end of the simulation. Then, during the IF stage, the first Program Counter (PC) is used as input. The Instruction Register (IR) takes the value of the memory location pertained to by PC. This is the first instruction, whose first six bits are then analyzed in the ID stage. Since instructions differ mainly in the EX, MEM and WB stages, each instruction is treated differently. The conditions to be followed during these stages are indicated in the corresponding entry of each instruction in the library files *main.lib*, *special.lib* and *fparith.lib*. Each instruction entry found in any of these libraries include the EX After noting these conditions, appropriate methods are performed. After the WB stage, the Next Program Counter (NPC) is checked and is compared against the value of PC of the last instruction noted earlier. If the NPC is greater, this means that the end of the program code has been reached and that the simulation is finished. Else, the next instruction is fetched and the simulation continues.

4.4 Infinite Loop

An instruction is said to be an infinite loop when it has exceeded the intended frequency of execution, usually brought about by logical errors made. To ensure accurate results, the system is equipped with the ability to detect infinite loops. This facility is also presented in the simulation options windows, wherein the user defines the number of times a certain instruction is executed before it can be considered an infinite loop. This becomes the threshold of the frequency of every instruction execution, and is applicable to both unpipelined and pipelined algorithms. To monitor infinite loops, each instruction is then assigned a counter, which counts the number of times that particular instruction is executed during simulation. If the counter value exceeds the threshold defined, that instruction is said to be an infinite loop. Whenever an infinite loop is encountered in either the unpipelined or the pipelined algorithms, simulation is terminated for that particular algorithm. Simulation of other algorithms continues unless an infinite loop is also met.

For instance, a user defines ten (10) as the frequency threshold in the simulation options window. This means that if a certain instruction is executed more than 10 times as indicated by the instruction's counter, it will be considered an infinite loop. There may be cases wherein an infinite loop is encountered only in one pipeline algorithm and not in the others. An instruction may be loop infinitely in pipeline 1, but not in the unpipelined execution and in pipeline 2. In this case, only the simulation of pipeline 1 terminates; simulation of unpipelined execution and pipeline 2 continues.

4.5 Backtracking and Statistics

Backtracking allows user to see the values stored in the pipeline registers, as well as the other registers, at any clock cycle. The user may do so by clicking on the clock cycle desired in the pipeline diagram displayed. Once clicked, the DLX data path diagram for that clock cycle is shown. During step mode, the option for backtracking is always available since instructions are processed per clock cycle. However, if the simulation is in one-pass mode, the user may choose to backtrack only after the whole simulation process has finished.

After program simulation, statistical data is displayed. This includes the number of hazards encountered.

During simulation, the outputs obtained in the data path, along with statistics variables, are stored in a text file. This makes statistics gathering and backtracking easier since every clock cycle is taken into account. Aside from the pipeline states, the data in the memory, registers (GPRs and FPs) and pipeline registers are stored. For every program code simulated, two text files are created – one for the unpipelined execution and another one for Pipeline 1 and Pipeline 2. The only difference in text file formats of the unpipelined and pipelined execution is the presence of the pipeline registers. All other data needed to be stored are included in all text files.

In the first line of the text files, the clock cycle is stored. The following lines correspond to the pipeline stages, memory and the register values for that clock cycle, including the pipeline register values for the pipelined execution. The data to be placed in the pipeline stages are the states of the components involved and executing at that clock cycle. The instruction currently executing at a particular pipeline stage are also noted through the special register IR. This is considered as the "block" of data for that clock cycle. A blank line is then inserted in the file, signifying the end of the "block" for that particular clock cycle. Thus, another "block" of data follows the blank line, following the same format until the last "block" of data is appended onto the text file.

Once the user has chosen a clock cycle, the three text files are accessed. The "blocks" of data stored in each text file is searched by their clock cycle. Upon finding the desired clock cycle, all data

within the block are acquired and displayed on the output window.

As with the statistical variable included in the text files, they store the number of hazards encountered as of the current clock cycle. However, statistical data are only displayed after simulation has finished and are not shown during backtracking.

5. CONCLUSIONS AND FUTURE WORK

DARC 2 provides an effective environment for the simulation and exploration of the DLX Architecture. The animation facility in the system proves useful for allowing the students to visualize the interaction of different components employed in the DLX architecture. It allows students to understand better the flow of data through the architecture and how each instruction is executed, while removing the difficulty that is often experienced with manual tracing and redrawing of pipeline and data path diagrams. With all the improvements incorporated in DARC2, it is hopeful that new batch of students will have a clearer understanding of the pipeline concept. It is hopeful that this will boost the appreciation of studying computer architecture even higher and thus, creating great possibilities of constructing new architectures.

Subsequent improvement involves adding advanced concepts such as superpipelining, superscalar execution, cache memory, branch target buffers and others to the simulators. Eventually, the simulator will evolve to a DLX Virtual machine. This is similar to the "Java machine" concept. In the virtual machine, actual DLX code will be executed in x86 machine. The DLX Virtual Machine projects will involve modules relating to runtime operating system, a compiler module to convert high-level language to DLX and others. With the research framework in place, our university is excited with the revival of Computer Architecture field.

6. ACKNOWLEDGEMENT

The author would like to acknowledge the 1st generation DARC development team composed of Mr. Jonathan Lee, Jonathan Ray Roque, John Jerrick Sy, and Aldrich Nino Lorenzo. This project, which is the development team's undergraduate thesis project also won grand prize award sponsored by a local software institution.

7. REFERENCES

[1] Uy, Roger Luis, Lee, J., Roque, J.R. Sy, J.J., and A.N. Lorenzo. *DARC.* Undergraduate Thesis, De La Salle University, Manila, Philippines, 2003.

[2] Hennessy, John L., and Patterson, David A. *Computer Architecture: A Quantitative Approach 3rd Edition*. Morgan Kaufmann Publishers, 2003.

MKit Simulator for Introduction of Computer Architecture

Seikoh Nishita
Department of Computer Science,
Faculty of Engineering, Takushoku University
snishita@cs.takushoku-u.ac.jp

Abstract

For the introduction of computer architecture in computer science, highly simplified specification of CPU, and visualization of CPU operation are important. This paper describes a CPU simulator we have developed. It has interfaces to show inner state of CPU and to manipulate the simulator in several ways, which are changed among with progress of students' learning.

1. Introduction

The introductory education on the computer architecture involves several topics: about the relationship between assembly and machine language, the connection of data-path and data-path elements as the static construction of a processor, the operation for instructions as dynamic structure, and the control by control unit and control signals. These topics focus different aspects of the processor. For example, instruction set is focused on the topic about the relationship of languages. On the other hand, processor block is focused on the topics of static and dynamic structure of CPU. And Control unit and control signals are focused on the topic about control.

This paper describes a CPU simulator, "MKit simulator" that we have developed. MKit simulator has several interfaces for students to check and manipulate MKit processor. These interfaces are draw panels in the simulator window, and input methods like buttons. The draw panel shows inner state of the processor, and the input methods are used to manipulate the processor and the simulator. These interfaces emphasize respectively various aspect of the processor. We correspond these interfaces with the topics that have same aspect of the processor. Our simulator provides students a part of the interfaces for every assignment for the corresponding topic. Since there are a number of topics students learn during an exercise, our simulator switches interfaces to use for every assignment.

The next section gives the specification of CPU for our simulator. Section 3 then describes simulator designing. Section 4 describes assignments we have designed for an exercise class with our simulator. Section 5 then describes implementation of the simulator. Section 6 discusses the relation of our simulator and other simulators that have been proposed for introductory education of computer architecture.

2. Specification of MKit

Before we discuss the designing of MKit processor, we describe our background of computer architecture education in our department. We have one course on computer science, and there are three classes closely concerned with computer architecture. The first is a class on introduction of computer science, where first year students learn basic concept of computer architecture with a simple CPU, MKit [4]. The second is a basic exercise on computer science, where our simulator is used. And the third is a class on computer architecture, where students learn advanced. The goal of the basic exercise class is students' making certain of the basic concept of computer architecture that they have learnt at the first year class.

We adopt MKit as the processor for our simulator. In deciding the processor we take care to keep the design as simple as possible, and to link the topic of first year class and the exercise class closely. When students are going to have the basic exercise class, they have learnt computer architecture with MKit processor. Therefore students are familiar with MKit, and they don't take relatively long time learn the specification of MKit than new specifications. As the result they can concentrate their thought to essential of computer architecture and assignments in the exercise.

MKit CPU is a 16-bit word accumulator machine. It has only the direct addressing as the addressing mode

to access memory. The memory access is 16-bit word addressable. Making the machine word addressable only simplifies the operation of CPU. Moreover care was taken to keep the correspondence between assembly language instructions and the machine instruction as one-to-one relationship. As the result, the address width is reduced to 12 bits, and the operation code occupies 4 bits of a word. This configuration simplifies the hexadecimal number representation in the machine language. MKit only supports two instruction formats as shown in Figure 1.

The instruction set of MKit consists of arithmetic calculations, jump/branch operations and memory access (load and store) operations (as shown in Table 2). It has instructions for a subroutine call "JL" and a return operation "RET". These instructions make the processor complicated, but they are straightforwardly integrated into the instruction set and the operation of the processor. Moreover they encourage students' learning of the concept on subroutine call operation without stacks.

The data-paths of the processor are not based on bus structure. The data-paths connect the data-path elements directly. And the instructions take multiple clock cycles to execute. Data are roughly flown clockwise along with data-path in a processor block diagram shown in Figure 3. One difference from other processors for introduction of computer architecture is that the processor has a return address register, RAR. RAR is used for the subroutine call and return operation. Since RAR is not stack, this micro-architecture does not support multiple subroutine call. But the micro-architecture shows an implementation for subroutine call, and it can be simply extended by replacing a register with a stack for RAR.

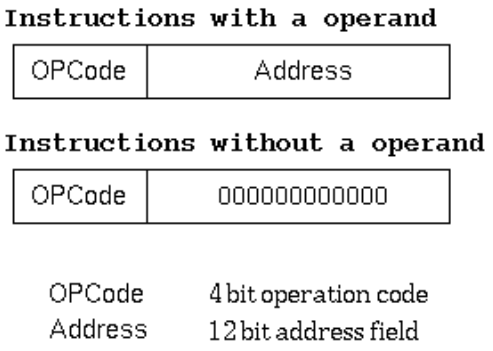


Figure 1. Instruction encoding formats

Sort	*Nmenonic*	*Semantics*
Memory access	LD n	ACC <- Mem[n]
	ST n	Mem[n] <- ACC
c n	ADD n	ACC+Mem[n]
	SUB n	Mem[n]
	SFR	ACC <- ACC>>1
	SFL	ACC <- ACC<<1
Jump and Branch	JMP n	PC <- n
	JPZ n	PC <- n (if ACC=0)
	JPN n	PC <- n (if ACC<0)
	JPO n	PC <- n (overflow)
	JPL n	PC value is kept, then PC<-n
	RET	PC value is restored
Control	HLT	halt

Table 2. Instruction Set

3. Simulator Designing

The basic concept of computer architecture for introductory education consists of several topics. On this paper, we divide the basic concept into following 3 topics.

- Instruction set architecture, and translation of assembly language instructions to machine language instructions
- Data-path connection with data-path elements (static construction of the processor except control unit), and the operation of the processor: instruction fetch and execution (dynamic construction)
- Control unit and control signals. Static and dynamic construction of the processor.

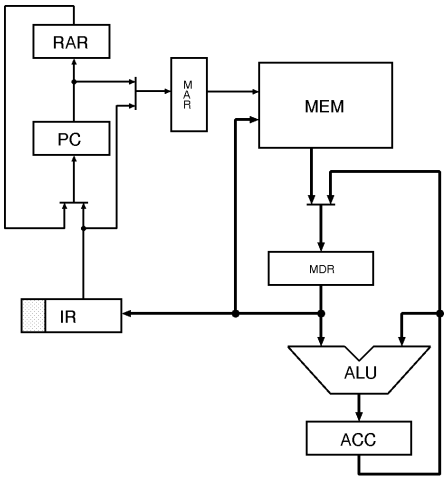


Figure 3. Processor Block Diagram

We also assume that assignments are given along with these topics.

The processor is placed as various objects on these topics respectively. For example, on the topic about instruction set architecture, the processor is regarded as a machine that executes instructions. For the topic about data-path connection, the processor is regarded as a structure represented by the processor block diagram. On the topic about control unit, the processor is regarded as a structure represented by the diagram with the control unit and lines for control signals.

Moreover these topics have their adequate assignments, which need functions to manipulate the processor. For example, functions for execution all instruction and execution one instruction are helpful to make certain of the meanings of instructions. Functions for step-by-step execution is useful to learn the operation of the processor.

Interfaces of CPU simulator emphasize the aspects of the processor, and they give the functions to manipulate. The interfaces are divided in interfaces to express inner status of the processor, and interfaces to manipulate the processor. These interfaces are often implemented as drawing panels or push buttons in a same window.

Our idea for designing a simulator is based on multiple uses of the expression interfaces and manipulation interface, and it is based on a concept of correspondence between the interfaces and the topics. Since the processor is regarded various object as the topics, we adopt the expression interfaces in order to represent the aspects of the processor. We also select the manipulation interfaces in order to design assignments.

On this paper, we design following manipulation interfaces from M1 to M5.

M1 is for execution all instructions in memory unit. It is implemented with one button, which activates virtual processor to run a program one by one along with clocks. This process stops till the processor decodes the halt instruction. This interface is suitable, when the processor is regarded as a machine to execute instructions.

M2 is for execution one instruction in memory unit. It is same as M1 except the fact that it activates virtual processor to run only single instruction.

M3 is for execution one-step of the process. It is same as M1 except the fact that it activates virtual processor to run only one step (one clock). Since one step of the execution may not be represented in semantics of the instructions, this interface is not suitable for students to check instruction architecture. But it is suitable for checking the inner operation of the processor, because it helps students to analyze the operation in detail.

M4 is for assertion of control signals directly. It is implemented with buttons, which assert all control signals in the processor. This interface is suitable for the topic about control unit and signals. It helps students learning of control signals and deriving their sequence of the fetch and execution operation.

M5 is for making time chart of control signals for instruction fetch and execution operation of all instructions, in order to specify the control unit. The branch instructions have two time charts for the condition of the branch. This interface is also suitable to the topic about the control unit and signals. Since we assume that students don't learn the subject of the logic circuit designing, there is no interface for designing control unit with logic circuit. Instead of logic circuit, we adopt time charts. Though we need to show how to interpret and write time charts, it doesn't take longer time for explanation about time chart than about logic circuit designing. Moreover we take advantage of supporting education by specifying the control unit. That is, combination of the interface M5 and M2 facilitates students to learn that the control unit generates the sequence of control signals automatically and these signals activate the operation of the processor.

We also use auxiliary interfaces to reset the processor, to suspend the execution and to adjust the speed of the execution.

In order to present the inner state of the processor, we make visible/invisible attributes for sorts in the processor block, i.e. control unit, path of control signals, data-path and data-path elements. By these attributes, objects are shown/hidden on the processor block diagram. Following list shows the attributes of the every sort in the processor.

- Data-path: “data-path” and “data-flow” attributes. The former is for static representation of data-paths, and the latter is for dynamic representation of flowing data.
- Data-path elements: “elements”, “value”, and “flag” attributes. The “value” attribute is used to show data kept at registers.
- Control unit and control signal: “control” attributes. The “control” attributes is used to show the both of the control unit and the lines of control signals.

We design the expression interfaces from E1 to E5 by specifying all attributes of the sorts in the processor block.

E1: “data-path-flow”, “elements”, “value” and “flag” of primary data-path elements are visible.
E2: all attributes except "control" are visible.
E3: all attributes are visible.
E4: all attributes except "control" are visible.
E5: all attributes except "control" and "value"

As addition of these interfaces, we also use a message box for explanation of the operation. In order to design the expression interface we define the word "primary data-path elements" for elements referred in the semantics of the instruction set architecture. The primary data-path elements of MKit are the program counter, the instruction register, the accumulator, and the memory unit.

All attributes specify simply their visibility, that is, if an attribute of a sort is visible, objects of the sort are always drawn in same position of the simulator window. This specification of the expression interface makes it simple to represent the inner status of the processor and makes it easy for students to grasp the operation.

4. Interface and Assignment Design

We introduced interfaces to express status and manipulate CPU simulator in previous section. In practically use of the simulator, the expression interfaces and the manipulation interfaces are combined along with assignments for the topics of computer architecture. We discuss with the assignments and combinations of interfaces we have designed. The assignments are assumed to given in the following order.

