Workshop on Computer Architecture Education

Sunday, June 5, 2005

Embedded Computer Architectures in the MPSoC Age

Wayne Wolf
Dept. of Electrical Engineering
Princeton University
wolf@princeton.edu

1. Introduction

Embedded computers are no longer used as simple controllers. Instead, high-performance embedded processors perform complex algorithms and are linked together to form multiprocessors. Embedded computing provides students different take on computer system design because of the requirements imposed on these systems:

- Embedded computing systems generally require real-time performance. Real-time and average-time performance are very different animals.
- Battery-powered embedded systems must meet very stringent energy requirements [Aus04].
- Although the software in embedded systems can be changed to optimize the overall system, the software must also meet the specifications of the application.

As such, an architecturally-oriented embedded systems class emphasizes somewhat different concepts than a traditional, general-purpose computer architecture class. An embedded computing architecture class must use a methodology to help students quickly get their hands around an unfamiliar application. They must explore a broad range of architectures. They should also explore trade-offs between architectural modifications and software modifications to meet system goals.

Distributed embedded systems, which are built from networks of embedded processors, are also widely deployed. This paper will concentrate, however, on systems-on-chips.

1. Multiprocessor Systems-on-Chips

Multiprocessor systems-on-chips (MPSoCs) [Jer04] are, first of all, systems-on-chips. They implement complete applications on a single chip. (Although as Rich Page points out, most systems-on-chips are marketing single-chip solutions---they use one chip plus all the other chips that you need to make the SoC work.) MPSoCs are systems-on-chips that include one or more programmable processors.

Systems-on-chips are generally adapted to the application to meet performance, power, and cost goals. Although modern VLSI fabrication technology provides us with very large chips, applications keep getting larger. Some markets are large enough that specialized architectures are inevitable and desirable.

Multiprocessor systems-on-chips try to balance specialization and programmability. Programmable processors allow the SoC to be programmed after fabrication; MPSoCs are often referred to as **platforms** because they allow for many implementations of a given type of system. Programmability offers many advantages: the same chip can be used in several products, reducing product cost; design tasks can be compartmentalized; and the platform chip may have a longer shelf life than a highly specialized SoC.

Because these are systems-on-chips, they generally aren't traditional symmetric multiprocessors. They may use hardwired function units in addition to programmable processors. They may use several different instruction sets. They may have non-uniform memory spaces supported by asymmetric networks.

Many multiprocessor systems-on-chips are now available for several types of applications:

- Mobile multimedia requires both high performance and low energy consumption. The ST Nomadik and TI OMAP architectures are MPSoCs that provide specialized architectures for audio, video, and communications.
- Home multimedia is not as tightly constrained on power as mobile multimedia but requires very high performance for applications like HDTV. The Philips Nexperia architecture is a well-known MPSoC for set-top box applications.
- Networking requires very high performance and provides some opportunities for specialized parallelism. Network processors from Intel, Cisco, and others use heterogeneous architectures to process packets at high rates.

3. Architectural Challenges

Embedded computing and MPSoCs make for a full employment act for computer architects. We are in no danger of running out of applications that can make use of large amounts of computing power and that can support the design effort required to create an efficient application-specific platform. Several specific challenges flow out of our continuing need to design MPSoCs.

Configurable processors, such as those provided by Tensilica, allow the SoC designer a convenient way of quickly building processors with customized instruction sets. One area in which designers need help is figuring out which instruction set extensions should actually be implemented. Another important goal is figuring out how to connecting configurable processors into multiprocessor networks.

Hardware/software co-design [DeM01] is another way to increase system performance for a particular application. Accelerators, when properly designed, can significantly and efficiently increase performance. However, the application must be carefully analyzed to be sure that an accelerator actually improves overall performance.

Heterogeneous multiprocessors for embedded applications generally implement pipelines of processes. Our own smart camera system [Oze05] is an example of a pipelineable application. The smart camera processes video in real time, using a number of distinct steps. The amount of work performed by these stages is generally data dependent and buffers are required to smooth out rates. As video data is processed, it is boiled down in size so that data rates at the end of the process are trivial compared to the input video data rates. Pipelined application architectures bring up both hardware and software questions about buffer management and rate control.

Networks for embedded systems are another important challenge. Several networks have been proposed for on-chip use. Many of these are general-purpose networks designed to be used in many different systems. However, our own experiments indicate that asymmetric networks offer significant advantages.

Balancing generality with efficiency is a key goal in MPSoC architectures. As we pointed out elsewhere [Wol05] even relatively simple consumer devices must now implement a wide range of functions. Consider what must be performed by simple devices like digital music players or digital cameras in addition to their core functions:

- User interface.
- Cryptography.
- Networking, either through Internet or specialized protocols.
- Digital rights management.
- File systems that are compatible with PC file systems.

This wide range of functions arguably calls for a general-purpose processor; on the other hand, some of these functions may call for application-specific hardware to meet performance/power goals. We do not yet fully understand the architectural implications of the networked consumer device.

Overall, methodology is an important aspect of embedded system design that does not often come into play in general-purpose systems [Wol00]. Because embedded system designers need to design many systems and do so in a predictable amount of time with a predictable number of people, they need to develop methodologies that allow them to repeatably make reasonable decisions in new design domains. Giving students an insight into the design process can be as important as showing them specific design outcomes.

4. Benchmarks

Benchmarks are at least important in embedded computing as they are in general-purpose computing. When you are designing an application-specific system, the wrong choice of a benchmark program or input data for that program can lead to fatal misjudgments.

I believe that larger programs make more useful design examples for embedded computing for several reasons. First, high-performance embedded systems typically run several different types of algorithms; it takes a certain amount of code to exhibit all that complexity. Second, larger programs do a better job of exercising multi-tasking. Third, they give students a more realistic taste of the nature of embedded software and performance analysis.

However, it is hard to get good benchmarks and data sets. Although several reference implementations of various standards are available, they can be very hard to use. Reference implementations may make inappropriate use of dynamic memory; they may also use inefficient algorithms for critical modules. For example, many reference video encoders come with full-search motion estimation, even though that algorithm is not used in practice. Measurements made on unrealistic algorithms will lead to bad design decisions.

5. Labs

Laboratories are a critical part of an embedded systems course. As embedded systems become more complex, it becomes harder to create an enriching set of labs for students.

Most instructors worry about the cost of lab equipment, particularly if they want to reach a broad audience. Although many microprocessor manufacturers and third parties sell evaluation boards, the associated development system is a hidden cost of these boards. Some vendors provide software along with the board while others charge a good deal of money for development systems. Ideally, students should be able to install on their own machines student versions of the development systems they use in labs; in the FPGA world, Xilinx is an excellent model for how to make devices and tools accessible to students.

Instructors can select from among a large number of uniprocessors, but it is hard to find a good experimental setup for multiprocessors. The TI OMAP processor is one of the very few embedded multiprocessors for which there exists an even moderately-priced development board, but that board is still expensive and the software environment is complex.

Much development work must be done on simulators, both in the real world and in class. Uniprocessor performance and power simulators are widely available. Although several open-source multiprocessor simulators are available, most of them are designed for symmetric multiprocessors and cannot be easily modified to handle heterogeneous multiprocessors. The MESH simulator from CMU was developed to handle heterogeneous multiprocessors as seen in systems-on-chips.

6. Conclusions

We live in an exciting time in which we have the opportunity to develop a new generation of courses on high-performance embedded computing. But because these are complex systems, instructors have to be prepared to invest time to set up lectures and labs that mate their students' interests with the applications that drive system-on-chip and large-scale distributed embedded systems. Although each institution has its own special requirements, particularly for labs, group effort may help us all build this new generation of courses.

6. References

[Aus04] Todd Austin, David Blaauw, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Wayne Wolf, "Mobile Supercomputers," *IEEE Computer*, 37(5), May 2004, pp. 81-83.

[DeM01] Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, eds., *Readings in Hardware/Software Co-Design*, Morgan Kaufman, 2001.

[Jer05] Ahmed A. Jerraya and Wayne Wolf, "Hardware/software interface codesign for embedded systems," *IEEE Computer,* 38(2), February 2005, pp. 63-69.

[Oze05] I. Burak Ozer, Tiehan Lu, and Wayne Wolf, "Design of a real-time gesture recognition system," *IEEE Signal Processing Magazine*, 22(3), May 2005, pp. 57-64.

[Wol00] Wayne Wolf, *Computers as Components: Principles of Embedded Computing System Design*, Morgan Kaufman, 2000.

[Wol05] Wayne Wolf, "Multimedia applications of systems-on-chips," in *Proceedings, DATE '05 Designers' Forum*, ACM Press, 2005, pp. 86-89.

.

Special Session on Embedded Systems Education

Introduction

As computational components continue to decrease in size and increase in performance, they are being embedded into devices in new and innovative ways leading to a proliferation of embedded systems in our society heretofore never witnessed. These devices introduce design and engineering challenges not always seen in general-purpose computing platforms, which are often the focus of modern computer engineering curricula. For example, embedded applications often include real-time behavior, multiprocessing, complex computations, reactive input/output, and require long-term deployment using only remote power sources. In addition, strict design constraints such as memory, power, size, cost, and time-to-market limitations, again not encountered in the design of most general-purpose systems, are the norm when designing embedded devices.

To produce graduates capable of addressing the specific issues applicable to embedded systems, it is necessary to incorporate these concepts into the computer engineering undergraduate curriculum. However, it is difficult to introduce such a broad range of topics crossing many application domains into general undergraduate education. In many cases, students are presented with discrete concepts in many different classes that are applicable to embedded systems but are never presented a system-level view of the field. This typically gives students the puzzle pieces but not the ability to connect the pieces to produce the full picture. In other cases, embedded systems education has been relegated to "teach-the-tool" and "teach-the-technology" approaches, where students learn one particular processor, development environment, or software tool.

In order to advance the field of embedded computing and prepare future graduates for success as embedded systems engineers, a more systematic approach to embedded systems education is necessary. This approach must provide students the fundamental concepts required of the field while also providing the more general understanding of the system-level concepts.

Toward this end, the Special Session on Embedded Systems Education held in conjunction with the Workshop on Computer Architecture Education (WCAE) will provide a venue for researchers and educators to exchange ideas related to embedded systems and embedded systems education. The intent is that attendees and organizers will gain insightful information through paper presentations, an informal panel discussion, and interactions with others involved in the workshop.

Kenneth Ricks
Organizer, Special Session on Embedded Systems Education, WCAE 2005

Embedded Systems Courses at RIT

Roy S. Czernikowski
Department of Computer Engineering
Rochester Institute of Technology
rsceec@rit.edu

James R Vallino
Department of Software Engineering
Rochester Institute of Technology
J.Vallino@se.rit.edu

Abstract

A three-course sequence of cross-disciplinary real-time and embedded systems courses has been introduced at RIT[•]*. We are teaching these courses in a studio-lab environment teaming computer engineering and software engineering students. The courses introduce students to programming both microcontrollers and more sophisticated targets, use of a commercial real-time operating system and development environment, modeling and performance engineering of these systems, and their interactions with physical systems.*

1. Introduction

Embedded computers are now ubiquitous, often in common products where they are invisible to the user. These embedded processors provide special purpose functionality not found in general-purpose applications familiar to desktop computer users. The standard computing curricula concentrate primarily on general-purpose desktop applications and do not provide students with the opportunity to gain the necessary skills for engineering software in real-time and embedded systems.

2. Real-time and embedded systems at RIT

In Rochester Institute of Technology's computer engineering program, senior projects often focus on real-time and embedded systems, but there was no formal instruction in the engineering of these systems. The software engineering program had an embedded systems application domain comprising three courses: two standard operating systems courses offered by computer science and a concurrent programming course from computer engineering. None of these courses directly addresses issues in developing real-time or embedded software; they were chosen because they were the closest courses relevant to the domain. We decided that the best way to address these shortcomings in the real-time and embedded domain in both the computer engineering and software engineering curricula was to adopt a cross-disciplinary approach. The presence of students from both programs created a unique opportunity for synergy at RIT. The computer engineering students possess knowledge of electronics and control systems along with software development skills at the lower-levels. The software engineering students possess significant knowledge of how to engineer complex software systems including the design and modeling of those systems. Developing software for real-time and embedded systems is where the skills of these two groups intersect.

In July, 2003, we started work on the laboratory and the development of a three-course sequence. Each of these upper-division courses is four academic quarter credit hours and meets for ten weeks of classes having a pair of two-hour studio sessions per week. In the studio-lab environment each class session mixes lecture material with hands-on exercises and projects in a flexible format. These courses are cross-listed in the software engineering and computer engineering programs. Registration is initially controlled with the goal of having an even mix between students from the two programs. To the extent possible we ensure that all project teams have a member from both computer engineering and software engineering. The students will bring together expertise from two domains and apply a common engineering approach for solving real-time and embedded system development problems. To this point, we have offered the first two courses in the sequence several times. The third course is currently being offered for the first time in the spring 2005 academic quarter. The remainder of this paper describes our laboratory facilities, the syllabus for the three courses we developed and some initial results of the internal and external evaluation of the program.

[•] Sections of this paper will also be presented at the Frontiers in Education 2005 Conference in October 2005.

Our funding came from the award of a National Science Foundation Course, Curriculum and Laboratory Improvement Adaptation and Implementation grant. We identified the School of Computing and Software Engineering at Southern Polytechnic State University and the Department of Computer Science and Engineering at Arizona State University as the collaborating institutions that would provide course materials for adaptation into the courses we developed.

3. Laboratory hardware facilities

The studio lab developed for these courses consists of twelve student stations and an instructor's station. The instructor's station is configured with classroom control software that enables the capture, control and display of any of the student stations on the classroom video projector. Each student station is positioned to allow a pair of students to work together. Each station has a modern personal computer for software development and a 486-based single board computer as a target system. We are using a Diamond Systems [1] pc-104 board with timers, A/D converters, D/A converters, and digital I/O capability for the target systems. See Figure 1.

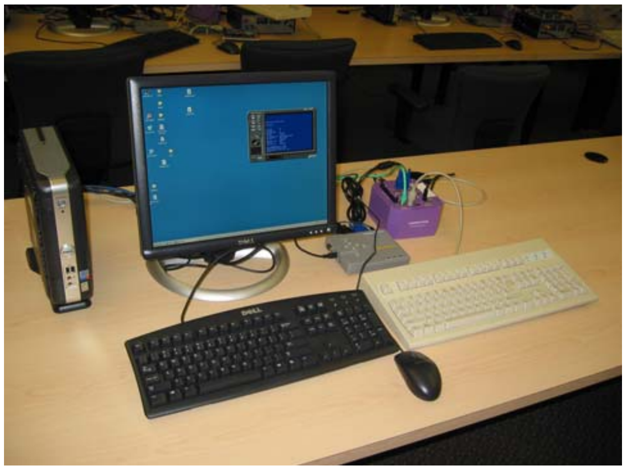


Figure 1 – PC Development environment and Diamond Systems pc-104 board target system showing picture-in-picture target system console.

To reduce the clutter in the student's work area we eliminated the second monitor often attached to the target system. Students can view the output from the target system in a number of ways. For text-based standard output, the target system development software provides a redirected console on the development system. We also have the VGA output converted to S-video and then fed into a USB S-video digitizer. The digitizer's software provides a picture-in-picture display shown in Figure 1. Finally, for projects that are generating VGA graphics output the student can view the full resolution video through the second input channel on the development station's dual-input monitor.

For the experiments involving programming a microcontroller, each station is also provided with a Motorola 68HC12 board, a custom designed interface board on which is mounted the microcontroller board, a custom binary LED-switch board for elementary binary input and output, a signal generator and a power supply.

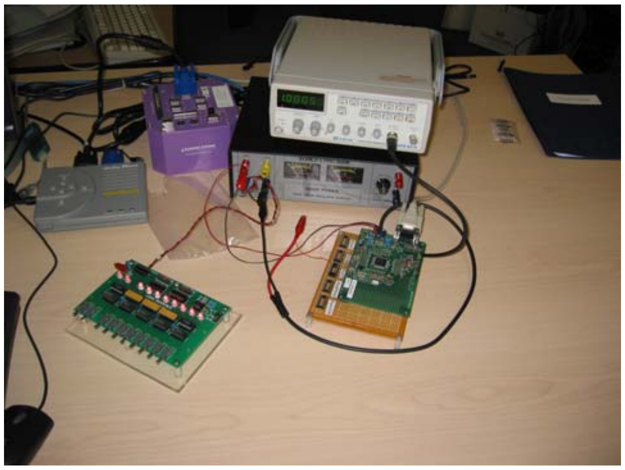


Figure 2 – M68HC12 Microcontroller, interface board, LED-Switch Board, Signal Generator and Power Supply.

The last pieces of hardware to mention are primarily used in the third course in the sequence. This course covers performance engineering of real-time and embedded systems. To motivate the need for system tuning of real-time systems we use the control of physical systems. The two systems we choose for the laboratory are from Quanser Systems [8]. We selected their inverted pendulum and ball and balance beam systems shown in Figures 3 and 4 respectively. In the third course the students also experiment with hardware/software co-design on a Digilent Spartan 3 FPGA board [2] shown in Figure 5. There is one FPGA system at each student station.

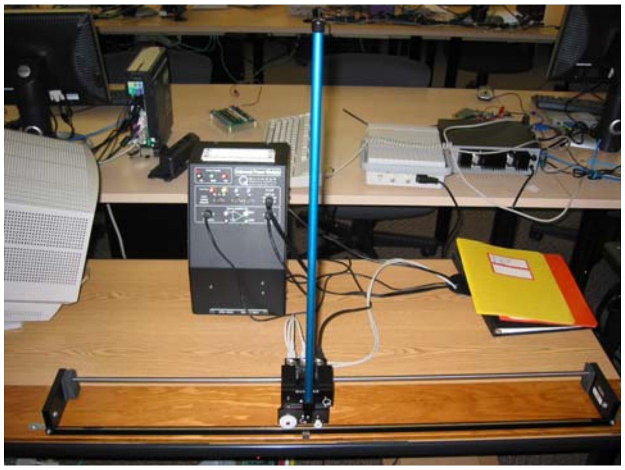


Figure 3 – Quanser System Inverted Pendulum

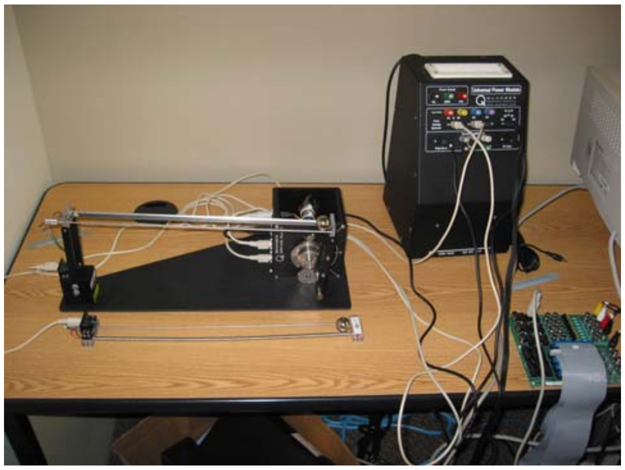


Figure 4 – Quanser System Ball and Balance Beam.

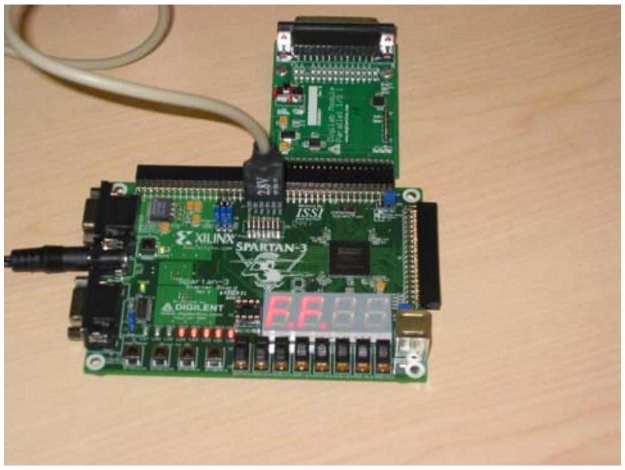


Figure 5 – Digilent Spartan 3 FPGA Board

4. Laboratory software facilities

There is a set of software tools to complement the hardware in the laboratory. The development stations are running the Windows XP Professional operating system. The MGTEK MiniIDE [7] supports assembly language programming on the 68HC12 microcontroller. We received a software grant from Wind River Systems [11] allowing the use of VxWorks and the Tornado integrated development environment. This is the commercial real-time operating system that the students utilize in the laboratory. Matlab and Simulink from The MathWorks [6] are used for simulating and controlling the Quanser experiments. We received software grants from IBM [4] for the Rational Rose development suite and Rational Rose Real-Time as UML modeling tools. Finally, the students work with Rhapsody from I-Logix [5] as a UML modeling tool. Rhapsody's statechart modeling and code generation features are used heavily in the second course in the sequence.

5. Course concepts

We designed a sequence of three courses that provides the student with broad exposure to the real-time and embedded systems domain. The first course, Real-Time and Embedded Systems, provides a general introduction to the area. We expect that this course will have the largest appeal across both disciplines with some aspects particularly attractive to both the computer engineering and software engineering students. The second course, Modeling of Real-Time Systems, has a stronger software engineering flavor. It covers UML modeling of real-time and embedded systems. The third course, titled Performance Engineering of Real-Time and Embedded Systems, deals with measurement of system performance, implementation of time-critical software and the fluid hardware/software boundary. The next sections describe these three courses in detail.

6. Real-time and Embedded Systems course

The first course in this elective sequence is titled Real-Time and Embedded Systems. It presents a general road map of real-time and embedded systems. It introduces a representative family of microcontrollers that exemplify unique positive features as well as limitations of microcontrollers in embedded and real-time systems. These microcontrollers are used as external, independent performance monitors of more complex real-time systems targeted on more robust platforms. The majority of this course presents material on a commercial real-time operating system and using it for programming projects on development systems and embedded target systems. Some fundamental material on real-time operating systems is also presented. This course was first offered at RIT in the spring of 2003. It has since been offered three more times. The textbook for the course is *Real-Time Systems and Software* by Shaw [9].

The topics covered by the class provide an introduction to the area. Class discussion focuses primarily on the fundamentals of real-time systems. The project work spans the range from microcontroller assembly programming through to application development under a commercial real-time operating system.

The topics covered by the Embedded and Real-Time Systems course include:

- Introduction to Real-Time and Embedded Systems
- Microcontrollers
- Software Architectures for Real-Time Operating Systems
- Requirements and Design Specifications
- Decision Tables and Finite State Machines
- Scheduling in Real-Time Systems
- Programming for a commercial real-time operating system

- Development for Embedded Target Systems
- Design Patterns for Real-Time Systems
- Language Support for Real-Time
- Real-Time and Embedded Systems Taxonomy
- Safety Critical Systems

There are several programming project assignments given to the students. A pair of students works on each assignment. As was mentioned previously, to the extent that the registration numbers permit, a software engineering and computer engineering student are paired together. This course has a mix of projects that allows the computer engineering student to provide the lead on some and the software engineering student to lead the others. The project assignments for this course are:

Microcontroller programming: students program the 68HC12 microcontroller to act as an interval timer and as an independent system performance measurement device. The microcontrollers used assembly language programs to measure and tabulate the inter-arrival times, the "jitter", of a series of 1000 pulses for several experiments described later. The microcontroller's timers have no difficulty measuring the arrival times or interarrival times of the pulses to 1.0 microsecond resolution.

Real-Time Operating System multi-tasking primitives: the main goal for this project is to have the students become familiar with programming under a commercial real-time operating system. Using VxWorks as an example of a commercial real-time operating system, students learn how to program using its concurrency and synchronization primitives. The team must implement a concurrent system such as a transit simulation or an automated factory. The programming is done within a simulated target system running on the development station.

Real-Time Operating System performance measurements: there are two smaller projects that fall into this category. These programs run on the target systems. Both projects make use of the microcontroller project as a timing device. In the first project the students learn how to schedule a periodic task under VxWorks. This task is toggling a bit on the printer port. The microcontroller timer measures the inter-arrival time and jitter of these software-generated periodic pulses. The second project measures the interrupt response time of the target system by having the microcontroller measure the time between generating an interrupt signal to the target and receiving its response. These two projects are run on the target systems, and the microcontroller collects 1000 samples with 1.0 microsecond resolution and displays the results.

Final project: there is a final programming project. This project is usually of student motivated with each team thinking of a project. We have seen implementations of user-level drivers for the devices on the target system, an ultrasound distance measurement, simple video games, and a digital oscilloscope.

Students are presented with two different embedded processors and development environments and are confronted with the strengths and weaknesses of each platform/architecture and environment.

Using Bloom's Taxonomy as a guide, the learning outcomes for this course are given in Table 1.

Table 1
Learning Outcomes for Real-Time and Embedded Systems Course

Knowledge	
	• List the scheduling algorithms commonly used in real-time systems. • Describe the steps required to build, install and run a software system on an embedded processor.
Comprehension	
	• Discuss the event sequence for responding to an interrupt.
Application	
	• Apply software engineering practices to the development of several small real-time systems. • Demonstrate the use of a micro-controller as an event timer. • Design and implement measurement tools to collect system performance data. • Design and implement a concurrent system on a real-time operating system.
Analysis	
	• Measure the performance of a real-time operating system.
Synthesis	
	• Design and implement a small-scale real-time application on a real-time operating system.

7. Modeling of Real-Time Systems course

The second course is titled Modeling of Real-Time Systems. The course takes an engineering approach to the design of these systems by analyzing a model of the system before beginning implementation. The course discusses primarily UML based methodologies. Implementations of real-time systems are developed manually from the models and using automated tools to generate the code. At this point, this course has run twice. *Doing Hard Time* by Douglass [3] is the textbook for the course.

Topics covered by the Modeling of Real-Time Systems course include:

- Introduction to Modeling of Real-Time Systems
- Basic Concepts of Real-Time Systems

- Basic Concepts of Safety-Critical Systems
- Use case analysis for real-time systems
- Structural object analysis for real-time systems
- Behavioral Analysis using statecharts
- Design patterns for real-time and safety-critical systems
- Threading and Schedulability
- Real-Time Frameworks

This course has the strongest software engineering emphasis. The projects progress through phases in the standard waterfall process model with emphasis on analysis and design of the software system. For the software engineering students this is continued practice in the UML modeling that they do in all the courses in their program. The application areas chosen for the projects, i.e. embedded systems, are significantly different from the typical desktop and GUI-over-database projects that they see in their other courses. In this course the software engineering students take the lead on most projects. Many computer engineering students have not done any UML modeling since their second-year software engineering course. The project assignments for this course are:

Requirements and Architectural Design: this assignment starts with the user manual for a consumer electronic device. It requires the students to identify the actors in the system and do a use case analysis. This is then followed by an architectural design and high-level class structural design. A home blood pressure monitor and a digital video recorder are two devices that students have modeled for this project.

Design and Implementation: this assignment starts with a clear statement of requirements and requires the team to do a class-level design and implementation. We have used both end-user applications, (such as a four-function calculator), and a simulation (of a controller for a chilled water air conditioning system). The implementation language is Java with the team implementing a graphical user interface to control the program.

Code Generation: through this course we place an emphasis on statecharts as a mechanism for behavior modeling of real-time and embedded systems. In this project the students explore the code generation features of the modeling tool they use. The teams create a statechart-based definition of the behavior and automatically generate C++ code for the application. Typically, the team will be able to create a fully-functioning application entirely from within the statechart model. This is not to say that the team writes no C++ code. Some adornments to states are code snippets that get built into the code that the tool auto-generates. For this project we have used a four-function calculator and garage door opener controller.

Final Project: this project is a modeling exercise done as a take-home final exam. Each student does a thorough identification of actors, a use case analysis, class structural design and system dynamic modeling using sequence diagrams and statecharts. There is no implementation of the systems which to date have been a power window controller for a car and a reverse vending machine that accepts containers for recycling at a local supermarket.

Using Bloom's Taxonomy the learning outcomes for this course are given in Table 2.

Table 2
Learning Outcomes for Modeling Real-Time Systems Course

Knowledge	
	• Specify the characteristics of real-time and safety critical systems.
Comprehension	
	• Discuss the software process for the development of real-time systems and contrast it with development for a standard application. • Identify architectural and design patterns for real-time and safety critical systems.
Application	
	• Apply architectural and design patterns in the analysis and design of real-time systems.
Analysis	
	• Model the dynamic behavior of a real-time system using statecharts. • Describe the requirements for simple real-time systems using use cases. • Model the structure of a real-time system using UML class diagrams.
Synthesis	
	• Implement a simple system on a real-time operating system.

8. Performance Engineering of Real-Time and Embedded Systems course

The third course is Performance Engineering of Real-Time and Embedded Systems. This course is first being offered during the spring quarter of 2005. As of this writing, aspects of the course are still under development. The course is roughly divided in half with the first and second parts emphasizing performance of real-time systems and embedded systems, respectively. This course has an unusual combination of topics and we have not identified a single textbook that is suitable. We are covering the course topics with handouts and other on-line resources for the students.

Topics covered by the Performance Engineering of Real-Time and Embedded Systems course include:

- Performance measurements for real-time and embedded systems

- Profiling of program execution in embedded systems
- Exploration of linear control systems
- Interpretation of linear control parameters
- Hardware system description languages
- Hardware/software co-design

The real-time part of the course presents the control of physical systems on an intuitive level. The intent is to give exposure to control system structure and performance rather than have student design control systems. The software engineers have no background in controls. The computer engineering students are able to contribute to the analytical and control algorithms from their required control systems courses and will take the lead on these projects. Students perform experiments with the inverted pendulum system and a ball and balance beam. These experiments highlight the effect of parameter tuning and system load on control of the physical apparatus. In future offerings, this set of experiments will culminate with student implementations of software controllers.