4.1 An assignment about the instruction set architecture

On this assignment, students translate assembly language instructions to machine language instructions. Then students make certain of the semantics of the instructions they have learnt. During the assignment, students write instructions into memory unit of our simulator, and execute instructions by MKit virtually.

We choose the interface E1, M1 and M2 for the assignment (Figure 4). Since the interface E1 draws only flowing data and primary data-path elements, E1 facilitate students to concentrate the semantics of instructions during the assignment. And because E1 does not draw whole processor block diagram, it is also suitable for students who have not learnt the processor block as micro architecture yet.

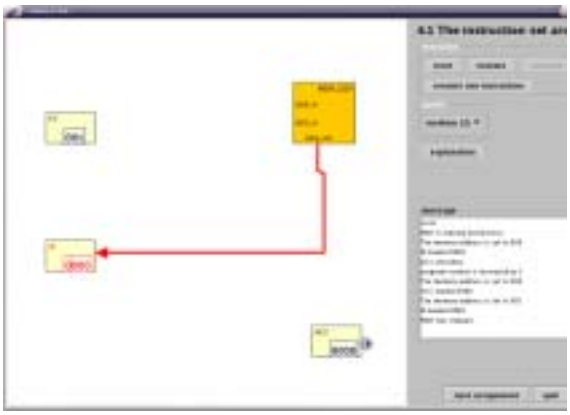


Figure 4. The Interface for Assignment of Instruction Set

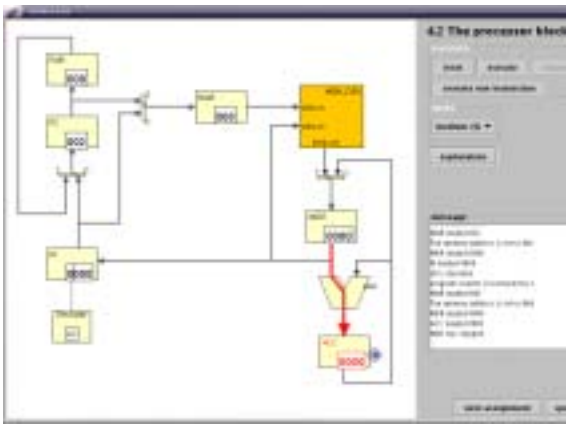


Figure 5. The Interface for Assignment of Processor Block

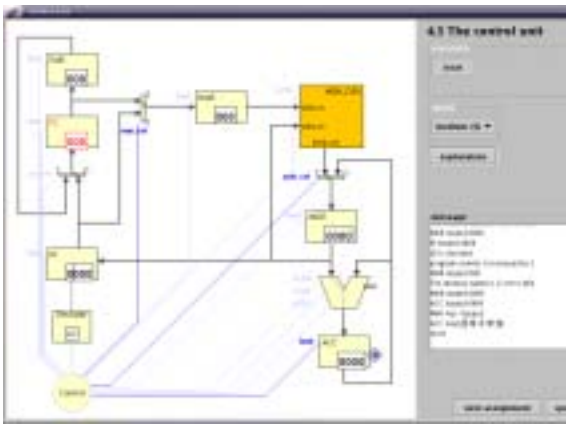


Figure 6. The Interface for Assignment of Control Unit

The interface M1 and M2 is used to execute instructions. These interfaces make students to confirm the semantics and a part of the processor operation, by checking flowing data on the simulator window. We note that the there is not the interface M3, which is used for executing step by step along with the clock. Since there are steps that don't bring any change of the primary data-path elements, the step-by-step execution is not suitable for the interface E1.

4.2 An assignment about the processor block

On this assignment, students confirm the static structure of the processor by the processor block diagram. Then students execute instructions in the memory unit, and make certain of the processor operation on the processor block diagram. This assignment supports students to confirm that there are registers and data-path among the primary data-path elements in order to propagate data.

We choose the interface E2, M1, M2 and M3 for the assignment (Figure 5). The interface E2 draws the processor block diagram except the control unit. E2 makes students concentrate to learn the processor block. Students can also execute instructions and check the operation of the processor by the interface M1. When students want to check the detail of the process, they can use M2 and M3 too.

4.3 An assignment about the control unit

We design three assignments for the topic on the control unit: an assignment about the static construction of control unit, two assignments about the operation of the control. These three assignments are described from this section to the section 4.5.

On this assignment, students make certain of the control unit and the lines of control signals. The goal of the assignment is that students confirm the existence and roles of the control unit and control signals.

We choose the interface E3 for the assignment (Figure 6). The interface E3 shows all sorts of the processor including the control unit and the lines of control signals. The minutest diagram of processor block makes students to learn and remember that the process of CPU arises from the control unit and signals.

Since there is no interface for manipulation, all students can do is to confirm the static construction of the processor block diagram. The next assignment makes demands of students for more active learning.

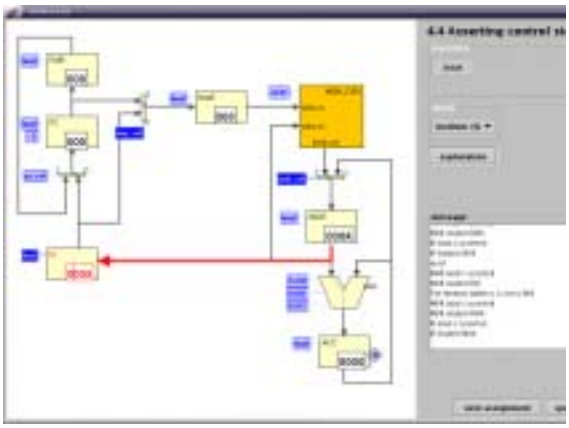


Figure 7. The Interface to Assert Signals

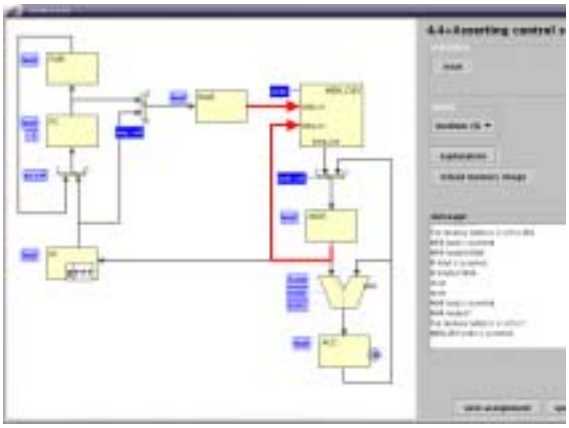


Figure 8. The Interface where register values are hidden

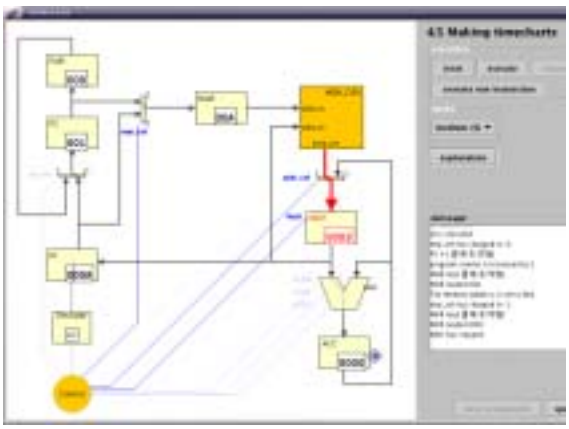


Figure 9. The Interface for Assignment of Making Time Chart

4.4 An assignment to control CPU by asserting control signals directly

On this assignment, students derive the sequence of control signals for fetch and execution of all instructions. We aim students make certain of the instruction fetch, decode, and execution in the CPU process, through this assignment.

We adopt the interface E4 (or E5) and M4 (Figure 7). In order to assign student to derive the sequence of control signals, we make students assert the signals via buttons provided by M4. In order to execute instructions, they derive the correct sequence of control signals on the basis of their knowledge they have learnt the first 2 assignments.

The interface E4 draws the processor block diagram except the control unit and the lines for signals. We indicate the lack of the control unit, and students have to perform the role of the control unit.

The use of the interface E5 (Figure 8) makes this assignment more difficult, because E5 does not show values kept by registers. Therefore students check only OPcode and flags, and then they assert control signals with the use of E5. The interface E5 helps students to learn the control unit has only these values as their input.

The interface M4 provides buttons for all control signals. The buttons are drawn of the processor block diagram at the draw panel. Drawing the buttons on the diagram makes manipulation of the buttons intuitive.

4.5 An assignment to control CPU with time chart

On this assignment, students make time charts for fetch and execution of all instructions. Then they execute instructions in the memory unit with the time charts they specified. The goal of this assignment is that students summarize the sequences of control signals, and they confirm the role of control unit that works automatically.

The virtual processor in the previous assignment was incomplete in the sense of the control unit absence. On this assignment, students make the processor complete by the time charts for the control unit.

We choose the interface E3 and M5 (Figure 9). The interface E3 is used in order for students to check the values of control signals, and to make it easy to try and error for specifying time charts. The interface M5 shows time charts of all control signals for fetch and execution of all instructions. And it accepts mouse clicks to modify signals (Figure 10).

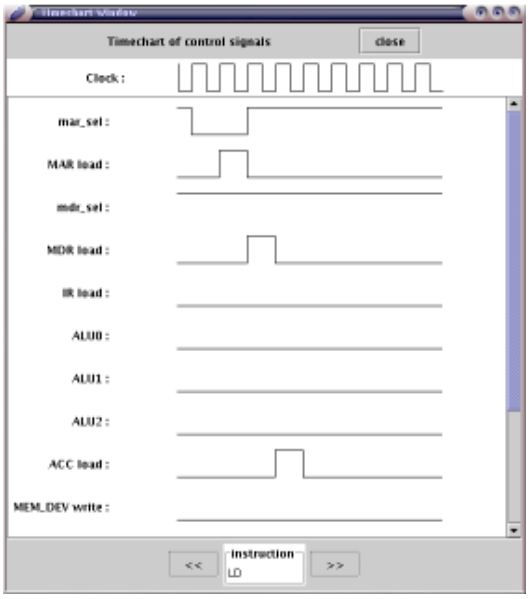


Figure 10. The Interface to Make Time Chart

5. Simulator Implementation

We have implemented MKit simulator with Java 1.4. Our implementation is based on object-oriented concept, where the parts of the processor (data-path, data-path elements, control units, control signals) are treated as objects. Each object has its attribute for the expression interface. And each object has methods to implement the manipulation interfaces.

Objects have their own positions on the simulator window respectively. If the current interface is changed, objects are drawn/hidden at the positions according to the attributes. As an advantage of this behavior of object, it makes students' grasp of CPU status easier, because a part of the single processor block diagram is always drawn, and no object changes it's position among the interfaces.

In order to design other exercises and assignments, we can change the order of the interfaces and the attributes of the interfaces. But in order to change the interfaces, we need to modify Java programs on current status of our implementation. There is other root for refinement of the current implementation. That is, our source package will be divided into three modules, i.e. the virtual processor, interfaces for expression and interfaces for manipulation. This refinement will make the simulator more flexible and highly extendable for modification of the interfaces and the virtual processor.

6. Related Works

In order to support computer architecture education, several CPU simulators and simple specifications of CPU have been proposed [1,2,3]. These simple specifications are designed deliberately for first or second-year students to learn essential of computer architecture, and to put advanced usage for operating system or compiler in their perspective.

Following is a list of policies for designing processor.

1) The specification is designed simple enough for first or second year students to learn essential of CPU operation.
2) The specification is designed in order to put advanced application for operating system or compiler in the perspective.
3) The specification that students have already learnt is adopted

In studies on CPU simulator, the items 1,2 are treated important factors. But our policy to decide the specification of processor is based on the item 1,3. This fact gives rise to a difference between ours and other specifications. That is, our specification is designed as a simple accumulator machine, while the specifications are based on register machines with load-store architecture. Since our idea is the correspondence between the interfaces of simulator and assignments, we can essentially apply the same idea to other CPU simulators for the introductory education of computer science.

The interfaces we designed are also used in CPU simulators. For example, the expression interface, E3 is same as the simulator for The Simple CPU [1]. The manipulation interface, M4 is same as one of interfaces of RTLsim [3]. One of differences between these simulators and our simulator is that these simulators use their own interfaces, while our simulator have a number of interfaces and use some of them along with assignments.

7. Conclusions

This paper describes a CPU simulator for introductory education of computer architecture. Our approach to design the simulator is based on the correspondence between the interfaces of our simulator and assignments of an exercise class on computer architecture. The simulator selects and provides a part of the interfaces for students along with an assignment. The simulator has single virtual CPU to simulate, and single processor block diagram to express the status of CPU. This framework helps teachers to design various assignments and exercises with CPU simulator on uniformed framework, and it also reduce students' extra learning overhead for CPU specification and use of the simulator.

We are applying the simulator to an exercise class at our Department of Takushoku University. We are going to evaluate the simulator with the practical use in the exercise class.

References

[1] J.R. Arias and D.F. Garcia, “Introducing computer architecture education in the first course of computer science career,” *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, July 1999, pp. 37-39.

[2] H.B. Diab and I. Demashkieh, “A computer-aided teaching package for microprocessor systems education,” IEEE Transaction on Education, vol.34, no.2, 1991.

[3] M. Pearson, D. Armstrong and T. McGregor, “Using Custom Hardware and Simulation to Support Computer Systems Teaching,” Proceedings of Workshop on Computer Architecture Education 2002, pp.19-26, 2002.

[4] K. Murakami and T. Ishikawa, "Introduction to Logic Circuit for Computer Systems" (Printed in Japanese), Kyoritsu Syuppan, pp.123-159, 1996.

PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education

Vijay Janapa Reddi, Alex Settle, and Daniel A. Connors
University of Colorado, Boulder.
{vijay.janapareddi, settle, dconnors}@colorado.edu

Robert S. Cohn
Intel Corporation
robert.s.cohn@intel.com

Abstract

Computer architecture embraces a tremendous number of ever-changing inter-connected concepts and information, yet computer architecture education is very often static, seemingly motionless. Computer architecture is commonly taught using simple piecewise methods of explaining how the hardware performs a given task, rather than characterizing the interaction of software and hardware. Visualization tools allow students to interactively explore basic concepts in computer architecture but are limited in their ability to engage students in research and design concepts. Likewise as the development of simulation models such as caches, branch predictors, and pipelines aid student understanding of architecture components, such models have limitations in the workloads that can be examined because of issues with execution time and environment. Overall, to effectively understand modern architectures, it is simply essential to experiment the characteristics of real application workloads. Likewise, understanding program behavior is necessary to effective programming, comprehension of architecture bottlenecks, and hardware design. Computer architecture education must include experience in analyzing program behavior and workload characteristics using effective tools. To explore workload characteristic analysis in computer architecture design, we propose using *PIN*, a binary instrumentation tool for computer architecture research and education projects.

1 Introduction

New applications and programming models are constantly emerging to complement new and improving hardware technology and paradigms. It is becoming essential to understand the workload characteristics of applications in order to design effective architectures. Often, in order to understand program behavior on a specific processor, students must have a significant amount of knowledge of the underlying hardware and the control and data flow of the application. For instance, modern performance is a confluence of many components working together - branch predictors, caches, pipelines, etc. Even with a deeply rooted understanding of the architecture, it is often extremely difficult to comprehend the flow and resource usage of the program because of the immense amount of data that needs to be collected and analyzed in order to study such behavior.

There are various tools available for computer architecture education. These tools can be divided into several categories, architecture visualization systems, simulation environments, and hardware event monitoring programs. Each of these categories play a role in bridging the divide between pedantic methods of illustrating computer architecture and real-world dynamic examination of architecture concepts. There are several areas of knowledge and skills that these tools address. For instance, architecture simulators provide insight into microarchitecture design and the behavior of the individual hardware components. These simulators are usually designed so that students can integrate additional emulated hardware components into the overall simulation system and analyze the impact of design parameters on the simulated processor. Likewise, performance monitoring support allows students to realize the performance impact of the different architecture components and compiler optimizations have on the overall system. Most importantly, monitoring systems provide accurate feedback on real workload applications.

Simulators and performance monitoring systems may not be sufficient for computer architecture education because they do not allow large amounts of information to be collected on a per-instance level at the instruction granularity. Likewise, profiling techniques, such as *gprof* and *pixie*, only provide coarse-grain profiling information and are not suitable for detailed computer architecture concept exploration. Rather it is necessary for students to attribute profile information to the instruction level of the program. Thus, tools that provide infrastructure for performing data analysis of both software and hardware events can be extremely valuable to the computer architecture fundamentals of performance analysis, design, and architecture validation. Overall, such integrated computer architecture examination results in a deeper and more detailed comprehension of the collected data.