The embedded systems part of the course uses our target system as the computing element running the VxWorks commercial real-time operating system. We deliberately chose a rather slow (100MHz clock) 486 processor for our target systems so that we could more easily monitor loading effects. This is close to power management policies in low-power embedded devices that prolong battery life by slowing the clock speed. In subsequent course offerings, input and output devices will be connected through an FPGA I/O controller. Students will measure initial system performance when the I/O controller is a pass-through interface between the processor and the devices. The current offering has the students performing a set of JPEG image compressions, first using an all-software approach on the target system, and then off-loading some of the computations to an attached FPGA board. The students will then be able to make a hardware-software co-design tradeoff by placing more device control functionality in the FPGA. At each step the students will measure the change in system performance as the boundary between hardware and software is moved.

Using Bloom's Taxonomy the learning outcomes for this course are given in Table 3.

Table 3
Learning Outcomes for Performance Engineering of Real-Time and Embedded Systems Course

Knowledge	
	• Identify PID control modes • Identify the major characteristics of a Field-Programmable Gate Array (FPGA)
Comprehension	
	• Distinguish differences between PID control modes • Contrast effects of system parameters on control of a physical system.
Application	
	• Profile the execution of an embedded system • Be able to program an FPGA doing minor revisions to VHDL code
Analysis	
	• Describe hardware/software tradeoffs in the design of an embedded system. • Analyze the profiling data to determine which areas of the program would benefit most from performance tuning. • Compare performance of systems based on performance data.
Synthesis	
	• Design a test and measurement plan to collect system performance data. • Demonstrate the effects of moving the hardware/software boundary in a design

9. Evaluation plan

This project has two components in its evaluation plan.

External evaluation: a faculty member from one of our collaborating institutions evaluated our work at the end of the first year in May 2004. At this same time we had an external review by someone working in local industry developing real-time and embedded systems. Near the end of the NSF funding period in June 2005 we will again arrange a review by faculty from our collaborating institutions and local industrial representatives.

Course evaluations and surveys: students enrolled in the courses are given concept surveys at the beginning and end of each course to assess their domain learning through each course. Course evaluations will ask students to assess the course materials, the laboratory environment, the teaching effectiveness and whether the course has increased their interest in real-time and embedded systems or helped them get a co-op or full-time position.

10. Future work

This section describes some areas for improvement that have been identified and other activities for the future.

- One challenge has been to develop courses interesting to the software engineers and computer engineers. The Modeling course is very well liked by the software engineering students but is not as attractive to the computer engineers. We need to balance the topics better so as to make the composite more attractive to both groups of students. Even the SE students suggest that we select projects with more explicit time-dependent requirements. We will also consider designing a project that requires implementation on the Java Micro Edition platform.

- The main exposure to VxWorks is in our first course. We do not have a strict prerequisite structure within these three courses thus we are hesitant to put projects requiring implementation on VxWorks in the other two courses. We need to create a very succinct tutorial on writing applications for VxWorks that we can use in the two courses that currently do not cover the RTOS in detail. It took us quite a while to settle on a configuration for VxWorks in the lab that could easily support 13 simultaneous target systems and give easy distribution of new VxWorks images. We next need to work on giving students the necessary control to create their own images when their project is developing a kernel-level driver. We will also investigate the use of a real-time variant of Linux in these courses.
- The lack of a suitable textbook for the performance engineering course is an issue for that course. We will assess the best approach to follow after the course has run for its first time in our spring 2005 term.
- There are other devices that we would like to have students use with their project work. At the top of the list would be interfacing to cheap USB webcams. Unfortunately, we have not yet identified any cameras that publish their USB interface.
- A last element of dissemination of our work, which will take place at the end of the project, is to collect all of our course materials, projects, exams, etc. onto a password protected website and publicize its availability to the engineering education community.
- The facilities are mostly in place now and this has attracted the attention of other faculty members. We already have one faculty member scheduled to develop a fourth course to be taught in the lab next year.

11. Acknowledgements

This project is being conducted under the sponsorship of a National Science Foundation grant under the Course, Curriculum and Laboratory Improvement Program (NSF DUE-0311269) and in collaboration with Professor Yann-Hang Lee of Arizona State University and Professor Ronald Schroeder of Southern Polytechnic State University. We would also like to thank Mr. Todd Mosher of Alstom Transport Systems for his review of our project's first year.

12. References

[1] Diamond Systems, http://www.diamondsystems.com.

[2] Diligent, http://www.digilentinc.com.

[3] Douglass, B. P., *Doing Hard Time – Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison Wesley, Reading, 1999.

[4] IBM Rational Software, http://www.rational.com.

[5] I-Logix, http://www.ilogix.com.

[6] The MathWorks, http://www.mathworks.com.

[7] MGTEK, http://www.mgtwk.com/miniide.

[8] Quanser Systems, http://www.quanser.com.

[9] Shaw, A. C., *Real-Time Systems and Software*, John Wiley & Sons, Inc., New York, 2001.

[10] Starnes, T, "Microcomputers Infest the Home", Gartner Research, Inc. 2002.

[11] Wind River Systems, http://www.windriver.com

Experiences with the Blackfin Architecture for Embedded Systems Education

Diana Franklin John Seng

Dept. of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
{franklin,jseng}@csc.calpoly.edu

Abstract

In the course of a major curriculum change at California Polytechnic State University, the embedded processing course was redesigned. During this process, the course had the opportunity to purchase new hardware. Analog Device's Blackfin processor was chosen based mostly on cost, but also on performance, development environment, and documentation.

We first present our goals in the class. We then give an overview of the Blackfin architecture and how the Blackfin fits in with many of our goals. We then present the implementation of an expansion board developed to interface with Blackfin's EZ-KIT Lite board.

We present our experiences with this setup in the hopes that others who might be thinking of a similar curricular change can learn from our successes and failures. We outline the strengths and weaknesses of the Blackfin architecture as an educational platform, followed by a discussion of our experiences and a presentation of the support materials we developed to accompany the course, including lecture material and laboratories. Finally, we discuss our future directions for our uses with the board.

1. Introduction

Designing the curriculum for an embedded processing course is especially difficult in today's schools because of the many conflicting goals in curricular design. The ideal would be cheap, flexible, powerful hardware. This be shipped with an industrial-strength, intuitive, feature-rich development environment. Finally, there would be a textbook available that is targeted towards students rather than a manual targeted at professionals. If we take a step back and look at the entire curriculum, we would also like a processor that could be used for a wide array of classes, such as digital signal processing, as well as student projects.

Unfortunately, such a bundle of technology, and educational materials does not exist in a low-cost package. At California Polytechnic State University, San Luis Obispo, we chose to use Analog Device's Blackfin processor. It satisfies several of the above goals, mainly that it is cheap, general and powerful hardware, coupled with a good development environment, but it was not without disadvantages. Our students used the manuals, augmented by lecture slides, but had no textbooks.

In this paper, we explore the tradeoffs that are involved in designing a single class, CPE 316, Embedded Systems, at California Polytechnic State University, San Luis Obispo. We describe our design and how it relates to those tradeoffs. Finaly, we augmented the original hardware and developed a detailed set of lecture slides that follow the Blackfin, which currently has no textbook written for it. We provide the class materials that we developed on-line at http://www.csc.calpoly.edu/ franklin/316/Bundle.tgz

We begin by analyzing our curricular goals for the embedded systems class in Section 3. We continue in Section 4 by describing the Blackfin architecture, our architecture of choice, and the development environment provided. Section 5 presents the expansion board design and the flexibility it gives to the labs. Sections 7 and 8 give a brief summary of our lectures and labs from several instantiations of the class. We give ideas for future development and conclude in Section 10.

2. Related Work

Embedded processing has become increasingly important, and with its rise in industrial significance, the best way to teach the concepts has been studied by several educators.

Many groups have looked at high-level approaches to improving embedded processing education in the curriculum. Michigan State University proposed an approach to integrate embedded processing into the whole curriculum rather than a single course [1]. A full curriculum targeted towards embedded processing, including design from math classes and engineering classes on up, has also been proposed [3]. They stress that high-level principles, not specific information commercial companies might want, should be emphasized.

We take on many of the practical matters in designing an embedded processing course. We assume that the core topics have already been decided. Our job is to convey this information in a way that fits well with the rest of the curriculum, is up to date, is not too costly, and fulfills as any educational goals as possible.

Course	Integration	Financial
textbook	unlike MIPS	inexpensive boards
intuitive software	parallelism	multiple courses
breadboard access		DSP

Table 1. Summary of goals

3. Goals

As with any course development, there were disparate goals in designing this course. We categorize our goals in one of three categories. First, we had the normal goals that anyone does with an embedded processing course, that of conveying the information for the course in the most painless, efficient manner. Second, we had issues with integrating this course with the rest of the curriculum. Finally, we had financial considerations to minimize the amount of hardware necessary to purchase. These goals are summarized in Table 1.

CPE 316 at Cal Poly follows a year of digital design and computer architecture. They have covered the first 7 chapters of the P&H architecture text, ”Computer Organization & Design” [4]. They have not yet covered interfacing processor and peripherals or parallel processing. The students have also taken at least a year of Java programming. The two courses that are not in the prerequisite chain are C programming and assembly programming (other than small portions in the architecture course). Most students had taken one quarter of C, though not all. Almost no students were familiar with particular C keywords integral to interfacing with devices.

Within the embedded processing course, we had several goals. The hardware needs to be easy to use, with a development environment that was intuitive and quick for the students to pick up. Cal Poly is on the quarter systems, so the students cannot waste much time learning new environments. In order to allow control of interesting devices, it needs a mechanism for students to connect their own breadboard to the processor. Finally, the course needs a textbook. There were two choices deemed acceptable - a textbook that is not tied to any single processor coupled with manuals, or a textbook that was specific to our hardware. The former is possibly more realistic for the workplace, although the latter is easier on the students.

No course is in isolation, so there are higher-level goals to consider. Prior to this, the major language is MIPS because of its use in the P&H's architecture book [4]. Students should have exposure to a variety of languages, so an assembly language that illustrates a new set of features is useful. Finally, the students have not yet been exposed to parallel processing, so a language that allows parallel instructions is desirable.

The financial considerations are listed last, but in this economy in a public school, they often become the overriding factor. The boards must be either donated or inexpensive. In order to amortize the cost of the boards, they should be used for multiple classes. To this end, the processor should be powerful and capable of digital signal processing tasks.

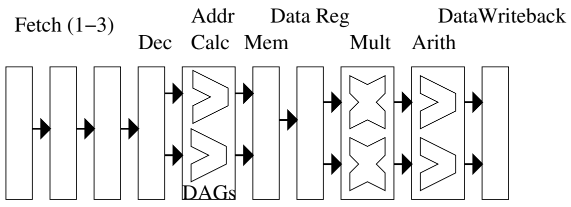


Figure 1. The 10-stage Blackfin Pipeline.

In the end, we were able to satisfy almost every goal except for the textbook. In the rest of the paper, we present how we satisfy the goals through the use of the EZ-Kit Lite Blackfin board and special hardware attached to it. For a textbook, we used a combination of detailed lecture slides and helpful laboratories. We were not satisfied enough with the general textbooks we found to require the students to purchase them. Because this was a senior level course, we expected that this was a more gentle introduction to the resources that will be available on the job.

4. Blackfin Architecture

The Blackfin is a hybrid microcontroller and digital signal processor. We used the EZ-KIT Lite, which was obtained at an educational discount from Analog Devices. We now present the interesting details about the Blackfin environment we had, split up into architecture, assembly language, software development environment, and EZ-KIT Lite board.

4.1 Architecture

The Blackfin is an in-order, multi-issue processor. The pipeline has two data paths throughout. The processing core consists of a 10-stage pipeline. The pipeline is depicted in Figure 1.

Instruction fetch requires three stages, with a decode stage fourth. It fetches 64 bits each cycle, though serial instructions require only 16 or 32 bits. It only executes 64 bits in a single cycle in the presence of a 3-wide parallel instruction.

Stages five and six are for memory operations and branches address calculations. It employs two Data Address Generators (DAGs) for address calculations. Once the branch address is calculated, it uses static branch prediction to go to the predicated destination.

The Blackfin reads the data register file in cycle seven, and then performs computations in cycle eight and nine. For computation, it performs multiplication first and then has an alu for accumulation or any other arithmetic operation. It also includes special-purpose video units. Data results are written in cycle ten.

There are two register files - eight 32-bit data registers and 8 pointer registers. It also has several special-purpose registers for looping and memory address calculations. In addition, there are two 40-bit accumulators, one associated with each multiply/ALU pair.

The Blackfin has three caches - two data and one instruction cache. There is also a very small instruction buffer in the fetch unit that can hold short loops. In each cycle, you can perform a load from all three caches. It may not perform two loads to the same data cache in the same cycle.

The architecture of the Blackfin itself presented an excellent opportunity to reinforce the ideas taught in the computer architecture course. The pipeline was still in order, but it had more pipeline stages and the stages were performed in a different order than the MIPS processor. The students were also able to learn about static branch prediction, which was not emphasized in the previous course. Finally, the presence of dual data caches allows students to think consciously about when their data is accessed in order to place data such that you can access both caches in the same cycle.

4.2 Assembly Language Features

The Blackfin ISA has several unique features beyond the simple MIPS instruction set. The main differences are the address calculation features, control features, variable data widths, and parallel processing.

The DAG allows for a very rich set of addressing modes. In general, one can access a memory location at a constant offset from a register index and increment the index in a single instruction. Furthermore, it allows circular addressing with a stride, automatically wrapping the pointer around when it reaches the end of the buffer. It also has bit-reverse addressing specifically designed for the FFT algorithm.

In order to maintain high performance with a 9-stage pipeline, the Blackfin needs support for branching. The Blackfin provides two major mechanisms to alleviate control hazards. First, it provides static branch prediction. Any branch can be labeled to be predicted taken. Unfortunately, this only saves four out of eight stall cycles. The address is not calculated until cycle four, so for loops with a known number of iterations, the Blackfin provides a zero-overhead loop mechanism in the fetch unit. It can keep track of two nested loops at once. The entire loop is buffered in the unit, along with the counter and the beginning and end program counters. This automatically provides the proper instruction, with no stalls, until the loop is complete.

The Blackfin provides support for 16-bit operations as well as 32-bit operations. It can either perform a single 16-bit operation on each ALU or have each 16-bit half of a 32-bit number be treated as a separate 16-bit value for the purposes of arithmetic operations. This allows one to perform four 16-bit operations in a single cycle when employing both ALUs.

Finally, as referred to above, the Blackfin allows limited parallelism. It may perform two 16-bit and one 32-bit operation at once, drawn from a list of parallelizable operations. Only one store may be performed each cycle, though one can perform two loads. There are two DAGs, so address offsets and updates may also be performed in parallel.

This instruction set satisfied all of the educational goals of the assembly language. The advanced branching instructions allowed for an excellent tie-in of core architectural material to the course, and the parallel instructions provide a unique opportunity. This was especially important because even correctly predicted branches as well as unconditional jumps had a 4-cycle penalty. The ability to control branches in the assembly language and think about the performance ramification makes the knowledge more concrete.

4.3 Software

The software environment needed to be intuitive and easy to pick up, especially in our quarter system. We use the Analog Devices' Visual DSP++ as an integrated development environment for the class. Visual DSP++ is designed to be used with the EZ-KIT lite, a processor simulator, or with a JTAG interface. This program allows programming the board in Blackfin assembly or C and provides an overall interface which is highly similar to other integrated development environments.

The only problem with this software is a combination of hardware problems and the license server. Occasionally, it gets into a state in which the student can no longer control the hardware. If they close the program, the license does not always return to the license server right away. Upon attempting to restart the program, the license server will say it is out of licenses. This requires a license server restart.

4.4 EZ-KIT Lite Board

The EZ-KIT Lite Board provides I/O opportunities for students with the Blackfin chip. They provide flexibility in augmenting their design by having flash memory that can be used to configure different input and output pins.

The most basic functions that are fun and easy to use are the LEDs and the pushbuttons. The sample codes that come with the board are simple for the instructor to understand. The board also includes more advanced features like audio/video and bus protocols.

There were two problems with the board. The board has a set of switches on it that, if changed, cause the board to act in odd ways. At Cal Poly, the labs are open to allow senior project students to use hardware for their projects, but they are not monitored at all times. Students will sometimes flip the switches, and it is difficult to tell. This caused several students to lose whole lab periods getting the hardware to work again.

Second, there is no good access for connecting a student breadboard. Section 5 describes the expansion board designed to give students access to several input and output pins on the EZ-KIT Lite board.

4.5 Documentation

The Blackfin architecture has a Hardware Reference Manual(HWR) as well as a separate Instruction Set Architecture Manual(ISA). In addition, the EZ-KIT Lite Board has a manual. These manuals are all electronic. Students may request hard-copies as well, though they are very large and heavy.

The Blackfin HWR and ISA manuals are very well indexed and easy to navigate through Acrobat Reader. The EZ-KIT Lite is a little more difficult to utilize efficiently. We found that the students were more comfortable with the physical versions of the books and had not had much experience with electronic manuals. In retrospect, I wish I had done a half a lecture on how to navigate the manuals effectively.

4.6 Discussion

The Blackfin 533x on the EZ-KIT Lite board satisfied our hardware goals. It had an intuitive environment, though not bug-free, it was inexpensive, it had an assembly language sufficiently different from MIPS, allowed for parallel execution, and had the functionality for digital signal processing. The only thing it lacked was a simple interface to a student breadboard.

Our experience with this hardware was mostly positive. When problems occurred, though, it was very difficult to track them down. It could be the students' software, the hardware switches, the connection to the development environment, or a bad state. When restarted, sometimes the license server would then fail.

To alleviate this, students should be counseled early in the class to save working versions of their code to determine whether a problem is with their code or the board. In addition, the students need easy access to someone who has the authority to restart the license server.

5. Expansion Board Design

Although the Analog Devices' EZ-KIT lite board is highly integrated and provides excellent performance, the board is not designed to be readily used in an educational environment. Several of the board pins are connected to other chips and are not available for use through on-board pin headers. Unfortunately, the board does not provide easy access to input/output pins. What the board does provide is a 3-socket expansion interface intended to be used with other Analog Devices' expansion cards. Each socket is a 90-pin connector with a fine pitch spacing. We use this interface to connect a custom expansion board for use in a class lab environment.

Our expansion board contains simple circuitry to buffer some of the input/outputs pins on the board. One fact to note when using the I/O pins of the Blackfin is that the I/O pins on the Blackfin processor use 3.3V interface circuitry. Connecting 5V circuits directly to the I/O pins would damage the Blackfin. Instead, we used voltage level conversion buffers to allow 5V circuitry to be used during the labs.

The expansion board design provides a modest number of digital inputs and digital outputs. The design allows software to control 8 digital outputs and 8 digital inputs. Should more inputs and/or more outputs be required, an SPI I/O port expander would be good for that purpose. Also, a CD4094, would work well as an output expander because of the shift-and-store inteface it provides.

A 24-pin ribbon cable is used to connect the students' breadboard with the Analog Devices' board. On one end of the cable is a polarized connector which connects with the expansion board, and on the other end is a 24-pin DIP socket which plugs directly into a breadboard.

6. Textbook

Currently, no textbook exists for that targets the Blackfin architecture. We considered a more general textbook, such as Computers as Components [5]. Although this was useful to use as instructors, and we incorporated some of the publicly available on-line slides into the lectures, it was at such a high level that we made it a recommended textbook, not a required textbook.

This meant that are lecture notes were the only resource the students had beyond the manuals. Our lectures slides are a combination of high-level, general material, followed by specific information for the Blackfin architecture.

7. Lectures

The lecture slides were a combination of theoretical material and Blackfin-specific implementation. The figures for the Blackfin-specific material were obtained from the hardware and ISA manuals [2].

We are releasing the slides so that they may be used as a building block for someone to tailor their own slides if they wish. They are by no means complete and will continue to be developed as the class is taught more often.

7.1 Lecture Topics

We have created a set of lectures that cover the core embedded processing subjects as well as additional special topics that are related to architecture and embedded processing in general. The core topics are:

- Memory-Mapped I/O / Polling

- Interrupts
- Timers
- Ports / Buses
- DMA and Power
- analog / digital conversion

We also added several topics, ranging from architectural lectures to tie the chip back to concepts introduced in the architecture classes to pure C and assembly programming techniques.

- Blackfin Overview / ISA describes the overall architecture as well as giving examples on branching mechanisms and loading and storing.
- Blackfin Pipeline gives details on what each pipeline stage performs including timing diagrams of instruction sequences and their stall cycles.
- Blackfin Calling Convention presents generic function call convention with the specific rules of the Blackfin processors. It also covers the difference in calling convention between concentional functions and interrupt handlers.
- Static Branch Prediction gives details on the zero-overhead-loop instructions, static branch prediction, and conditional instructions. It includes timing diagrams and statistical performance problems. Finally, it relates the branch penalties to what stages operations occur in the pipeline.
- Parallel Processing covers statically scheduled parallel programming, Blackfin parallel instructions, loop unrolling, and software pipelining.
- C for Assembly Programs presents C keywords that range from necessary to useful when programming with devices. First a brief overview of memory regions and scope in C. The keywords *volatile, register, static, inline* are shown. A memory example of exploiting two data banks is given. It moves on to several Blackfin-specific tricks like the keyword *restrict*, making easily-recognizable circular buffers. Finally, it shows how to interface C functions with assembly functions and use inline assembly.
- Optimizing Code introduces the idea of profiling, Amdahl's Law, and test input sets. It then presents several optimization techniques like DMA, data locality, and some simple examples of branch removal.

7.2 Discussion

The additional topics were taught only in the second instantiation. This led to some different observances in the lab work for the course.

The first time this course was taught, before the C for Assembly lecture was included, students strongly preferred using assembly in the laboratories. After the addition of the C lecture, students were much more comfortable using C, and more than half of the students used C when they were given a choice.

Before the calling convention lecture, students had very little idea of how, from a register point of view, the handler should be written. Some students were reserving registers to be used as communication between the main loop and the ISR, whereas others were destroying random registers without realizing that this would affect the registers used in the main loop. This greatly enhanced the understanding of both the unpredictability of when the ISR is called and the importance of register usage conventions.

For the rest of the extra lectures, they are very much bonus material intended to reinforce concepts learned in either assembly language courses or architecture courses. An embedded processing course is the ideal place to do this, since this is sometimes the first time students have needed to program in a meaningful way at this level. In previous courses, they often felt the assembly language was just an educational task with no real purpose. Once they see the usefulness, one needs only to bring in a performance-critical problem in order to expand the focus of the course. This gives the opportunity to teach about profiling and high-level code optimizations all the way down to branch prediction, code scheduling, and pipelining. It can serve as a great culmination of all of the software and hardware skills the students have learned.

8. Labs

Our labs were designed with a few goals in mind. First, we wanted to target the skill sets of polling, interrupts, and control. Second, we wanted to make the labs interesting so that by the end of the quarter, the students could imagine themselves building a robot if called upon to do so. Finally, we needed to fit everything in a 10-week course. The labs below are not from a single instantiation of the class, but chosen from various instantiations of the class. They are not the entire assignments, either, but the portions that the students found the most fun. The actual assignments had small pieces that are not mentioned, culminating in a larger assignment at the end. The full text of the labs can be perused at: "http://www.csc.calpoly.edu/∼franklin/316/Labs.html."

8.1 Polling

The original polling labs were fairly uninteresting, only requiring the students to respond to button presses by changing patterns on the LEDs.

A proposed future lab would create a Simon Says game where the LEDs would light up in a certain sequence, and the player would need to repeat that sequence with the buttons. The computer would keep generating faster and longer sequences until the player could no longer get the sequence correct.

8.2 Interrupts

The interrupt lab was a ping-pong game, where the LEDs represent the ball, and the buttons represent the paddles. A player can lose by either pressing the button at the wrong time or not pressing the button when the ball is there. At the end, display a message that indicates both who won and why they won. As the game continues, the ball needs to accelerate.

This lab served several purposes. It was fun for the students, required thought as to how to detect all the ways to lose, and allowed for some flexibility in design by having them decide how to display the loss. Several students even implemented extra functionality by allowing a game reset with one of the other buttons. In one instantiation of the course, this was the most successful lab.

8.3 Nested Interrupts

The nested interrupts assignment was a part of the interrupts lab. They were to display morse code depending on what button was pressed, but allow interruption of displaying the different patterns depending on which other button was pressed. They were to implementing the displaying of the pattern in the interrupt service routine, not in the main program.

8.4 Timers

For this lab, the students built a dimmer. The light's brightness was controlled by the amount of time the light was turned on. Timers controlled the light turning on and off. When one button is pressed, the light gets dimmer, and another causes the light to get brighter.

The students enjoyed this lab very much. The biggest mistake was to change how often the light turned on and off without ever turning the light on for a longer period than it was turned off.

8.5 Advanced labs

In various instantiations of the class, the last lab involved the students receiving input from external devices, performing some operation and producing output for an external device. These devices could be hooked up to the breadboard.

Servo Lab The servo lab used a standard hobby servo that is controlled by a 1-2ms pulse with a period of 20 ms. If the pulse width is 1ms, it is turned all the way to the left. At 2ms, it is turned all the way to the right. You can place it anywhere in between by adjusting the width between 1-2ms. The period must stay constant at 20ms.

The servo was controlled by the buttons. There were two instantiations - two buttons set them to far left and far right, while the two middle buttons made the servo rotate slowly to the left or right. In the other version, all four buttons determined four positions for the servo to point.

Potentiometer A potentiometer dial, when rotated, adjusts the power it is sending between 0 and 5 volts. This is then connected to an ADC0831 and read in by the students.

The ADC0831 interface was the most complex the students encountered. They needed to transmit a chip select signal along with a clock to the ADC0831 and then sample the incoming bit 8 times in order to obtain the 8-bit value for the volt.

Students did not realize how precise the timing needed to be about putting the chip select down before beginning the clock, and then waiting a cycle before beginning the sample.

The potentiometer was used to control the LEDs. The LEDs could either display the 8-bit number in binary, or it could look more like a voltmeter with the number of lights growing from one side or another.

The potentiometer and servo can be combined to have the potentiometer control the servo. This involves more coordination for the students, but they thoroughly enjoyed getting the hardware to work. This lab was a highpoint for many of the students.

8.6 Discussion

There are many ways to design the labs. In our quarter-system environment, we felt the need to streamline the labs so that the students could learn the most concepts in the shortest amount of time. This led to tradeoffs in how the labs were structured as well as to how much information was provided to them.

When designing the labs, we had a trade-off between small problems that targeted specific skills and large labs that would take fewer different files. Due to a combination of the development environment and the fact that they were initially coding in assembly, the overhead with beginning a new program was quite large. In retrospect, it is important that different parts merely build on each other and do not require a new codebase. What were listed above are the core projects, although the actual labs often include some smaller, simpler parts before building to the full lab. The intent of the smaller parts was to allow for more partial credit if students could not get the whole thing working. In the future, teaching the students about how to break down large projects in order to test them thoroughly would have been better than cutting the projects up into different parts that did not directly build on each other.

There were also differences in how we implemented the labs. The first instantiation of the course provided students with only the manuals, requiring them to begin from scratch. The second instantiation of the course provided sample code (often the code similar to that shipped with the board) so that they could use that as a baseline and modify it for the specific assignment. In order to try to ensure the students took the time to understand the given code, a set of questions was asked about the sample code and turned in. This definitely made it quicker for students who could learn from sample code to finish the projects. Several groups that understood the concepts completed early labs in very little time. Struggling students resorted to some method of random code replacement, not truly understanding the sample code and often not making the changes to it in the right places. It is clear that for strong students, the sample code method removed much of the tedium that would have been involved and took nothing away from the learning. For the struggling students, however, it is unclear which was better. With no sample code, they do not know what to generate on their own, so it would take much more time to solve the labs. On the other hand, if they solve the early lab, that would give them a more solid foundation to solve later labs. With sample code, they could more easily fool themselves into thinking they were not so lost.

9. Future Work

Since the course is still in its first year, it will continue to be developed in the coming years. In the future, we will be augmenting our slides with material and improving the laboratory projects.

For the lectures, the more general material was not integrated seamlessly into the Blackfin-specific details. More work will be done in the following year with obtaining support materials for the students and integrating them into the lecture slides.

In addition, more hardware components can give new and interesting laboratory assignments. There are a variety of labs that could be added for a course that is a semester long. This would open up the possibility of an open-ended project for the last month of the course. In addition, we did not have time to touch upon code optimization in the laboratories. We could give a task and have the students learn how to profile code, time their code with on-board timers, and have a contest as to which group had the fastest solution. The students were very excited about such a prospect.

10. Conclusion

Embedded processing courses will always have a difficult time keeping up with technology because students work at the assembly level. Textbooks are hard-pressed to keep up with the new hardware offerings, and schools face many pressures when choosing a development platform.

We give analysis on what problems were faced in designing our embedded processing course. We found a hardware / software environment that serves most of the goals set out, and we have augmented the available materials with our own. Our materials are now publicly available. The course was largely successful, with just a few changes needed in the material to present in order to make up for the lack of a textbook. We hope others who choose the same setup will be able to learn from our contributions of materials and experiences.

References

[1] B. Chang, D. Rover, and M. Mutka. A multi-pronged approach to bringing embedded systems into undergraduate education. In *ASEE*, 1998.

[2] A. Devices. *Blackfin Processor Family Manuals*. ADI, 2005.

[3] S. Guangfan, W. Peidong, L. Jinbao, and W. Kaizhu. A curiculum design and consideration for the embedded systems. In *ICITA204*, 2004.

[4] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann, 2004.

[5] W. Wolf. *Computers as Components*. Morgan Kaufmann, 2001.