In this paper, we present PIN, a binary instrumentation tool that can be used as a teaching aid by in-

structors in computer organization education to facilitate the study of real-world workloads and their impact on the design of architectures. In addition to discussing the benefits of using PIN in the area of education, this paper includes a complete description of PIN's extended profiling features and methods of operation. Overall, we demonstrate that PIN can be used to replace the time and complexity of using trace files and software simulators to explore computer architecture concepts and design.

In the following section, we present an overview of tools that have been used in computer architecture courses, followed by the motivation of using PIN in computer architecture education in Section 3. In Section 4 we present some projects to illustrate how PIN can be used to teach. Following that we present our conclusion.

2 Background in Computer Architecture Education Tools and Methods

Although various tools are integrated into computer architecture curriculum, we believe that the best systems for computer architecture education should primarily involve:

- Simulated Design Environments
- Performance Analysis
- Workload Characterization

We elaborate on each of the above points and present arguments for and against them as we percieve fit in the classroom environment.

Simulated Design Environments: Instructors often place much emphasis on designing architectural models in the classroom. Simulators are used as a means of getting the students to explore design issues or are even as models to deepen the students understanding of architecture concepts.

Design simulators such as the Liberty Simulation Environment [7] allow students to design architectural components that interact within simulation modeling framework. Such simulators allow them to gain a deep understanding of how various components interact with one another and could potentially create a bottleneck. Others such as MipsIt[2], DLXide[12] etc. are used in the classroom because they help students better understand the theoretical concepts taught and open up research ideas. There exist numerous other simulators that could be used in the classroom, [5] has a lengthy listing of such tools which could very effectively be integrated into the coursework. Building a simulation environment allows students to comprehend the details of real hardware and provides greater understanding of the subtlties which give rise to complex designs. However, when designing simulators, significant time is spent developing software rather than doing the actual performance analysis on their simulator. Simulation environments often abstract the concept of real workloads on real hardware too much. Some students have difficulty relating the effect of the simulation environment to the actual program and hardware.

The simulators students develop are often trace driven which are severly limiting as we elaborate here. The trace files used for simulation can be immensely large, several megabytes per benchmark, and are therefore difficult to distribute as inputs for programs. On average, for ASCII trace file formats with simple execution information (program counter, opcode, 1 field of behavior), for every 2 million instructions there is overhead of about 80MBytes of trace file. Burtscher[11] reports that even with specifically tailored trace-compression algorithms applied to the execution trace, real workloads will exceed several gigabytes of space.

Trace files already reflect control flow of the program which was used to generate the traces. Thus, it voids the students an opportunity to investigate changes to both the application (compilation, rewriting algorithms) and the underlying architecture. Another issue with trace-based simulators is the problem of simulation time, generally such simulators take days to model the entire run of a real program. Our own evaluation of trace-based efficiency has determined that trace-reading (parsing and I/O) consumes nearly 40% of a simple cache simulator program.

Performance Analysis: Hardware event monitors are currently being extensively invested into by chip developers. The Itanium Processor Family (IPF), IA-32, POWER, and Alpha systems provide various event monitors for use. However only a few combinations may be active at any given time; the combinations are often limiting in the events that can be simultaneously monitored. Nevertheless, providing hardware counters has facilitated the development of interesting tools. Tools such as Perfmon[6] and PAPI[10] access the underlying event counters to develop applicaton profiles and other interesting reports that are valuable in performing program analysis.

In the classroom, awareness of hardware event monitoring tools is essential because they are actively being used in the industry to study the performance of programs on the underlying hardware. As mentioned previously, current architectures currently come heavily armed with an arsenal of hardware counters.

Interesting works such as vertical profiling[14], dynamic optimizations and code caches [8] take advantage of event monitors to perform performance critical analysis on programs to help boost the applications performance.

The primary limitation of simply showing absolute counter values to students would not prove effective as a method of teaching because the students lack an

insight on how the various performance counters could be co-related. A very basic example is to realize that using a counter to realize the total instruction count for a program divided by the total execution time of the program in cycles results in getting the average cycles per instruction.

Hardware event monitors are not flexible in that they come as part of the hardware and are not easily customizable to suit the users needs. The event monitors usually perform sampling, they do not guarantee sequence of execution as outputs. Their inflexbility limits their use in the classroom.

Workload Characterization: The arguments presented thus far are to bring forth the realization that minimal emphasis if any has been placed on presenting students with real workloads in the classroom, an essential component to the coursework. Simulators and hardware event monitors though valuable are limited in their contributions to the class environment. The solution to their limitations is using tools that facilitate a means of observing real program behavior on real hardware. Tools that facilitate binary instrumentation - PIN[13], Dyninst[3], Atom[1], etc.

The use of such tools requires no compiliation of the source, any binary application can be directly used as input into the program. This gives the flexibility of being able to study the nature of numerous programs since the only requirement is the binary itself. It voids the requirement of traces for simulators or source codes for studying program behavior.

Since these programs along with their binary input sets run directly on the machine, their execution times are short which allows the study of programs for their entire run. It opens up new venues of concepts that may be presented in the classroom, concepts such as phase behavior. Phase behavior requires that programs be run for a very long period of time, a requirement that cannot be met by simulators, but one that binary instrumentation tools can.

Furthermore, instrumentation tools allow quick implementation of ideas and do not require complex infrastructure. This would be suitable in the classroom because students may quickly implement new concepts and test the concept's effectiveness. This fosters a research oriented environment in the class which motivates students to investigate deeper into the subject. The instrumentation tools would allow students to explore real workloads of varying characteristics, from scientific and engineering programs and even to commercial products.

There exist varied tools that could be incorportated into the architecture curriculum; the essential note is that instructors should realize that it is extremely beneficial for students to have a broad idea of the tools available for teaching and performing real workload studies. The idea is much similar to making a decision in selecting the appropriate programming language when designing a softare applicaton. Thus it is essential for instructors to design their course material such the students are exposed to multiple tools through the run of their computer architecture coursework.

3 PIN - An Approach to using Binary Instrumentation Tools in Education

3.1 About PIN

A tool that we believe fills in the missing pieces of the previously described tools is PIN[13]. PIN is provided free of charge from Intel. It currently runs on the Itanium systems, but work is under way to support ARM and the IA32 architectures. It provides a functionality similar to the ATOM[1] tool for Compaq Tru64 Unix.

The user writes instrumentation and analysis routines. Instrumentation routines insert calls to analysis routines into an application. They determine how an application is instrumented. The analysis routines are called while the program executes and can record information like the effective address of a memory instruction or the direction of a branch instruction. The instrumentation is customizable; the user decides where analysis calls are inserted, the arguments to the analysis routines, and what the analysis routines do.

PIN inserts instrumentation into an application at run time. It sees every instruction in the user process that is executed, including the dynamic loader and all shared libraries. The instrumentation and analysis execute in the same address space as the application, and can see all the application's data.

PIN passes instructions or a sequence of instructions (trace) to an instrumentation routine. The instrumentation routine can inspect the instructions, looking at the opcode class and its register and literal arguments. The instrumentation routine may insert a call to an analysis routine before or after an instruction. PIN tries to make the instrumentation and its own execution transparent to the application. It does not use the same memory stack or heap area (brk) as the application, and maps addresses in a special area. Addresses of local variables (stack) and addresses returned by calls to brk, malloc and mmap will not be changed when PIN is active.

3.2 Using PIN

Presented in Table 1 is a simple example that gives the instruction count of a program including all the shared library calls made by the application program. The sample program is run by executing: *$ pintool – /bin/ls* at the shell prompt.

Lines 13 and 14 register callback functions with PIN. The function "Instrument Instruction" is the *Instrumentation* function that is called on every instruction and "Finish" is the function called upon termi-

```
      /* Analysis Function */
1     void AnalyzeInstruction() {
2          icount++;
3     }
4
      /* Instrumentation Function */
5     void InstrumentInstruction(INS ins, void *v) {
6          PIN_InsertCall(
               /* Call analysis func. before instr. is executed */
               IPOINT_BEFORE,
               /* Current instruction */
               ins,
               /* Call analysis func. before instr. is executed */
               (AFUNPTR) AnalyzeInstruction,
               /* End of PIN_InsertCall's argument list */
               IARG_END);
7     }
8
      /* Executed at end of program */
9     VOID Finish(int n, void *v) {
10         cout << "ICount : " << icount;
11    }

      /* Register callback functions */
12    int main(int argc, char *argv[]) {
13         PIN_AddInstrumentInstructionFunction(Instruction, 0);
14         PIN_AddFiniFunction(Finish, 0);
15         PIN_StartProgram();
16    }
```

Table 1: PIN tool to count the total number of instructions in a program

```
1     void InstrumentInstruction(INS ins, VOID *v) {
2        /* Query the opcode */
3        switch(INS_Category(ins)) {
4           case TYPE_CAT_BRANCH:
5               PIN_InsertCall(IPOINT_BEFORE,
                    ins,
                    /* Call the branch prediction program */
                    (AFUNPTR) Branch_Predictor,
                    /* The instruction address */
                    IARG_IP_SLOT,
                    /* Non-zero if branch will be taken; otherwise 0 */
                    IARG_BRANCH_TAKEN,
                    IARG_END);
6               break;

7           case TYPE_CAT_STORE:
8           case TYPE_CAT_LOAD:
9               PIN_InsertCall(IPOINT_BEFORE,
                    ins,
                    /* Call the data cache program */
                    (AFUNPTR) Data_Cache,
                    /* The memory address */
                    IARG_EA,
                    IARG_END);
10              break;

11          default:
12              break;
13    }
```

Table 2: PIN tool that interfaces with Data cache and Branch prediction simulators

nation of execution of an application. The *Analysis* function for every instruction is specified through PIN_InsertCall on line 6. The instrumentation function is called only the first time an instruction is executed. The analysis function, AnalyzeInstruction() is called every time the instruction is executed.

Data Cache & Branch Predictor Simulation Interface: The pin tool in Table 1 can be changed to support simulations easily by changing the instrumentation function. The instrumentation function in Table 2 easily integrates a data cache and a branch prediction simulator into one tool while still providing the previous instruction count analysis. Detailed opcode analysis is avoided here for simplicity. Precise opcode detail is available in the actual source at [13].

The simulator codes for cache and branch prediction can be written in entirely separate modules, compiled and linked with the pin tool. Also, due to PIN's inherent interface with the hardware, simulator code sizes

IPOINT	
IPOINT_BEFORE	Call before the instruction/procedure is executed.
IPOINT_AFTER	Call after the instruction/procedure is executed.
IPOINT_TAKEN_BRANCH	Call after the instruction executes and before the target is executed. Only supported for IP relative branches.

Table 3: Instrumentation Points (IPOINTs) for PIN_InsertCall(*IPOINT*, INS, AFUNPTR, iarg1, iarg2, ..., iargN, IARG_END)

are small in relation to real simulators. The data cache and branch predictor simulator code size are approximately only 20 lines. The development time is dramatically shortened because the need to build surrounding infrastructure to understand the instruction set architecture (ISA) is no longer required.

The PIN_InsertCall(IPOINT, INS, AFUNPTR, iarg1, iarg2, ..., iargN, IARG_END) function is the key to instrumenting any binary in the PIN environment. Details of the various instrumentation points (IPOINT's) that can be placed for every instruction or every procedure call are provided in Table 3. The function can take up to a maximum of eight arguments to facilitate various types of instrumentation and is capable of instrumenting both procedures and instructions. Table 4 describes the various Instrumentation Arguments (IARG's) that may be passed into the analysis function. The type AFUNPTR defines the analysis function to be called during the run of the program. Only a few of the IARG's and IPOINT's are listed for conciseness.

4 In the Classroom

4.1 Students and Projects

Understanding concepts allows the principles of many different disjoint areas to be leveraged in solving problems and developing skillful intuition. In turn, teaching is about exposing the underlying principles of ideas in ways that are both clear and logical. A good approach to teaching computer architecture is to be able to teach a concept and immediately illustrate a working system to students. The PIN infrastructure can be used to illustrate such ideas and allow students to cultivate and exercise their creativity and intuitions in projects. Often, course projects are limited in scope because of time, however, by integrating PIN with existing tools for use in a project, more structured ideas can be realized. Furthermore, it is evident that the act of learning an existing tool for a project is similar to real engineering situations in becoming assimilated to a particular design team.

One of the most common projects in computer architecture is to build concept simulators to enhance understanding. These projects include instruction cache, data cache and branch prediction simulators. Such assignments are very costly in the amount of time the students spend building the interface to reading the trace

IARG	
IARG_IP_SLOT	Memory address of an instruction, where the low 4 bits encode the slot number (e.g. 0, 1, 2).
IARG_IP	Memory address of the bundle containing this instruction.
IARG_EA	For a load or store, the effective address of the memory location accessed by an instruction. Only valid for IPOINT_BEFORE.
IARG_QP_VALUE	The value of the qualifying predicate for this instruction. Only valid for IPOINT_BEFORE.
IARG_REG_VALUE	The value of a register, register name follows.
IARG_BRANCH_TAKEN	Non-zero if the branch will be taken, otherwise 0. Only valid for IPOINT_BEFORE.
IARG_BRANCH_TARGET_ADDR	The target address of a branch.
IARG_FALLTHROUGH_ADDR	The IP and slot of the next instruction to be executed. If this instruction is a branch, it is assumed that the branch is not taken.
IARG_THREAD_ID	Thread id, first thread is 0, successive threads are 1, 2, ...

Table 4: Instrumentation Arguments (IARGS) for PIN_InsertCall(IPOINT, INS, AFUNPTR, *iarg1, iarg2, ..., iargN*, IARG_END)

format. This often results in students being limited to building only one or two simulators per semester due to time constraints, furthermore the time is spent on details of programming/software engineering and not on analyzing the results of the architecture simulators.

However, if the students had an opportunity to first interact with the various simulators as the class progresses and are assigned simple assignments of optimizing a pre-built simulator by changing the parameters etc. they would be able to get a good feel of how design parameters affect performance. Further more, it would allow the instructor to design the course such that at the end of the semester a project could be assigned where the students could pick a simulator that intrigued them and build it from bottom up. The advantage of that is that students often reach deep into their work when they are keenly interested in it. Exploiting that in them would guarantee that they extract the most from the class.

4.2 Applying PIN

In Section 2 we gave a brief introduction to PIN as a tool and hereby wish to reflect upon why PIN would prove effective in the classroom environment. PIN has certain characteristics which we believe makes it a unique experience for projects in the classroom. They are as follows:

Unique simulation environment: PIN's simulation environment is perceived uniquely by students because they realize the input program are binary tools that they use regularly such as ls, sort, grep etc. instead of presenting their simulators with traces. The students run the programs on real hardware rather than on an abstract software layers which limit some students from understanding how a simulator is working.

Reduced development time: The development time for simulators is dramatically cut short and thus allows students to focus more on actual data analysis; students often loose precious time in just building the simulator infrastructure.

Flexibility: PIN is a valuable teaching tool because of its flexibility in being able to support simulation environments as well as being to monitor compiled binaries both statically and dynamically. Often students are expected to cope with multiple tools because no one tool provides enough flexibility to be able to last through the run of the course. The students could use PIN all through their computer architecture without having to change tracks to using a new environment.

To give a generic view of how PIN could be used in the classroom; we present through Figure 1 the various analysis that students can do as part of their class projects in computer architecture coursework. The example illustrates cache and value profile modules being used on the entire program. This can be achieved by instrumenting every single instruction in the program using the PIN callback function PIN_AddInstrumentInstructionFuction(...). In procedural level instrumentation for call_A() and call_C(), instrumentation is injected again by using the PIN_AddInstrumentInstructionFuction(...) function; however the instrumentation range is dedicated only to the range of instructions that fall in the scope of those procedure calls. Detailed application programming interface (API) is available online at [13]. We see that call_A() and call_C(); data capture varies from collecting fine grained opcode statistics to generic profiling.

The currently released PIN kit contains various tools for use, a few of which are the data cache, branch predictor simulators, a tool to measure instruction counts and to analyze the latency of load instructions. Also contained are tools that perform profiling of the program; time spent in procedural calls etc.. A tool to collect detailed program traces is also available. Details of all the tools mentioned are available at the pin website [13].

4.3 PIN's accessory tools/libraries

PIN provides fine-grained analysis with excellent flexibility however a limitation that often tends to exist is with students being unable to analyze the data being collected. Thus we provide complementary tools to help the students.

Statistical analysis package. The students are provided with *data analysis* packages. These play a significant part in using PIN as a teaching tool because it voids the students from having to analyze raw data by hand. Collected data that has been processed as in Table 5 and Figure 2 make it easy for students to comprehend their program behavior better.