SPIMbot: An Engaging, Problem-based Approach to Teaching Assembly Language Programming

Craig Zilles
Department of Computer Science
University of Illinois at Urbana-Champaign
zilles@cs.uiuc.edu

ABSTRACT

This paper describes SPIMbot, an extension to James Larus's widely-used MIPS simulator SPIM, that allows virtual robots to be controlled by writing programs in the MIPS assembly language. SPIMbot was written to provide an engaging environment to motivate students to learn assembly language concepts. The SPIMbot tool allows the development of scenarios—in which students must program the robot to perform certain tasks—and provides the means to compete two robots against each other.

In our sophomore/junior-level class, we structure the programming component as a collection of structured assignments that produce sub-components for the robot; these sub-components are then used in a final open-ended programming assignment to produce an entry for a SPIMbot tournament. In our experience, this has been an effective means of engaging students, with many students investing time to aggressively optimize their implementations. SPIMbot has been effectively used in large classes and its source code is freely available [8].

1. Introduction

As one of their "Seven Principles for Good Practice in Undergraduate Education", Chickering and Gamson [1] list **emphasizing time on task** as number 5. They state:

> Time plus energy equals learning. There is no substitute for time on task.

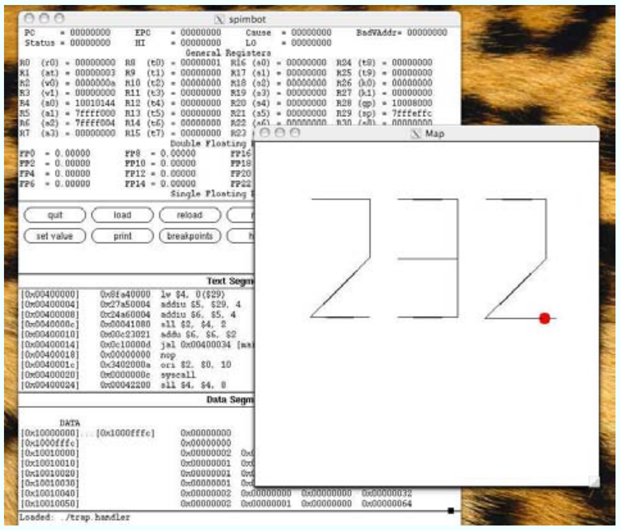
Thus one of our chief tasks as undergraduate educators is to develop activities that encourage our students to spend time on the course concepts and approach them with desire to master them. This paper describes one such set of activities, focused on teaching concepts related to assembly language programming.

In the remainder of this section, we describe the motivation for this work (Section 1.1) and abstractly how we use SPIMbot to achieve our pedagogical goals. After discussing the capabilities of the software (Section 2), we discuss, in detail, how it was used in the Spring 2004 semester (Section 3). We conclude, in Section 4, with a discussion of student feedback that supports our assertion that SPIMbot is an engaging way for students to learn assembly language programming concepts.

1.1 Motivation

In teaching assembly programming in our Computer Science curriculum[1], we have two primary goals: 1) to provide students a mental model of how a computer executes their high-level language (HLL) programs, and 2) to provide the background knowledge necessary for later courses on compilers and operating systems. To this end, we teach the students about instruction sets, stacks and their management (including recursion), calling conventions, floating point arithmetic, instruction encoding, I/O interfacing, and interrupt handling.

If one is not careful, these topics can come across as dry.



```
PC     = 00000000   EPC    = 00000000   Cause  = 00000000   BadVAddr= 00000000
Status = 00000000   HI     = 00000000   LO     = 00000000
                            General Registers
R0  (r0) = 00000000  R8  (t0) = 00000001  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000003  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 0000000a  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 10010144  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = 7ffff000  R13 (t5) = 00000000  R21 (s5) = 00000000  R29 (sp) = 7fffeffc
R6  (a2) = 7ffff004  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000000  R23
                            Double Floating
FP0  = 0.00000      FP8  = 0.00000      FP16
FP2  = 0.00000      FP10 = 0.00000      FP18
FP4  = 0.00000      FP12 = 0.00000      FP20
FP6  = 0.00000      FP14 = 0.00000      FP22
                            Single Floating

                            Text Segm
[0x00400000]    0x8fa40000  lw $4, 0($29)
[0x00400004]    0x27a50004  addiu $5, $29, 4
[0x00400008]    0x24a60004  addiu $6, $5, 4
[0x0040000c]    0x00041080  sll $2, $4, 2
[0x00400010]    0x00c23021  addu $6, $6, $2
[0x00400014]    0x0c10000d  jal 0x00400034 [ma
[0x00400018]    0x00000000  nop
[0x0040001c]    0x3402000a  ori $2, $0, 10
[0x00400020]    0x0000000c  syscall
[0x00400024]    0x00042200  sll $4, $4, 8
                            Data Segm

       DATA
[0x10000000]...[0x1000fffc]     0x00000000
[0x1000fffc]                    0x00000000
[0x10010000]                    0x00000002  0x
[0x10010010]                    0x00000001  0x
[0x10010020]                    0x00000001  0x
[0x10010030]                    0x00000001  0x
[0x10010040]                    0x00000002  0x00000000  0x00000000  0x00000032
[0x10010050]                    0x00000002  0x00000001  0x00000000  0x00000064
Loaded: ./trap.handler
```

Figure 1: Example SPIMbot screen shot. *The map window shows the robot's current location, orientation, and virtual environment; in this scenario, SPIMbot can turn on/off a paint trail allowing it to write out messages. Behind the map window is the main window (unmodified from xspim) that shows the MIPS processor's machine state.*

[1] Assembly programming is taught in the context of the second semester-long class in a required two-class sequence in computer architecture. The first class in the sequence teaches digital fundamentals: the digital abstraction, combinational logic, finite-state machines, and basic architecture concepts (*e.g.*, a single-cycle implementation). The second class covers three main topics: assembly programming, machine organization, and memory and I/O systems; each topic receiving roughly a third of a semester. As our undergraduates predominantly pursue software-oriented (rather than hardware-oriented) careers, the goal of this second class is to provide the practical understanding of computer hardware necessary to be an effective programmer. Most students continue their architecture sequence, taking a third course in either high-performance architecture or embedded systems.

The students' limited programming experience (this class is early in the curriculum) coupled with the inherent inefficiency of assembly programming can limit the scope of programming assignments. Furthermore, the demands of grading, especially in large enrollment classes where some form of automation is necessary, require most assignments to be rather structured. Examples of common assembly programming assignments found at many universities include: producing the Fibonacci sequence, string manipulation (reversing a string, `toupper()`, etc.), and sorting arrays. In many cases, HLL source is provided, reducing such assignments to somewhat mechanical translation.

The goal of SPIMbot was to produce an environment for teaching assembly programming that was fun and interesting, to motivate students to want to learn the material. While there is a long history of using robots for instruction (*e.g.*, [5]), the author's inspiration came from Patricia Teller's presentation [7] at the 2003 Workshop for Computer Architecture Education. In their semester-long course on assembly programming concepts, students program 68HC11-based robots to escape from mazes and chase other robots. Pedagogically, programming robots has three appealing features: 1) it is visceral: students like seeing their code control motions and actions of objects in the physical world, 2) it is cognitively challenging: debugging requires mapping robot behavior back to the behavior specified in the code, and 3) it provides a non-contrived way to expose students to I/O programming.

The problem with (physical) robots is one of logistics; in a high enrollment class—we have 100-150 students per semester—acquiring, maintaining, and scheduling sufficient resources is prohibitive. In contrast, virtual robots are cheap, plentiful, take-up no space, require no maintenance, yet (for students accustomed to interpreting computer-rendered virtual realities) still provide the fundamental qualities of physical robots.

1.2 How we use SPIMbot

The central part of our implementation is the SPIMbot tournament, a friendly competition between the programs that the students write. The contest presents a challenging, multi-part task for the robots to perform. We use this concrete task to motivate the presentation of the desired assembly language concepts and the problem solving/design process.

As most of our students have not been exposed to assembly language previously, the SPIMbot tournament is the last activity in our assembly language segment. We work up to the contest by solving isolated sub-problems as programming assignments. We start with small structured assignments and then move onto larger structured assignments before attempting the contest (a large open-ended assignment). This structure lets us provide the students with early, motivating successes.

Although it is the last assignment, we present the contest first, because it allows us to model a problem solving process: a top-down design, followed by a bottom-up implementation. In class, we brainstorm approaches to the contest task, making it clear that there are multiple approaches. Then, we identify sub-tasks necessary for accomplishing the contest goal; these sub-tasks make up the structured programming assignments leading up to the contest. The contest itself challenges students to figure out what they need to implement and requires them to integrate the components they've completed in previous assignments.

When it comes to covering the desired course material, the fact that SPIMbot exists only in a virtual reality can be an advantage, as we can structure that reality to include those concepts that we want to teach. For example, two concepts that we cover in the course are recursion and the implementation of linked-data structures. To incorporate these concepts into our programming assignments, our Spring 2004 contest (see Section 3) involved an I/O device that returned its output as a tree, requiring students to write a recursive procedure to traverse the nodes of the tree.

After the students have submitted their contest entries, we use one class period to hold a tournament. With each competition lasting about 15 seconds, a double-elimination tournament for 32 teams can easily be held in a 50-minute class period. While this class time could be used for other purposes, we believe that it successfully motivates students to be actively engaged with course material *outside of class* achieving our objectives.

A Note on Competition: As competition can be demotivating if not handled properly [2, 3], we take a number of steps to alleviate the potential downsides of competition: 1) performance in the competition is responsible for a minimal fraction (about 1 percent) of student's final grade, 2) students compete as teams, reducing the pressure on individuals, and 3) teams select team names allowing students to compete anonymously.

2. SPIMbot Software

SPIMbot is an extension of James Larus's widely-used MIPS simulator SPIM [4]. SPIMbot involves three major enhancements: 1) a framework for simulating robots and their interactions with a virtual world, 2) a 2-D graphical display to visualize the robots and their environments, and 3) support for concurrently simulating multiple programs—each on their own virtual processor—allowing multiple robots to be simultaneously active in a single virtual world.

Simulating the virtual world requires tracking and updating the state of the robots and other objects in the simulated world. In addition to location, orientation, and velocity, we have to keep track of the state of any I/O devices. Updating the world involves computing new locations for objects based on their current velocities. Collision detection is performed to update an object's velocity/orientation (*e.g.*, when a robot runs into a wall) and to allow interaction between robots and simulated objects (*e.g.*, when a robot picks up an object or pushes a button). Events in the virtual world can also trigger events in the MIPS processor, either updating the state of an I/O device and/or triggering an interrupt.

To interact with the virtual world, SPIMbot provides the robot programmer an (extensible) array of input/output devices. These virtual I/O devices, like real I/O devices, have their I/O registers mapped to memory addresses and, thus, are accessed using normal loads and stores. Simple examples include "sensors" that tell SPIMbot its or the opponent's (X,Y) coordinates and "actuators" to control its orientation. The SPIMbot code is structured so that the collection of I/O devices can easily be extended for a particular scenario. Furthermore, SPIMbot includes a programmable interrupt controller (PIC) that allows individual device interrupts to be enabled/disabled. Standard interrupts include the "bonk" interrupt (raised when SPIMbot runs into something) and timer interrupts (SPIMbot includes a programmable timer). The collection of interrupts can also be extended.

To achieve a tight coupling between the virtual world and the simulated MIPS code, we interleave the simulation of the virtual world with that of the MIPS code. Every *cycle* we execute a single instruction for each robot and update the physical world based on the actions of the robots. Simulating multiple concurrent robots required eliminating the use of global variables in SPIM's parsing and simulation of MIPS code; while currently we only simulate two robots, this could easily be extended to any number. As there can be interactions between the robots, we alternate each cycle which robot is simulated first in an attempt to be fair.

The graphics are currently decidedly low tech—XWindows drawing primitives are used to draw geometric shapes (lines, boxes, circles, etc.)—but this appears to actually have two advantages: 1) it is very simple; a minimal amount of development time is required to add the rendering code for a new scenario, and 2) it is not distracting; students can focus on what the graphics represent instead of the graphics themselves. Because the graphics are not demanding, smooth animation can be achieved without state-of-the-art hardware. In part this is because the graphical display need not be rendered every cycle. Currently, we re-draw every 1024 cycles and can achieve a refresh rate over 60 Hz on a 1GHz laptop.

3. Example Scenarios

In this section, we discuss one scenario in detail to demonstrate how we organize the competition and the assignments that lead up to it and, then, discuss two other competitions more briefly to demonstrate the expressiveness of SPIMbot.

3.1 Spring 2004: Token Collection

In the Spring 2004 semester, the competition revolved around collecting "tokens": 15 tokens were randomly placed on a square map, tokens could be collected by driving over them, and the location of tokens can be divined by using an I/O device called the "scanner." The winning robot was the one that collected the most tokens by the end of competition.

Writing a program to compete in the contest involved: 1) allocating memory for the results of a scan, 2) communicating with the scanner to initiate a scan, 3) handling the scanner's interrupt, 4) searching the tree-like data structure returned by the scanner for the location of tokens, and 5) repeatedly orienting SPIMbot toward a token and recognizing when it has arrived, until all tokens have been collected. As this represents a relatively difficult programming assignment for students at this point in the curriculum, we broke out major components of the program as individual programming assignments. Below is a list of the structured assignments that led up to the contest:

1. A SPIMbot introduction: write a simple interpreter that reads a string of commands (*e.g.*, turn, wait, paint on/off) and invokes provided functions that perform these actions. *Introduces students to SPIM/SPIMbot and exposes students to loops, arrays, calling functions, control flow and I/O interfacing.*

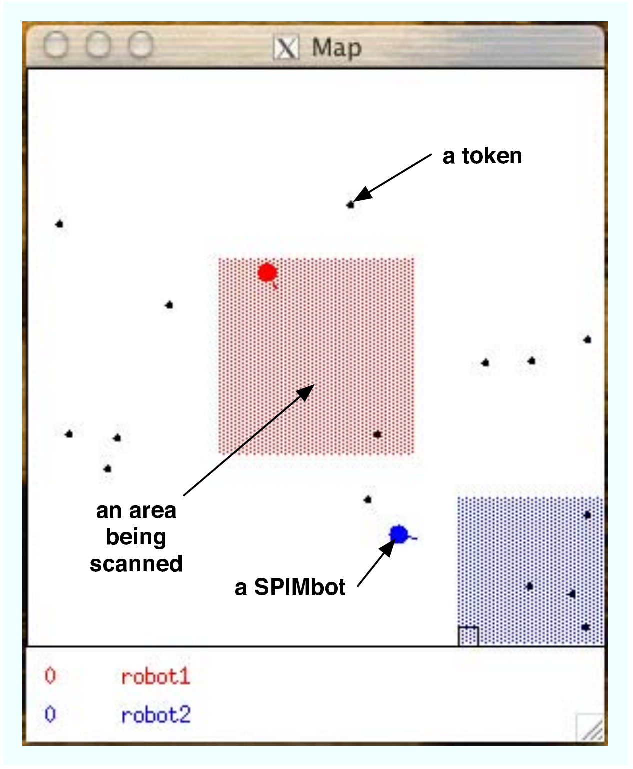


Figure 2: SPIMbot token collection competition.

2. Arctangent Approximation: given the (x,y) location of 2 points, compute the angle to drive from one to the other using a Maclaurin series expansion. *Exposes students to computing in floating point.*

3. Tree Traversal: SPIMbot's scanner returns the location of the tokens embedded as leaves of a tree-like data structure. Students write a recursive function that traverses the tree. *Exposes students to linked data structures and recursive functions in assembly.*

4. Interrupt Handler: write an interrupt handler for the timer interrupt that commands SPIMbot to turn 90 degrees and resets the timer, resulting in SPIMbot driving in a square. *Introduces students to writing interrupt handlers.*

While the solutions to these assignments can be integrated into a working contest entry, designing a competitive entry requires a little more effort. Three activities dominate the execution time of most of the robots: scanner latency, tree traversal, and collecting tokens. In a straight-forward implementation, which scans the whole map at once, these activities are performed completely sequentially (Figure 3a).

A higher performance implementation can be developed which pipelines the scan/traversal/collection process. The scanner can be programmed to scan only a portion of the map at a time, and its latency is largely a function of the area scanned. Once a small portion of the map has been scanned, the robot can begin collecting tokens from that portion while it requests the scan of the next region. In this way, much of the scan latency can be overlapped with the latency of tree traversals and token collection. Students

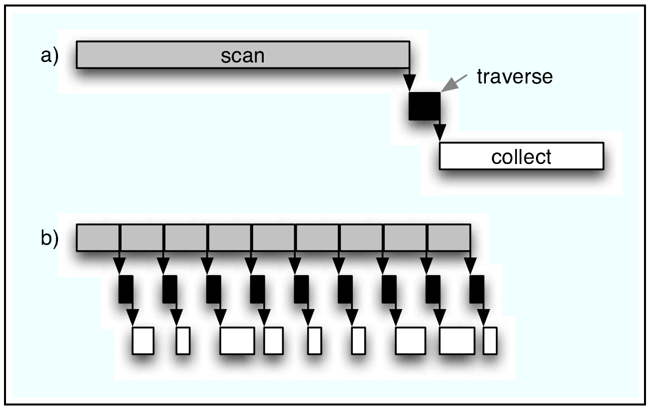


Figure 3: Pipelining the three sub-tasks reduces the latency of the task. *By scanning one-ninth of the map at a time, the pipelined version (b) overlaps the collection of tokens with the scanner latency, completing the task significantly before the non-pipelined version (a).*

found that breaking the map into 9-36 pieces and pipelining the processing of those pieces resulted in good performance. Another enhancement that students developed was driving to the center of the region currently being scanned after all known tokens had been collected.

Developing such a pipelined solution requires managing concurrent activities and demonstrates the importance of interrupts. The students learn first hand that their interrupt handler must avoid clobbering the applications registers, because it could be called at any time. It also demonstrates that pipelining—a concept we introduce in the machine organization portion of the class—is not a concept that is restricted to hardware.

3.2 Fall 2004: Block Pushing

In the Fall 2004 semester, the contest revolved around pushing blocks onto your side of the map (see Figure 4). The contest had a fixed running time and the winner was the one with the most blocks when time ran out. Elementary physics were implemented so that robots could push blocks, which in turn could push other blocks. An I/O device was provided that could be queried to provide the location of each of the blocks.

Like the token collecting contest, we integrated a computational challenge into the contest. Initially, a most of the blocks are "locked" to one or both of the robots. When a robot runs into a locked block, an interrupt is triggered and the robot receives a six character string. This string is a scrambled version of common english word, which, if unscrambled, can be used to unlock the block for this robot so that it can be pushed. As machine problems leading up to the contest, students wrote a string compare function, a function that would do a binary search of a sorted dictionary looking for a given word, and a recursive function that produces every permutation for a 6 character word. These functions can be integrated to unscramble the scrambled clues.

Because we provided the dictionary to the students ahead of time, there was a significant opportunity to optimize the unscrambling function by offline computation. The following is representative of what the winning robots did: 1) sort

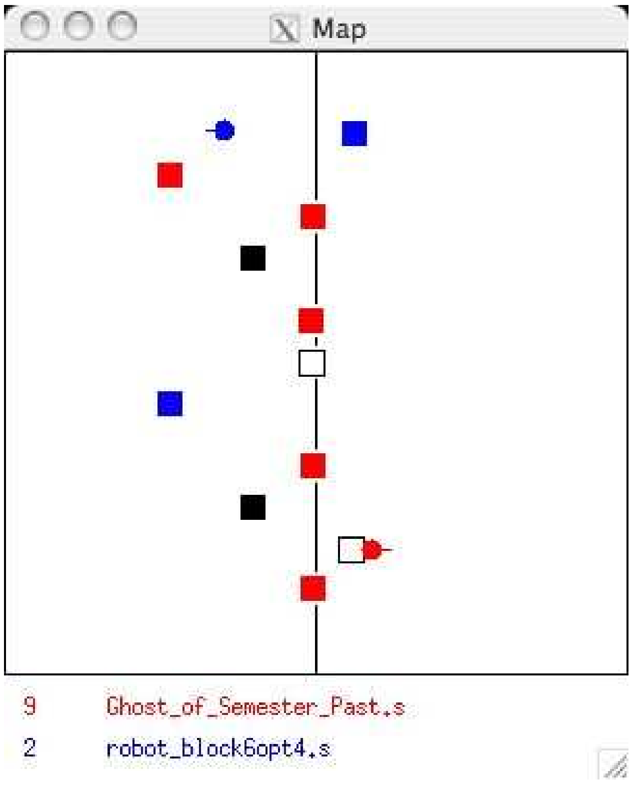


Figure 4: SPIMbot block pushing competition.

the characters in the scrambled word into a canonical order (*i.e.*, alphabetical order), 2) as only the 26 lower case letters were used, each ascii character could be represented in 5 bits; use this insight to translate the 6 char string into an integer (6 * 5 bits = 30 bits), 3) do a binary search on a precomputed table that maps these canonical integers to the strings they encode.

3.3 Spring 2005: Maze Traversal

This Spring semester our contest goal was to completely traverse a maze without being able to see the walls (see Figure 5). Since the mazes we generate are *unicursal* (*i.e.*, there are no isolated islands), the "right-hand rule" (*i.e.*, never letting your right hand leave the wall) can be used traverse the whole maze. Alas, SPIMbot does not have arms, much less hands, but the right-hand rule algorithm can be implemented with two interrupt handlers, by periodically checking to see whether a wall is still to the right of you, as follows: 1) request timer interrupts at a period so that roughly one is received for each square visited; when a timer interrupt is received, turn right and request another timer interrupt, and 2) when you run into a wall (which triggers a "bonk" interrupt), turn left. This was assigned to the students as a machine problem.

The computational challenge for this contest was to sort an array of double precision floats to find the Nth highest number (for varying N). Each time the correct value was identified, the SPIMbot was provided additional "energy"; energy could be used to drive faster, or, in large amounts, to drive through walls for short periods of time. Incorrect answers were penalized, so that the expected value of random guessing would be negative.

As a machine problem early in the semester, the students implemented a bubble sort, but there is clearly much opportunity to do better. A number of students implemented quicksort with the optimization of, at each stage, only sorting the partition that contains the Nth number. A few groups recognized that because a small error rate could be tolerated, the computation could be done on the integer pipeline only loading the top word of the doubles; this optimization saves one cycle on each of the load, because lw.d is translated into two instructions. The winning group realized that, because multiple guesses were allowed, the penalty for incorrect guesses was low enough that it was more efficient to guess an expected range for the Nth value (based on the properties of our random number generator) and perform a single pass over the array guessing any number in that range. In this way, their robot could maintain full energy while constanting driving through walls; their run time was minimized by finding the shortest path that visited every square.

Scenario Implementation Time: After the Spring 2004 semester, we re-factored SPIMbot's implementation to decouple the scenario-specific aspects from the core of SPIMbot's implementation. With these changes in place, it is rather straight-forward to implement new scenarios, by implementing a collection of functions for supporting scenario-specific initialization, physics, drawing, and I/O devices. The Fall 2004 scenario required about a day to prepare; this accounts for the time to implement both the SPIMbot code, as well as the MIPS code to test the scenario (which includes solutions to most of the structured assignments). The Spring 2005 scenario took longer to implement (perhaps a 40-hour week of programming time), but was largely completed by an undergraduate.

4. Student Reaction

The Spring 2004 students had a quite positive opinion of the SPIMbot assignments and student anecdotes suggest that they found it engaging. Students were asked in an anonymous electronic survey to rate their enjoyment of the SPIMbot assignments on a 5-point scale (5: "very much so" to 1: "not at all"). Of the 88 out of 99 students that responded, the mode was a 5 and the mean was just under 4 (see Figure 6).

In the course evaluations, six students commented specifically about SPIMbot when asked "What do you like about this course?", including the following quotes:

> "I really liked the SpimBot Tournament. That was the coolest thing I have done in a class. It makes it a lot more fun".
>
> "I liked the MP's, especially the SPIMbot Tournament and how the MP was designed to make us think of optimizations for ourselves."
>
> "... I also really liked the SPIMbot tournament"

The feedback was not uniformly positive, suggesting that there remain opportunities for improvement. One student mentioned SPIMbot in response to the question "What do you NOT like about this course?", giving the following response:

> "Spimbot. Pointless, difficult, and closed source, so hard to see exactly what was happening, so it's not entirely useful".

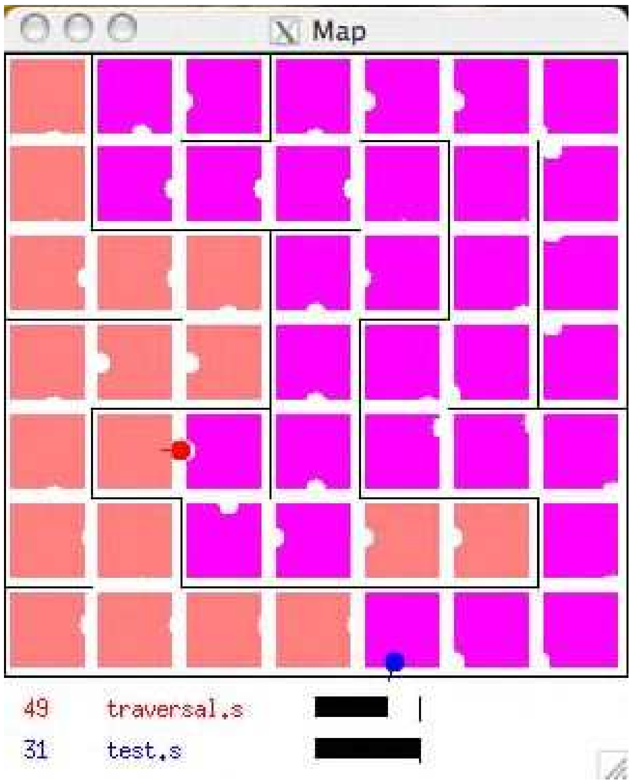


Figure 5: SPIMbot maze traversal competition. *The red squares are those that have only been visited by the red robot; the purple squares have been visited by both the red and blue robots (red + blue = purple). The black bars below the map indicate energy.*

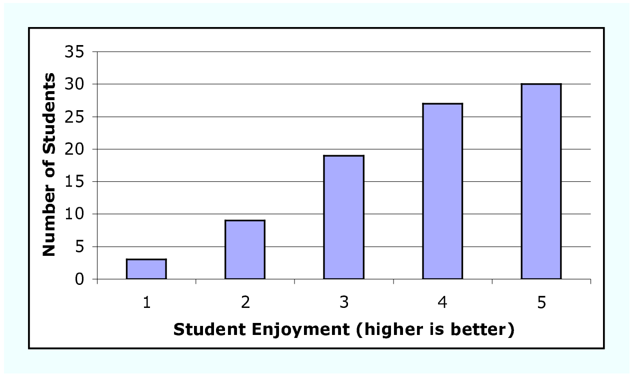


Figure 6: SPIMbot achieved high-level of student enjoyment. *Data shown for the 88 (out of 99) responding students for the Spring 2004 semester.*

We have addressed this comment in more recent contests by providing the SPIMbot source to the students when the contest is assigned. By having the source, students can run SPIMbot inside a debugger which helps them debug problems relating to interrupts, which are challenging to identify from SPIM's built-in debugger. We encourage students to inspect the code by stating that they are free to exploit any bugs they find[2]. A number of students do inspect the source; in the Spring 2005 contest, we received many comments about an unused "SPIMBOT_CHEATER" `#define` statement that was left in the code from when we were developing and testing the scenario. As the ability to efficiently read source code is a skill that comes with practice (one not emphasized early in our curriculum), organizing the contest in this way motivates some students to study the code.

Another measure of student engagement is the effort they expended. Along with their source code, students handed in a short write-up describing any noteworthy aspects (generally optimizations) of their program. Of the 30 teams, over 3/4's of the teams attempted optimizations with half completing significant optimizations:

- 15 teams (50%) described aggressive optimizations like segmenting the scan and the aforementioned pipelining,
- 5 teams (17%) described modest optimizations like greedily picking up the closest known token at any time,
- 7 teams (23%) reported attempting no optimization, and
- 3 teams (10%) reported attempting aggressive optimizations, but failed to get them working, requiring them to submit unoptimized versions.

Some of the teams that aggressively optimized their code reported trying a variety of techniques or parameterizing their code and tuning those parameters. Here are two student comments:

> "We tried many different strategies, including sorting the nodes in order of increasing distance from the spimbot, using an algorithm which heads toward the closest node to spimbot each time spimbot moves toward a new token, rescanning token locations to determine if they have been picked up, and breaking up the scans into different sizes. After trying all of these, we found that the only one which sped up the collection of tokens was breaking down the scan."

> "Our program does scans of size 25 thus giving 36 scans. We found this to be optimal because we started out with scans of size 5 doing 900 scans and found it to speed up as we approached 36. We even went down to 16 and found it slowed down as the scan sizes got bigger. Thus we have an optimal scan size."

[2]Interestingly, the first thing that many students look for is a way to write to the memory image of the other robot, which provides a nice segue to discussing virtual memory, also covered in the course.

In the Fall 2004 semester, we had students report the number of hours they contributed to the development of their SPIMbot programs. While there was some variability, most students spent 10-20 hours each, working in teams of 2-3 students.

A final metric of effort that students expended on their contest entry is the number of lines of code. While lines of code is a metric of little practical utility, it outlines the of the work and the amount of effort the students put into it. The assignments that the students handed in ranged from 186 to over 608 lines of code and data segments (not counting blank lines and those containing only comments), with most in the 200-400 lines of code range. For comparison, there were about 130 lines of code provided in solutions that most students incorporated into their designs.