CUT - Colorado Utility Tool: Data generated in Table 5 shows detailed analysis of a load instruction at memory address 0x200000000000db20 that was pro-

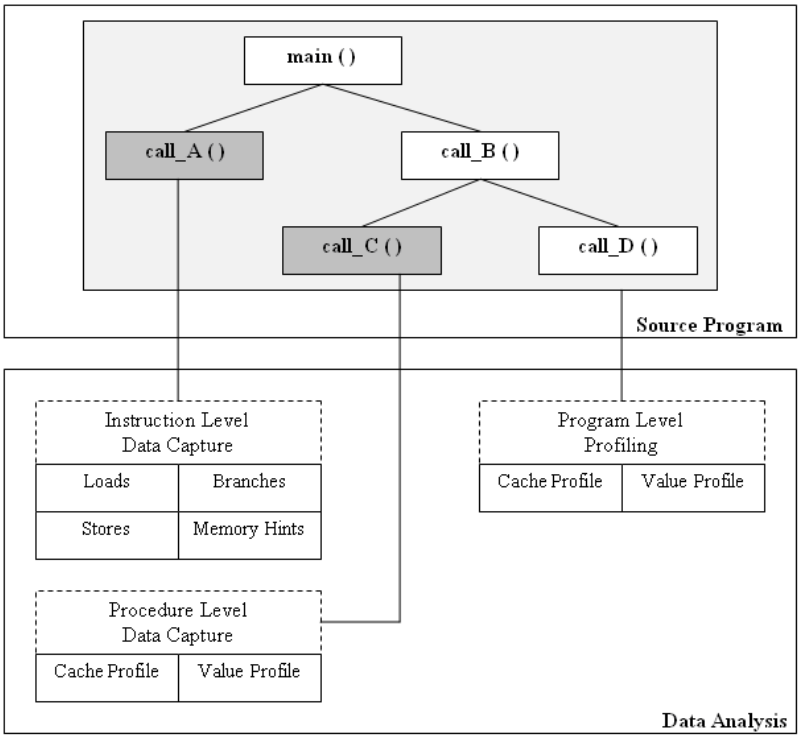


Figure 1: Instruction and Program level data capture

Colorado Utility Package (CUT): Data Analysis	
0x2000000000000db20-samples	7
0x2000000000000db20-mean	119
0x2000000000000db20-stddev	220.596
0x2000000000000db20-conf90	175.095
0x2000000000000db20-conf95	220.724
0x2000000000000db20-conf99	335.541
0x2000000000000db20-quantile-samples	7
0x2000000000000db20-Samples-6	3
0x2000000000000db20-Percent-6	42.8571
0x2000000000000db20-CumPercent-6	42.8571
0x2000000000000db20-Samples-12	1
0x2000000000000db20-Percent-12	14.2857
0x2000000000000db20-CumPercent-12	57.1429
0x2000000000000db20-Samples-45	1
0x2000000000000db20-Percent-45	14.2857
0x2000000000000db20-CumPercent-45	71.4286
0x2000000000000db20-Samples-158	1
0x2000000000000db20-Percent-158	14.2857
0x2000000000000db20-CumPercent-158	85.7143
0x2000000000000db20-Samples-604	1
0x2000000000000db20-Percent-604	14.2857
0x2000000000000db20-CumPercent-604	100

Table 5: Program level statistical analysis of a load instruction at address 0x2000000000000db20

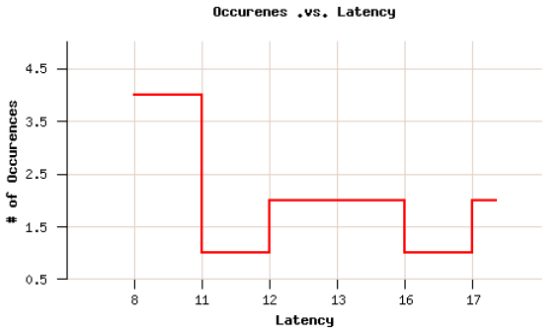


Figure 2: Dynamically generated load latency histogram

filed through the entire run of the program. The data is interpreted as follows: The first five lines reflect the samples, mean, standard deviation followed by the confidence intervals respectively. Thereafter the data reflects each of the individual samples; the load latencies every time the instruction was executed. *quantile-samples* is the total number of occurrences of this load instruction. Samples-*x* and its corresponding entry represent the frequency of occurrence of the load for latency *x*. Respectively followed by the percentage and cumulative percentages of occurrences of that latency for the given load instruction. The graph in Figure 2 reflects another data set; the graph was generated automatically through our CUT package.

Our analysis's package allows students to easily create digestible reports and graphs for post-run analysis.

Profiling structure library: Aside from PIN students are provided with a number of code modules for increasing the flexibility of the system as well as reducing development time. First, a set of data structure modules are provided that include generic caches, hash tables, time-line event record books, and symbol tables. In addition, a library module for value and memory address profiling is available for seamless integration with PIN instrumentation calls. The value profiler can be directed to keep a top*N* value (TNV) table for register operand values. The address profiler can track constant, stride, and finite-context matched patterns of addresses for load and store instructions.

Sampling interface: PIN provides a sampling interface that directs the binary instrumentation process. There are several management controls (known as PIN-pointing) which support triggering of the user-inserted instrumentation calls after an initialization period of instruction execution events or for periodic sampling. More detailed controls allow the instrumented code events to be called for a set interval of instruction executions after each periodic point has been reached.

4.4 PIN Projects

Numerous projects could be given out to students to select. The following are a few of those that could be used as a guide line:

Architectural models: It is helpful for students to see how the various architecture units of the system are performing while a program is running. A student could select/write the modules of interest such as: a Register Stack Engine(RSE), branch predictor, cache simulator etc.

Profiling: Profiling is a very common occurrence when studying program behavior. It serves as the fundamental step prior to doing in depth analysis. Thus profiling in conjunction with the data analysis packages would facilitate the study and generation of reports that reflect how the system performed through the run of the program based on the analysis the user has asked for. Some of the profiling tools could extend from simply collecting the opcode frequencies to observing the load referencing patterns where students may study how far ahead loads were fetched and record the actual use of the load. Yet another profiling tool could be one that looked for redundant loads and stores to the same location. PIN can facilitate this by looking at the source and destination registers and comparing them to see if the values are identical.

Trace collection: PIN is able of collecting traces of the programs as they execute. While there exist many tools that facilitate such a feature; the uniqueness of PIN is that the trace could actually consist of register values that are present at the time the instruction is being executed.

4.5 Future Development

Run-time program analysis is vital to understanding the essence of computer programming and not limited to comprehending the effectiveness of modern architecture designs. It applies to program writers at all levels, from students to software developers and especially to those involved in that area of research. It is vital to understand how a high level language such as C/C++ gets transformed into low-level code that runs on the underlying modern architecture since it affects the performance of the machine.

Lately interest has been growing in optimizing programming dynamically during their execution time - dynamic optimizations. With binary instrumentation tools, interesting research topics such as code caches, feed-back directed optimizations etc. may be simplified and presented in the classroom as projects to encourage research interests in students. Students are not aware of such concepts and presenting them with such ideas could give way to newborn interests in pursing the field further.

5 Conclusion

Instrumentation tools can be a vital teaching tool in the classrooms. We propose PIN as such a tool because it presents the students with live runs real compiled binaries on real hardware on a custom simulator if desired while also facilitating fine/coarse grained analysis and instrumentation functions. We believe that the ability to be able to merge all those into one program to be used as a teaching tool is of tremendous significance.

The tool would be extremely vital in helping students understand how programs are to be analyzed and how how their behavior can be monitored while still being able to teach them and making them understand the architecture upon which their computer programs run.

Acknowledgments

We would like to thank the Intel and Hewlett-Packard Corporations for the donation of Itanium Processor Family (IPF) systems. The systems allowed us to gain valuable insight and experience with the Itanium version of the PIN tool. Grant assistance in support of this work was provided by the Intel Corporation. We also extend our gratitude to Professor Dirk Grunwald at University of Colorado at Boulder for sharing with us his data analysis package - the Colorado Utility Tool (CUT).

References

[1] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI'94), pages 196–205, 1994.

[2] 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

[3] Bryan Buck and Jeffrey K.Hollingsworth. An API for runtime code patching. The International Journal of High Performance Computing Applications, 14(4):317-329, Winter 2000.

[4] Doug Burger and Todd M. Austin and Steve Bennett "Evaluating Future Microprocessors: The SimpleScalar Tool Set" Technical Report 1996-1308, 1996.

[5] Greg Wolffe, William Yurcik, Hugh Osborne, and Mark Holliday, published in the Proceedings of the 33rd Technical Symposium of Computer Science Education (SIGCSE 2002), ACM Press, Northern Kentucky USA, Feb/March 2002.

[6] http://www.hpl.hp.com/research/linux/perfmon/index.php4

[7] Jason Blome, Manish Vachhajarani, Neil Vachhajarani, and David I. August., "The Liberty simulation environment as a pedagogical tool,", Workshop on Computer Architecture Education (WCAE), June 2003.

[8] Kim Hazelwood and Michael D. Smith. "Generational Cache Management of Code Traces in Dynamic Optimization Systems," 36th Annual International Symposium on Microarchitecture (MICRO-36). San Diego, December 2003, pp. 169-179.

[9] L.DeRose, Y. Zhang, and D. Reed. Svpablo: A multilanguage performance analysis system. In Proc. 10th International Conference on Computer Performance Evaluation - Modeling Techniques and Tools- Performance Tools '98, pages 352–355, 1998.

[10] London, K., Moore, S., Mucci, P., Seymour, K., Luczak, R. "The PAPI Cross-Platform Interface to Hardware Performance Counters," Department of Defense Users' Group Conference Proceedings, June 18-21, 2001.

[11] M. Burtscher. VPC3: A Fast and Effective Trace-Compression Algorithm. Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS04). June 2004.

[12] P.Lopez. http://www.gap.upv.es/people/plopez/english.html

[13] Robert S. Cohn, Intel Corporation. http://systems.cs.colorado.edu/Pin

[14] Vertical Profiling: Understanding the Behavior of Object-Oriented Applications Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, Michael Hind, 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.

A Simulation Applet for Microcoding Exercises

Roland N. Ibbett
Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh
Edinburgh EH9 3JZ, UK

Abstract

At the University of Edinburgh we have used a Hierarchical Computer Architecture design and Simulation Environment (HASE) to build a number of architectural models for use in research and teaching. Within HASE, the JavaHASE facility allows models to be translated into applets which can be accessed via the WWW.

The Edinburgh Microcodable Microprocessor Applet (EMMA) was created in response to a need to provide students with a reliable practical experiment on processor design in a Computer Design course. There are currently two versions, a Basic model that can execute single-cycle arithmetic operations and an Enhanced model that can also execute multiply and divide. The Basic model was used successfully by a class of about 80 students in 2003.

I. INTRODUCTION

At the University of Edinburgh we have used a Hierarchical Computer Architecture design and Simulation Environment (HASE) to build a number of architectural models for use in research and teaching. Within HASE, the JavaHASE facility [1] allows models to be translated into applets which can be accessed via the WWW[1]. JavaHASE applets are programmable simulation models with visualisation capabilities that allow activities taking place within the model (data movements, state changes, register/memory content changes, etc) to be displayed on-screen dynamically. Models of Tomasulo's algorithm and the DLX [2] and DASH [3] architectures and, most recently, a microcodable processor (the Edinburgh Microcoded Microprocessor Applet (EMMA)), are currently being used in teaching at Edinburgh. Each model is supported by a web site describing the architecture and the model.

In order for students to be able to carry out exercises using the models, the applets require access to cut and paste facilities, using the clipboard, on the client machine. Although standard security managers for applets do not allow access to the clipboard, the security manager can be configured using a Java policy file to allow clipboard access to applets from specified URLs.

According to Wolffe *et al*'s classification [4], most of the JavaHASE DLX applets [5] are Enhanced Microarchitecture Simulators. These applets are designed to show students what happens inside a processor as programs are executed in a simple pipelined system, in a system with a scoreboard and in a dual-issue (VLIW) system with predication. In a practical exercise involving the DLX with Scoreboard applet shown in Figure 1, for example, students are given an assembly code sequence representing a simple implementation of a scalar (dot) product loop and are asked to run the simulation and note where hazards occur. They are then asked to reorder the code to eliminate or at least reduce the effects of these hazards. As a further optimisation they are asked to unroll the loop to include two iterations of the algorithm in one program loop.

The JavaHASE Tomasulo's algorithm applet (Figure 2) is both an Enhanced Microarchitecture Simulator and a Historical Machine Simulator in Wolffe *et al*'s classification, since it models closely the original system used in

[1] www.icsa.inf.ed.ac.uk/research/groups/hase/

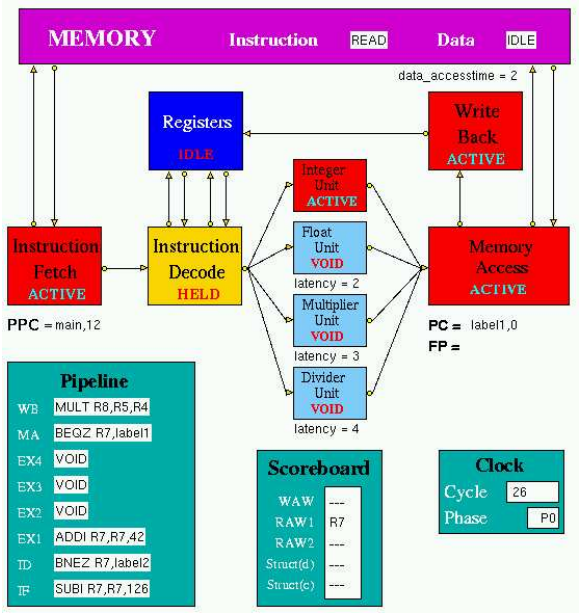


Figure 1: DLX with Scoreboard

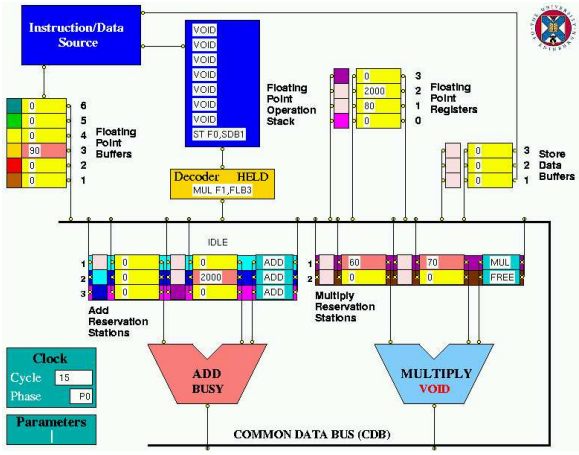


Figure 2: Tomasulo's Algorithm Applet

the IBM System/360 Model 91 Floating-point Unit [6]. The algorithm is difficult to explain to students without a dynamic demonstration, so the applet is designed to show, in particular, how the tags move around the system during program execution. The 360 processor and memory are represented in the model by an Instruction/Data Source Unit which stores a sequence of instructions and a set of data values. These are sent to the Floating-point Operation Stack in sequence. The website explains the operation of the algorithm in terms of the sequence of instructions contained in the applet when it is downloaded but instructors and students can load their own code and data into the applet's memories.

The JavaHASE DASH Cluster Model is also, in a sense, a Historical Machine Simulator, though it was designed to show the operation of the DASH snoopy bus cache coherence protocol rather than to be a complete historical model and is therefore better thought of as a Multi-Processor Simulator in Wolffe *et al*'s classification scheme. The cluster consists of four nodes attached via a cluster bus to a memory. Each node contains a primary cache, a secondary cache and processor that is simply modelled as a source of addresses.

In a practical exercise, students are asked to display the contents of the processors (*i.e.* the lists of addresses) and the caches and, by running the animation in single shot mode, to observe what happens as the simulation proceeds. They are then asked to submit listings of the addresses in each of the four processors, annotated to show the responses of the caches to each access.

EMMA was created in response to a need to provide students with a reliable practical experiment on processor design in a Computer Design course and falls into Wolffe *et al*'s Simple Hypothetical Machine Simulator category. It exists in two versions, the Basic version (EMMA-1) can execute single-cycle arithmetic operations whilst the Enhanced version (EMMA-2) can also execute multiply and divide. Of the simulators identified in [4], only the MicroArchitecture Simulator[2] appears to offer similar facilities to EMMA but it is not web-based and requires the use of MacOS. Most currently available processor simulation applets are targeted at demonstrating processor operation at the register transfer level by allowing students to enter their own assembly code, *e.g.* the Little Man Computer[3] and cpu-sim[4]. cpu-sim shares some characteristics with EMMA in that it shows the movement of information inside a processor by means a 'worm' moving along the data paths

[2] www.dslextreme.com/users/fabrizioo/msim.html
[3] www.acs.ilstu.edu/faculty/javila/lmc/
[4] www.cs.gordon.edu/courses/cs111/

as assembly code instructions are executed, but the micro-execution sequence for each instruction is predefined within the model.