In light of the age-old challenge of teaching a student body with a diversity of aptitude (*i.e.*, "How can we teach so that all of the students learn the fundamentals, while still pushing the best students?"), perhaps the SPIMbot tournament's best use is providing the best students a challenge that pushes them.

5. Future Work

As it stands, SPIMbot is derived from SPIM which is only a functional simulator: each instruction takes a single cycle. Given that our course teaches pipelining and cache fundamentals, it would be desirable to enhance SPIM (as was done for CLSPIM [6]) to model pipeline and cache stalls. In this way, the course material would be unified in this final project and students would be exposed to a more realistic optimization scenario.

6. Acknowledgments

This work was supported by NSF CAREER award 434 CCF 03-47260.

7. REFERENCES

[1] A. W. Chickering and Z. F. Gamson. Seven principles for good practive in undergraduate education. *American Association for Higher Education Bulletin*, 39:3–7, 1987.

[2] B. G. Davis. *Tools for Teaching*. Jossey-Bass, San Francisco, CA, 2001.

[3] E. M. F. III and L. Silvestri. Effects of Rewards, Competition and Outcome on Intrinsic Motivation. *Journal of Instructional Psychology*, 19:3–8, 1992.

[4] J. Larus. SPIM: A MIPS R2000/R3000 Simulator. http://www.cs.wisc.edu/~larus/spim.html.

[5] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, New York, 1980.

[6] A. Rogers and S. Rosenberg. Cycle level SPIM. Technical report, Department of Computer Science, Princeton University, Princeton, NJ, October 1993.

[7] P. Teller, M. Nieto, and S. Roach. Combining Learning Strategies and Tools in a First Course in Computer Architecture. In *Workshop on Computer Architecture Education, held in conjunction with the 30th Annual International Symposium on Computer Architecture*, June 2003.

[8] C. Zilles. SPIMbot. http://www-faculty.cs.uiuc.edu/~zilles/spimbot.

QUILT: A GUI-based Integrated Circuit Floorplanning Environment for Computer Architecture Research and Education*

Gregory J. Briggs, Edwin J. Tan, Nicholas A. Nelson
Electrical and Computer Engineering
University of Rochester
Rochester, NY 14627
{*grbriggs*, *etan*, *ninelson*}*@ece.rochester.edu*

David H. Albonesi
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853
albonesi@csl.cornell.edu

Abstract

*In this paper, we describe a graphic editing tool called QUILT (**Q**uick **U**tility for **I**ntegrated circuit **L**ayout and **T**emperature modeling). QUILT permits users to rapidly build floorplans of integrated circuits, providing both a visual aid as well as an input to the HotSpot simulator. The tool provides numerous features for estimating circuit performance, such as interconnect delay, and for generating graphical images for publications. As a graphical and easy to use tool, QUILT is well suited for both research and coursework purposes.*

1. Introduction

An essential element of computer architecture education, whether at the level of an undergraduate homework assignment or doctoral-level research, is investigating the tradeoffs among multiple design criteria, such as performance, cost, and power dissipation. The most hands-on, real world, avenue for exploring computer engineering tradeoffs is designing, testing, and fabricating different integrated circuits and comparing their characteristics. However, the prohibitively high time and monetary costs of these activities make them useful only for small circuits or long term projects. Hardware emulation via FPGAs provides more rapid turnaround time yet suffers from two major limitations. First, the density of FPGAs significantly lags that of full-custom CMOS designs, making it necessary to span several FPGAs for large, microprocessor-level emulations. Second, the internal hardware structure of FPGAs differs considerably from a full-custom design making it difficult to correlate performance and power measurements from the FPGA to that of the full-custom design.

For these reasons, the use of software simulation for exploring design tradeoffs is very popular in computer architecture research and education, permitting large system investigations to be performed very rapidly. Although not as accurate as real hardware, simulators produce results for large scale systems in a matter of minutes or hours. Thus, many tools, both proprietary and public-domain, have been developed to study the performance, power, and temperature aspects of different architectures. The reasonable accuracy and rapid turnaround times of these tools makes them highly appealing.

However, most popular simulation toolsets are text-based and command-line driven. When used for tasks such as floorplanning, such interfaces are tedious and lead to frequent and hard-to-detect errors. For instance, the input to the HotSpot temperature modeling tool [7, 12] is a text file containing a listing of x,y coordinates and sizes of the functional units. In the past, researchers modified this file using text editors and manually computed each coordinate, a tedious and error-prone approach. Thus, the prime motivation for designing the QUILT tool described in this paper was to provide a more productive means to utilize HotSpot. However, in the course of development, it became apparent that QUILT would also be useful in estimating IC transistor counts, rough floorplanning for very large scale integration (VLSI) layout, producing graphics which can be used in reports and presentations, and as a general educational tool. Research has shown that the use of graphical user interfaces (GUIs) can increase productivity and also help to reinforce concepts learned in the classroom [1, 9].

The rest of this paper discusses the QUILT tool and is organized as follows. Section 2 describes the operation of QUILT in detail. Section 3 gives a brief overview of the technical aspects of the software. Examples of how QUILT can be used in an academic environment are given in Section 4. New features that can be added onto the current version of the tool are discussed in Section 5, and conclusions are provided in Section 6.

*This research was supported in part by National Science Foundation grant CCR-0304574.

2. Detailed Description of the QUILT Tool

QUILT allows one to easily build a floorplan as an input for various simulation tools. The current version is optimized for use with HotSpot. Users can quickly adjust their designs by simply "pushing polygons", and running the HotSpot simulation again. This removes the tedium of manually computing functional unit coordinates and allows users to focus on exploring design issues. Functional units can be easily moved and resized. On-chip interconnects are also simulated in detail.

2.1. General Structure

QUILT is a standalone Java application. The main function of the tool is to generate an input text file for a simulator while viewing or editing a graphical representation of the IC floorplan in a GUI. The tool was tested with a simulator based on SimpleScalar 3.0b [3] with Wattch [2] and HotSpot [7, 12] extensions.

QUILT reads and writes HotSpot floorplan coordinate text files, and also colorizes the floorplan based on power or temperature trace files. The tool can be started from the command line and an existing floorplan file can be specified as a command line option, or it can be given a shortcut icon and associated with `.flp` files similar to most GUI programs.

In addition to a text-based coordinate file, QUILT leverages Java's capability to generate JPEG files to produce layout images that can be used in documents.

QUILT requires various parameters for the technology node of interest. For our research with QUILT, we obtained these from the 2003 ITRS Roadmap [10]. Changing to a different node is a simple matter of replacing some constants with the desired node's "Roadmap" values.

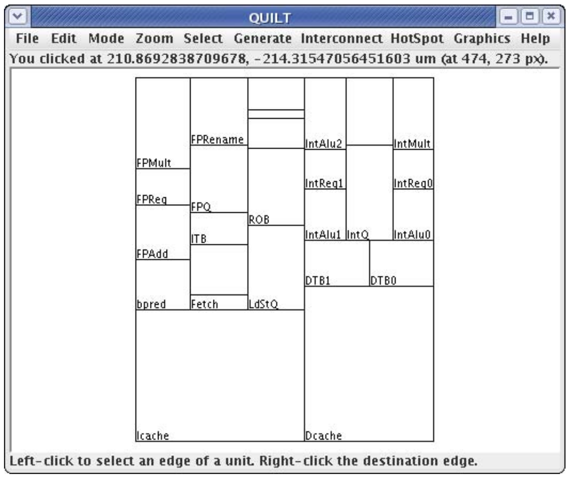
2.2. User Interface

When the tool is started, a GUI window is displayed which has the look and feel of a typical drawing editor. The drop-down menus are located on the top of the window. If an input file was specified at start up, the editing area will display a floorplan of the circuit layout. Figure 1 shows QUILT displaying a sample processor, in this case a modified version of the Alpha 21264 floorplan.

2.3. Floorplan Generation

A floorplan can be created from scratch if desired. The drop-down menus Edit, Mode, Zoom, Select and Generate are used to draw and edit a basic floorplan.

New units can be created by specifying their name and dimensions. SRAMs can be created by choosing a memory size. Level 2 cache can automatically be laid out to surround the core. The Generate menu is shown in Figure 2.



You clicked at 210.8692838709678, -214.31547056451603 um (at 474, 273 px).

Left-click to select an edge of a unit. Right-click the destination edge.

Figure 1. QUILT displaying a sample processor (modified Alpha 21264) floorplan

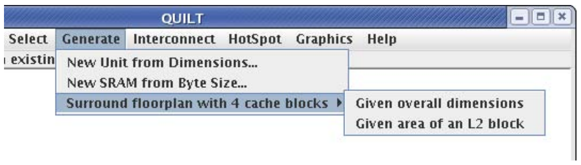

QUILT
Select Generate Interconnect HotSpot Graphics Help
existin
New Unit from Dimensions...
New SRAM from Byte Size...
Surround floorplan with 4 cache blocks
Given overall dimensions
Given area of an L2 block

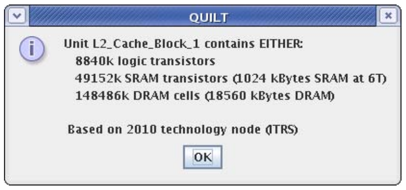
Figure 2. QUILT's Generate drop-down menu

Once units have been created, they can be resized and moved using three possible editing modes. The first mode, Move, simply allows a unit to be translated to new coordinates. The second mode, Resize (constant area), is useful in that one can adjust the dimensions of a unit, while still retaining the original area and thus the same functional capability (for example, to keep the number of bytes of an SRAM constant). Finally, the third mode allows the dimensions to be changed without constraint.

QUILT can compute the transistor count for a particular functional unit and technology node projected by the ITRS Roadmap. Figure 3 shows the pop-up window, obtained by selecting the function in the File menu, displaying this information. This operation is useful in estimating the total number of transistors used in a design.

The Zoom and Select menus make it easy to zoom in to, or select a certain part of, a unit, respectively.

Lastly, the Edit menu (Figure 4) covers typical edit operations, as well as a few extra operations that have proven useful. The Join very close edges operation is especially useful when importing HotSpot floorplans that had been made by hand. These floorplans often contain small calculation errors. Another function is Show overlapping and underlapping points which is useful in verifying that there


QUILT

Unit L2_Cache_Block_1 contains EITHER:
8840k logic transistors
49152k SRAM transistors (1024 kBytes SRAM at 6T)
148486k DRAM cells (18560 kBytes DRAM)

Based on 2010 technology node (ITRS)

Figure 3. Window displaying functional unit transistor count

are no spaces in the floorplan. A common cause of HotSpot floorplan errors are gaps between unit edges, which act as insulators during temperature simulation.

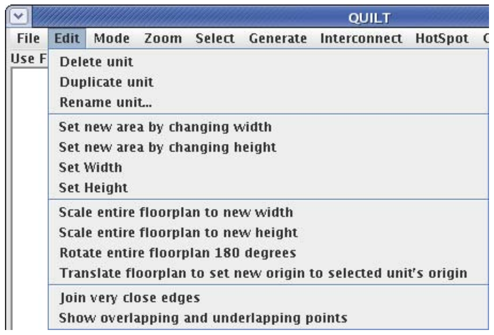
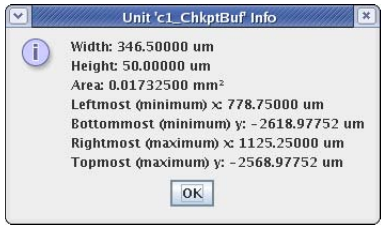

QUILT
File
Edit
Mode
Zoom
Select
Generate
Interconnect
HotSpot
Use F
Delete unit
Duplicate unit
Rename unit...
Set new area by changing width
Set new area by changing height
Set Width
Set Height
Scale entire floorplan to new width
Scale entire floorplan to new height
Rotate entire floorplan 180 degrees
Translate floorplan to set new origin to selected unit's origin
Join very close edges
Show overlapping and underlapping points

Figure 4. QUILT's Edit **drop-down menu**

To recapitulate information regarding a particular unit, an option called Show Unit Info in the File menu displays a window listing the unit's width, height, area and x,y coordinates.


Unit 'c1_ChkptBuf' Info

Width: 346.50000 um
Height: 50.00000 um
Area: 0.01732500 mm²
Leftmost (minimum) x: 778.75000 um
Bottommost (minimum) y: -2618.97752 um
Rightmost (maximum) x: 1125.25000 um
Topmost (maximum) y: -2568.97752 um

Figure 5. Details of a functional unit in a pop-up window

2.4. Interconnect

Interconnect delay is of growing importance for computer architects. QUILT models multiple types of interconnects, and can be easily extended to other approaches.

Conventional metal interconnects as well as optical interconnects are currently modeled, based on estimates

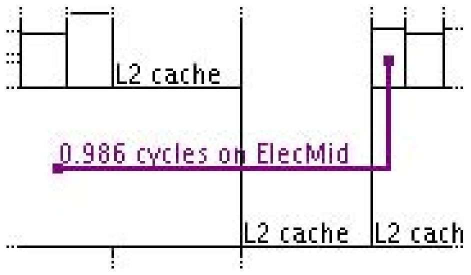


Figure 6. QUILT interconnect delay estimator

from [4]. After two endpoints have been selected for communication, the program presents a list of the estimated delays using each type of interconnect. The user can select the desired type. Electrical connections are automatically routed in a simple Manhattan style. Optical interconnects are modeled as point-to-point links. The area consumed by the interconnect is also depicted (visible for interconnects that are many bits wide). Finally, the connection delay is expressed in terms of clock cycles, for ease of comparison. This is shown in Figure 6.

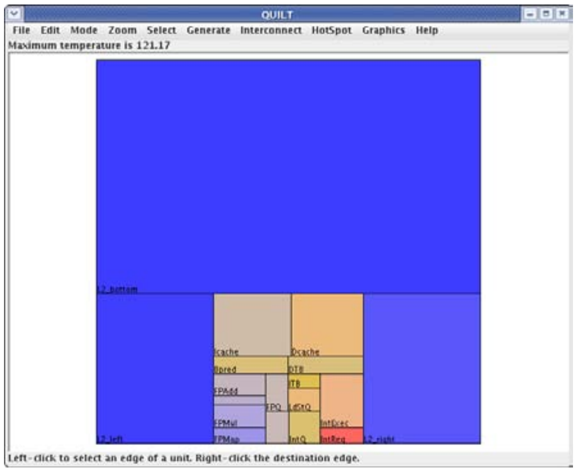
2.5. HotSpot Usage

The primary file format of QUILT is the `.flp` (floorplan) file used with HotSpot. However, one can do much more with QUILT than just save files for use with HotSpot.

A single menu function takes care of saving the floorplan file, running HotSpot, and depicting the results within the editor. The floorplan is automatically colored to indicate cool (blue) and hot (red) functional units. HotSpot supports running from a power trace file, which means that one does not have to wait for a SimpleScalar simulation to finish. Rather, the power trace file lists the power dissipation in each unit and HotSpot just needs to recompute the thermal interactions.

Once a user has compiled HotSpot, QUILT can quickly run a HotSpot simulation and immediately color the floorplan according to temperature. The user can rearrange functional units according to thermal constraints and re-simulate instantly. This can be very useful for design space exploration.

To use QUILT in this way, one must first produce a power trace file which is a listing of power consumed by the processor units. The simulator in HotSpot called `sim-template` generates this trace file as part of its output. The time it takes for results to be produced is on the order of seconds. An example using the demonstration files included with HotSpot is shown in Figure 7. The cool caches are colored with blue hues and the hot integer units are indicated with red hues.



Maximum temperature is 121.17

Left-click to select an edge of a unit. Right-click the destination edge.

Figure 7. QUILT displaying a temperature-colored floorplan

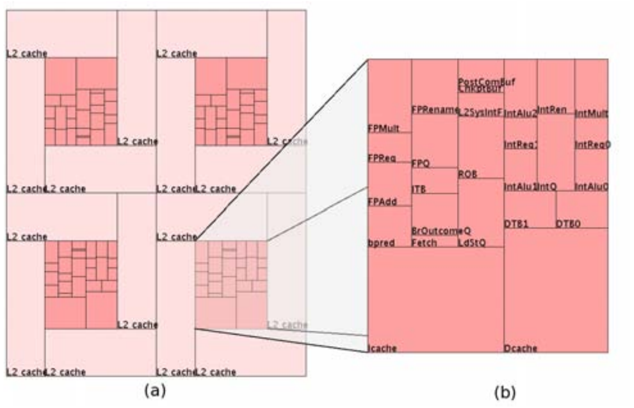


Figure 9. The "zoom effect" for a chip multiprocessor image: (a) floorplan for a proposed 4-core multiprocessing fault-tolerant processor, and (b) closeup of one core [11]

2.6. Graphic Image Generation

Graphics are frequently needed to clarify ideas and depict results for reporting purposes. In the past, producing these graphics involved manually adjusting, resizing and recoloring a large number of functional units. QUILT largely automates these tasks. Single menu commands, shown in Figure 8, permit the floorplan to be recolored, fonts resized, and labels changed.

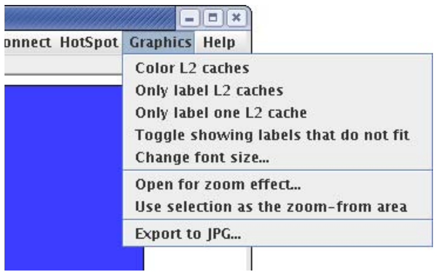


Figure 8. QUILT's Graphics **drop-down menu**

The coloring may reflect either temperature gradients or IC functionality such as caches. Additionally, for multi-core processors, there is a "zoom effect" generator. This creates a graphic containing a picture of the entire processor with "zoom" lines leading to an image of a single core, as shown in Figure 9.

3. Implementation

QUILT was developed using the Java Virtual Machine environment which makes the software portable across many computing platforms, a common need in academic settings. In its current implementation, the tool comes as a single 70kB file which is easily distributable and does not require tremendous computing resources to run.

This single file is a Java Archive (JAR) file, and in many operating systems, can be executed simply by double clicking on it. Since the JAR file is actually a compressed archive, a user who wishes to modify QUILT can uncompress it to obtain the complete source code.

QUILT takes full advantage of the Java object model. Each functional unit displayed is actually an instance of the `Unit` class, and interconnects are members of the `InterconnectLine` class. The technology node is also encapsulated in a separate class, as are many other components of the software.

Sun's `javax.swing` package was used to render the graphical interface. The actual floorplan editing area was made by extending the `JComponent` class. By selecting Sun's standard graphical interface library, QUILT's source code should be easier to understand and extend.

4. Teaching and Research using QUILT

Simulators are widely used in computer architecture education, as they permit designs to be analyzed relatively quickly and cheaply. However, text-based simulators are not intuitive and they are prone to errors that can be corrected only with careful scrutiny. Although QUILT itself is not a full simulator in the strictest sense (its main task is to provide a front-end for, and a graphical representation of, the data generated for or used by text-based simulators), it enables students to comprehend results more effectively. Studies conducted by Felder and Brent [5] using a questionnaire developed by Felder and Soloman [6] indicated that

82% and 63% of engineering students are visual and sensing learners, respectively. The researchers defined sensing as oriented towards facts and hands-on methods and visual learning pertains to information presented in pictures and diagrams.

A simple exercise using QUILT can be organized in three steps: configuration, simulation, and analysis. The first step requires the student to modify an existing floorplan of a IC chip by using the drawing tools as seen in Figures 2 and 4. An example is shown in Figure 1. If thermal simulation is to be done, the functional unit names should correspond to the power outputs listed by Wattch. The simulation step is actually not performed in QUILT but through HotSpot and/or SimpleScalar. The student can monitor and analyze data visually (Figure 7) while the SimpleScalar simulation is in progress or when it is completed.

There are many other exercises that are possible. For instance, students can compare layouts for temperature versus interconnect delay, or examine the thermal impact of adding newly proposed units, splitting units, etc.

In most computer engineering curricula, computer architecture is taught at two levels: an introductory level course targeted towards undergraduates, and a more advanced course designed for upperclass and graduate students. Due to the complexity of simulators such as SimpleScalar, Wattch and HotSpot, exercises involving their use and modification are usually carried out only in advanced courses, even though they are excellent teaching tools. A graphical based tool such as QUILT permits the instructor to introduce architectural concepts and simulation skills early in a student's education.

Using QUILT as a research tool is not much different from a classroom exercise. A further step would probably involve producing graphics such as the temperature-colored (Figure 7) or multicore floorplan (Figure 9) required as part of the documentation following the research. The authors have used QUILT to generate results for two papers [8, 11].

5. Future Work

Although the authors have used QUILT for their own work, the tool still has room for improvement. For example, when making presentation graphics, XFig and PostScript outputs would be useful. Other areas of improvement are in ease of use, modeling, and new functional unit generation.

From an educational standpoint, ease of use is important. An on-line help system could be added. Additionally, the editor should support "drag-and-drop" of unit placement, similar to other vector graphics editors.

The supported models could also be improved. Interconnect models could be more detailed and more types of interconnects could be added. Delay uncertainty based on temperature could be calculated. Also, more technology node specifications could be added to the system; an easy way to scale a floorplan to a different feature size would also be useful. The transistor count window currently shows three different numbers and a correct interpretation requires the user to decide if the functional unit is of a logic, SRAM or DRAM type. QUILT could be improved to automatically determine the unit's function type and display the appropriate transistor count.

When generating new units, it would be useful to be able to create items like queues and register files based on parameters such as number of ports and byte size. An example would be for QUILT to read SimpleScalar `.cfg` configuration files to automatically generate functional units. ALUs could be pre-defined given an integer width. Such items are impossible to produce exactly, but can be estimated based on current processor designs.

Finally, the program has been designed with modularity and ease of extension in mind. The community is invited to implement any new features they desire and to share them so that all may benefit.

6. Conclusion

In the past, computer architecture simulators tend to be text-based which makes debugging and analysis an inconvenient process. This distracts computer architects from focusing on the main task of designing and verifying new designs. Previous research on human learning and cognition has shown that visual activity enhances the pedagogical experience.

QUILT acts as an interface between raw text data and the user. It can run on a variety of computing platforms which makes it accessible to many users. Using QUILT enables users to make changes to IC layout quickly and to evaluate and analyze the results of their modifications. One of the features not seen in other tools is the ability to generate graphics for hard copies or for use in presentations and documentation.

Finally, QUILT addresses the issues of temperature and interconnect. These are two areas of growing importance for future microprocessors, and need increased emphasis in the classroom. This tool provides interactive visualizations which are effective in helping to meet that need.

7. Acknowledgments

We wish to acknowledge Joseph Toscano for his references to the visual studies performed on human learning.

References

[1] D. Bodemer et al. The active integration of information during learning with dynamic and interactive visualisations. *Learning and Instruction*, 14(3):325–341, 2004.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, Vancouver, Canada, Jun 2000.

[3] D. Burger and T. Austin. The SimpleScalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison Computer Sciences Department, Jun 1997.

[4] G. Chen et al. Electrical and optical on-chip interconnects in future microprocessors. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, Kobe, Japan, May 2005.

[5] R. M. Felder and R. Brent. Understanding Student Differences. *Journal of Engineering Education*, 94(1):57–72, Jan 2005.

[6] R. M. Felder and B. A. Soloman. Index of Learning Styles. World Wide Web, `http://www.ncsu.edu/felder-public/ILSpage.html`.

[7] W. Huang et al. Compact Thermal Modeling for Temperature-Aware Design. In *Proceedings of the 41st Design Automation Conference*, pages 878–883, San Diego, CA, Jun 2004.

[8] N. A. Nelson et al. Allevating Thermal Constraints while Maintaining Performance via Silicon-Based On-Chip Optical Interconnects. In *Workshop on Unique Chips and Systems*, pages 45–52, Austin, TX, Mar 2005.

[9] O. K. Park and R. Hopkins. Instructional conditions for using dynamic visual displays: A review. *Instructional Science*, 21:427–449, 1993.

[10] Process, Integration, Device and Structures. *The International Technology Roadmap for Semiconductors*. World Wide Web, `http://public.itrs.net/Files/2003PIDS/PIDS2003.pdf`, 2003 edition, 2003.

[11] M. W. Rashid et al. Exploiting Coarse-Grain Verification Parallelism for Power-Efficient Fault Tolerance. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Sep 2005.

[12] K. Skadron et al. Temperature-Aware Microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 2–13, San Diego, CA, Jun 2003.

The ‘Little Man Storage’ Model

Larry Brumbaugh William Yurcik

National Center for Supercomputing Applications (NCSA)
University of Illinois Urbana-Champaign
{ljbrumb,byurcik}@ncsa.uiuc.edu

Abstract

A simple but powerful storage model is described that has close correlation to generic storage systems. Extending the Little Man Computer paradigm developed by Stuart Madnick and John Donovan during the 1960s at MIT (where it was taught to all undergraduate computer science students), this paper describes a comparable development undertaken for disk and tape storage devices. A “Little Man Storage” paradigm is proposed to simplify the explanation of how storage devices function and how data is maintained by those devices.

1. Overview

For over forty years the Little Man Computer (LMC) paradigm has proved to be a simple but powerful and long-lived tool for teaching computer architecture to undergraduates in a field where a product is considered obsolete after 5 years (8 generations!). The authors of this paper have taught for many years with LMC simulators and have documented how LMC simulators can be useful teaching tools [1-4]. However, as computer architectures have evolved over time, subsystems within computers have also grown in complexity and capability such that their operation can no longer be effectively explained to undergraduates without new educational support.

In this paper we propose a new paradigm for teaching about storage systems, a core embedded subsystem coordinated with the larger computer architecture that has grown in complexity and capability to necessitate separate treatment. In fact many storage systems today have under-utilized processor capabilities such that we feel teaching storage systems may actually have an impact on future developments.

We propose a “Little Man Storage” model for teaching about storage systems consisting of elements similar to “Little Man Computer”. By using ecological design in which model elements have intuitive meaning from human experience, we believe that a Little Man Storage (LMS) model may provide benefit in courses where storage systems are studied comparable to the impact of Little Man Computer. The LMS paradigm is consistent with the SNIA Shared Storage Model [5] that was developed to help standardize storage concepts across vendor platforms. This paper provides a conceptual overview of LMS as a precursor to a simulator implementation. It is our hope for feedback that can be incorporated into near-term development. This paper is meant as a discussion of educational techniques for communicating complex concepts in a learning environment and not as a tutorial, we assume readers a basic understanding of disk storage devices and how they store and manage data.

The remainder of the paper is organized as follows: after reviewing the LMC paradigm in Section 2, the LMS model is described in Section 3. In Section 4 the relevant LMS conceptual elements identified. Section 5 compares/ contrasts LMC and LMS to highlight our contribution. In Sections 6 and 7 file storage and data management are modeled. The discussion and examples focus exclusively on disk storage. An example is given of a typical storage processing operation that illustrates the individual steps within the operation and examples are also given that show the changes that occur in the storage device itself. Although not discussed in this paper, a small subset of this material can be used to illustrate tape/cartridge processing.

2. The Little Man Computer Paradigm

The LMC paradigm has stood the test of time as a conceptual device for helping students understand the processing that takes place inside a computer. One of its greatest strengths is its simplicity. The paradigm consists of a walled mailroom, 100 mailboxes numbered 00 through 99, a calculator, a two digit location counter, an input basket, and an output basket. Each mailbox is designed to hold a single slip of paper upon which is written a three digit decimal number. Note that each mailbox has a unique address and the contents of each mailbox are separate from its address. The calculator can be used for input/output operations, temporarily store numbers, and to add and subtract numbers. The two-digit location counter is used to increment the count each time the Little Man executes an instruction. The location counter has a reset located outside of the mailroom. Finally there is the "Little Man" himself, depicted as a cartoon character, who performs tasks within the walled mailroom. Figure 1 illustrates the major components of the LMC paradigm. Other than the reset switch for the location counter, the only communication a user has with the Little Man is via slips of paper with three digit numbers put into the input basket or retrieved from the output basket.

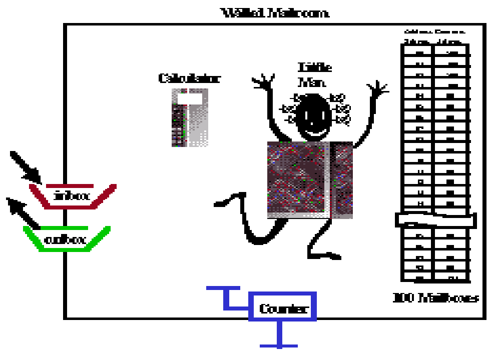


Figure 1. Little Man Computer and the Walled Mailroom

The authors have written several papers [1-4] describing use of a LMC simulator to enhance the quality of computer science courses, specifically those that emphasize architecture, hardware/software, and operating systems concepts. The two simulators developed by the authors are part of a larger worldwide effort to construct LMC simulators some of which are described in [3]. We feel these widespread developments validate both the utility and continuous interest in the LMC paradigm.

3. The LMS Model

We intend to leverage the LMC paradigm with corresponding conceptual analogies. In particular, the basic philosophy utilized in the LMC model is to minimize the functional details and physical structure while still allowing the important conceptual features to be clearly illustrated. The LMS model described here would have been valid with the disks of 30 years ago. However, more importantly it provides insight into modern storage systems. Furthermore, this paper describes a model, not a working simulation, but all the moveable pieces for the working simulation are presented.

Recall that a disk storage device contains several moveable components including: a) the revolving platters where data are stored, b) an access arm that moves to the designated location for the data and c) a mechanism for copying data between the buffers and the hard drive during input and output operations. Little Man Storage itself, again depicted as a cartoon character, performs all three of these functions.