II. EMMA

A. *Processor Architecture*

EMMA is a load/store, register-register arithmetic processor implemented as shown in Figure 3. It was designed with simplicity and elegance in mind but was nevertheless intended to give students a feel for issues which can arise in the design of real systems. It uses a Harvard architecture, with separate instruction and data memories. This is not only convenient from a simulation perspective (instructions are represented using a C++ struct which allows them to be displayed in readable assembly code format whilst data is represented in 32-bit integer format) but also reflects the use of separate instruction and data caches in most real microprocessors. The microcode word is also 32 bits, divided up into eight hex characters, with one or more hex characters being assigned to each unit.

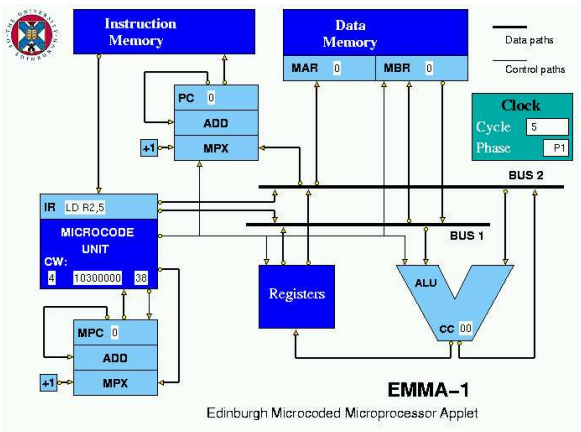


Figure 3: Basic EMMA

The units making up the processor itself are two data buses (BUS 1 and BUS 2), an ALU, a Register Unit (containing 16 32-bit registers, with R0 always set to 0), a Program Counter, a Microcode Program Counter and a Microcode Unit (which also contains the Microcode Memory). The Program Counter (PC) is, in effect, the memory address register for the Instruction Memory with the Instruction Memory buffer register being the Instruction Register (IR) in the Microcode Unit. The Data Memory has built-in Memory Address and Memory Buffer Registers (MAR and MBR) connected to BUS 1 and BUS 2 respectively.

EMMA operates on a two phase clock (shown as P0 and P1 in the Clock entity display). In clock cycles in which they are active, each unit executes its internal actions in the first phase of the clock (P0) and sends out a result packet in the second phase (P1). The Microcode Unit, for example, reads its microcode memory in P0 and sends the appropriate microcode fields to other units, if they are to be activated, in P1.

B. *Instruction Set*

The instruction set is prefined in the applets. In the Basic version it includes absolute jumps (JUMP and JREG) and relative branches (BEQZ and BNEG), loads (LD, LDL, LDX) and stores (ST and STX), register-register operations (ADD, SUB, AND, OR, XOR, SLL, SRL, SRA) and register-literal arithmetic operations (ADDL, etc), as shown in Table 1, and a STOP instruction, which stops the simulation. RD is the destination register, RS, RS1 and RS2 are source registers and L is a Literal (immediate) operand.

JUMP	PC = L
JREG	PC = (RS)
BEQZ	PC = PC + L if ALU = 0
BNEG	PC = PC + L if ALU -ve
LD	RD = Memory(RS)
LDL	RD = L
LDX	RD = Memory(RS + L)
ST	Memory(RD) = RS
ADD	RD = RS1 + RS2
ADDL	RD = RS1 + L

Table 1: Basic Instruction Set

The Enhanced version also includes multiply (MUL, MULL) and divide (DIV, DIVL) operations and two undefined register-register operations (OP1 and OP2) which can be used for functions of the student's own choice, *e.g.* RD = RD + (RS1 * RS2).

C. *Processor Design*

1) *Microcode Unit:* The Microcode Unit has one input from the Instruction Memory and one from the Microprogram Counter (MPC). (There is also an input, not shown in the display, carrying the Condition Code from the ALU.) The microcode memory contained in the Microcode Unit is addressed by the function field of the Instruction Register in the clock cycle in which a new instruction is received from the Instruction Memory and by the Microcode Program Counter (MPC) in subsequent clock cycles. (Erroneous microcode can cause both to occur simultaneously; this is automatically detected and displayed as an error.)

The Microcode Unit has an output control path to each of the units it controls and three output data paths, one to the MPC and one to each of the buses. These data paths are activated by bits in the most significant hex character in the microcode word (Table 2). This position was chosen because it leaves the most significant bit unused and thus avoids problems with negative numbers.

The second hex character controls conditional execution. The Microcode Unit always sends a control code to the MPC but only sends to the other units if

- neither conditional bit is set
- the *execute if condition is true* bit is set and the condition is true
- the *execute if condition is false* bit is set and the condition is false

Conditional branches can thus be implemented by executing either a microcode instruction that adds the Literal operand to PC or an instruction that adds +1 to PC.

2) *PC and MPC:* The Program Counter and Microprogram Counter units behave identically. They contain the relevant register together with an adder which receives one input from the register itself and the other from a multiplexer (MPX) which has inputs of +1 or a value taken from BUS 2 in the case of PC or the Microcode Unit in the case of MPC. Each has two outputs: Output1 is connected back to the adder (and is permanently enabled), Output2 is enabled under microcode control.

Whenever PC or MPC is activated by receipt of a microcode packet (it should also have received appropriate data packets), it enables the appropriate inputs to the multiplexer and sends the result of the addition to Output1, and thence back to its own adder input and to Output2 if the corresponding microcode bit is set.

3) *The Buses:* The buses (BUS 1 and BUS 2) have a number of input connections but should of course receive data from only one of them in any one clock period (in P1). Inputs which do not receive data are set to zero. The inputs are internally ORed together (simulating a wired-OR bus) and the result sent to all the outputs.

4) *Data Memory:* The microcode for the Data Memory contains two bits which control its input registers (MAR and MBR), one bit which activates its output register (MBR) and a Read/Write bit. For a read operation (Read/Write = 0), the address received from BUS 2 in P1 of one clock period is copied into MAR in P0 of the next clock period, the memory is read and the result copied into MBR. In P1 the value in MBR is sent to BUS 1. For a write operation (Read/Write = 1), the address sent from BUS 2 is copied into MAR in P0 of next clock period, the data value sent from BUS 1 is copied into MBR and the value is written into the memory.

5) *Registers:* The Registers unit contains 16 general purpose registers, with R0 being permanently set to 0. It receives input values from the ALU and has two outputs connected to BUS 1 and BUS 2. Whenever the Microcode Unit sends a microcode command to the Registers, it appends the appropriate source and destination register numbers extracted from the instruction in IR. In an instruction such as ADD RD RS1 RS2, the value in register RS1 is sent to BUS 1 and that in RS2 to BUS 2.

6) *ALU:* In the Basic version of EMMA, the ALU has two microcode control fields, one to control its inputs (one from each of BUS1 and BUS2) and outputs (one to the Registers and one to BUS2) and one for the function. It executes Add, Subtract, AND, OR, XOR, Shift Left Logical, Shift Right Logical and Shift Right Arithmetic. At the end of each operation it sets the Condition Code bits: CC0 = 0 if the result is 0, CC1 = 1 if the result is negative.

D. *Microcode Format*

The Microcode Unit contains a 128-word microcode memory, with each word having the following format:

Label	Microcode Word	Address

The Label is a string of characters used only for readability purposes. The Microcode Word is represented in hexadecimal format and is structured as shown in Table 2. The bits are clustered into hex characters, two for the Microcode Unit itself, one each for the MPC and PC units, one for the Registers, one for the Memory and two for the ALU.

The (integer) Address field is used for jumps within the microcode memory. The first 32 locations form a jump table indexed by the function number, except for location 0 which contains the (one) microcode word needed to implement the JUMP instruction.

If MPC is loaded with a new address (*i.e.* not MPC+1), there is a 1-clock delay before the new value is returned to the Microcde Unit. In this branch slot, the Microcode unit sets the microcode word to 0x00D0000, thus incrementing MPC automatically in the next clock.

III. ENHANCED EMMA

The Enhanced version of EMMA differs from the Basic version in a number of ways. The ALU is shown in more detail (Figure 4) and contains an additional counter (D) which is required for the implementation of divide. The microcode memory is doubled in size to 256 words and is logically divided into two sections, the Standard Microcode memory (addresses 0-127) and the Alternative Microcode memory (addresses 128-255). The Alternative Microcode allows for the implementation of multi-cycle operations such as multiply and divide. Address 255 is special in that when accessed, it halts the simulation and can thus be used to prevent an attempt to divide by 0, for example.

Unit	Bit	Signal
00	00	Not used
	01	Addr/Lit to BUS1
Mcode	02	Addr/Lit to BUS2
	03	Mcode Addr to MPC
01	04	Select ~CC0
	05	Select CC1
Mcode	06	Execute if True
	07	Execute if False
02	08	Input1 from MPC
	09	Input2 (+ 1)
MPC	10	Input3 from Mcode
	11	O/p To Mcode
03	12	Input1 from PC
	13	Input2 + 1
PC	14	Input3 from BUS2
	15	O/p to I_Memory
04	16	Not used
	17	Write
Regs	18	Source1 to BUS1
	19	Source2 to BUS2
05	20	MAR Input
	21	MBR Input
Data	22	Read/Write
Memory	23	MBR Output
06	24	Input1
	25	Input2
ALU	26	O/p to Regs
	27	O/p BUS2
07	28	Not used
	29	Function
ALU	30	Function
	31	Function

Table 2: Microcode Format

Words in the Alternative Microcode memory have the same basic format as words in the Standard Microcode memory but the Data Memory field in the standard format is replaced by an additional ALU field used to

control the extra facilities needed in the ALU to implement multiply and divide operations (Table 3). In addition, the Address field in the microcode word is sent to the ALU for use as a Literal (immediate) operand.

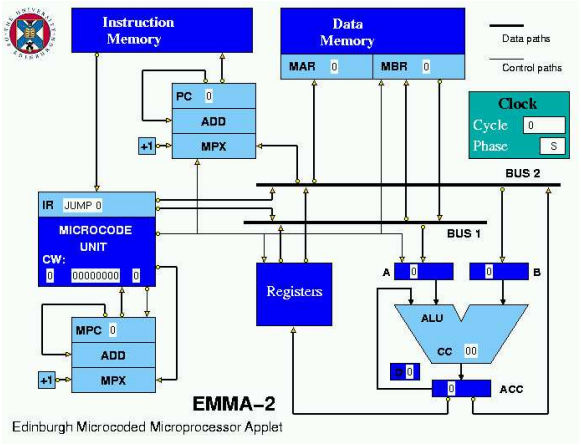


Figure 4: Enhanced EMMA

The Enhanced version of EMMA is upwardly compatible with the Basic version, so microcode written for the Basic version can be used unchanged in the Standard Microcode section of the Enhanced version and will achieve the same effects.

Unit	Bit	Signal
04	16	Dest. to BUS2
	17	Write
Regs	18	Source1 to BUS1
	19	Source2 to BUS2
05	20	Inhibit A to alu
	21	Inhibit B to alu
ALU	22	Enable ACC to alu
	23	Inhibit B to alu
		if A<31> = 0
06	24	Input1
	25	Input2
ALU	26	O/p to Regs
	27	O/p BUS2
07	28	Function
	29	Function
ALU	30	Function
	31	Function

Table 3: Alternative Microcode Format

Other differences when using the Alternative format are that the Destination register in the Registers unit can be routed to BUS 2 and the ALU has extra functions as shown in Table 4. Four of these functions operate directly on the two input registers (A and B) and two manipulate the counter (D) which can be used to implement Division.

NOP
Reverse Subtract
Negate A
Negate B
Shift A → by literal & set CC
Shift B ← by literal
Set D = 17
Decrement D & set CC

Table 4: Extra ALU Functions

The shift functions take as their argument the literal value in the microcode. This argument can be positive (for use in multiplication) or negative (for use in division). When the argument is negative the shift is in the opposite direction to that shown in Table 4. In the case of a left shift on A, the value shifted into the least significant bit is the inverse of CC1, (*i.e.* = 1 if the value in ACC is non-negative). Normally the argument is +1 or -1 but -16 is also required for division.

A. *Muliply and Divide*

1) *Multiply:* The ALU is designed to be able to multiply together the numbers in A and B by repeatedly adding the value in input B to the accumulator (ACC). In each cycle, either the value in B or zero is added to ACC according to the value of the least significant bit of A; A is then shifted right one place and B is shifted left. The operation stops when A becomes zero (whenever A is shifted, the Condition Code bits are set according to the new value in A). This algorithm only works for positive values of A. If A is initially negative, it must first be negated and the final result negated before being returned to the destination register.

2) *Divide:* The ALU is designed to be able to divide 16-bit numbers by first shifting the divisor in input B left 16 places and then repeatedly subtracting it from the value

in ACC, testing for a negative result and shifting B right one place. If the result is negative, B is added back to ACC before repeating the cycle. At the start of the operation, the Quotient, in A, is set to zero (this can be done by loading it from the bus but without sending a value to the bus beforehand). After each subtraction, A is shifted left one place with the value shifted into the least significant position being 1 if the value in ACC is non-negative (as described above). At the end of the operation, A contains the quotient of the result, whilst the remainder is in ACC. However, there is no way in the current version of EMMA to return each value to a different register.

This algorithm only works for 16-bit positive numbers. If either value is negative, it must be negated at the start and the result negated, if appropriate, at the end. The ALU simulation code itself checks its inputs and stops the simulation, with a warning, if a number is out of range. It is also essential not to proceed with a divide if the divisor is zero.

IV. USING THE APPLETS

When downloaded, each applet contains the microcode for the JUMP and LD instructions. The applet automatically executes a JUMP 0 instruction at the start of each simulation. The Data Memory contains values in each of its first 32 locations equal to their address whilst the assembly code contained in the Instruction Memory consists of just two instructions:

```
LD R2 5
STOP
```

Suggested exercises for the students are:

- Using the Basic version, implement the remaining microcode for all single cycle instructions and write an appropriate assembly code program to demonstrate that they work.
- Using the Enhanced version, implement the multiply and divide instructions and demonstrate that they work on suitable test data, including all possible combinations of positive and negative numbers.
- Implement OP1 and OP2 with new functions.

The Basic version requires around 100 lines of microcode in total, including the jump table at the start. The arithmetic instructions (ADD/ADDL, etc) require 3 lines each, BEQZ, BNEG and JREG each require 2, while LD, LDL and LDX require 2, 3 and 5 respectively. The Basic version was used by around 80 students in Autumn 2003. Most of them got most of the instructions working correctly, though some had difficulties, particularly with LDX and STX. Student reaction to using the applet was positive; they felt that it gave them a good insight into how a processor works.

In the Enhanced version, MUL/MULL and DIV/DIVL can be implemented using 18 and 43 microcode instructions respectively. MULL and DIVL use most of the same code as MUL and DIV but each requires different initial microcode instructions to load the operands at the start.

V. CONCLUSION

JavaHASE applets have been successfully used in teaching in a number of courses, either as demonstrations or for practical exercises. Because they are accessible via the WWW, they can be used by students who wish to work off campus or at times outside normal hours. The most recent applet (EMMA) has been used as a replacement for aging laboratory equipment.

VI. Acknowlegements

The development of HASE has been supported by the UK EPSRC through grants GR/J43295 and GR/K19716. Particular thanks are due to Frederic Mallet who created the JavaHASE facility, to the Computer Design class of 2003/4 for their forbearance during development of the Basic model and to Eric McKenzie who taught them.

References

[1] F. Mallet and R.N. Ibbett, "JavaHASE: A Web-based simulation environment", *SCSC'03*, SCS, 2003.

[2] J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 1996.

[3] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens et al "The DASH prototype: Implementation and Performance", *Proc ICSA*, 1992, pp 82-103.

[4] G.S. Wolffe, W. Yurcik, H. Osborne and M.A. Holliday, "Teaching Computer Organization/Architecture With Limited Resources Using Simulators", *SIGCSE'02*, 2002,

[5] R.N. Ibbett and F. Mallet, "Computer Architecture Simulation Applets For Use In Teaching", *Proc FIE 2003* IEEE, Nov 2003.

[6] R.M Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research & Development, Vol 11*, 1967, pp 25-33.