4. LMS Hardware

The LMS disk device consists of two platters where data can be stored on both sides of a platter. Both the top and bottom surfaces of each platter surface contain three concentric tracks. Hence, the storage device consists of three cylinders. Each track consists of eight areas and all areas store exactly 512 bytes of data. Table 1 specifies the numbering scheme used to identify actual locations on the device. Figure 2 shows both sides of a platter.

Table 1. Basic Hardware Components of LMS

Storage Device Components	ID Numbering Scheme for the Component
3 cylinders	0, 1, 2 (independent of platter surface)
4 tracks per cylinders	0, 1, 2, 3 (0/1 1st platter & 2/3 2nd platter)
8 areas per track	0, 1, 2, 3, 4, 5, 6, 7 (same data each area)

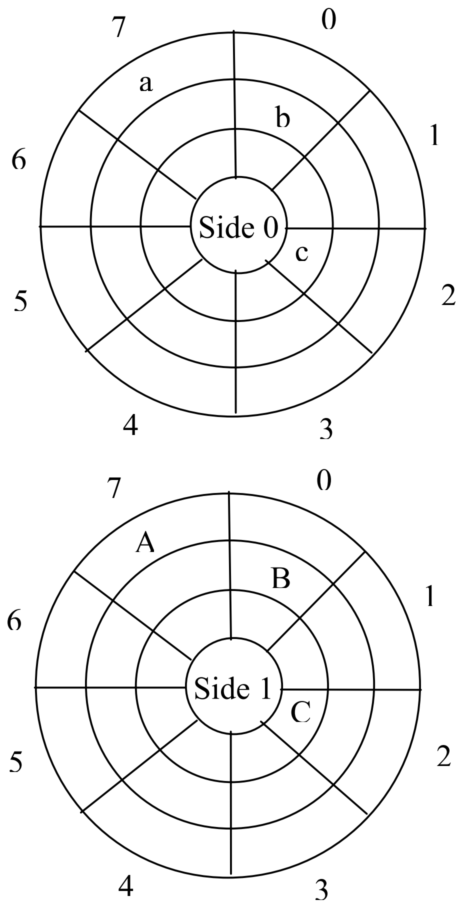


Figure 2. Both Sides of a Disk Platter

Areas can be referenced with values from 000 to 237. Address *xyz* identifies the location of the cylinder, platter, and area respectively. The small size of the storage device allows decimal numbers to be used for all three values, which simplifies addressing. Total disk capacity is 48K (=3 cylinders * 4 tracks/cylinder * 8 areas/ track * .5K bytes/area. Figure 2 shows one of the two platters in the storage device. The three area locations denoted by a, b, and c in Figure 2 have addresses of 007, 100 and 202 respectively. Area locations A, B and C have addresses of 017, 110 and 212 respectively. An alternative approach that was briefly considered that numbered the areas from 00 to 95.

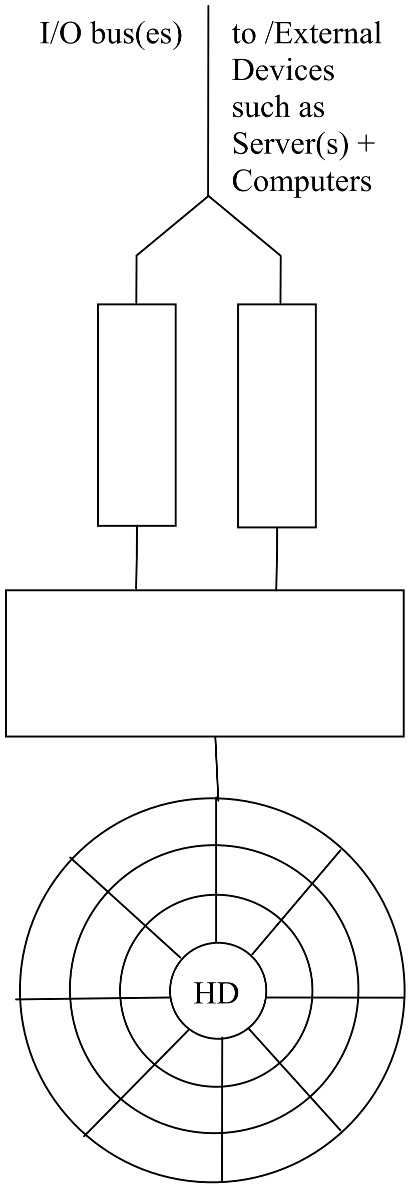


Figure 3. Physical Components that comprise an LMS Storage Device

The LMS model consists of the physical components shown in Figure 3. The disk controller is 'Little Man' (cartoon character) who provides the intelligence for disk operation and can perform a limited number of simple functions. In particular, LMS decodes and executes the commands sent to it from the attached server/computer. In implementing the commands, LMS uses one of its arms to read-data-from and write-data-to the hard drive (HD). The HD consists of the platters where data is actually stored. Communication paths called I/O buses connect the storage device to the source/destination of its data. Buffers are intermediate storage areas (pieces of paper) where data is placed both prior to copying it to storage and after retrieving it from storage and before sending it to the external device. There is one buffer (piece of paper) for data going in each direction.

In adhering to the LMC simplification principle, the disk contains no cache. Likewise, there are no auxiliary or reserved areas/tracks that can be used to replace parts of the disk that become defective. If part of an LMS device becomes inoperable, there is no way to designate processing options. No timing considerations are provided for any of the electromechanical components of the devices. Little Man Storage performs all the physical processing associated with the device. This includes using one arm to rotate the platters in the HD, using the other arm to move over a specific cylinder and then with the same arm copying the data to/from the HD.

5. Comparing Little Man Computer and Little Man Storage

Table 2 provides a comparison of the environments provided by the two paradigms and the types of physical acts that the Little Man must perform in each of them.

Table 2. Comparing LMC and LMS Characteristics

Environment/Physical Act Compared	Little Man Computer (LMC)	Little Man Storage (LMS)
historic relevance of paradigm	from 1960's to present	from 1970's to present
type of hardware device described	computer	disk storage device
actual hardware location of Little Man intelligence	CPU control unit	storage controller
locations where data is stored	100 mailboxes (00-99)	96 disk areas (000-237)
methods for performing I/O operations	read/write slips of paper	read/write disk areas
programmable device?	yes	no

6. File Storage and Data Management

The LMS storage device consists of 96 areas where 94 areas are used to store data and 2 areas are reserved to help manage the other 94. Area 000 contains a LIST of all files stored on the device. This is the only (the root) LIST on the disk. It is of fixed size (one area) and cannot be expanded. Table 3 shows the values stored in the LIST for several files. The location of the initial data in the file is specified in the Area Start Location as a (cylinder, platter, area) location. For simplicity, there are no attributes that can be assigned to a file. When a file is created, LMS adds a new row in the LIST. A new row is always added following the last or bottommost current LIST entry. If a file is deleted, its line in the LIST is erased. This is denoted as <blank> in Table 3.

Table 3. LIST Structure for the Disk

File Name	Size (Bytes)	Area Start	Creation Date
ALPHA.doc	10	006	06/06/2005
X.Y.Z	5000	128	09/18/1997
<blank>	-	-	-
NextFile1234.txt	0	225	12/25/2002
**************	-	-	-

Area 001 is used to manage the data areas that the device contains. Each of the 94 data areas either holds data associated with a file or is a free (unused) area. New files and additions to existing files obtain their storage from the free areas. It is the job of LMS to utilize this information in area 001 to retrieve and store files. LMS must also modify this information when necessary.

Initially, when the disk is first formatted, LMS marks areas 002 through 237 as free. This information is kept in a Free-Area-List. Whenever a file is created, one or more of the free areas are assigned to hold its data. When a file is deleted, the areas where its data were stored are returned to the Free-Area-List. Area 001 holds the Area Utilization List (AUL), where LMS stores information about the data areas. There are 96 entries in the AUL. The first two are used to manage the Free-Area-List and are described in the next section. The others entries are either used to identify the storage areas assigned to individual files or are a part of the Free-Area-List. Table 4 shows the initial portion of an AUL after 2 files have been written to the storage device. One file occupies 4 areas (002, 003, 005 and 006) while the second file occupies a single area (004). A value of 999 identifies the final area in a file. Note that areas 007 and 008 are either part of the same file or both are free areas. Free areas are shown in italics. LMS itself does all of this reading and writing of information.

Table 4. Contents of Area 001 Showing Storage Allocation after Two Files are Written

Area Number	Next Area Location in File	
000	007	(first free area) *
001	00N	(last free area) *
002	003	(file continuation)
003	005	(file continuation)
004	999	(end of file)
005	006	(file continuation)
006	999	(end of file)
007	008	
. . .	. . .	
237	999	

Table 3 shows that the LIST entry for a file identifies only the first area assigned to it. The rest of the file location information is stored in the AUL. The AUL identifies the areas that are linked together to provide storage for the file. The final area contains a Next Area Location value of 999, meaning this is the last area associated with the file. Storage for a file need not be in contiguous areas. The areas that are not assigned to any file are tied together in the Free-Area-List. The areas at the beginning of this list are used to satisfy subsequent requests for storage. The Table 4 structure is actually an oversimplification used to clarify processing details. In reality, the AUL only needs to contain the rightmost column of values since LMS can determine the Area Number from its physical position in the list (by counting from the beginning of the list).

7. Additional Storage Model Parameters

LMS must remember three important values. It uses the first value to find an initial free area for new files and additions to existing files. This value is stored as the very first entry in the AUL (see Table 4). When additional storage is needed, LMS looks in this location and begins writing data to the corresponding area it identifies. Additional free areas can then be determined using the Free-Area-List. Once the last free area needed for the current processing operation is determined, its Next Area Location (the next free area) becomes the new first value in the AUL. Similarly, the second entry in the AUL identifies the final area in the Free-Area-List. When a file is deleted, its areas are added to the Free-Area-List following the area identified in the second AUL entry. The final area added to the list becomes the new value in location 2 of the AUL.

The third important value is the final entry in the LIST, which is identified by following it with a 'fake' file name entry of '******************'. The LIST is a white board where LMS writes entries for new files at the bottom of the board and erases entries for deleted files. Once the bottom of the board is reached, the LIST is considered full and must be 'reorganized'. If there are unused erased rows on the board, rows on the bottom are copied to the currently erased rows and then erased from the bottom of the board. Following the LMC principle of simplicity, the LMS model places restrictions on the LIST structure and on the number of files that can be stored. With some effort this limit can be raised and subdirectories can also be used. Since this clearly will result in a more complexity, it is not discussed here.

Whenever a file is created, it is assigned one initial area. If no data are written to the file, LMS writes ***End-of-File*** at the beginning of the area. An area is never split between two distinct files. Hence, every file requires at least one area of storage and the maximum number of files is 94. An alternative approach that was strongly considered assigns the Start Location entry in the LIST for an empty file to a special value such as 999.

8. Storage Processing Operations

In the same manner that the CPU of a computer executes instructions, a storage device controller such as Little Man Storage is capable of executing a pre-defined group of commands that create, delete, store, retrieve and process data. Although some storage devices support a wider range of operations, we limit LMS to five commands as shown in Table 5. LMS processes complete files and individual records must be identified in the application programs (since storage devices are unaware of logical records). Each buffer can hold one area of data. A physical record consists of all the data in an area. LMS determines the actual location of a physical record that it needs by combining information from the command itself, the LIST, and the AUL. Each command is composed of steps in the same way that CPU instructions are composed of steps. EXAMPLE 1 illustrates the steps performed as part of a Read File command.

Table 5. Basic I/O Commands Supported by LMS

Command	OpCode	Processing Performed by Command
Create File	00	Write an entry in the LIST, including create date, etc. Initialize one Free-Area-List area to ***End-of-File***.
Delete File	01	Erase the file entry from the LIST. Return all AUL entries associated with the file to the Free-Area-List.
Read File	02	Begin in the LIST and then go through the corresponding AUL entries. With the alternative approach noted above, can also start in the AUL table.
Write File	03	Add data starting with the first area on the Free-Area-List. Write ***End-of-File*** after the last record is written.
Append File	04	Follow the AUL entries for the file to the one containing 999. Add new records in a new area and replace 999 with new area number.

All commands have the same basic syntax |op-code|filename|optional data|. In the case of Write and Append commands, the data to be written immediately follows the command code and file name. Op-codes are 1 byte in length, while file names are 20 bytes and can contain any printable characters. For example, |3|MY-NEW-INFO |*****| is a command to write 5 asterisks to a file called MY-NEW_INFO.

EXAMPLE 1: A paper is placed in the input buffer that says to get the data in the ALPHA.doc file. LMS looks at the command in the buffer and reads it, noting the command code (02) and the file name. LMS looks in the LIST and sees that initial data in ALPHA.doc begins in area 002. It rotates the disk until that area can be accessed. It copies the data from area 006 to the output buffer. LMS then looks in the AUL and notes the entry for area 006 identifies additional ALPHA.doc data in area 013. It uses one arm to move the disk to this location and the other arm to copy the data from 013 to the output buffer. This processing continues for every area where ALPHA.doc data is stored. When an AUL entry of 999 is found, the Read File operation is complete.

9. Detailed Processing Examples

Two examples are now given to illustrate all of the LMS components discussed to this point. Throughout all of these examples an unrealistic assumption is made that every operation is performed successfully. There is no way to recover from an invalid or incorrect operation.

EXAMPLE 2: It is assumed that the HD is formatted and all 94 data areas are free. File AA is created and several small records are written to it. File BB is created and enough records are written to it to fill three areas. Several additional records are then added to AA, requiring a new area to be allocated using an Append File command. A third file GG is created, but no records are written to it. Finally, file DD is created and three areas have data written to them. Figure 4 shows the relevant areas following the processing. The first two areas contain the LIST and the AUL.

	0	1	2	3	4	5	6	7
00	-	-	AA	BB	BB	BB	AA	GG
01	DD	DD	DD	-	-	-	-	-
02	-	-	-	-	-	-	-	-

Figure 4. Disk Status Following the I/O Operations in EXAMPLE 2

EXAMPLE 3: This example begins immediately after the processing in EXAMPLE 2 has completed. File AA is deleted. Two new files called SS and RR are created and one byte of data is written to each file. Additional records are then written to SS. Figure 5 shows the relevant areas following the processing.

	0	1	2	3	4	5	6	7
00	-	-	-	BB	BB	BB	-	GG
01	DD	DD	DD	SS	RR	SS	-	-
02	-	-	-	-	-	-	-	-

Figure 5. Disk Status Following the I/O Operations in EXAMPLE 3

10. Summary

We have introduced a new Little Man Storage model for teaching about computer storage systems. While this paper focuses primarily on conveying disk storage concepts, work is underway for developing a Little Man Storage software simulator that extends the storage concepts demonstrated beyond disks. Results from the educational use of this model will also provide feedback on the effectiveness of this model in targeted learning environments.

11. References

[1] W. Yurcik and L. Brumbaugh, “Using LMC Simulator Assembly Language to Illustrate Major Programming Concepts,” *Info. Systems Education Conf. (ISECON),* 2001.

[2] W. Yurcik and L. Brumbaugh, “A Web-Based Little Man Computer Simulator,” *32nd Technical Symposium of Computer Science Education (SIGCSE),* pp. 204-208, 2001.

[3] W. Yurcik and H. Osborne, “A Crowd of Little Man Computers: Visual Computer Simulator Teaching Tools,” *Winter Simulation Conference (WSC),* 2001.

[4] W. Yurcik, J. Vila, and L. Brumbaugh, "An Interactive Web-Based Simulation of a General Computer Architecture," *IEEE Intl. Conf. on Engineering & Computer Education (ICECE*), 2000.

[5] *SNIA Shared Storage Model White Paper.* <http://www.snia.org/tech_activities/shared_storage_model/ SNIA-SSM-text-2003-04-13.pdf>

An Emulated Computer with Assembler for Teaching Undergraduate Computer Architecture

Timothy Daryl Stanley, PhD
Brigham Young University Hawaii, #1854
55-220 Kulanui Street
Laie, Hawaii 96762-1294
(808) 293-3388
stanleyt@byuh.edu

Mu Wang
Brigham Young University Hawaii, #1854
55-220 Kulanui Street
Laie, Hawaii 96762-1294
mw024@byuh.edu

Abstract

An eight-bit computer has been designed using an open source logic emulation package called "Multimedia Logic" from www.softronix.com. The intent of the project was to make clear to computer science students how the data path and control lines work to provide computer functionality.

This computer is an excellent teaching aid because:

1. All registers, ALU outputs, control lines, and memory outputs are instrumented.
2. Instructions can be executed with a single step switch or run with a clock.
3. The architecture is quite simple, with separate memory devices for data and instructions.
4. It is supported with an assembler patterned after the MIPS assembler used with the SPIM simulator.
5. An ASCII output display is available.

The instruction set designed for this computer includes: Add from memory, Add immediate, Load from memory to the input register, save from the output register to memory, jump to the address given by the immediate, jump to the address given by the immediate if the last add produced a zero result, and halt.

The design includes an instruction format of three bits of operation code followed by five bits of immediate.

Using this design as a launching point, students have been encouraged to design their own computers. Some excellent designs have been submitted. These include an elaborate multi-cycle 16-bit design, and many application specific designs.

This paper provides details of this computer design, assembler and example programs as well as descriptions of designs submitted by students.

Categories and Subject Descriptors

B.6 Hardware / Logic Design / Simulation.

C.1.1 Computer Systems Organization / Computer Architectures

General Terms

Design, Human Factors, Theory

Keywords

Logic Simulation, Computer Design, Binary Visualization, Multimedia Logic

1. Introduction

The concepts of computer architecture are some times very difficult for beginning computer science students to visualize because the action is all happening at the electron level in microscopic circuits. By building on knowledge from other courses students may be able to visualize what is happening in circuits, but many layers of abstraction are involved. For example, if one builds a computer with TTL circuits, there is a level of abstraction in the relation ship between circuit pin outs and logic elements. There is also a complex chain of detail between circuits that is visible only with logic probes or additional expensive instrumentations. Also when a student has spent the time to understand and master the breadboard circuit the semester is over, the circuit is disassembled and used for the next class.

The emulated logic approach the authors have developed overcomes these limitations in understanding the details of computer architecture. The circuits are designed by "wiring" up logic elements with all data inputs coming in on the left, control signals coming in from the bottom, and outputs exiting from the right. The high level devices like memory circuits and ALU's look like the devices in schematic diagrams, making these devices easier to visualize. By designing simple circuits the operation of the individual components can be understood. At the completion of the class the students can take the design with them.

While the focus of this paper is an emulated computer for teaching architecture, a series of introductory circuits used to develop an understanding of the components that make up a digital computer are also provided. Many of the concepts of digital logic are difficult to grasp without practical experience. Some use prototyping boards with small scale digital circuits to design and build examples of digital devices [1]. Others use a hardware design language, like Verilog, to illustrate and teach digital logic concepts [5]. One school even uses students actors to emulate instruction flow in a computer [6].

The 8-bit computer will be thoroughly documented starting in section three.

2. Component Learning Projects and Outcomes

A number of projects built and demonstrated by students will be given in this section. We will start with simple projects and advance to more complex designs. Each design will be demonstrated with the presentation of this paper at the conference.

2.1 Calculator with Binary and Hexadecimal Outputs

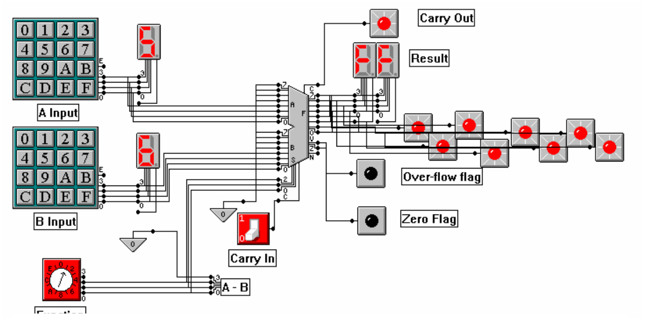


Figure 1. Calculator with Binary and Hexadecimal Outputs

The first project, illustrated in fig 1, is a calculator that takes two four-bit inputs, from hexadecimal keypads, and provides an output in both binary and hexadecimal, based on a function selected. The function is selected with the selector switch. The functions available in the ALU are: addition, subtraction, multiplication, division, equal, less than, shift right and shift left.

This is a nice project to start with as it builds on the ALU device example that is provided with Multimedia Logic.

The learning outcomes of this project are: familiarity with the ALU, comparing hexadecimal and binary, exploring properties of binary numbers under operations like the 5-6 operation shown in figure 1 to see the two's complement binary notation of a negative one.

2.2 Scanned Memory to Output Display

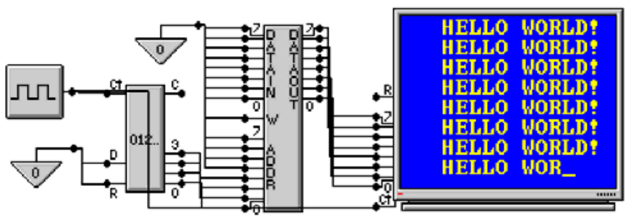


Figure 2. Scanned Memory to Output Display

This project, shown figure 2, connects the output of a memory device to an ANSII display device. Then by sequentially scanning the memory addresses with a counter connected to a clock, the content of the memory is sent to the display. In this case the content of the memory is " HELLO WORLD! ". For this project, only the first sixteen locations in memory are used, however, with an 8-bit counter, 256 locations could be used.

In Multimedia Logic the memory contents can be read from a "text" file or written to during the simulation. In this case the memory contents are loaded from a file and the memory is treated as a read-only memory (ROM).

Learning outcomes from this project are: an understanding of the relationship between memory address lines and data output lines, understanding counters, and clock oscillators, and synchronous data transfer from memory to display.

2.3 Programmable Calculator

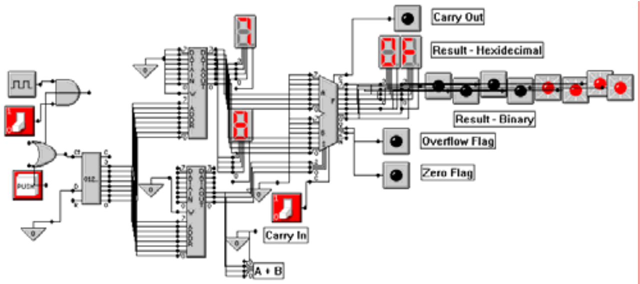


Figure 3. Programmable Calculator

This project, shown in figure 3, is a combination of the first two, using scanned memory to provide functions and data to an ALU. This project begins the comparison to a real computer, with the upper memory serving as data memory, the lower memory which provides functions to the ALU as a program memory, and the counter as a program counter.

The learning outcomes of this design are observation of the different things that a series of binary lines can be, from instructions to data to addresses, to clock pulses. This is where we also learn about data paths and control paths.

2.4 Four-Bit Adder

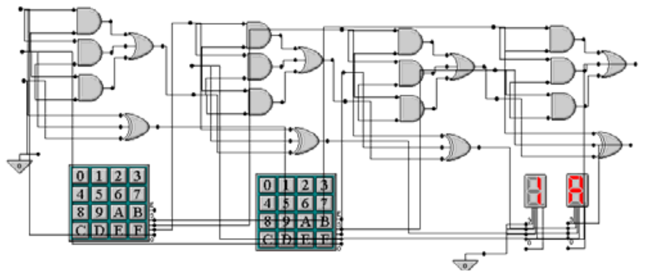


Figure 4. Four-bit Adder

These next two projects are designed to understand the inner workings of an ALU. The first, shown in figure 4, is a ripple carry binary adder. Two four-bit values are provided on the hexadecimal key pads and the results of the addition are displayed on the seven segment displays. By inverting the B inputs and making the C input for the first stage one the adder can be converted to a subtraction unit, illustrating the algorithm for converting a binary number to its two's complement negative.

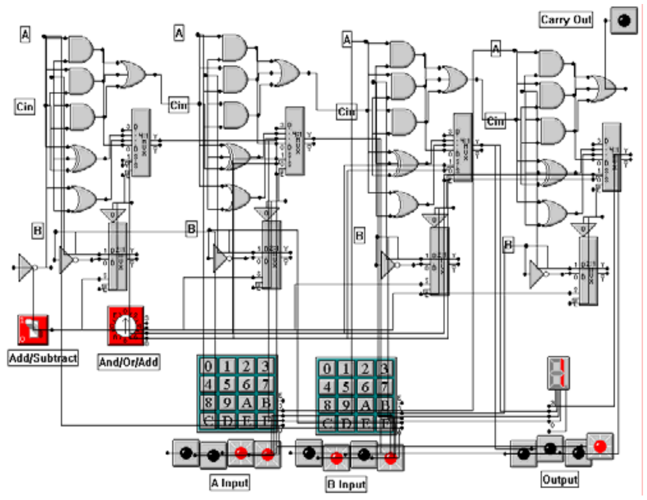


Figure 5. Four-bit ALU

The most important learning outcome of this design is an appreciation for how logic circuits can perform the kinds of operations we see computers perform.

2.5 Four-Bit ALU

This project, shown in figure 5, illustrates the complexity in the design of an ALU. This ALU, designed after the one-bit ALU from Patterson (Figure 6), can And, Or, Add, and Subtract. It is very useful for illustrating the bitwise operations of And and Or. For example the output illustrated above is the bit wise And of 3 and 5.

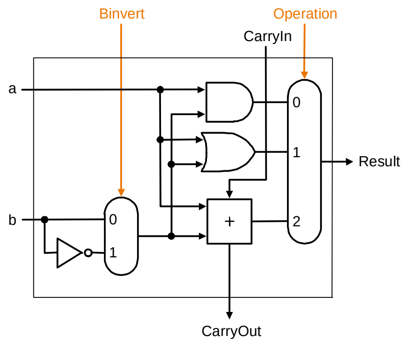


Figure 6. One-bit ALU from Patterson [4]

Learning outcomes of this project include an appreciation of how multiplexers make possible the control path in a computer—and again, an appreciation of how gates can be combined to produce computer functions.

3. An Emulated Computer for Teaching Computer Architecture

Providing a computer that is very well instrumented, visible on one page and easy to demonstrate, was the main goal of this design effort. In my computer architecture classes I ask my students to design an emulated computer. This design was one I produced to illustrate what I wanted from my students. I suggested they start with an instruction set and register design and build a computer from this foundation. For this eight-bit computer, an instruction format of three bits of operation code and five bits of immediate was chosen. This instruction format provides for eight instructions. These with mnemonics are:

1. adi - Add the immediate value to the input register and place in the output register,
2. adm - Add memory location addressed by the immediate to the input register and place in the output register,
3. lmi - Load the contents of the memory location addressed by the immediate value to the input register,
4. som - Save the output register to the memory location addressed by the immediate value,
5. ji - Jump to the location given in the immediate,

6. jzi - Jump to the location given by the immediate if the result of the last addition was zero,
7. om - Output the data from the memory location addressed by the immediate to the output display device,
8. hlt - Halt operation.

The physical architecture was to use two separate memories, to hold the data and program. This parallels the MIPS emulator PC SPIM which has a ".data" segment of memory holding constant data and a ".text" segment that contains the machine instructions. This construction simplifies the data path of the computer, but limits the capability to do recursion. The design includes an input register and an output register.

This design is a complete eight-bit, single cycle, stored program computer. The data paths are connected at the start of the clock cycle at then at the clock transition registers and memory are writing enabled. This enables demonstration of the inputs to commands being set up and then the operation being executed.

One non-physical device available in the logic emulator used is a binary controlled text display. This device can be seen just below the vertical column of control line indicators. This display shows one of sixteen lines of text, depending on the binary inputs to the device. In this case the device is used to show the operation being set up in the computer.

The memory devices can be used as read-only devices reading content from an underlying file, or they can be initialized with a file and altered dynamically during program execution. For registers memory devices with all address lines grounded are used.

One limitation of this emulation package is the absence of a 2-by-8 multiplexer. As a result the multiplexers are assembled by stacking a series of 1-by-2 multiplexers

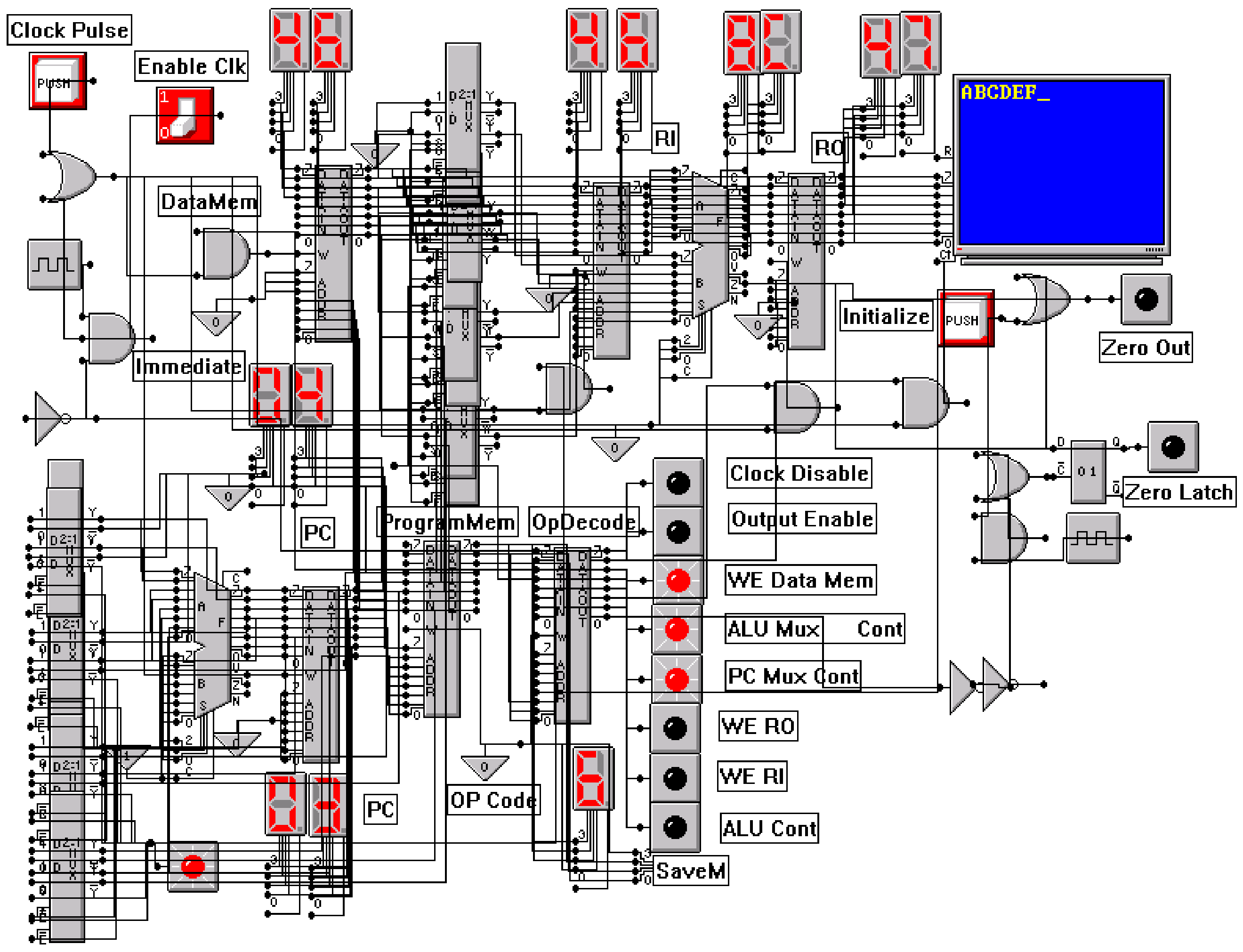


Figure 7. Eight-bit teaching computer design implemented in multimedia logic

partially overlapping one another. Since this emulator is published with its source code, I have built versions of this computer using a version of the software with a modified ALU that has an A out and a B out instruction. Then the multiplexer stacks can be replaced with ALUs. I have not included this design because it uses ALUs in a non-standard way and because the design could not be used with the emulator down loaded from the emulator's web site.

4. Sample Programs for the Computer

With this set of instructions a number of demonstration programs have been written. The file underlying the memory has a format that includes two hexadecimal digits that are the memory content for each line. The memory ignores any additional information on the line. So following the operation code or data a comment can be given. This allows instruction documentation information to be included with each line. These include a program to send a string in data memory to the output display device, a program with an up counting loop and a down counting loop to display the letters of the alphabet and halt at Z, and a program to display various size boxes on the display.

The design includes two ALUs, one incrementing the program counter and one performing the additions. Memory devices include a data memory, program memory, input register, output register, program counter, and an operation decode ROM. The nicest feature of implementing a computer design this way, rather that in a breadboard, is the much greater instrumentation of registers, and data lines. One can see each value as the computer steps through the program.

Three sample programs are included in this section.

4.1 Sample Program 1, ABCs.

This first program was designed to be simple but use all eight of the operations of this computer. It consists of a loop that counts up one memory location from ASCII A to ASCII Z, and counts down in another location to halt the computer after 26 letters. To implement this program the memory contents in the following tables are place into the data and program memories. Note that in these tables that the two hex digits in each line are the actual output from the memory device and the rest of the line is a comment. Data and program memory files are shown if tables 1 and 2 below. The output is shown with figure 7 above.

00	zero (not used)
19	Hex for character count in alphabet
ff	Twos complement negative one
fe	Twos complement negative two (not used)
41	ASCII code for letter A
41	(not used)
41	"
00	"
00	"
00	"

Table 1. Data memory content for program 1

c4	Output from memory location 04
44	Load input register from memory location 4
01	Add I (01) to input register
64	Save output register in memory location 04
c4	Output from memory location 04
41	load input register from memory location 01
22	Add from memory location 02
61	Save output register to memory location 02
aa	Jump if last calculation result was zero to 0a
80	Jump to memory location 00(+1)
e0	Halt execution

Table 2. Program memory content for program 1

4.2 Sample Program 2, Hello World.

The second program was to be the simplest possible, like the "Hello World" used to introduce all programming languages. For this program a string in the data memory is sent character by character to the output screen and then the program loops back to the beginning. The lack of instructions to update program memory based on calculations prevents the use of simple iteration to implement this program. The data and program memory files are show in tables 3 and 4 and the output is shown in figure 8.

20	Space
20	Space
48	H
45	E
4c	L
4c	L
4f	O
20	Space
57	W
4f	O
52	R
4c	L
44	D
21	!
0d	New Line

Table 3. Data memory content for program 2

c0	Output from memory location 00
c1	Output from memory location 01
c2	Output from memory location 02
c3	Output from memory location 03
c4	Output from memory location 04
c5	Output from memory location 05
c6	Output from memory location 06
c7	Output from memory location 07
c8	Output from memory location 08
c9	Output from memory location 09
ca	Output from memory location 0A
cb	Output from memory location 0B
cc	Output from memory location 0C
cd	Output from memory location 0D
ce	Output from memory location 0e
80	Jump to Zero (+1)

Table 4. Program memory content for program 2

4.3 Sample Program 3, Triangle.

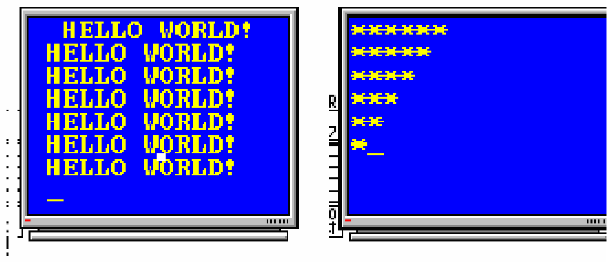
This program was written to test the assembler discussed in the next section. It uses two nested loops to print a triangle on the output screen. Data and program memory files are given below and the output is shown in figure 8.

06	column (size of triangle)
03	row (not used in program)
00	column step
00	row step
ff	negative one (allows decrementing)
00	zero
2a	symbol "*"
0d	new line

Table 5. Data memory content for program 3

45		Load input register from memory location 5 (zero)
20		Add memory location 0 (column) to input register
62		Save result in memory location 2 (column step)
63		Save result in memory location 3 (row step)
c6	lp1:	Output from memory location 6 (symbol "*")
44		Load input register from memory location 4 (neg one)
22		Add from memory location 2 (column step)
62		Save result in memory location 2 (colmn step)
a9		Jump on zero to lp2:
83		Jump to lp1:
23	lp2:	Add from memory location 3 (row step)
63		Save result in memory location 3 (row step)
b6		Jump on zero to :hlt
c7		Output from memory location 7 (new line)
45		Load input register from memory location 5 (zero)
20		Add memory location 0 (column) to input register
44		Load input register from memory location 4 (neg one)
20		Add memory location 0 (column) to input register
60		Save result in memory location 0 (column)
45		Load input register from memory location 4 (neg one)
20		Add memory location 0 (column) to input register
62		Save result in memory location 2 (column step)
83		Jump to lp1:
eo	hlt:	Halt

Table 6. Program memory content for program 3



```
 HELLO WORLD!
HELLO WORLD!
HELLO WORLD!
HELLO WORLD!
HELLO WORLD!
HELLO WORLD!
HELLO WORLD!
_
```

```
******
*****
****
***
**
*_
```

Figure 8. Output screens for programs 2 and 3

5. The Assembler in PERL

To add to the utility of this computer, an assembler was designed in the PERL language. As the assembler runs it generates text files that can be loaded into the data and program memory in the simulated computer. The assembler allows symbolic linking between the data and the program and allows symbolic naming of jump locations. The assembler was patterned after the assembler imbedded in the MIPS emulator PC SPIM.

This assembler starts execution by asking the user for data and program memory file names. Then the user sees the screen from the table below which gives a review of the instruction set of this computer and then provides a sample input file to show the syntax that must be used. When the line with the stop command is given, the program closes the files and returns.

6. Student Computer Designs

Using this computer and its design process as an example, computer architecture students have been required to design a computer of their own from the registers and instruction set to layout and implementation with example programs. The first design from a student team was an elaborate 16-bit design that used eight cycles to decode and execute each instruction with the idea of demonstrating a pipeline implementation. This computer consisted of eight pages of logic. While this computer represents a great deal of effort on the part of the students involved, it is not as useful for demonstration because parts of the display are on separate pages and can not be viewed simultaneously.

Some students had difficulty designing a computer starting with operations and layout. For these students the approach that seemed to work best was to start with an application they would like to demonstrate on their computer and then design a computer to meet that requirement. Some examples of the application-motivated designs were for an electronic door lock and a "Whack a Mole" game.

```
**********      Operation Code      **********
*******************************************
*********    adi- Add Immediate          *********
*********    adm- Add Memory             *********
*********    lmi- Load Mem -> Ri         *********
*********    som- Save Ro-> Mem          *********
*********    ji- Jump Immediate          *********
*********    jzi- J on z Im              *********
*********    om- Out Mem Im              *********
*********    Hlt- Halt                   *********
*******************************************


************ Sample  Input *************
*******************************************
.data
(PLease Input Data for DataMem.)
numlet:26d
negone:ffh
acode:41h
.text
(PLease Input Data for ProgramMem.)
omi acode
start:lmi acode
adi 01d
som acode
om acode
lmi negone
adm numlet
som numlet
jzi stop
ji start
stop:hlt
*********************************************
```

Table 7. Assembler Output

7. Comments from Students

In this section, student's comments are provided to show the value of this approach to teaching the inner-workings of a computer. One student, Daniel McCallum, wrote in an email [2] after completing Computer Organization:

"Multimedia Logic has helped me a lot to comprehend many of the complex ideas behind the workings of a computer. It helps me see things visually and can look at things one step at a time. For example how an ALU works made a lot more sense when I could put it together and take it apart myself, using Multimedia Logic. Another big aspect of Multimedia Logic was that I can see all the different switches, gates, etc. visually and have come to understand how basically a computer does what it does."

Several students commented that they now understood how circuits make computers and how computer functions can be made from simple switching logic devices. Students that previously used breadboard devices commented that understanding what was going on was much easier in the emulated environment because each register can be instrumented individually.

8. Limitations of Multimedia Logic

One difficulty encountered with Multimedia Logic is the unexplained dropping of wires from saved files. This occurs the first time a new file is saved and seems to be a problem with overlapping components. For example a horizontal row of eight light-emitting diodes will lose connection to every other light when saved, if they are placed adjacent to each other and are vertically lined up. The "work-around" for this problem is to stagger the lights slightly in the vertical direction. This vertical staggering can be seen in figures 1, 3 and 5.

9. Summary

A number of designs built in Multimedia Logic have shown to be useful to students in gaining an understanding the inner workings of a computer and related technology. Students in computer architecture classes have successfully used this tool to design many eight-bit and even two sixteen-bit computers, most with single cycle designs, but two with multi-cycle designs. Through this experience the details of how switches can make computers becomes very clear.

10. Acknowledgments

Thanks to George Mills of www.softronix.com, who has graciously made his product, Multimedia Logic, available for free download and included the source code.

And a special thanks to the faculty and students of our computer science department who have encouraged me in this effort by their enthusiastic support.

11. References

[1] Hoffman, Mark E., *The Case for More Digital Logic in Computer Architecture*, Conferences in Research and Practice in Information Technology, Vol. 30.

[2] McCallum, Daniel, Email of 6/25/2004.

[3] Mills, George, www.softronix.com, Multimedia Logic download kit and source kit.

[4] Patterson, David A., and Hennessy, John L. *Computer Organization and Design, the Hardware/Software Interface, 2nd Edition,* Morgan Kaufmann Publishers .

[5] Patterson, David A., and Hennessy, John L. *Computer Organization and Design, the Hardware/Software Interface, 3rd Edition,* Morgan Kaufmann Publishers .

[6] Powers, Kris D., "Teaching Computer Architecture in Introductory Computing: Why? And How?" Sixth Australasian Computing Education Conference (ACE2004), Dunedin.

[7] Wolffe, Greg, Yurcik, William, Osborne, Hugh, and Holliday, Mark, "Teaching Computer Organization/Architecture With Limited Resources Using Simulators", SIGCSE 2002, ACM Press, Northern Kentucky USA, Feb/March 2002.

An Embedded Systems Course and Course Sequence

Kenneth G. Ricks*
Electrical and Computer Engineering
The University of Alabama
Tuscaloosa, AL 35487, USA
kricks@coe.eng.ua.edu
* Contact Author

William A. Stapleton
Electrical and Computer Engineering
The University of Alabama
Tuscaloosa, AL 35487, USA

D. Jeff Jackson
Electrical and Computer Engineering
The University of Alabama
Tuscaloosa, AL 35487, USA

Abstract

Recently, the University of Alabama Department of Electrical and Computer Engineering adopted curricular changes to incorporate embedded systems into its computer engineering core course sequence. One of the major changes implemented was the creation of a senior lecture/laboratory combination specifically dedicated to embedded systems. This paper describes the specific lecture and laboratory content of this senior-level course and how this course fits within the new curriculum a The University of Alabama.

1. Background/Introduction

The faculty of the Computer Engineering program at The University of Alabama has undertaken a project of pedagogical improvement by incorporating a focus on embedded systems that is pervasive throughout the computer engineering curriculum. There are several driving factors behind this decision. Embedded systems represent a major fraction of the digital systems market as indicated by the fact that embedded systems represent a key technology in the automotive, consumer electronics, industrial automation, military and aerospace applications, office automation, telecommunication and data-communication industries [1-3]. There is also significant regional interest in embedded systems with several major automotive and other manufacturing industries located in the state of Alabama and surrounding areas [4].

As much as 98% of all 32-bit microprocessors currently in use worldwide are used in embedded systems [5]. However, most computer engineering programs teach programming and design skills that are appropriate for a general-purpose computer operating under control of a commercial operating system rather than for the more specialized embedded systems [6]. Additionally, instruction in embedded systems can increase opportunities for breadth in a curriculum as these systems naturally involve hardware and software components that interface to various electrical, mechanical, and chemical processes. Thus embedded systems education is an excellent example of an area of study that requires depth and rigor while maintaining breadth required for meeting emerging workforce and education needs of U.S. industry [4, 7].

The rapid proliferation of embedded systems requires an increasing number of engineers trained in microcontroller-based systems, real-time concepts, hardware/software co-design, distributed processing, hardware/software integration, and system-level issues in embedded systems design. Instructional material is just beginning to appear in this area and the development of this focus area, associated instructional materials, and evaluation materials will allow us to better serve our students and, more importantly, to provide material for this emerging area that can be adapted for use by others.

This embedded systems focus is important in the context of distinguishing our programs at The University of Alabama. The embedded systems focus will directly affect three degree programs: Computer Engineering, Computer Science, and Electrical Engineering. The majority of computer engineering programs deal primarily with design and programming for general-purpose computers. Traditionally, we also have offered a broad exposure to computer engineering topics in our curriculum and conducted research in a number of areas. Recent self-assessments of our program utilizing both the IEEE/ACM model computer engineering curriculum [8] and a set of nationally recognized and comparable programs led us to choose to adopt a more focused curriculum model. Because of our limited size and resources, we believe that focusing both our education and research efforts on a single theme, namely embedded systems, will allow us to progress in both areas. A web-based search for "embedded systems education" using the ASEE database and internet search engines reveals a scarcity of programs focusing on embedded systems, particularly in the U.S. Southeastern region. We believe that successful implementation of this focused effort in a niche area will serve as a model for many other similarly sized programs [4].

2. The University of Alabama Computer Engineering Core Course Sequence

The plan for reforming the curriculum will involve each of the courses in the Computer Engineering program shown in Figure 1. In this figure, the arrows denote a prerequisite relationship between the courses. The comprehensive plan builds upon each of these courses to provide an enriched experience for the students.

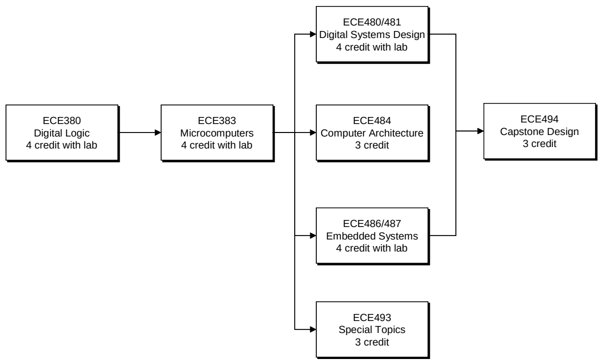


Figure 1. Computer Engineering Core Curriculum with Embedded Systems Focus

The first course in the sequence of Figure 1, ECE 380 - Digital Logic, is a four-hour lecture/laboratory combination class incorporating traditional combinational and sequential logic design and digital design using VHDL. The embedded systems theme is incorporated into this class through exercises that, for example, include digital counter designs in the context of watchdog timers common in embedded processors, pulse width modulation (PWM) circuit design, and complex state machine designs for typical embedded system tasks such as bus arbitration. Altera's Quartus II electronic design automation software is used to provide the students with system design and simulation experience. This course is required for students in all three directly affected engineering disciplines. Along with the nature of the subject material, this student diversity makes this course especially well-suited for the incorporation of multidisciplinary team-based learning. Finally, basic designs from exercises in this course are used as components in larger, more complex designs in subsequent courses. Proper design techniques as well as design reuse are stressed.

The second course, ECE 383 – Microcomputers, builds on a foundation of traditional architectural topics such as register, memory, bus, and instruction set design to incorporate embedded systems topics such as peripheral interfacing, analog-to-digital (A/D) conversion, device control, interrupt management, and system reliability. Metrowerks CodeWarrior is used to provide a modern development environment for programming and debugging the software portions of system design. Students expand the use of Altera's Quartus II software introduced in ECE 380 to produce custom interface logic to connect a microprocessor with a variety of peripheral devices. We also introduce the basic use of Mentor Graphics software for facilitating hardware/software co-design and board-level design issues. As with ECE 380, this course is required for students in all three directly affected engineering disciplines facilitating the incorporation of multidisciplinary team-based learning.

The third course, ECE 480/481 - Digital Systems Design, is a four hour lecture/laboratory combination class that focuses on the design and test of digital systems components including basic arithmetic and logic components, and digital systems interfaces including PWM designs, and mouse, keyboard and video display drivers. VHDL-based designs are implemented on FPGA devices. System-on-a-Programmable-Chip design methodologies are introduced. Special emphasis on testing includes an introduction to device-embedded logic analyzers and their use for debugging SoPC designs. Specific topical material introduced includes hardware description languages, electronic design automation, logic circuit testing and testable design, SOC design and intellectual property (IP) cores. Software tools for electronic design automation from Altera and Mentor Graphics corporations are used, allowing students previously exposed to these toolsets to become more proficient in their use. More advanced features of these toolsets are introduced including floor planning, advanced timing analysis, and synthesis options. Additional toolsets are introduced including both design-for-test and hardware/software co-design for embedded processors. Additionally, the Mentor Graphics toolset includes capabilities for engineering project management that are used to manage the execution of best design practices throughout project assignments. Specific embedded systems concepts that are covered include embedded processor design, peripheral integration and SOC solutions for embedded systems. Integration of custom hardware and software with existing components is emphasized. Hardware/software co-design is addressed by integrating and expanding basic projects from the first two courses: ECE 380 and ECE 383 [4].

The fourth course, ECE 484 – Computer Architecture, is a three hour course that incorporates embedded systems concepts into the context of computer architectural issues. Traditional computing architectures are introduced, evaluated, and contrasted with embedded systems architectures [9]. Specifically, architectural design tradeoffs associated with the processor(s), input/output (I/O), and memory are discussed. Performance evaluation and analysis is also contrasted between a general-purpose MIPS architecture and architectures used in embedded systems. Hardware/software co-design is introduced, and the relationships between the software and hardware components of computing systems are discussed.

The fifth course, ECE 486/487 – Embedded Systems, is a four hour lecture/laboratory combination class. It is described in detail in the following sections of this paper.

The sixth course, ECE 493 – Special Topics, provides flexibility in the curriculum by allowing advanced embedded systems concepts to be introduced on a regular as-needed basis. Such topics would include, but are not limited to, real-time systems, distributed embedded systems, hardware/software co-design methodologies and design verification/validation/testing.

The seventh course, ECE 494 – Capstone Design, culminates the undergraduate engineering design experience by providing a semester-long, team-oriented design project building on the skills learned in a previous senior-level lecture/laboratory course. Candidate lecture/laboratory courses preceding the Capstone Design course include ECE 480/481 Digital Systems Design and ECE 486/487 Embedded Systems. All facets of the previously introduced software tools will be exercised in this course. Design projects such as programmable logic devices and SOC solutions in robotic car competitions [10] and projects following the IEEE Computer Society International Design Competition model [11] will be used. Since the design is team oriented, this course also provides the opportunity to assess student teaming skills and the pedagogies used throughout the curriculum for instruction in teaming [4].

3. ECE 486/487 Embedded Systems

The ECE 486/487 Embedded Systems lecture/laboratory course is a new course resulting from the curriculum reform activities. The following sections describe the concepts covered in the lecture, how these concepts relate to the IEEE/ACM model curriculum, the laboratory activities, and the hardware and software currently used for the laboratory assignments.

3.1 Lecture Material

The course begins with an introduction to embedded systems. This portion of the lecture provides general definitions of embedded systems, examples of common embedded systems, and distinguishes embedded systems from other types of computing systems. Also, general characteristics of embedded systems are given and functional and non-functional metrics used to evaluate system design and performance are described. Background material such as Moore's Law is presented to explain the broad emergence of embedded systems throughout our society. This leads to a justification of embedded systems as a focus area within computer engineering and the corresponding need for embedded systems education. This material corresponds to various core components of the IEEE/ACM model curriculum including "History and overview of embedded systems – ESY0" and "Classification of embedded systems – ESY6", as well as one elective component of the model called "Software engineering considerations – ESY7".

The next set of lectures is designed to concentrate on the design of embedded systems. Specifically, ad-hoc, top-down, and bottom-up design methodologies are shown to be inadequate as general-purpose methodologies due to the varying system requirements and characteristics across multiple embedded systems applications. Hardware/software co-design is introduced and compared to the other methodologies. Its uses a domain-independent process abstraction to describe system behavior which delays hardware and software allocation and mapping decisions making it more suitable as a general-purpose approach for these applications. The main goals of this concept are that embedded systems designers must be able to perform hardware and software design tradeoffs and analysis. Computational models used to describe system behavior are also introduced. These lecture concepts correlate to several of the components in the "Software engineering considerations – ESY7" section of the model curriculum which is recommended as elective material [8].

The aforementioned lecture materials represent a high-level, abstract view of embedded systems. Some of these concepts, particularly the design methodologies, are difficult for students to grasp, and students have indicated that these sections of the lecture are their least favorite. The following sets of lectures deal with more tangible concepts that are more easily mapped to hands-on laboratory assignments. Students have indicated a higher level of interest in this material.

The next set of lectures is designed to discuss typical I/O activities and related concepts required of embedded systems. Specifically, data acquisition, A/D conversion, digital-to-analog (D/A) conversion, sampling rates, the Nyquist rule, A/D resolution, "system" resolution, PWM, timers, timer resolution, communication protocols, direct memory access, and specific I/O devices such as keypads, and UARTs are discussed. Many of these concepts are introduced earlier in the course sequence, but in this case a concerted effort is made to put these concepts into a "system" context. For example, A/D conversion is introduced in ECE 383 in the context of an on-chip converter incorporated with the microprocessor. In ECE 486, A/D conversion is again discussed, but this time it is seen as part of a data acquisition system and the A/D converter is incorporated as an off-chip I/O peripheral device. In this case, the A/D converter resolution and sampling rate are compared to the requirements of the "system" within the context of the specific real-world data being collected. These topics are listed as components in two parts of the model curriculum including "Fundamentals of embedded systems – ESY1" (core) and "Hardware considerations – ESY3" (elective) [8].

The different architectures to support interfacing required for the I/O activities previously mentioned is the focus of another set of lectures. In particular, bus-based architectures are discussed and specific designs are created. Bus communication protocols are compared, master-slave relationships are defined, and system activities are decomposed into atomic bus transactions. Bus arbitration is introduced, multiprocessor bus architectures are described, and bus saturation is defined and explored. Finally, interrupt-driven and polled I/O are described, compared, and contrasted in terms of hardware design, software design, and system performance. All of these topics satisfy many of the components in the following parts of the model curriculum: "Language issues – ESY2" (core), "Hardware considerations – ESY3" (elective), "Mapping between languages and hardware – ESY4" (core), "Classification of embedded systems – ESY6" (core), "Particular techniques and applications – ESY8" (elective), and "High integrity software systems – ESY10" (elective) [8].

Another set of lectures is designed to address memory concepts. These lectures cover different memory technologies and discuss particular applications of each. The technologies are compared and contrasted based upon their operational characteristics. Also, memory system hierarchical design and caching are introduced. The localities of reference upon which memory system design is based are used to show the importance of memory system design and its effect on overall system performance. The particular aspect of the model curriculum incorporated into these lectures is "Mapping between languages and hardware – ESY4" (core) [8].

The last set of lectures is designed to introduce real-time issues. Real-time systems are defined and the various types are compared and contrasted. Real-time operating systems are discussed and their performance goals are described as they relate to I/O activities and memory operation addressed in earlier lectures. For example, at this point students seem to recognize and understand the effects of caching on real-time performance and the minimization of interrupt latency with real-time operating systems. The students have shown genuine excitement about being able to relate such concepts. Scheduling is also introduced at this point. Since we have already defined the process abstraction and the concurrent process model of computation, it is easy to address process scheduling, preemption, non-preemption, priority-based scheduling, and priority assignments based upon popular algorithms such as the rate-monotonic algorithm. These topics correlate to the following parts of the model curriculum: "Language issues – ESY2" (core), "Mapping between languages and hardware – ESY4" (core), "Real-time operating systems – ESY5" (elective), and "Classification of embedded systems – ESY6" (core) [8].

3.2 Laboratory Hardware and Software

The hardware and software dedicated to the embedded systems laboratory assignments uses a single-bus architecture built around the VMEbus. The VMEbus is a standardized bus protocol designed for I/O intensive operations and often used in industrial, military, and aerospace embedded applications [12]. Each of the three lab stations consists of two single-board-computers (SBC) connected to the VMEbus, one 6U-sized combination VMEbus CDROM drive and hard drive for each SBC, and one shared A/D board consisting of 64 differential analog input channels also connected to the VMEbus. One SBC is loaded with the Windows XP Professional operating system and the second SBC is loaded with Redhat Linux version 9.0 running the 2.4.20-6 Linux kernel. A customized library of software functions compatible with the C programming language is available for use on each platform. The functions make interfacing to the VMEbus address space easy and eliminate the need for timely driver development for the specific hardware used. Each of the three lab stations allows for remote login via the Internet. This promotes sharing of the hardware. Remote login does not provide for interacting directly with the equipment in some cases, for example setting up analog input into the A/D board. But, it does allow for software development which accounts for a majority of the time spent using the stations.

Although the VMEbus is seen almost exclusively in industrial, military, and aerospace applications, it is surprisingly useful for academic embedded systems activities. In addition to using its asynchronous protocol as an example of such bus communications, the flexibility of the VMEbus makes it perfect for demonstrating many other topics discussed in the IEEE/ACM model curriculum. For example, SBCs can be easily added to a VMEbus backplane to produce a multiprocessor. The SBCs can be the same producing a homogenous multiprocessor, or each can be different, even executing different operating systems, to produce a heterogeneous multiprocessor. Various memory configurations can be set up by adding global memory cards to a VMEbus system. Multiprocessors and shared memory provide the opportunity to address mutual exclusion, concurrency, and inter-process communication issues. Various operating systems including real-time operating systems are readily available for VMEbus SBCs. With such an operating system, detailed timing analysis of system performance and real-time scheduling concepts can be investigated. The VMEbus supports various bus arbitration methods, has a prioritized 7-level interrupt protocol, supports multiple bus masters, has a data transfer rate of 40 Mbytes per second, and is standardized. Its thorough I/O support makes it easy to study polled I/O, interrupt-driven I/O, standard and memory-mapped I/O configurations, arbitration for multiple interrupting devices, starvation, and bus saturation concepts. One final benefit of the VMEbus is that there are many vendors and many choices for VMEbus devices making off-the-shelf components common, relatively inexpensive, and simple to use.

3.3 Laboratory Activities

The laboratory activities are chosen to supplement the lecture material. Each assignment is made with the goal of supporting the "system" concept of an embedded system. So, in each case, overall system performance is a concern. Based upon the data presented in [13], the C programming language is used for approximately 80% of all embedded systems, and assembly language is used for approximately 10%. Since assembly language is the choice for earlier courses in the UA sequence, such as ECE 383, this is the best time to introduce C as a high-level programming language suitable for embedded applications. By doing so, the laboratory addresses a core topic in the IEEE/ACM model curriculum called "Language Issues – ESY2". This specifically refers to a need for the description of various programming languages used in embedded systems and the specification of a guide for when such languages are appropriate [8]. Finally, each assignment will use the VMEbus systems described in the previous section or will involve a software simulation of some embedded systems component.

Another important aspect of the laboratory assignments is that the technical data necessary to program the hardware and to use the custom C software libraries is not presented in a formal fashion. Instead, students are responsible for gathering the necessary information from the technical documentation accompanying the laboratory hardware and software, i.e. technical manuals. This type of experience is invaluable to embedded systems engineers who will be faced with this task early and often in their careers, often dealing with documentation that is poorly written and filled with errors. Thus, the laboratory activities provide an opportunity to assess student learning in an unstructured environment.