The SimCore/Alpha Functional Simulator

Kenji Kise[†,‡], Takahiro Katagiri[†,‡], Hiroki Honda[†], and Toshitsugu Yuba[†]
[†] Graduate School of Information Systems
The University of Electro-Communications
[‡] PRESTO, Japan Science and Technology Agency (JST)
{kis, katagiri, honda, yuba}@is.uec.ac.jp

Abstract

We have developed a function-level processor simulator, SimCore/Alpha Functional Simulator Version 2.0 (SimCore Version 2.0), for processor architecture research and processor education. This paper describes the design and implementation of SimCore Version 2.0. The main features of SimCore Version 2.0 are as follows: (1) It offers many functions as a function-level simulator. (2) It is implemented compactly with 2,800 lines in C++. (3) It separates the function of the program loader. (4) No global variable is used, and so it improves the readability and function. (5) It offers a powerful verification mechanism. (6) It operates on many platforms. (7) Compared with sim-fast in the SimpleScalar Tool Set, SimCore Version 2.0 attains a 19% improvement in simulation speed.

1 Introduction

Various processor simulators [1, 2] are used as tools for processor architecture research or processor education. The environment in which a processor simulator can perform is improving dramatically due to the increased speed of PCs and the growing use of PC clusters. However, the time needed for simulator construction increases as the architectural idea to be implemented increases in complexity. In many cases the evaluation with a simulator can be finished in several weeks, although several months are needed for the construction of the simulator, even if the simulator is developed with existing tools.

SimpleScalar Tool Set [3] and SPIM [4] are well-known processor simulators used for such purposes as processor research and education. But, since SimpleScalar can be implemented as a high-speed simulation, it is not a code that can easily be modified. Similarly, SPIM may not be readable [1]. Although there has been much research on processor simulator speedup [5], there are few simulators which make readability a priority.

[1] We say readable to mean *enjoyable and easy to read.*

In addition to its utilization in processor research and education, a processor simulator is vital as a module in a parallel computer simulator, embedded system emulator, and so on. In these various uses, in addition to high speed, a process simulator must have high readability and be easy to use.

We previously constructed an Alpha [6] processor simulator named SimAlpha [7]. Its design policy was to keep the source code readable and simple. This paper describes the design and implementation of a new generation of processor simulator named SimCore/Alpha Functional Simulator Version 2.0 (SimCore Version 2.0) derived from SimAlpha. SimCore Version 2.0 is a function-level simulator, which satisfies the requirements of high readability and high-speed execution simultaneously. In general, function-level simulators cannot be used to measure processor performance. However, it can be used to collect basic data, such as the number of executed instructions, the hit ratio of a branch prediction and cache, and the ideal instruction level parallelism of a program. It is also vital as a base for preliminary evaluation of detailed clock-level simulation and as a module for verification.

The rest of this paper is organized as follows. Section 2 describes the design and implementation of SimCore. Section 3 reports our quantitative evaluation results. Section 4 is a discussion of related works. Section 5 contains some concluding remarks.

2 Design and implementation

SimCore Version 2.0 features the addition of various functions, an increased number of platforms, and simulation speedup, while inheriting the high readability of Version 1.0. In this section, the function of SimCore is summarized, then the design and implementation issues are discussed.

2.1 Functions

SimCore offers many functions as a function-level or instruction-level simulator. Except for the fact that the tar-

```
 1  # SimCore-Loader a.out > aout.txt
 2  # SimCore aout.txt
 3  SimCore/Alpha Functional Simulator Ver 2.0
 4  hello, world
 5
 6  ==========================================
 7  == SimCore Version 2.0 2004-01-08
 8  == 0 million code ( 2070 code) 0.019 MIPS
 9  == SimCore takes 0 min 0 second(109375 usec)
10  == SimCore starts at Wed Apr 7 18:34:20 2004
```

Figure 1. A sample output of the "hello, world" program.

```
 1    lines   words   bytes  filename
 2       19      54     525  sim.cc
 3       75     188    2067  chip.cc
 4      231     833    6833  instruction.cc
 5      297     872    7221  memory.cc
 6      468    1711   12817  arithmetic.cc
 7      168     401    3915  debug.cc
 8      578    1100   12783  syscall.cc
 9      378    1021    9865  etc.cc
10      550    1724   16565  define.h
11     2764    7904   72591  ---- total ----
```

Figure 2. The number of lines, words and bytes of SimCore code.

get architecture is fixed to the Alpha processor, it has almost the same function as sim-fast in SimpleScalar.

A fundamental function is to simulate an application program for each instruction. As an example, the commands to simulate the well-known program which prints "hello, world" and its output are shown in Figure 1. Part of the output is formatted so that it is easy to see. The 1st line is a command to translate the application program into a SimCore input file (see Section 2.3). The 2nd line is the command to start SimCore. The 4th line is the output of the application program. The 6th through 10th lines are a report of the simulation. In this example, 2,070 instructions were executed and 109,375 μsec was required for the simulation.

Other implemented functions are enumerated below:

- The function to measure the frequency of appearance of the executed instructions (instruction mix).
- The function to display the history of the executed system calls.
- The interactive debugging function to display the contents of the memory or register at a specified time.
- The debugging support function to output the contents of the main registers for all executed instructions.

2.2 Compact description

A compact description was one goal of the design policy of SimCore. SimCore is written in C++ and the code size containing the include file is small at **2,764 lines**. The number of lines of each file is summarized in Figure 2. Because functions are somewhat different, a direct comparison cannot be made, but the number of lines of other simulators are given for reference. The code for compiling sim-fast in SimpleScalar is 15,566 lines. The code of SPIM is 14,213 lines.

The main reason why such a compact description was attained is that the fundamental software architecture differs from a conventional simulator design.

The part of the main loop in SimpleScalar which processes one instruction is shown in Figure 3. Only a portion

```
 1  switch (op){
 2   case ADDQ:
 3     regs.regs_R[(inst & 0x1f)] =
 4       regs.regs_R[(inst >> 21) & 0x1f] +
 5       regs.regs_R[(inst >> 16) & 0x1f];
 6     break;
 7   case MULQ:
 8     regs.regs_R[(inst & 0x1f)] =
 9       regs.regs_R[(inst >> 21) & 0x1f] *
10       regs.regs_R[(inst >> 16) & 0x1f];
11     break;
12  }
```

Figure 3. The main loop implementation of sim-fast.

of an add instruction (ADDQ) and a multiply instruction (MULQ) is shown.

The type of instruction to be processed is distinguished in the switch sentence of the 1st line in Figure 3. Then, the block from the case sentence of the 2nd line through the 6th line describes the operation of addition. The sum of the value of the registers is calculated in the 4th and 5th lines, and the result is stored in a register in the 3rd line. The block from the 7th line through the 11th line describes the operation of multiplication. This style, which describes the operation of each instruction independently, is named an **unfolded description style**. This style permits an easy change of operation of an instruction, and the addition of a new instruction. Since the processing which is necessary for each instruction is described, there is the advantage that a high-speed simulation is possible. On the other hand, because the same description appears in two or more parts, there are the drawbacks that management of the source code becomes complicated and the amount of code becomes large.

In order to remove these drawbacks, a style can be considered which extracts the common part of the operation of each instruction and describes its operation gradually with reference to the pipeline structure of a microprocessor. We name this style a **folded description style**.

SimCore adopts a folded description style. The code of the method step which processes one instruction in SimCore

```
 1  int simple_chip::step(){
 2    p->Fetch(&ev->as->pc); /* pipe stage 0 */
 3    p->Slot();             /* pipe stage 1 */
 4    p->Issue();            /* pipe stage 3 */
 5    p->RegisterRead();     /* pipe stage 4 */
 6    p->Execute();          /* pipe stage 5 */
 7    p->Memory();           /* pipe stage 6 */
 8    p->WriteBack();
 9    return ev->sys->running;
10  }
```

Figure 4. The main loop implementation of SimCore. SimCore adopts a folded description style.

```
1  /* SimCore 1.0 Image File */
2  /*** Registers ***/
3  /@reg 16 0000000000000003
4  /@pc  32 0000000120007d80
5  /*** Memory    ***/
6  @11ff97000 00000003
7  @11ff97008 1ff97138
```

Figure 5. A sample SimCore execution image file.

is shown in Figure 4. The class instruction which holds the information for processing one instruction is defined. The p in Figure 4 is the object of the class instruction. The values of the private variables are gradually determined by calling the method (Fetch, Slot, Issue, RegisterRead, Execute, Memory, or WriteBack) of object p which corresponds to the pipeline stage.

A folded description style is close to the description of the microprocessor implemented in a hardware description language such as verilog-HDL. For this reason, the advantage is that the operation of the hardware can be easily captured. Since the common operation is described in one place, management of the code becomes easy. On the other hand, compared with the unfolded description style which describes the operation of each instruction independently, the folded description style is a disadvantage with respect to extendibility and simulation speed.

2.3 SimCore and program loader

SimCore does not have the function of a program loader, which may be regarded as an additional function of a processor simulator.

A simulation is started using the execution image of the original format. An example of the execution image file of SimCore is shown in Figure 5. This execution image file is in text format and consists of two parts. In the first part, values are assigned to some of the registers. In the example of Figure 5, the hexadecimal value 3 is assigned to the 16th register, and the value of 120007d80 is assigned to the program counter. In the second part, the value of some memory is assigned in the same manner. In the example of Figure 5, the value 1ff97138 is assigned to the memory of address 11ff97008.

The execution image file is created from the Alpha binary file with a program named SimCore-Loader [8]. As an example, the command to simulate li (lisp interpreter) from SPEC CINT95 on sim-fast of SimpleScalar is shown.

```
1  $ sim-fast li train.lsp
```

The corresponding command of SimCore is shown. The command in the 1st line generates the execution image file. The simulation is started by the command in the 2nd line.

```
1  $ SimCore-Loader li train.lsp > aout.txt
2  $ SimCore aout.txt
```

The simulation of an identical application is repeated many times with various simulation parameters. In SimCore, once an execution image file is created, only the name of the execution image file is specified to run the simulator. This mitigates any mistake at the time the simulation starts. Moreover, in the utilization of SimCore, knowledge of the executable file form of ELF or COFF[9] is not required. This is an advantage because users can concentrate on the description of the behavior of the processor simulator.

2.4 Elimination of global variables

No global variable is used in the SimCore code. On the other hand, many global variables are used in SimpleScalar and SPIM.

The readability of the source code is improved by eliminating global variables. In addition, elimination of global variables is important from the viewpoint of the function of a simulator. For example, let's consider the measurement of the branch prediction and the cache behavior when switching two or more tasks or threads in a processor. In this case, it is necessary to switch the application at a fixed interval. Since SimCore does not use a global variable, it is possible to describe such behavior compactly.

The main function to switch two applications with a 5,000-instruction interval is shown in Figure 6. In the 5th line and 6th lines, the simple_chip type objects p1 and p2 are generated. Then, 5,000 instructions of one application are processed by the for loop in the 9th line. The method step called in the 9th line is described in Figure 4. Similarly, 5,000 instructions of another application are processed by the for loop in the 10th line. It becomes possible to measure the behavior of the branch prediction or cache in the task-switching environment by inserting a branch or cache module in the code shown in Figure 6.

As shown in Figure 6, SimCore can generate two or more simulation images with a compact description. This makes it easy to use SimCore as a module of a complicated computer system or a parallel computer system. In contrast, it is difficult to generate two or more simulation images in a

```
 1  int main(int argc, char *argv[]){
 2    char *p1 = argv[argc-1]; /* program name*/
 3    char *p2 = argv[argc-2]; /* program name*/
 4
 5    simple_chip *c1=new simple_chip(p1, argv);
 6    simple_chip *c2=new simple_chip(p2, argv);
 7
 8    for(int i=0; i<100; i++){
 9      for(int j=0; j<5000; j++) c1->step();
10      for(int j=0; j<5000; j++) c2->step();
11    }
12
13    delete c2;
14    delete c1;
15    return 0;
16  }
```

Figure 6. The main function, which switches two applications with a 5,000-instruction interval.

single process with a conventional simulator implemented using global variables.

In addition to the elimination of global variables, in order to make it readable, neither goto statements nor conditional compilation is used.

2.5 Verification feature

A processor simulator is complicated software and there is the possibility that it may have various bugs. At the time of development, sufficient verification is necessary.

In conventional simulators, it was difficult to make two or more simulation images in one process due to the existence of global variables. For this reason, the correct execution result was saved to a file and another simulator was verified by comparison with this file. As a means of high-speed verification, SimCore offers a function which embeds the object of SimCore for another simulator such as SimpleScalar. We offer the C language interface to realize this feature.

At the time of development of SimCore Version 2.0, the behavior was verified using this verification feature. Whenever the simulator executed one instruction, all values of the architecture state (a program counter, 32 integer registers, 32 floating point registers) of SimCore and the architecture state of SimpleScalar were compared and we confirmed that the two architecture states were identical during the 20 benchmark simulations of SPEC CINT95 and CINT2000.

Two or more simulation images can easily be generated in one process, as shown in Figure 6. By using this feature, any bug of new simulators under development is discovered at an early stage. Also, by using the feature, developers can prove the validity of their simulator.

2.6 Platforms

SimCore Version 2.0 operates on more platforms than did the previous version. The platforms where the correctness of operation has been verified are enumerated. On seven platforms, operation has been verified with dhrystone and 20 programs from SPEC CINT95 and CINT2000.

- Pentium 4, RedHat 7.3, GCC version 2.96
- Pentium 4, RedHat 7.3, Intel C++ 7.1/8.0
- Pentium 4, RedHat 7.3, PGI Compiler 5.1
- Pentium 4, Cygwin version 2.340, GCC 3.2
- Pentium 4, FreeBSD 4.9, GCC 2.95.4
- Opteron, Turbo Linux 8, GCC version 3.2.2
- Alpha 21264, Tru64, GCC version 2.95.2

On two platforms, operation has been verified with dhrystone.

- UltraSPARCIII, Solaris, GCC version 2.95.3
- MIPS R14000, IRIX6.5, MIPSpro C++

SimCore operates on these major platforms. Because a processor simulator is used in various environments, it needs to support many platforms. In contrast, SimpleScalar Version 3.0 has not been compiled with either an Intel compiler or a MIPSpro compiler.

2.7 Simulation speedup

This section discusses the tuning technique implemented in the main loop of SimCore while keeping the high readability of the source code.

2.7.1 Pipeline frontend reuse

A simple main loop without optimization is shown in Figure 7. As discussed in Section 2.2, one instruction is executed by calling seven methods corresponding to the instruction pipeline from the 3rd line through the 9th line of Figure 7.

Three methods, Fetch, Slot and Issue, in the simple main loop in Figure 7 take charge of fetching the 32-bit instruction code, decoding, and calculating an immediate value, respectively. These methods correspond to the pipeline frontend. If the simulator is processing the statically same instruction, identical processing is repeated each time in these methods. Therefore, the calculation result obtained is saved in memory and it is possible to improve the simulation speed using the calculation result. This speedup technique is called pipeline frontend reuse.

The main loop with the pipeline frontend reuse is shown in Figure 8. The pointer array ib of the type instruction (the 3rd line), which contains the past calculation result,

```
 1  void simple_chip::loop_simple(){
 2    while(ev->sys->running){
 3      p->Fetch(&ev->as->pc); /* pipe stage 0*/
 4      p->Slot();             /* pipe stage 1*/
 5      p->Issue();            /* pipe stage 3*/
 6      p->RegisterRead();     /* pipe stage 4*/
 7      p->Execute();          /* pipe stage 5*/
 8      p->Memory();           /* pipe stage 6*/
 9      p->WriteBack();
10
11      ev->e->retired_inst++;
12      house_keeper(p);
13    }
14  }
```

Figure 7. A simple implementation of the SimCore main loop.

is prepared in the same way as the direct-mapped instruction cache. The number specified by the constant IMSK is the number of entries of the array, and is set as a 64K entry. The index of the array is generated from the program counter (the 9th line). If the program counter from the array (the 10th line) and from the instruction currently executed differs (the 12th line), Fetch, Slot, and Issue (from the 13th line through the 15th line) are executed. Otherwise, the past history is used and the pipeline frontend (from the 13th line through the 15th line) is omitted.

In the method implemented in Figure 8, the rate at which the pipeline frontend can be omitted is the same as the high hit ratio of the direct-mapped instruction cache of the 64K entry. Therefore, in the execution of most instructions, it is possible to eliminate the processing of the pipeline frontend for Fetch, Slot, and Issue.

This method is not new in software engineering. However, the fact that this technique can be implemented without lessening readability is important.