The first two laboratory assignments involve the creation of a data acquisition system. The particular analog data collected from the real-world is not as much of a concern as how the data is collected and what is done with the data. For the first iteration of the course, the students collected environmental data including temperature, light, and humidity. The sensors and the circuitry required were pre-selected and set up for the students. This represents a case where practicing engineers are given an I/O component, i.e. a sensor package, with which to work and must integrate that package into the data acquisition system. In this way, the students can focus on system integration activities and avoid electronic design issues they should have been exposed to earlier in the curriculum and that tend to distract some students from the goal of the current exercise. For the first laboratory assignment, students create a data acquisition system that uses polled I/O to collect environmental data at a specified rate. The time required for the A/D conversion and the responsiveness of the overall system is collected. In the second lab, the students create the same data acquisition system that is interrupt-driven. In this case, the interrupt latency is measured and compared to the system timing of the polled I/O system. Creating the same functionality using two different approaches has proven to be a valuable technique in demonstrating important differences in performance and implementation. These two assignments also support many of the interfacing topics covered in the lecture portion of the course including general I/O configurations, writing interrupt-service routines, and decomposing bus-based communication into atomic bus transactions using master-slave relationships.

Another lab assignment that is used is that of creating a software simulation of a memory hierarchy. For this assignment, there is no direct connection to the VMEbus hardware, although students are encouraged to write their simulations using the lab stations to promote further familiarity with those systems. For this assignment, students are required to develop a simulation of a memory hierarchy configured according to user input. Once configured, the simulations must be able to accurately track memory performance given a set of memory references. Considering that many embedded applications have predictable workloads, memory performance prediction and configuration is a necessary component of embedded systems development.

The final laboratory assignment involves real-time scheduling. Like the previous lab, the students are asked to develop a software simulation of a real-time scheduler configured according to user input. Possible configuration options include preemption or non-preemption, static or dynamic priority assignment, periodic or aperiodic task execution, independent tasks or tasks having precedence constraints. This assignment incorporates many concepts discussed in the IEEE/ACM model curriculum and included as part of the lecture material. For example, real-time operating system issues are addressed, as well as different priority assignment algorithms such as rate-monotonic and earliest-deadline-first. Scheduling processes also ties back into the concurrent process model of computation mentioned earlier as a technique used to describe system behavior. Students can now see the effects of different functional decompositions and different granularities of decomposition.

4. Future changes to ECE 486/487

After the first complete offering of this course with its associated laboratory assignments, it is evident that several adjustments must be made. First, a complete co-design laboratory assignment must be produced to complement the lecture material on this subject. Co-design is a rather abstract topic for students to understand especially if they have little to no design experience. The problems encountered up to this point with introducing such an assignment include finding a suitable system with the scope appropriate for a 1-2 week assignment, a system that will provide obvious and limited design choices after using trade-off analysis, and conquering the learning curve associated with design environments using co-design.

A second addition to the course includes expanding the software simulation assignments to incorporate the VMEbus systems. Adding a real-time operating system to one SBC will make it easy to incorporate the VMEbus systems into the scheduling assignments. Also, the VMEbus SBCs have cache memories and configurable caching options including the ability to turn caching off to support hard, real-time applications. With limited effort, it should be straightforward to incorporate the VMEbus systems into the memory simulator assignments.

Finally, additional lab assignments must be introduced to complement other lecture topics such as multiprocessing. As embedded systems continue to increase in complexity, multiprocessing is becoming a necessary topic as opposed to an "advanced" topic and must be incorporated into the class. The VMEbus systems readily support multiprocessing and this must become a fundamental part of the course.

In addition to adding laboratory assignments to the course, the course lecture and lab materials must be generalized in such a way as to make them available for use by others. The generalized versions of the materials should incorporate feedback generated from student assessment of the current materials. Assessment strategies are currently being defined.

5. Conclusions

The University of Alabama has reformed its Computer Engineering curriculum in order to incorporate an embedded systems theme throughout its core course sequence. One large component of these changes involves the introduction of a senior-level lecture/laboratory combination course concentrating on embedded systems. This course is integrated into the core course sequence and its lecture topics are derived from the IEEE/ACM model computer engineering curriculum. The laboratory assignments are designed to complement the lecture topics, and they also incorporate many of the topics, both core topics and elective topics, mentioned in the model curriculum. The laboratory assignments make use of a system architecture designed around the VMEbus. The VMEbus is shown to provide a powerful, flexible platform from which to teach many of the concepts in the model curriculum.

6. References

[1] Gannod, G. C., Golshani, F., Huey, B., Lee, Y. H., Panchanathan, S., and Pheanis, D., "A Consortium-based Model for the Development of a Concentration Track in Embedded Systems", 2002 Proceedings of the American Society for Engineering Education Annual Conference and Exposition, session 1532.

[2] Wolf, W., "Rethinking embedded microprocessor education", In Proceedings of the 2001 American Society for Engineering Education Annual Conference and Exposition, Albuquerque, NM, 2001.

[3] Wolf, W., Madsen, J., "Embedded systems education for the future", In Proceedings of the IEEE, 88(1), pp. 23 . 30, January 2000.

[4] Stapleton, W. A., Ricks, K. G., Jackson, D. J., "Implementation of an Embedded Systems Curriculum" 20th International Conference on Computers and Their Applications (CATA'04), New Orleans, Louisiana: ISCA, pp. 302-307 (March 2005).

[5] Turley, J., "The Two Percent Solution," Embedded Systems Programming, December 2002, www.embedded.com/story/OEG20021217S0039.

[6] Ganssle, J., “A Call for a New Curriculum,” Embedded.Com, May 2002, www.embedded.com/ story/OEG20020530S0075.

[7] “From Analysis to Action: Undergraduate Education in Science, Mathematics, Engineering and Technology”, National Research Council, National Academy Press, Washington, DC, 1996, http://www.nap.edu/ catalog/9128.html.

[8] The IEEE Computer Society/ACM, Computing Curricula, www.computer.org/education/cc2001/.

[9] Hennessy, J., Patterson, D., *Computer Architecture: A Quantitative Approach*, 3rd Edition, Morgan Kaufmann, 2003.

[10] Georgia Institute of Technology, School of Electrical and Computer Engineering, http://users.ece.gatech.edu/ ~hamblen/4006/projects/nios_robot/ECE4006_page.html.

[11] IEEE Computer Society, Computer.org, Computer Society International Design Competition 2003, http://computer.org/CSIDC/.

[12] *IEEE Standard for A Versatile Backplane Bus: VMEbus*, ANSI/IEEEANSI/IEEE Std 1014-1987, 1987.

[13] Lewis, Daniel W., Fundamentals of Embedded Software, Prentice Hall, Upper Saddle river, New Jersey, 2002.

Hardware/Software Co-Design of Embedded Real-Time Systems from an Undergraduate Perspective

Kevin C. Kassner*
RF & Electronic Systems Department
Dynetics Corporation
Huntsville, AL 35806, USA
kevin.kassner@dynetics.com
*Contact author

Kenneth G. Ricks
Electrical and Computer Engineering
The University of Alabama
Tuscaloosa, AL, 35487, USA
kricks@coe.eng.ua.edu

Abstract

The increasing complexity of embedded systems parallels the difficulty of adequately preparing students to design them. Two topics key to the success of a graduate in the area of embedded systems are hardware/software co-design and real-time computing. This paper serves as a case study describing how an undergraduate applied hardware/software co-design in the design of a spectrum analyzer with real-time constraints for a Capstone senior design project. The goal of this work is to produce a co-design approach more suited for undergraduates having little design experience.

1. Introduction

How can we prepare our electrical and computer engineering students to design embedded systems? There is so much material to cover at the undergraduate level it hardly seems possible to adequately prepare students for a career in embedded systems development. Thus, educators are faced with the difficult task of selecting a subset of critical topics to include in their curriculum. Two critical topics are hardware/software (HW/SW) co-design and real-time computing. In spring 2004 The University of Alabama offered for the first time an embedded systems class at the undergraduate level. An educational result of this course was the design of a spectrum analyzer with real-time constraints which was successfully completed December 2004 as a Capstone Design project. This paper is an examination of how HW/SW co-design was employed in an undergraduate design class. Completion of such a project suggests that a student is well prepared for a career in embedded systems development. The remainder of this paper is organized as follows. First, some background material is presented describing HW/SW co-design. Traditional implementations are presented that lead to a customized implementation implemented by the author. A short description is then given about the specific design project undertaken in this effort. This is followed by a detailed description of the custom HW/SW co-design technique as applied to this specific project. Finally some conclusions and observations are made.

2. Background

Hardware/software co-design is a design methodology which exploits the synergism of hardware and software through their concurrent design [1] and achieves this by delaying the allocation decision. Hence, as much as possible is known about the system prior to allocating pieces of the system to the hardware or software domains. This methodology has two primary advantages; more time to evaluate tradeoffs and it creates better hardware/software interfaces. However, it requires engineers to be familiar with both hardware and software caveats. Any design methodology should:

- provide a checklist for the design process
- facilitate the communication of design team members
- help to predict costs
- aid in the creation of a working prototype
- aid in the creation of a timeline for the development cycle
- help with the identification of metrics
- aid with requirements specification, and
- assist with the development of test procedures.

The goal of HW/SW co-design is to do all of these things as well as allow designers to "predict" implementation, "incrementally refine" a design over "multiple levels of abstraction", and create a "working first implementation" [2]. HW/SW co-design is a cyclic design methodology. Implementations of HW/SW co-design are as varied as embedded systems themselves. Institutions and individuals tailor the methodology to fit their application and institutional framework. All these different implementations make it difficult to apply co-design, especially for an undergraduate student having limited design experience. An implementation of HW/SW co-design suitable for an undergraduate applying it (the methodology) for the first time was needed. To meet this requirement a custom version (shown in Figure 4) based upon Wolf's and Axelsson's descriptions of HW/SW co-design was created [2, 4].

Wolf's and Axelsson's implementations of HW/SW co-design are presented here for reference and comparison to the author's version. In [2], Wolf divides co-design into four major tasks:

- *partitioning* the function to be implemented into smaller, interacting pieces;
- *allocating* those partitions to microprocessors or other hardware units, where the function may be implemented directly in hardware or in software running on a microprocessor;
- *scheduling* the times at which functions are executed, which is important when several functional partitions share one hardware unit;
- *mapping* a generic functional description into an implementation on a particular set of components, either as software suitable for a given processor or logic which can be implemented from the given hardware libraries.

In [3] Wolf also describes HW/SW co-design in the following way: "Front end activities such as specification and architecture simultaneously consider hardware and software aspects. Similarly, back-end integration and testing consider the entire system. In the middle, however, development of hardware and software components can go on relatively independently – while testing of one will require stubs of the other, most of the hardware and software work can proceed relatively independently" [3]. A block diagram of the co-design process from [3] is shown in Figure 1. Wolf's two descriptions of HW/SW co-design are very different, yet they both demonstrate the core concept of delayed allocation.

Though the cyclic nature of co-design is missing from Figure 1, it is demonstrated in Axelsson's diagram shown in Figure 2. The structure of Figure 2 also emphasizes the delayed allocation decision by including allocation as a separate task in the design flow diagram. Axelsson [4] defines the tasks in his figure as follows:

- *System behavioral description*, giving an executable specification of what the system is supposed to do.

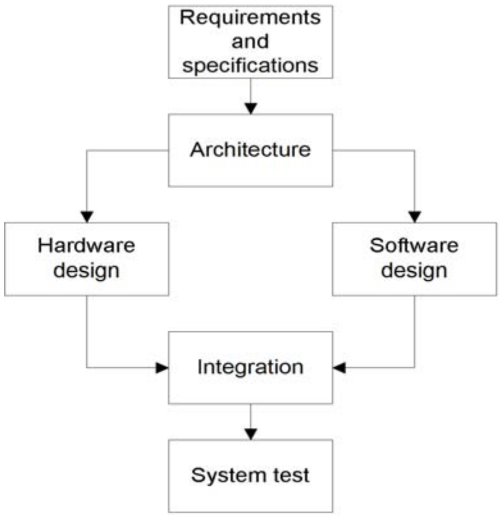


Figure 1. A simple HW/SW co-design methodology [3].

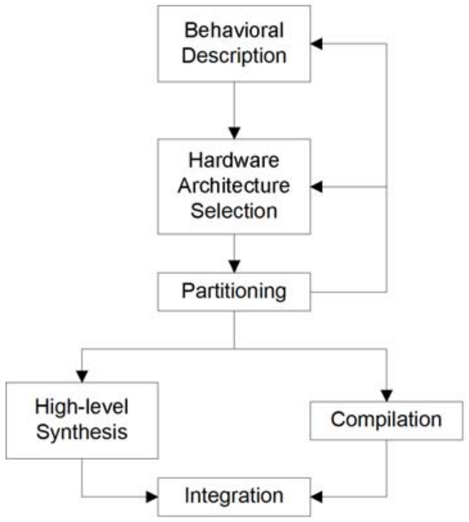
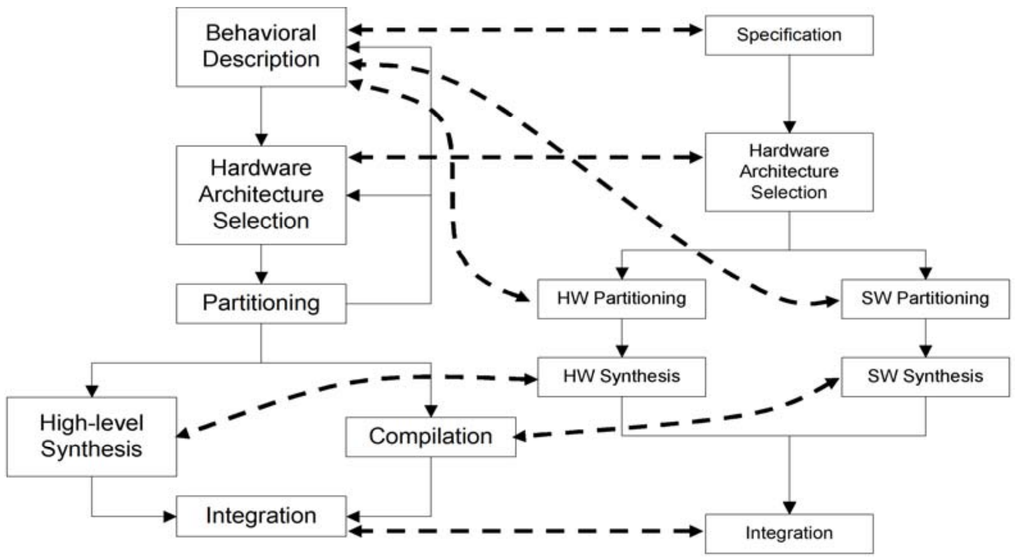


Figure 2. Axelsson's diagram of HW/SW Co-design [4].

- *Hardware architecture selection*, describing what hardware components should be used and how they should be connected.
- *Partitioning*, deciding which parts of the system behavior should be realized by what parts of the hardware architecture.

Please note that Axelsson's use of the term partitioning is analogous to our use of allocation thus far.

Figure 3 is a comparison of Axelsson's design flow diagram and a typical top-down model. This figure illustrates the advantage of a detailed behavioral description that is domain independent; the more information known about a system prior to hardware architecture selection the better.



HW/SW Co-Design Top-Down

Figure 3. Axelsson's diagram versus a typical top-down model.

Figure 4 shows the author's flow diagram for HW/SW co-design. The nomenclature used here is slightly different from that of Wolf and Axelsson.

- *Specification*, usually consists of a collection of metrics, both functional and non-functional, which provide a precise description of the top-level system attributes and requirements. Examples of metrics include throughput, latency, unit cost, NRE cost, power consumption, maintainability, and time-to-market.
- *Partitioning* is the action of breaking the system functionality into small domain-independent, concurrent and interacting/communicating processes. The size of the processes is called the granularity. The result of the partitioning step should be a fully defined behavioral description of the system, with well defined interfaces between processes. Performance requirements for the processes such as frequency, throughput, and latency should also be defined.
- *Allocation* is the action of assigning each process to either the hardware domain or the software domain. Communication bandwidth alternatives/limitations between hardware and software should be considered. For example, two processes exchanging lots of data frequently would likely best exist in the same domain.
- *Hardware Architecture* means describing what hardware components should be used and how they should be connected [4] to support the execution of the processes.
- *Mapping* is the selection of specific hardware components and mapping the processes onto parts of the hardware architecture. This includes mapping processes from the software domain to the processor(s) on which they will be executed. Much consideration should be given to the execution requirements of the processes. Manufacturability should be considered during component selection.
- *Synthesis* is the implementation of the hardware and software processes for the selected hardware.
- *Integration* is the recombination and testing of processes and interfaces after implementation.
- *Scheduling* is the assignment of resources to all system processes such that their execution requirements are satisfied including inter-process communication dependencies.

Those who are familiar with HW/SW co-design may not see the need to break the design process down into this many steps. However, undergraduates find this decomposition beneficial because it requires one to think about each step separately and consider trade-offs that may not have otherwise be considered. Figure 5 shows how this design flow compares to Axelsson's. Allocation is placed above hardware architecture because the allocation process provides helpful intuition going into the hardware architecture selection. This was done even though the first hardware architecture selection usually causes some immediate feedback into the allocation.

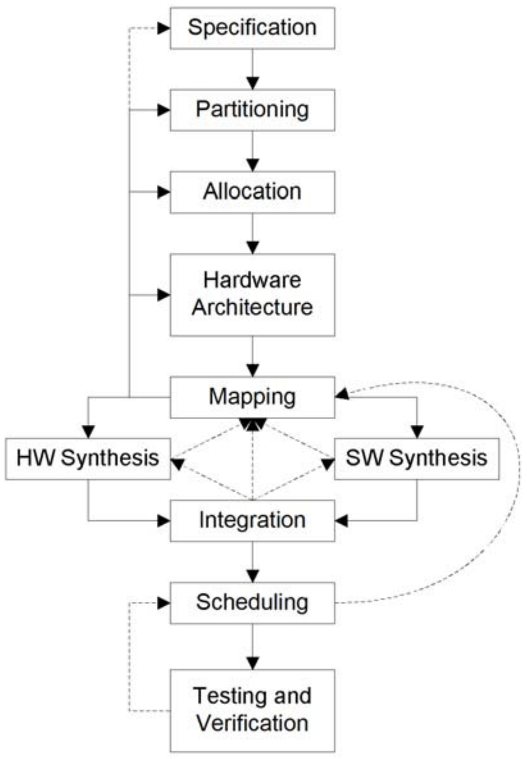


Figure 4. Customized diagram of HW/SW co-design. Dashed arrows indicate feedback paths that may not occur in every design.

Scheduling appears near the end of the design process, though a system schedule is defined in the partitioning step and considered throughout the design process. The finer granularity of the design tasks makes them more manageable for an undergraduate without much intuition gained through experience. The direct correlation to the definitions listed above serve as a reference to keep the student on track during each design task. For these reasons this design flow is believed to be much more accessible to undergraduates applying HW/SW co-design for the first time. The remainder of this paper is a case study of how this customized HW/SW co-design methodology was used in the design of a spectrum analyzer with real-time constraints for a Capstone senior design project.

3. Project Background

The project under examination is the design of an FFT based low-bandwidth real-time spectrum analyzer. The inspiration for the project was an ASIP designed by SiWorks Inc. This FFT processor is capable of computing a 1024-point FFT in just 250 clock cycles. Unfortunately these chips were not available for purchase during the initial stages of the design project. Ultimately the implementation technology used to compute the FFT was an FPGA. This resulted in a computational bandwidth well beyond that of our specifications and the analog interface. The customer for the design was the Department of Electrical and Computer Engineering at The University of Alabama for use in sophomore and junior level laboratories. The goal of the project was to design and build a beta prototype of a stand-alone spectrum analyzer with these basic requirements:

- enough bandwidth to view the spectrum of ADSL signals
- a flexible input interface for general purpose use
- VGA interface
- \$300 proposed maximum unit cost per thousand

The user interface and VGA resolution details were not specified. One of the primary metrics was the real-time requirement. These goals were met and surpassed with the exception of some op-amp stability issues and one known firmware bug. The specifications of the completed system are listed in Table 1.

Table 1.
Specifications of Completed System

Real-Time	-- Input data stream sampled continuously -- Every sample must be processed -- No results are to be discarded
FFT size	1024 points
Frequency Range	0 to 1.10 MHz
Resolution	1.95 kHz
Sample Frequency	4 MHz
Input Voltage Range	0 to 100V_{peak}
Input Impedance	1MΩ, 20pF
Input Range Selection	Automatic
System Latency	-- 0.5 ms (input to video processor) -- 80 ms (input to display)
Configuration Interface	PS/2 Mouse
Output Interface	VGA (640x480x6-bit color)
Power Source	Single Phase, 120V, 60Hz
Manufacturability	No BGA or leadless chip packages
Unit Cost per Thousand	\$87.44

4. Implementing HW/SW Co-Design

The original ad-hoc system diagram that was created prior to the application of HW/SW co-design is shown in Figure 6.

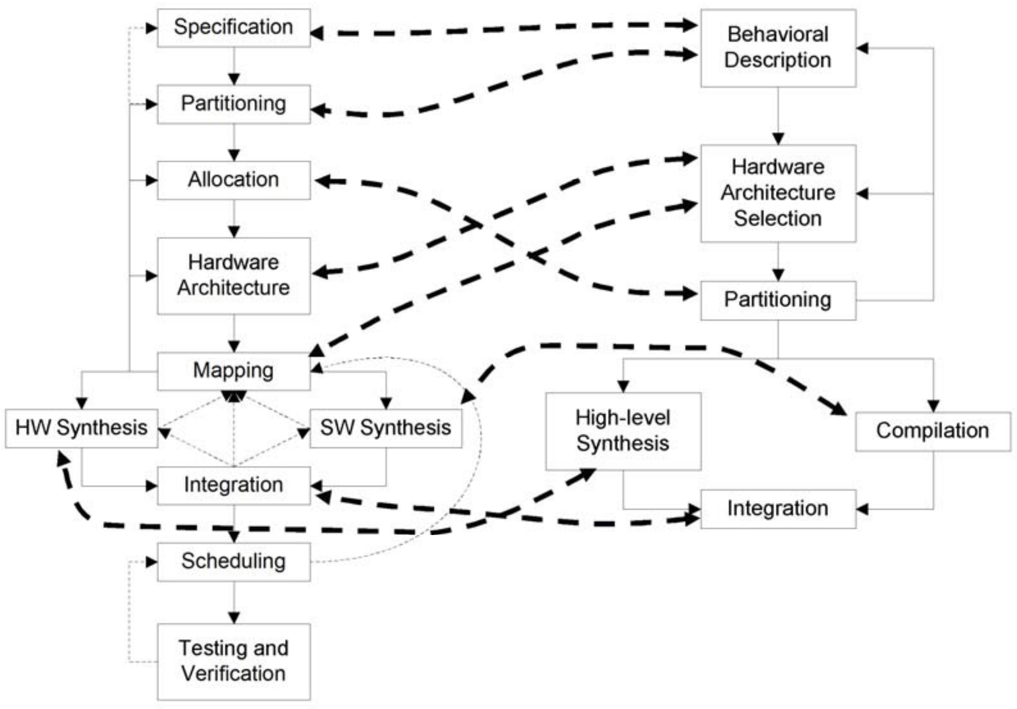


Figure 5. Customized design flow versus Axelsson's.

It is evident from the figure that partitioning, allocation, and hardware architecture selection were all occurring simultaneously. Early in a design process very little is known about how the system will function; therefore, at that point it is dangerous to attempt to define a hardware-architecture to support the operation of the system. Instead, Figure 7 shows a system partitioning resulting from a co-design approach. The immediate advantage of applying HW/SW co-design is a domain and architecture independent partitioning. Figure 8 shows one component of the system, the system control unit, decomposed into its constituent parts. This is the progression of partitioning that should continue until the processes are simple enough that they are readily implemented and the interfaces between them are fully defined, representing a system having the desired granularity. The partitioning step is also the time to define performance requirements for the processes such as frequency, throughput and latency. These will be important factors to consider in the mapping step to ensure that the final scheduling process will be successful.

During the allocation, hardware architecture, and mapping stages many tradeoffs must be analyzed before settling on a particular system implementation. It is during these stages of the co-design process that decisions must be made that may ultimately affect the partitioning and even the system specification. These are the feedback loops built into the co-design process that lead to multiple iterations through this process before project completion. For example, the original intent was to use an FPGA to implement a custom optimization of the FFT algorithm to achieve the desired performance. However, during initial *hardware architecture* selection it was realized that a sufficiently large FPGA would be cost prohibitive. The next alternative explored was an FFT ASIP although those found were not available (Zarlink PDSP16510, I&C Tech. STARFFT). Finally it was decided to use a DSP, the TI TMS320C6711, which is a 272-pin BGA device. It met the minimum performance requirements, was inexpensive and readily available. Having made this selection required a *re-partitioning* of the system. This second top-level partitioning is that shown in Figure 7. As part of the *allocation*, each process was assigned an anticipated implementation technology. At this point dual-port RAM was the chosen implementation technology for buffering. Unfortunately, dual-port RAM is very expensive in sizes as large as 1Kbyte. The memory did not need to be random access, so a 2Kbyte FIFO from TI, SN74V235, was used instead.

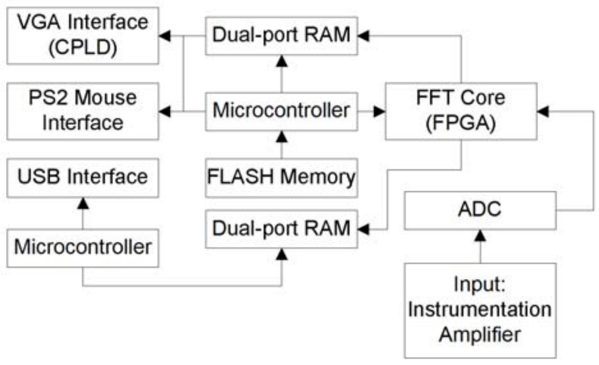


Figure 6. Original ad-hoc system partitioning.

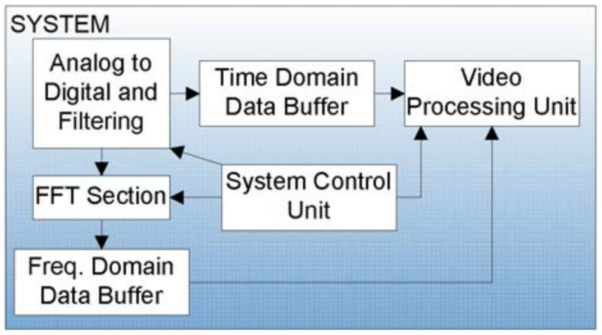


Figure 7. Co-design top-level system partitioning.

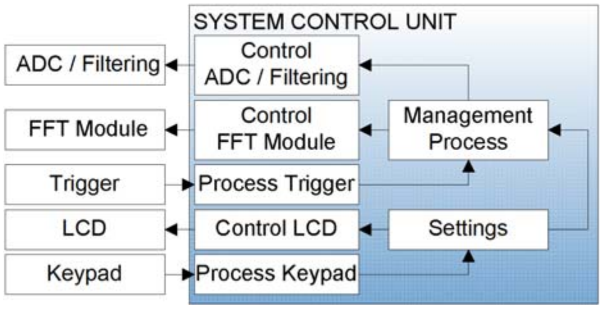


Figure 8. Partitioning of the system control unit.

As another example, there were concerns about the thermal characteristics of the circuit board and difficulty in mounting the device prior to proceeding with the *hardware architecture* using the TMS320C6711. The *specifications* were changed to include manufacturability, which meant no BGA parts. This required another tradeoff to a different DSP device, the TMS320C5402 which comes in a 144-pin QFP. This processor is capable of computing the FFT at an input sample rate of greater than 2MHz. The last frequency bin in the FFT corresponds to $F_{sample}/2$ providing a 1MHz bandwidth, just barely satisfying the minimum performance requirements. Therefore this change did not affect the *partitioning*, *allocation,* or the hardware architecture used for the TMS320C6711.

As another example of these feedback loops through the co-design process, changes were required to prevent aliasing. In order to prevent aliasing (frequencies above $F_{sample}/2$ from wrapping around into the low end of the spectrum), a low-pass filter was needed to attenuate the frequencies above $F_{sample}/2$ to less than the LSB of the input data; the input data being the output of a 10-bit ADC. However, the -3dB point of the filter needed to be 1MHz or higher to meet the performance requirements. To meet the minimum performance requirements two DSPs would need to be used in parallel, each one processing every other set of data, to compute the FFT. By putting two DSPs in parallel and using a six-pole Bessel low-pass filter the F_{sample} would be 4MHz. With this new configuration the -3dB point was calculated to be 1.10MHz. Again, these changes would ripple through all phases of the co-design process resulting in a new partitioning, allocation, and hardware architecture shown in Figure 10.

One final example of the need for feedback in the co-design process resulted from the introduction of new technology midway through the design process. In this case, it was discovered that Altera had recently made an FFT IP core available on their web site. The FFT IP core could be configured as a streaming FFT (one input and one output every clock cycle), meaning the entire system could be pipelined requiring little additional memory for buffering and greatly simplifying the overall implementation of the system. It was also determined that the Altera Cyclone EP1C12Q240C7, the largest FPGA offered by Altera or Xilinx and available in the QFP package, was available and within budget. The embedded memory blocks in the cyclone line of FPGAs are true dual-port RAM. A review of the *partitioning* showed that switching from the DSPs and external FIFOs would not require changing the algorithms; disregarding those for the VGA interface which would in fact be simplified due to the interfaces being completely internal to the FPGA. The decision was made to change the mapping to make use of this new technology. This resulted in yet another cycle through the co-design process starting with partitioning and continuing through allocation, hardware architecture, all the way to the system integration, scheduling and testing phases.

The final top-level system *partitioning* using the FPGA device is shown in Figure 9. The final partitioning shows remarkable similarity to the original system partitioning shown in Figure 7, with the exception that Figure 9 has significantly more detail at the top level. Finally, a screen shot of the output of the system and a photo of the finished spectrum analyzer are shown in Figure 11.

5. Conclusions

This case study demonstrates that the application of HW/SW co-design can be employed in senior design classes to increase the complexity of projects accomplishable by undergraduate students. The custom HW/SW co-design process presented here should be applicable to any embedded system. The structure of the design flow diagram and the accompanying definitions make it ideally suited for undergraduates.

6. References

[1] G. De Michell, R. K. Gupta, "Hardware/software co-design", Proceedings of the IEEE, Vol. 85, no. 3, March 1997, pp. 349.

[2] W. H. Wolf, "Hardware-Software Co-Design of Embedded Systems", in Proceedings of the IEEE, Vol. 82, no. 7, July 1994, pp. 967-989.

[3] W. H. Wolf, *Computers as Components, Principals of Embedded Computing System Design*, Morgan Kaufmann, New York, New York, 2001, pp. 502–503.

[4] J. Axelsson, "Hardware/Software Partitioning of Real-Time Systems", IEE Colloquium on Partitioning in Hardware-Software Codesigns, February 13, 1995, pp. 5/1-5/8.

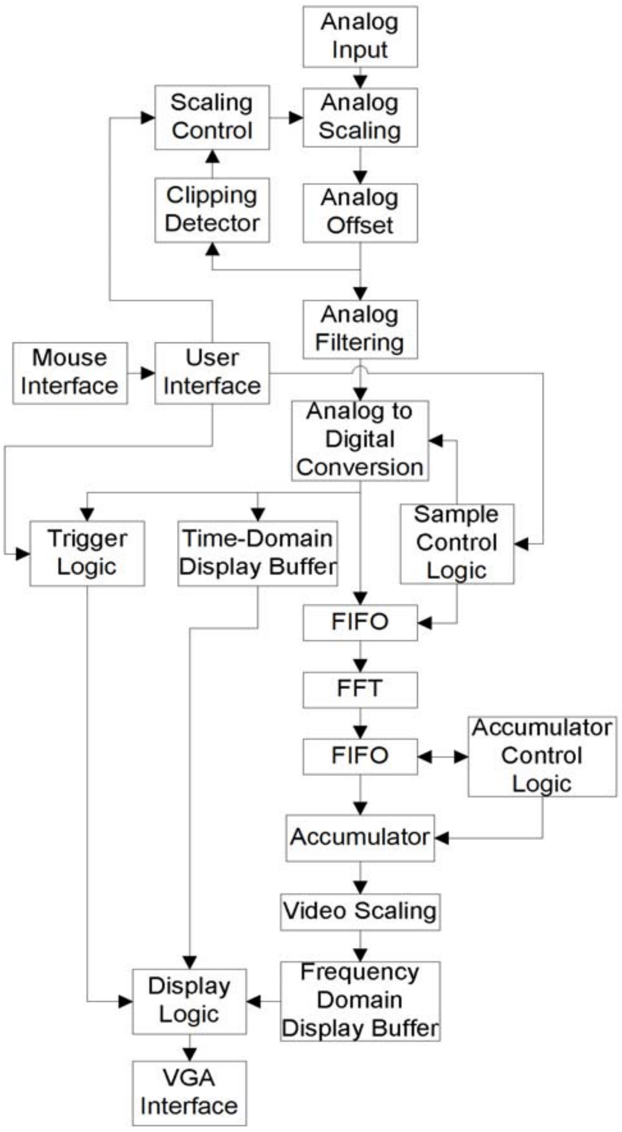


Figure 9. Final top-level system partitioning.

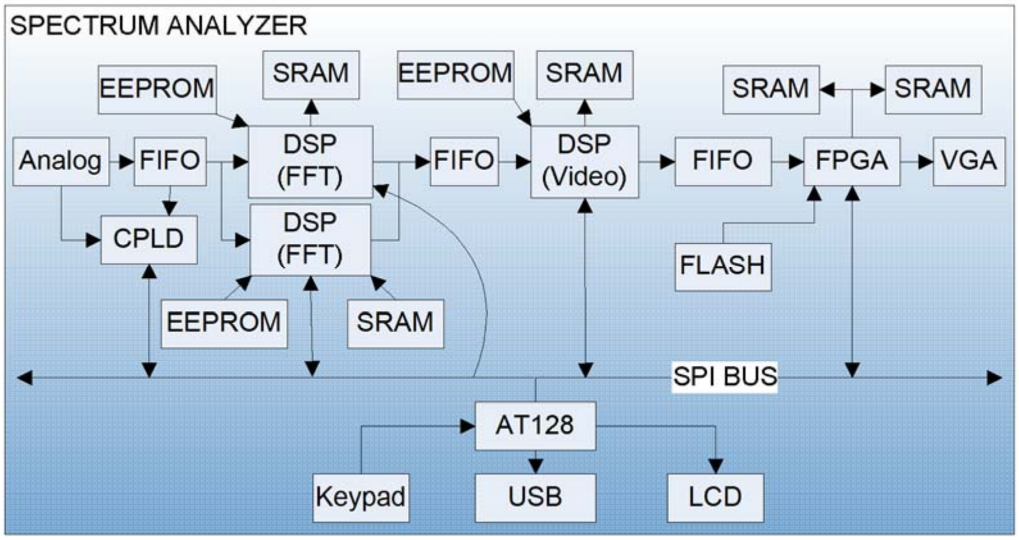


Figure 10. One version of the hardware architecture.

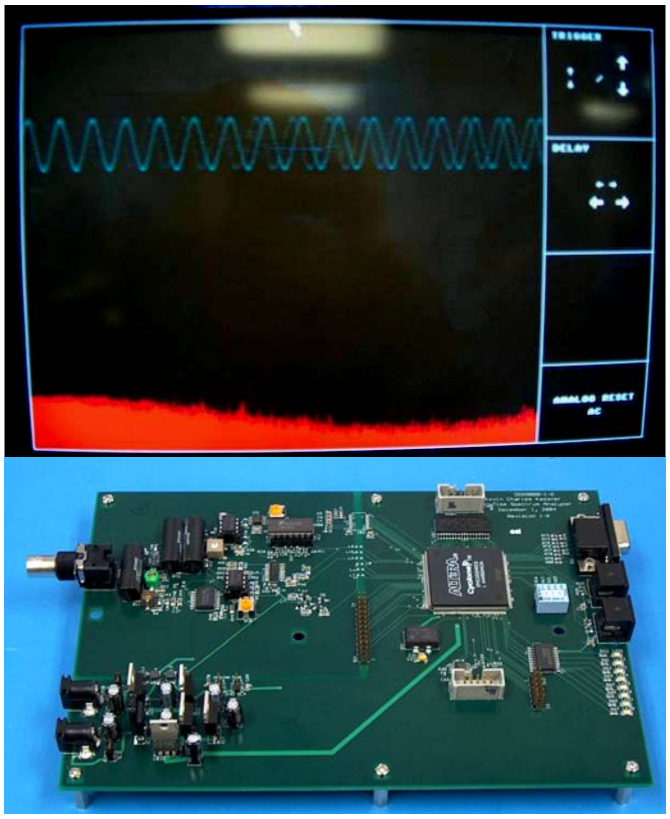


Figure 11. System output (top) and completed spectrum analyzer (bottom).

Teaching Microprocessor Systems Design Using a SoC and Embedded Linux Platform [1]

Yann-Hang Lee and Aung Oo
Department of Computer Science & Engineering
Arizona State University
yhlee@asu.edu, aung.oo@asu.edu

Abstract

In traditional microprocessor systems design courses, students learn to develop assembly language programs to control peripherals, handle interrupts, and perform I/O operations. We adopt a 32-bit StrongARM architecture on the Motorola MX1ADS board with Embedded Linux to present a modern microprocessor system design course. With this new platform, we use a high-level language to develop projects that accelerate the students' learning curve. Embedded Linux also provides the necessary flexibility and tool set required for students to debug their own projects. Our students' responded very positively to this change. They were excited about the renewed course structure, the updated learning environment, and the challenging projects.

1. Introduction

Embedded systems are designed for dedicated applications running in control systems. The unique feature of such systems is the capability to perform timely and predictable operations in response to concurrent requests arriving from the external environment. To create an effective embedded system one must properly employ the appropriate system architecture, hardware/software interfaces, peripheral devices, and software components. Currently, embedded systems companies are facing with a shortage of engineers having the appropriate skills to respond to market opportunities [8]. Therefore, embedded software engineering has emerged as a key element for curriculums in Computer Science, Computer Engineering, and Electrical Engineering at universities throughout the world.

To teach the subject of software/hardware integration and I/O interfaces, undergraduate computer science and engineering programs incorporate a microprocessor system and applications course. In the course, students develop assembly language programs to control peripherals, handle interrupts, and perform I/O operations. Then students perform experiments with a target single-board microprocessor system integrated with typical interface circuits such as programmable timers, serial ports and parallel ports. Unfortunately, this approach fails to keep pace with industry technology. This lag is prompted by the advent of rapid prototyping development of microelectronic systems that includes:

a. SoC-based platforms for embedded applications: The system-on-a-chip (SoC) devices have made great progress along with the ever-growing number of transistors that can be integrated on a chip.

b. Abundant I/O interfaces: Besides programmable timers, serial ports, and parallel ports, there are several new I/O standards designed for human interfaces, multimedia, networking, and inter-IC/device communication.

c. I/O programming with high-level languages: For software portability, modularity, and readability, high-level programming languages have been used in all levels of software development. An appropriate use of programming languages and software structures often leads to reusable embedded software.

Our traditional computer engineering curriculum also taught relatively outdated techniques in the subjects of software/hardware integration and interface. The "Microprocessor System Design" course emphasizes assembly language programming and exercises only a limited number of I/O interfaces. The course falls short in addressing state-of-the-art interfacing technology and emerging applications.

In our curriculum development project sponsored by the NSF EIA program, we redesigned the microprocessor system design class. Our goals were to provide a learning environment which aligned with emerging technology and improved the effectiveness of instruction. We also developed a laboratory environment which incorporated cutting-edge programming approaches to manage hardware components in SoC platforms. This renewed course goes beyond the inclusion of various interfaces and devices. The course focuses on the appropriate software

[1] This course development project is supported in part by NSF Educational Innovation Grant EIA-0122600, the Consortium for Embedded and Inter-Networking Technologies (CEINT), and Motorola University Program.

structures using a mixture of high-level and assembly language programming, I/O operations in modern operating systems, and reusable software components.

In this paper, we will explore the challenges and successes we encountered in implementing this new microprocessor system design class. The course serves as the first of three embedded system courses in our curriculum. Section 2 presents background information on the embedded system curriculum at Arizona State University (ASU). In Section 3, we will present the new course design followed by the course objectives, the course material and the setup of the laboratory environment for programming projects. Section 4 will cover some of our lessons learned and feedback from our students. In Section 5 we conclude our discussion.

2. Background

ASU, Motorola, and Intel formed a not-for-profit Consortium for Embedded and Inter-Networking Technologies (CEINT) in 2001 [3]. CEINT developed an infrastructure to support a strong curriculum in embedded systems. The end product was a concentrated path in Computer Systems Engineering, which consisted of an Embedded Systems Development, Embedded Systems Engineering, and Embedded Systems Capstone course [1].

We wanted to provide students with the opportunity to learn practical development techniques using the Embedded Systems Development course. To accomplish this goal, we chose Motorola MX1ADS boards using MontaVista's HardHat Linux Toolkit. Although we discussed both assembly level and high level programming development, C was the main language used for developing projects. This particular combination of programming language, development environment, and microcontroller architecture is rare for an introductory level embedded systems class.

At the same time, the students were challenged to get quickly up to speed on the fundamentals required to use the new development environment and tools. Most of the students did not have strong backgrounds in developing software for Linux. To lessen this steep learning curve, we provided laboratory demonstrations and walked through simple development projects in small groups. We also provided online tutorials, sample Linux drivers, and low level C code examples for students to study.

In this course, we introduced students to memory devices, memory controllers, buses, handling interrupts, DMA, timers, counters, UART, SPI, I2C, parallel I/O, keypad, LCD, touch panels, and A/D - D/A converters. The students also developed device drivers for timers, PWM, UART, gpio, and SPI eeprom as class projects. Other available features such as watchdog timer, blue tooth technology, USB, and CMOS sensors were left for more advanced courses in the sequence.

Assembly language teaches the students about the detailed architecture of the hardware. This gives students an appreciation for high level constructs implemented in assembly language [2]. However, implementing all software programs in assembly language neither practical nor desired. In fact, assembly-language programming is no longer the best choice for developing embedded systems, due to the availability of excellent compilers and the rising complexity of software projects [6][9].

3. Course Design

3.1. Course Objectives

The objectives of the course are to familiarize the students with hardware-software interfaces, hardware designs of microprocessor systems and peripheral devices and their communication protocols. Students work at acquiring technical knowledge and applying this knowledge to the development of programs for controlling peripheral devices and interfaces. Thus, the students learn to analyze and synthesize suitable solutions for building integrated hardware/software systems capable of interacting with external world.

3.2. Course Content

The revamped course places emphasis on software/hardware integration and I/O programming, the incorporation of the state-of-the-art SoC platforms, and emerging embedded system development tools. Our plan is to gear the integration of hardware modules to construct embedded systems and the programming models and characteristics of various I/O interfaces and peripherals. The course syllabus is established as follows:

Course Syllabus: Microprocessor System Design

Course Goals:

- Develop an understanding for using a CPU core as a component in system-level design.
- Develop the ability to integrate the CPU core with various interface units in embedded systems.
- Gain the necessary skills for programming and debugging I/O operations to manage peripherals for embedded applications.

Major topics covered:

- Introduction and review of instruction set and assembly language programming, instruction execution cycle and timing (4 lectures)
- C programming for embedded systems (2 lectures)
- Interrupts and I/O multiplexing (2 lectures)
- Parallel I/O interface and signal handshaking (1 lecture)

- Timers and counters (2 lectures)
- Serial communication: UART, SPI, and I2C (4 lectures)
- Keypad and LCD interfaces (3 lectures)
- Transducers and sensors, touch panels, A/D-D/A converters (3 lectures)
- Memory devices, SRAM, DRAM, flash memory, and SDRAM controller (3 lectures)
- Buses, access arbitration, timing, and bus protocols (2 lectures)

Laboratory projects:

- Introduction project on understanding the programming environment on a target development board.
- 3-4 small (1-2 weeks) assignments on programming and interfacing with various peripheral units.
- 2 medium (3-4 weeks) sized projects to build applications integrating multiple devices.

As shown in the syllabus, the course started with an introduction to the ARM architecture and instruction sets. We then discussed C programming for embedded systems which included accessing I/O registers, bit manipulation, C calling convention, and in-line assembly. The students used the ARM Software Development Toolkit (ARM SDT 2.02u) to develop and debug their assembly/C programs in an ARM instruction set simulator called an *ARMULATOR*.

Following the introduction to ARM architecture and programming, we presented the overall architecture of MX1 processor and the connection to peripheral interfaces. For the I/O interfaces and interrupt signals, we started the discussion with the general-purpose input/output (GPIO) and handshaking signals. Since most I/O functions and peripheral interfaces are multiplexed at the I/O pads, the lectures focused on the programming techniques for configuring I/O pins and functions. Similarly, interrupt multiplexing and configuration techniques were discussed, followed by interrupt vectors and ISR operations. This allowed us to look into each peripheral interface in subsequent lectures.

The peripheral interfaces covered in the class included a timer, pulse-width modulator, UART, SPI, I2C, LCD controller, and touch panel controller. The lectures addressed the basic design principles, the internal register configuration of the peripheral interfaces, and interrupt mechanisms. The timing diagrams of the signal waveforms at I/O pins were discussed to illustrate the interaction of programming model and device operations. In addition, the schematics of the MX1ADS development board were used to show the connections of MX1 processor with external interface circuits and devices. While discussing LCD and touch panel controllers, the lectures also encompassed general raster display devices and A/D converters.

After discussing the selected peripheral interfaces and the programming techniques, the lectures focused on the memory structure of microprocessor systems. Both the abstract model and physical memory architecture of the SRAM and DRAM were explored. We paid special attention to synchronous DRAM, their timing characteristics, and access modes. We used the Micron MT48LC32M8A2 as an example of SDRAM.

The interconnection mechanism of microprocessor systems is also an important subject of the course. We focused on the bus architecture and the protocols of PC's XT, AT, ISA, and PCI buses. The general bus designs, including synchronous/asynchronous, bus arbitration, and block transfer were also covered. The final topic covered optimization techniques of bus performance such as pipelined transfers and split transactions.

3.3. Hardware Platform for Lab Projects

Although our goal was to teach the general principles of the microcontroller architecture and system design, we desired to have a target platform available to students to use for experimentation. We decided to use a 32-bit RISC platform instead of a traditional 8-bit architecture such as the Intel 8051 and Motorola 6811. There were three motivating factors in choosing a 32-bit RISC architecture over an 8-bit architecture. First, we wanted to use a current technology so that students would be well prepared for a career in the embedded systems industry. Second, we wanted to introduce multiple peripheral devices and bus technologies that were only available on 32-bit architectures. And finally, we had received a large endowment from industry partners to provide equipment and classroom support for the 32-bit architecture.

The target hardware platform had to include a high performance SoC microprocessor for which popular interfaces were available and configurable. To acquire additional support to build the experimental environment, we contacted the Motorola's Dragonball University Program, sponsored by Motorola SPS in 2003. The University Program considered our approach for software/hardware integration as an effective instructional method for embedded systems software development, and donated thirty Dragonball MX1 development boards (MX1ADS) for our lab. Motorola also agreed to provide all necessary technical support to expedite the installation of lab equipment.

To facilitate various projects, the SoC-based development boards are accompanied with a peripheral board on which various devices are installed. Figure 1 depicts a typical development system that enables programming development for different I/O projects.

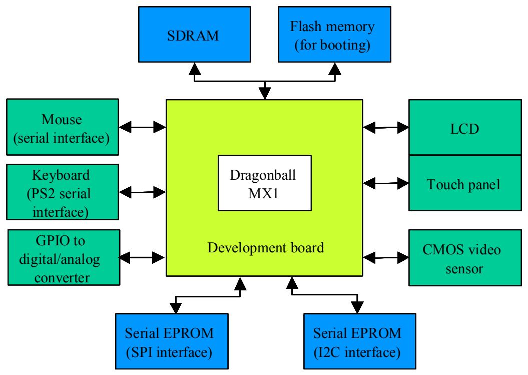

SDRAM
Flash memory (for booting)
Mouse (serial interface)
Keyboard (PS2 serial interface)
GPIO to digital/analog converter
Dragonball MX1
Development board
LCD
Touch panel
CMOS video sensor
Serial EPROM (SPI interface)
Serial EPROM (I2C interface)

Figure 1. The target development system for lab assignment

3.4. Software Platform for Lab Projects

Embedded Linux was chosen as the software platform on the MX1ADS boards. The fine modularity of Linux components allowed us to customize the Linux kernel for the course. Only the device drivers required to boot the target board were kept in the Embedded Linux build. This enabled students to load their drivers as modules. Additionally, Linux provided a rich set of freely available debugging tools and environments, such as *printk, strace, gdb*, *ksymops*, and *klogd*. With MontaVista's Linux, we established the software development environment shown in Figure 2.

Influence from industrial trends also played a

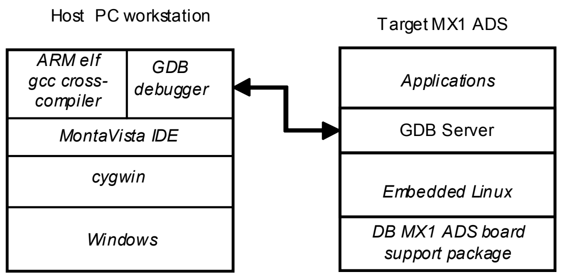


Figure 2. The target software development environment for MX1 ADS

significant role in our decision to use Linux. Currently, Linux is one of the preferred choices in the embedded system industry due to the availability of kernel source code without loyalties. This has lead toward recent trends of Linux becoming a dominant platform in embedded controllers. According to a survey conducted by the Venture Development Corporation, the estimated worldwide shipments of embedded Linux operating systems, add-on components, and related services reached over $60.0 million in 2003. This number is projected to reach over $115 million in 2006 [4].

In the target environment, students test their software components to manage peripheral devices. Since the I/O addresses are a part of the kernel address space and are protected, software components are developed as loadable device drivers modules. User applications use the drivers through standard file operations such as *open, close, read, write,* and *ioctl*. Interrupt service routines can also be registered as the modules are installed. This approach is quite attractive since the software for hardware interfaces are modular and embedded as a part of the operating system to support user applications. For students who have not taken any operating system courses, it may be challenging to comprehend the software structure and kernel APIs, and to develop kernel modules.

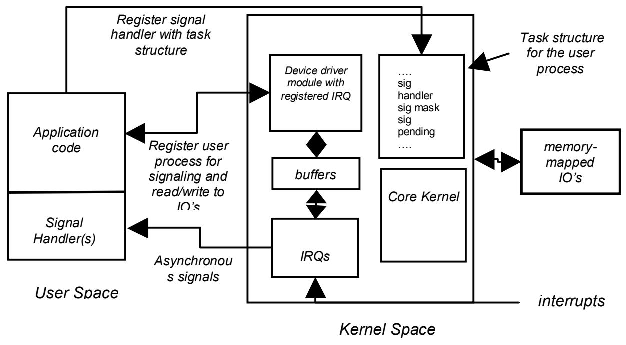

Register signal handler with task structure
Task structure for the user process
Device driver module with registered IRQ
....
sig
handler
sig mask
sig
pending
....
Application code
Register user process for signaling and read/write to IO's
buffers
memory-mapped IO's
Core Kernel
Signal Handler(s)
IRQs
Asynchronous signals
User Space
interrupts
Kernel Space

Figure 3. A pseudo driver for exercising kernel I/O address space and interrupts

To assist students with Linux specific driver development, we provided several example driver modules to illustrate the interactions between user applications and device drivers. One example is a pseudo driver, shown in Figure 3, which allows a user application to access memory locations in the I/O address space. When read or write functions are called, a command structure consisting of an I/O address and a data field is passed from the user application to the driver. The driver then reads from or writes to the I/O address. Hence, the student's application program can manipulate and access various control and status registers of peripheral controllers. To illustrate interrupt-driven data transfer, we added a ring buffer in the pseudo driver with which I/O data can be saved for subsequent read calls. Blocked driver function calls and the interaction with ISRs are demonstrated using a wait queue, *interruptible_sleep_on,* and *wake_up_interruptible* kernel functions. In addition, the pseudo driver makes use of asynchronous notification to emulate interrupts to user application programs. An ISR can invoke *kill_fasync* to signal a user application handler once it is registered. The signal handler can then take an action or pass the status changes to the main program. This pseudo driver also provides a great example to build character device drivers for some peripheral devices.

3.5. Sample Projects

To reduce the learning curve on Linux device driver development models and Linux kernel application programming interfaces (API), we provided a driver framework for each assignment. This allowed the students to concentrate on writing the hardware/software interface code rather than worrying about Linux's internal device driver interface. For example, the following segment of code is part of the driver framework we provided to students to develop a timer driver.

```
int init_module()
{
   int result;
   /* register our character device */
   result = register_chrdev(IO_major, driverName,  &IOBridge_fops);

   if (result < 0) {
      printk("<1>%s: Can't get major %d\n", driverName, IO_major);
      return result;
   }

   if (IO_major == 0)
   {
      IO_major = result;
   }

   // initialize hardware timer
   timer_init();

   // Register timer interrupt from the kernel.
   if (request_irq(TIMER_IRQ, timerISR, 0, "Timer2", NULL)) {
      printk("<1> Unable to get IRQ for Timer 2\n");
     unregister_chrdev(IO_major, driverName);
      return -EBUSY;
   }
   return 0;
}

void cleanup_module()  /* This function is called when we do rmmod. */
{
   printk("<1>Freed %s\n", driverName);
   free_irq(TIMER_IRQ, NULL);
   unregister_chrdev(IO_major, driverName);
}

void timer_init() {
}

void timerISR(int irq, void *dev_id, struct pt_regs *reg) {
}
```

In terms of projects, the platform enabled many development assignments with peripheral device controllers and hardware configurations. The following lists some sample projects given in the Fall of 2004.

1. Measurement of execution of the CRC-32 procedure with a hardware timer. The measurement was done in the eLinux environment on MX1ADS target board using MontaVista's DevRocket IDE on a Windows PC or Linux workstation.
2. Development of an interrupt-driven mouse driver for a serial mouse. The project employed a Microsoft 2-button serial mouse (Version 2.0A) attached to UART serial port. The driver compiles three mouse movement data packages and then reports any movement to the user applications.
3. Development of a driver for an external memory device. A Microchip 25LC640 EEPROM which consisted of 256 32-byte pages (or blocks) was used. The EEPROM contained an SPI interface. Hence, all commands and data transfer operations are done via a SPI bus controller. The project introduced students to the important concept of timing in device driver programming.

For the first project, we provided a Linux character driver capable of writing and reading registers on the target board. The students were tasked with developing an application to measure the execution time of a given program by using the hardware timer. This assignment introduced students to the Linux device driver model and software-hardware interface.

Next, the serial mouse driver project allowed students to apply their theoretical understanding of UART to develop an interrupt driven mouse driver. The driver uses an asynchronous I/O signal to communicate between the application and device driver in the kernel. We provided a framework for asynchronous I/O implementation in the Linux device driver.

The overall goal of the assignments was to reinforce classroom learning by providing the students with interesting projects. This gave them a greater understanding of theoretical concepts and a feeling of satisfaction upon completion of the projects [2].

4. Outcome and Evaluation

At the end of the semester, we surveyed the students about their learning experience. Twenty-eight out of forty-four students responded to the survey (64%). The survey questions are grouped into five categories: C programming, the Linux development environment, system architecture and system-level design, peripherals and projects, and overall satisfaction.

According to the survey, over 80% of the students agreed their understanding of C programming language has increased and that they were comfortable with developing device drivers using C. Even though the students were not familiar with the tools and development platform we used in class, we found that they were able to learn them quickly. About 73% of the students suggested that they were able to use the tools effectively at the end of semester.

The most challenging issue was the lack of proficiency in C programming and Linux development environments. We are planning to integrate a Linux environment in some prerequisite classes and add more emphasis on C in basic programming courses in the future.

5. Conclusion

Similar to many computer engineering curriculums, the microprocessor system design course at ASU has focused on teaching hardware/software interfacing and the management of peripheral devices. The previous approach of using assembly language and microcontroller-based platforms had been in place for more than a decade. It allowed the students to appreciate machine level processor operations and hand optimization to achieve the efficiency of assembly programs. However, with the advent of modern software development tools and the wide-spread use of embedded systems applications, a change in course material becomes inevitable.

There are a few important initiatives used in our approach for the microprocessor system design course. First, the use of assembly language for software development to control peripheral interfaces should be minimized. Students must be able to assess the cases where the use of assembly code can be justified. This would include encapsulating assembly code in well-defined interfaces and incorporating the code in software components as required. Second, the use of a broad set of peripheral interfaces including serial buses, LCD controller, touch panel, and data acquisition should be introduced. Finally, a practical software development and execution environment should be utilized so that students can gain familiarity with modern tools to build structured software components for embedded applications.

With these initiatives, the microprocessor system design course was transformed and introduced in the Fall of 2004. It was anticipated that knowledge gaps would exist in some of the prerequisite courses. Hence, we assumed that students may encounter difficulty with the required learning curve. However, we were surprised and satisfied with students' reception to the course. In general, students were excited about the new course structure, the updated learning environment, and the challenging projects, although complaints over the large amount of manuals and data sheets still existed. Overall, we believe this course was successful and we look forward to the development of the more advanced courses in the Embedded Systems curriculum.

6. References

[1] Gerald C. Gannod, et. al., "A Consortium-based Model for the Development of a Concentration Track in Embedded Systems", *Proceeding of the 2002 American Society for Engineering Education Annual Conference & Exposition.*

[2] Chris Hudson, "Teaching Microcontroller Technology – learning through play", *IEEE International Symposium on Engineering Education: Innovation in Teaching, Learning and Assessment*, Volume: Day 1, 4 January 2001.

[3] David C. Pheanis, "CEINT Internship Program", *33rd ASEE/IEEE Frontiers in Education Conference,* November 2003.

[4] Chris Lanfear, Steve Balacco, "The Embedded Software Strategic Market Intelligence Program, 2004", Venture Development Corporation, July 2004.

[5] Seongsoo Hong, "Embedded Linux Outlook in the PostPC Industry", *Proceeding of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2003.

[6] Frank Vahid, "Embedded System Design: UCR's Undergraduate Three-Course Sequence", *Proceedings of the 2003 IEEE International Conference on Microelectronic Systems Education*, 2003

[7] Naehyuck Chang and Ikhwan Lee, "Embedded System Hardware Design Course Track for CS Studnets", *Proceeding of the 2003 IEEE International Conference on Microelectronic Systems Education,* 2003.

[8] Shlomo Pri-Tal, John Robertson, Ben Huey, "An Arizona Ecosystem for embedded Systems", *IEEE International Conference on Performance, Computing, and Communications*, 4-6 April 2001.

[9] Konstantin Boldyshev, "Linux Assembly HOWTO," http://www.linuxselfhelp.com/HOWTO/Assembly-HOWTO/index.html.