2.7.2 Function call overhead elimination

By adopting pipeline frontend reuse, most of the execution time of SimCore is spent in the pipeline backend. At this time, the function call overhead in the 17th line through the 20th line in Figure 8 becomes notable. In order to reduce this overhead, processing of the four methods which organize the pipeline backend is described as one method, named BackEnd. The code after elimination of the function call overhead, which corresponds to the main loop from the 8th line through the 24th line in Figure 8, is shown in Figure 9. The code is replaced with the method BackEnd in the 10th line.

In SimCore Version 2.0, the main loop shown in Figure 9 is adopted in order to simultaneously attain a compact description and an improvement in speed.

```
 1  #define IMSK 0x0ffff  /* mask of inst_buf */
 2  void simple_chip::loop_reuse(){
 3    instruction **ib=new instruction*[IMSK+1];
 4    for(int i=0; i<IMSK+1; i++){
 5      ib[i] = new instruction(ev);
 6    }
 7
 8    while(ev->sys->running){
 9      int index = (ev->as->pc>>2) & IMSK;
10      instruction *pt = ib[index];
11
12      if(pt->Cpc!=ev->as->pc){
13        pt->Fetch(&ev->as->pc);
14        pt->Slot();
15        pt->Issue();
16      }
17      pt->RegisterRead();
18      pt->Execute();
19      pt->Memory();
20      pt->WriteBack();
21
22      ev->e->retired_inst++;
23      if(ev->sc->slow_mode) house_keeper(pt);
24    }
25  }
```

Figure 8. A main loop with pipeline frontend reuse.

3 Evaluation of simulation speed

In this section, as a quantitative evaluation of SimCore, the simulation speed of SimCore Version 2.0 is measured and compared with sim-fast in SimpleScalar. Moreover, the influence of the technique discussed in Section 2.7 is examined.

A total of eight benchmark programs from SPEC CINT95 are used to evaluate the simulation speed of SimCore. An input parameter is adjusted so that the number of simulated instructions is reduced from about 100 million to 200 million instructions. The binary of the benchmark programs is generated using a DEC C compiler with the optimization option O4.

The data in this section is measured using a PC with two Pentium4 Xeon 2.8 GHz processors and 2 GB main memory running RedHat Linux 7.3.

3.1 Simulation speed comparison

In this section, the simulation speed of SimCore is measured and compared with sim-fast in SimpleScalar. For the evaluation measure, the number of instructions processed per second (MIPS: Million Instructions processed Per Second) is used.

The evaluation result is summarized in Figure 10. The x-asis indicates benchmark names and the average of 8 benchmark programs. SimCore and sim-fast are compiled using GCC with the optimization option O3. The SimCore simulation speed is faster than sim-fast in all of the bench-

```
 1    while(ev->sys->running){
 2      int index = (ev->as->pc>>2) & IMSK;
 3      instruction *pt = ib[index];
 4
 5      if(pt->Cpc!=ev->as->pc){
 6        pt->Fetch(&ev->as->pc);
 7        pt->Slot();
 8        pt->Issue();
 9      }
10      pt->BackEnd();
11
12      ev->e->retired_inst++;
13      if(ev->sc->slow_mode) house_keeper(pt);
14    }
```

Figure 9. A main loop with pipeline frontend reuse and the elimination of function call overhead.

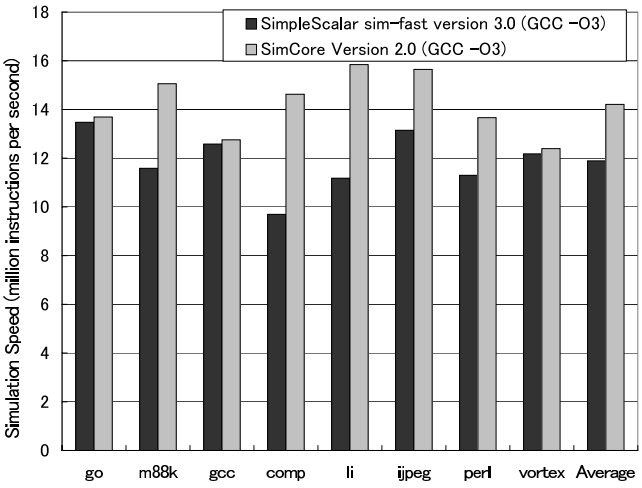


Figure 10. Simulation speeds of sim-fast and SimCore.

mark programs. In particular, in the simulation of compress (comp), SimCore attains the highest speed improvement, which is 50%. In the average of the eight benchmarks, the simulation speed of SimCore is 14.2 MIPS. A speed improvement of 19% is attained compared to the 11.9 MIPS simulation speed of sim-fast.

Next, the simulation speed measured with various compilers and optimization flags are summarized in Figure 11. Five sets of data are shown in this figure for each benchmark. The 1st and 2nd bars from the left are the same data shown in Figure 10.

The 3rd from the left is the data using the Intel C++ Version 7.1 compiler (icc) with optimization option O3 and optimization between files. The simulation speed in this case is an average of 15.3 MIPS. Sim-fast was not able to be compiled using the Intel C++ compiler. SimCore compiled using Intel C++ Version 7.1 compiler with optimization option O3 attains a 28% speed improvement compared with sim-fast compiled using GCC.

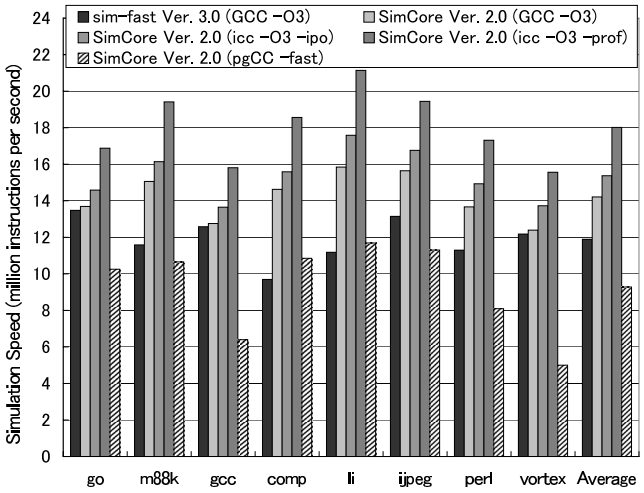


Figure 11. Simulation speed measured with various compilers and optimizations.

The 4th from the left is the data adding the optimization with profile information. As the profile data, the execution history of the dhrystone of a 10,000-times loop is used. The compile time including the execution time for acquiring this profile is very short at less than 5 seconds. The simulation speed becomes an average of 18.0 MIPS by using the profile optimization of the Intel C++ compiler. In this configuration, SimCore attains a 51% speed improvement compared with sim-fast.

The 5th data (on the right end) is the result of using a commercial PGI Compiler 5.1 (pgCC) with the -fast option. The simulation speed in this case is an average of 9.3 MIPS and is slower than the case using GCC.

From the evaluation results summarized in Figure 11, we confirm that when GCC is used, SimCore attains a 19% speed improvement compared with the simulation speed of sim-fast. When the profile information and the Intel C++ compiler is used, SimCore attains a 51% speed improvement.

3.2 Influence of the tuning methods

In this section, the influence of the tuning methods discussed in Section 2.7 is evaluated quantitatively.

The simulation speed of SimCore with versions of the main loop is summarized in Figure 12. The simulation speed shown here is the average of the eight benchmarks of SPEC CINT95.

The uppermost data in Figure 12 is the simulation speed with the main loop of the simple implementation shown in Figure 7. The simulation speed of this version of SimCore is 7.1 MIPS.

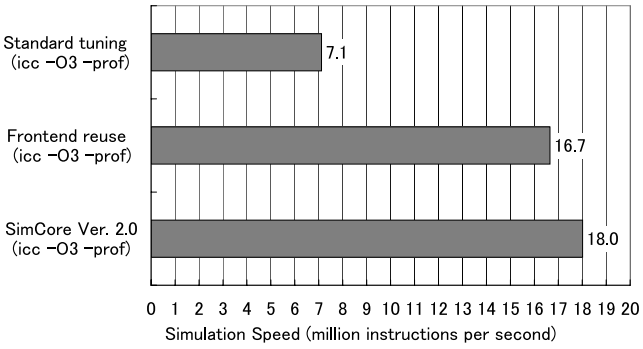


Figure 12. Influence of the tuning methods on SimCore.

The 2nd data is the simulation speed of SimCore with the main loop shown in Figure 8, in which pipeline frontend reuse is used. The simulation speed of this version is 16.7 MIPS. A speed improvement of more than twice is attained by using this technique.

The 3rd data is the simulation speed of SimCore Version 2.0 (Figure 9), in which pipeline frontend reuse and function call overhead elimination are used. The function call overhead elimination brings about a 7% improvement in speed compared to the 2nd data. The simulation speed of SimCore Version 2.0 reaches 18.0 MIPS.

As shown with these results, pipeline frontend reuse and function call overhead elimination attain the highest improvement in simulation speed, even though these are techniques which can be implemented compactly.

4 Related work

Various processor simulators are used as tools for processor architecture research or processor education. In addition, the demand for a faster processor simulator has been growing in recent years with the diversification of processors, including reconfigurable devices such as FPGAs. The focus of some research [10, 11] has been on speed improvement of various instruction set architectures. However, SimCore avoids complicating the source code in order to support many instruction sets. One of the features of SimCore is that it operates as a practical simulator with about 2,800 lines of code, which is a small amount.

There is much research on improving the speed of the processor simulator. In some research [12, 13], speed improvement is attained by lowering the accuracy of the simulation. However, the lower simulation accuracy makes verification difficult. Therefore, lessening the simulation accuracy is not allowed in the design policy of SimCore.

Concerning the speed improvement using reuse or memorization, a FastSim simulator [5] and scheduling calculation reuse have been proposed. In SimCore, pipeline frontend reuse is used as a technique which is realized with little changing of the code. Although this technique is not new, one of the features of SimCore is that techniques are selected in order to prioritize code readability and compact implementation.

SimpleScalar Tool Set [3] and SPIM [4] are well-known processor simulators used for purposes such as processor research and education. But, since SimpleScalar can be implemented in high-speed simulations, it is not a code that can easily be modified. Similarly, SPIM cannot be said to be readable. On the other hand, SimCore Version 2.0 satisfies the requirements of high readability and high-speed execution at the same time.

Historically, the development of SimCore for the C version began in March, 1999. Development of SimCore for the C++ version began in June, 1999. A processor simulator is an important tool, and it is advantageous to choose the most suitable tool, given many choices. As a tool for processor research and education, SimCore offers another choice.

5 Conclusions

We have developed a function-level processor simulator, SimCore/Alpha Functional Simulator Version 2.0 (SimCore Version 2.0), for processor architecture research and processor education. It satisfies the requirements for high readability and high-speed execution at the same time.

In this paper, we discussed the design and implementation of SimCore Version 2.0 in detail. The main features of SimCore Version 2.0 are as follows: (1) It offers many functions as a function-level simulator. (2) It is implemented compactly with 2,800 lines in C++. (3) It separates the function of the program loader. (4) Global variables are not used in order to improve the readability and function. (5) It offers a powerful verification mechanism. (6) It operates on many platforms. (7) Compared with sim-fast in the SimpleScalar Tool Set, SimCore Version 2.0 attains a 19% improvement in simulation speed.

For quantitative evaluations with SPEC CINT95 benchmarks, the simulation speed of SimCore Version 2.0 was measured and compared with sim-fast in the SimpleScalar Tool Set. We confirmed that when GCC is used, SimCore attains a 19% speed improvement compared with the simulation speed of sim-fast. And, when profile information and the Intel C++ compiler are used, SimCore attains a 51% speed improvement.

SimCore/Alpha Functional Simulator Version 2.0 is free software. The source code is downloadable from the following URL.

`http://www.yuba.is.uec.ac.jp/~kis/SimCore/`

References

[1] Shubhendu S. Mukherjee, Sarita V. Adve, Todd Austin, Joel Emer, and Peter S. Magnusson. Perfor-

mance simulation tools. *IEEE Computer*, 35(2):38–39, 2002.

[2] The microlib project. http://www.microlib.org/.

[3] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[4] David A. Patterson and John L. Hennessy. *Computer organization and design the hardware/software interface*. Morgan-Kaufmann Publishers, 1998.

[5] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 283–294, 1998.

[6] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):25–36, 1999.

[7] Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. simalpha version 1.0: simple and readable alpha processor simulator. *Lecture Note in Computer Science (LNCS)*, 2823:122–136, September 2003.

[8] Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. Implementation of simalpha-loader and construction of cross-development environment. Technical Report UEC-IS-2003-5, Graduate School of Information Systems, The University of Electro-Communications, 2003.

[9] John R. Levine. *Linkers and loaders*. Morgan-Kaufmann Publishers, 1999.

[10] Wai Sum Mong and Jianwen Zhu. A retargetable micro-architecture simulator. In *Proceedings of the 40th conference on Design automation*, pages 752–757, 2003.

[11] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 39th conference on Design automation*, pages 22–27, 2002.

[12] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, 1994.

[13] Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, 1996.

A Combined Virtual and Remotely Accessible Microprocessor Laboratory*

Helmut Bähring Jörg Keller Wolfram Schiffmann
FernUniversität Hagen
FB Informatik
58084 Hagen, Germany
{helmut.baehring|joerg.keller|wolfram.schiffmann}@fernuni-hagen.de

Abstract

We present a microprocessor lab that is accessible remotely, i.e. students can control the hardware in the lab from their computer at home. At the same time the lab also provides the features of a virtual lab, i.e. students conduct experiments on simulators. Both modes of the lab are run under a common user interface. This lab is suited especially for distance education, as students can develop and test their code offline, while still conducting their experiments on real hardware. This lab is thought to replace a traditional lab course where students had to be present on campus for one week fulltime, which is quite difficult to realize in distance education.

1 Introduction

FernUniversität is Germany's Distance Teaching University, serving also Austria and Switzerland and German speaking students in other parts of the world. The Computer Science Department offers study programmes leading to Bachelor, Master, and PhD degrees, respectively. Course texts are made available to the students in electronic form, preferably pdf, via a portal where students can access the courses they are enroled in. The course pages also provide study materials in addition to course texts, such as applets, link lists, multimedia enhancements, and the like. Additionally, paper copies of the course texts are sent via regular mail to the students. Assignments are submitted either in paper by mail or electronically, preferably via a system called WebAssign [9] that provides a web interface to the students and achieves automatic distribution of assignments to correctors and notification of submitting students about results and solutions when the assignments are corrected. WebAssign also allows to add modules for automatic grading of submissions, for simple cases such as multiple choice, but even for the submission of circuit schematics [5]. Students can ask questions or seek help over a number of communication channels: telephone, telefax, email, newsgroups, and (partly) video conferences over the Internet. Our method of teaching allows students to follow their study programme at any place and at any time. As more than 85% of our students work, this is a necessity in order to give these students a chance of success, as regular attendance of meetings at fixed times (even via Internet), is difficult or impossible for a majority of them.

Yet, the German computer science curriculum requires laboratory courses as part of the undergraduate study programme. For the reasons mentioned above, laboratory sessions on-campus are difficult to realize, even if they are compressed to a single one-week session close to the end of the semester, where that session is preceded by studies and preparations at home. This calls for virtual laboratories. While a virtual programming lab can be achieved with usage of the internet and tools for student collaboration, a virtual microprocessor lab proves more difficult.

First, while a number of simulator tools are available to create a virtual lab, they necessitate the installation of software on the student's computer. Yet, FernUniversität's students access the Internet frequently from their workplace, where they often do not have rights to install software. This would call for a lab with browser access, where all software is available as applets. However, students at home suffer from the amount of online time necessary in this case, and all students would suffer from the applet download times. Second, simulators never give a complete picture. Especially in applications where signals are to be checked or produced in a manner that includes a certain real-time behavior, a simulator is clearly not sufficient. While this can be cured by a remote lab, where a physical device is in the lab at FernUniversität and can be programmed, run and observed remotely, this again incurs long online times. Additionally, in order to serve a large number of students, the necessary amount of hardware installation would be enormous. We overcome these contradicting requirements and constraints by establishing a microprocessor lab that is both remotely accessible and virtual.

*This Research was partly funded by FernUniversität's Innovation Fund 2003.

Laboratory experiments are indispensable parts of most universities' engineering education programs. During the last decade different kinds of remote labs were developed. They can be divided into three categories: 1. electronic devices, 2. control engineering and robotics laboratories, and 3. digital logic and microprocessor systems. In the first category, the main focus is to measure the electrical characteristics of semiconductor devices. A good overview of current remote laboratories of this kind can be found in [4, 8]. In [1] remote laboratories for electrical and mechanical engineering are described. In [10] a control engineering laboratory is proposed where the students have to design, implement and test a discrete controller on a real plant that can be remotely accessed. Examples of remote laboratories for digital logic and microprocessor systems can be found in [6, 7]. In this paper we propose a new microprocessor lab that provides not only remote access but also an integrated simulation facility to prepare experiments while not being connected to the internet. To conduct the experiments remotely, the students can use the *same* graphical interface as used for the simulation. We are not aware of any other lab that tries to combine both modes under a single user interface. Also, our approach is in contrast to known online labs as it tries to reduce online time. Hence, we believe our approach to be novel.

The remainder of the paper is organized as follows. In Section 2, we detail the requirements and constraints of a microprocessor lab in distance education. In Section 3, we describe the implementation of our lab. In Section 4, we present some case studies of how our installed infrastructure will actually be used in lab courses. In Section 5, we give a conclusion and an outlook on further applications and activities.

2 Requirements and constraints for a virtual and remote microprocessor lab

If students want to get familiar with microprocessor programming and deployment, they must have access to a software development environment and a target system. Our computer engineering lab course[1] is divided into three phases. First of all, the students work through a course text that describes the basics for the experiments. For instance, the processors to be programmed are described, and the pitfalls and fallacies of assembler programming are explained. In the second phase they use development tools to practice machine language programming. The development system consists mainly of an editor and an assembler for compiling machine programs into the microprocessor's object code. These tools — cross-assemblers in particular — are widely and freely available, and are either installed locally at the students' computer or accessed remotely. In the third phase the students test their programs on a target system. For this purpose, we had to develop an appropriate target system that operates and looks like the real microprocessor system which is used in our laboratory on campus.

The target system should provide not only a simulator but also serve as a remote control for a real microprocessor system in Hagen, so that the execution of students' programs can be observed in real-time. In the simulator mode the students can use the target system as a *virtual* laboratory for testing the object code at home. Although they can test the functionality of their programs in this phase by simulation, they are not able to either check the real-time behavior, nor can they control external hardware that is connected to a real microprocessor system. In order to provide these facilities the students should be able to connect via the Internet to a real target microprocessor system located at the lab on the Hagen campus, i.e. they use a *remote* laboratory. By switching into the remote control mode the students will be able to remotely control an experimental setup in the lab, consisting of the processor itself and some additional hardware. By means of additional measurement devices that can also be controlled remotely the students can conduct experiments with external hardware (e.g. a traffic light panel). In addition to the software tools, web cams provide live pictures of how the setup in the lab behaves, e.g. which lights are on or off.

Students thus develop their programs locally and test them on a simulator. If they do not have the rights to install software on the computer they are using, they can also use a development system on a computer in Hagen via the Internet. When a student is confident that his program works, then he accesses the remote lab, uploads his program to the target system, and conducts his experiment on real hardware. If an error occurs or the real-time behavior differs from what is expected or required, he corrects the program until he is satisfied. In the end, he submits his results and his program via the WebAssign system to the correctors for grading. In this manner, the virtual lab mode saves valuable online time for many students, as they can develop at home in an offline mode. This also distinguishes our approach from online labs.

Despite the two modes of operation, students should not be required to learn two sets of user interfaces, as this would incur a lot of additional learning time. Hence, the virtual lab and the remote lab should share a common user interface. For example, a student should be aware whether he is running code on a simulator or on real hardware, but the controls for starting the program run should be the same.

[1] In the German curriculum, a basic computer engineering education is part of the computer science programme.

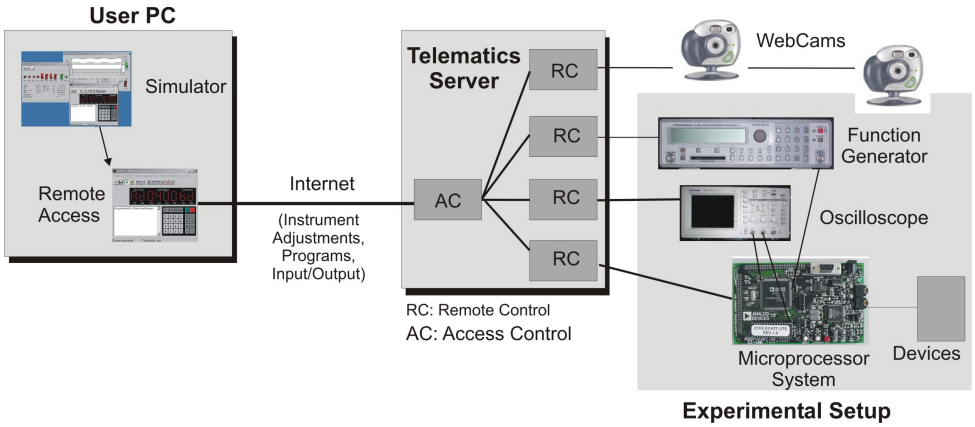


Figure 1: Structure of the microprocessor lab.

3 Implementation of the microprocessor lab

The remote laboratory is realized by a client-server approach. Students who want to use the remote lab have to download a number of client programs that serve as remote controls for the microprocessor system and the measurement equipment, such as an oscilloscope or a web cam. These clients connect to server processes that run on a server computer (named telematics server) in the laboratory at Hagen. Some of those clients also have simulator functionality (e.g. the microprocessor client), so students can seamlessly switch between the virtual lab and the remote lab. The server computer in turn is connected to the hardware devices in the lab: microprocessor card, oscilloscope, web cam, and so on. The microprocessor card itself is connected to some additional hardware such as a traffic light kit. The server also provides a central installation of the development software and the simulator for students who are not allowed to install software on their local computer. Those students may access the server either via a remote shell or session, or web-based via VNC [11]. Figure 1 outlines the scenario of our microprocessor lab.

To accomplish more complex experiments with attached devices, such as a radio controlled clock (see next section), the simulator establishes a TCP/IP-connection via the internet to the server at our laboratory, thus allowing students to switch seamlessly to the remote lab. After authentication, the user gets access to the remote control software of different components of the experimental setup. These components include one or more oscilloscopes, function generators and mainly the microprocessor system hardware. Attached to this system will be different devices, e.g. the above mentioned radio controlled clock or a model of a traffic light.

The user is able to upload his programs to the remote microprocessor system via the user interface and to start them there. He can adjust the already mentioned instruments remotely, perform different measurements, and gets back the results. To give him the possibility of observing the experiments, and thus the feeling of being in the laboratory room, all instruments and devices — but also the whole laboratory — are accessible by controllable web cams.

The client programs are preferably programmed in Java. The software is programmed in two parts: one part that is independent of the particular device for which the client is the frontend, and one part which realizes the particular device's frontend. The former part can be re-used for all devices, the latter part is programmed in a way that realization of an additional device can be done with the least possible effort. Using Java also had the advantage of being more independent from the type of computer the students have locally, than with any other realization. Figure 2 depicts the microprocessor user interface in a close-up.

Of course, the implementation of the lab is not restricted to the particular microprocessor card. There are also cards with a microcontroller and a digital signal processor, respectively, that can be accessed in the same manner. Also several other experiments can be realized, see Section 5.

4 Case studies

In this section we want to demonstrate how the virtual and remote lab can be used by the students. When the students have worked through the course text, which describes the basics for the experiments, they can conduct the experiments. In all cases they use the microcomputer simulator depicted in Figure 2. Note, that the graphical interface of this simulator can also used as a remote control during the remote laboratory experiment. By means of a pull down menu the user can

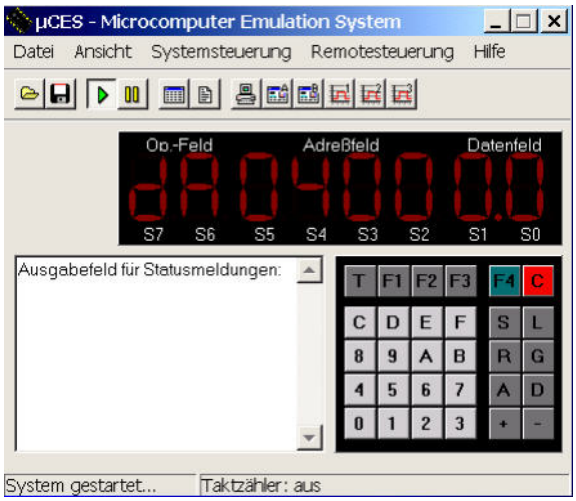


Figure 2: Microcomputer simulator with integrated remote control.

switch between the two modes. For the remote mode the IP of the remote lab server must be entered.

4.1 Decimal counter

In this experiment the students have to write a assembler program for a decimal counter that is incremented every second. At first, the experiment should be conducted by means of the microcomputer simulator. The students enter, edit and assemble the program on their home PC and load it into the simulator. Then the object code is launched and a four digit starting value is entered via the keyboard. The counter will be started by pressing the character '+' and the current counter value will be displayed on the LED display. If the experiment has been successfully conducted in the virtual laboratory, the students can switch to the remote laboratory by using the corresponding pull down menu. They enter the server IP and will get connected to a real microcomputer system located in the computer engineering laboratory at Hagen (see Figure 3). Now, the program runs on the real microcomputer while the display can be observed by means of a web cam.

4.2 Radio controlled clock signal

The objective in this experiment consists of writing a program that should emulate the signal of a radio controlled clock signal. While the development cycle is identical to that for the decimal counter experiment, we need an additional measurement device here to watch the generated signal. For this purpose we implemented a client-server program suite that allows to remotely control an oscilloscope at Hagen and to display the measured curves on the student's PC (see Figure 4).

5 Conclusions and outlook

We have realized a microprocessor lab course that serves the requirements of students in distance education: minimization of online time, avoidance of downloads and local installations if possible, and access to real hardware. At the same time, we fulfilled our curricular requirements: performing experiments complex enough to teach the students low-level programming of microprocessor hardware. This is accomplished by two modes of operation: virtual lab and remote lab. The switching between these modes is seamless, relieving the students from having to learn several user interfaces for the same device. Our own software development has been reduced to a minimum by re-use wherever possible. The achievements so far have resulted in a shortening of the on-campus phase from one week to two days for the next offering the lab course in spring 2005.

There are further experiments, for which virtual and remote versions have been developed, and which will complete the virtualization of the microprocessor lab in the future. These are: digital logic, where a GAL can be programmed remotely, a digital signal processor, which is used to implement a filter for a signal which is generated by a function generator, and a microcontroller which is programmed for a control task. The function generator can be controlled remotely in the same manner as the oscilloscope. The digital signal processor and the microcontroller can, in principle, be accessed in the same way as the microprocessor. However, the simulator for the digital signal processor was not freely available in Java, but provided with the development software for a particular operating system. Also, the user interface for the card itself is separate from the development software and not freely available. Hence, we did not find a way so far to provide students with a seamless switch between virtual and remote modes for this experiment. Furthermore, the control of the hardware necessitates a remote session or use of VNC.

The (partial) virtualization of the microprocessor lab continues our transformation to a web-based computer engineering education, as set out in [2], which started by integrating a circuit design tool, submission of assignments via the Internet, and semi-automatic assessment and grading of assignments with student circuits in a beginners course [5]. The integration of the simple scalar tool set [3] into a course on advanced computer architecture is still underway.

We still have to accomplish a kind of reservation scheme for the physical devices. Currently, the remote lab is allocated on a first-come-first-serve basis, which is not the optimal choice at times close to deadlines for assignments. We also plan to extend this type of lab access to other computer engineering lab courses.

Figure 3: Real microcomputer at Hagen university.

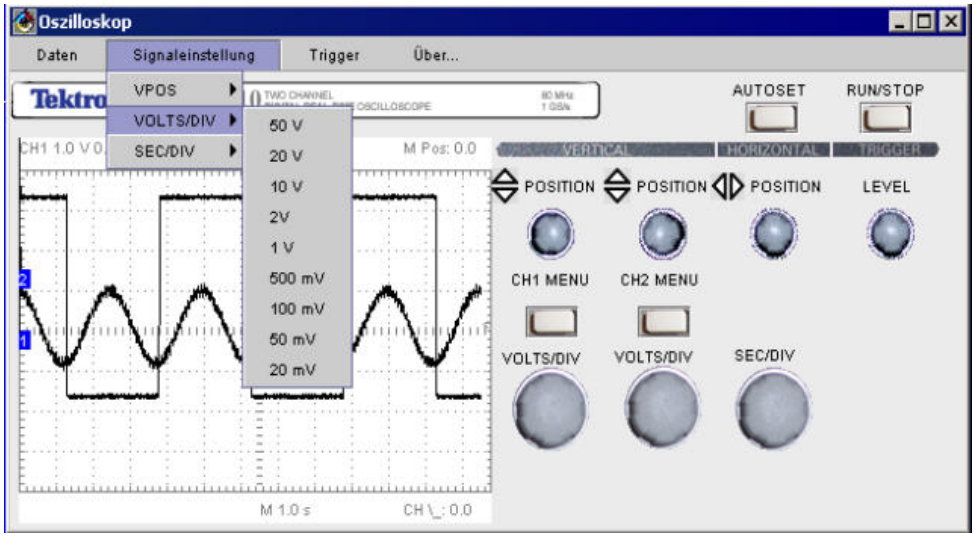


Figure 4: Remote control for the oscilloscope.

Acknowledgements

The research presented here was partially funded by FernUniversität's Innovation Fund 2003. We thank Bernhard Fechner for implementing part of the web interfaces for the lab devices.

References

[1] B. Alhalabi, D. Marcovitz, K. Hamza, S. Hsu. Remote Labs: An innovative leap in engineering distance education. IFAC, 2000.

[2] H. Bähring, J. Keller, W. Schiffmann. Deployment of New Media at FernUniversität Hagen. In Informationstechnik und Technische Informatik, vol. 43 no. 4, 2001, pp. 215-218.

[3] D. Burger, T. M. Austin. The simple scalar tool set. Tech. Rep. TR-1342. Computer Science Dept., Univ. of Wisconsin, Madison, 1997.

[4] T. A. Fjeldly, M. S. Shur. Lab on the Web: Running Real Electronics Experiments via the Internet, Wiley-VCH, 2003.

[5] U. Hönig, J. Keller, W. Schiffmann. Web-Based Exercises in Computer Engineering. In Proc. International Conference on Networked e-learning for European Universities, Granada, Nov. 2003.

[6] S.-J. Hsieh, P. Y. Hsieh, D. Zhang. Web-based simulations and intelligent tutoring system for programmable logic controller. ASSE/IEEE Frontiers in Education Conference, Session T3E, IEEE 2003.

[7] Y. Ko, T. M. Duman, A. Spanias. On-line laboratory for communication systems using J-DSP. ASSE/IEEE Frontiers in Education Conference, Session T3E, IEEE 2003.

[8] Z. Nedic, J. Machotka, A. Nafalski. Remote laboratories versus virtual and real laboratories. ASSE/IEEE Frontiers in Education Conference, Session T3E, IEEE 2003.

[9] H. W. Six, G. Ströhlein, J. Voss. Evaluation of WebAssign. In 20th World Conference on Open Learning and Distance Education (ICDE Congress). Düsseldorf, April 2001.

[10] J. Tuttas, B. Wagner. Distributed online laboratories. Intern. Conference on Engineering Education, 2001.

[11] RealVNC. `http://www.realvnc.com/`.

Open Forum on Textbook Pricing

Students have *always* complained about the price of textbooks. What's new about that? Why should you care?

With students actually staging protests on major campuses and both state legislatures and congress looking into the issue as parents take up the claims of their college-age children, it may be useful to take a closer look at the players, their interests, and at the true costs of publishing a first-rate textbook.

Denise Penrose, Senior Editor for Morgan Kaufmann Publishers invites you to spend some time with her discussing these issues. Here are some questions we will consider:

- Are students right that publishers are just greedy corporations taking more than their "fair share" of profits?
- How much does it really cost to develop a solid teaching text?
- Who wins and who loses in a used book exchange.
- What are “international student editions,” why are they priced less than “domestic editions,” and what's fair about that?
- How might you be unwittingly increasing the cost of books, increasing the price for your students?
- Why is it important for you to know the price of the books you select for your students? What are publishers doing to address these issues today?

Denise Penrose has more than 25 years of professional publishing experience including developing and acquiring university texts for computer science and engineering curricula, text references for computing professionals, and computer trade books for users; managing a production staff for a monthly high technology magazine; writing and editing stories for high-technology computer trade books for users; managing a production staff for a monthly high technology magazine; writing and editing stories for high-technology magazines; and developing an academic journal. She joined Morgan Kaufmann in 1996 in time to develop and publish the second edition of *Computer Organization and Design: The Hardware/Software Interface*. She publishes the computer architecture and the artificial intelligence programs for Morgan Kaufmann. She graduated cum laude from Smith College with a Bachelor or Arts degree in Comparative Literature.