Multi-processor programming in the embedded system curriculum

Proceedings

2008 Workshop on Embedded Systems Education

WESE 2008

Editors

Jeff Jackson
Peter Marwedel
Kenneth G. Ricks

Atlanta, Georgia, USA

October 23, 2008

Organizers' Message

We are pleased to present this fourth workshop on embedded system education associated with the 2008 Embedded Systems Week.

The program committee and additional reviewers selected seven regular papers and one keynote paper for inclusion in this workshop from international academic authors. We would like to thank the program committee members for their time and efforts in inviting papers, organizing reviewers, and for providing general assistance to the workshop organization. Without their efforts the workshop would not have been possible. We would like to thank the authors for their manuscript submissions and their timely response to implement improvements suggested by the reviewers. We would like to also thank the ESWEEK organizers and their support for the WESE2008 workshop. Finally, we would like to thank ARTIST for their support.

There exist many opportunities for international cooperation in the community of embedded systems researchers and educators and we look forward to those efforts.

Jeff Jackson
Peter Marwedel
Kenneth G. Ricks

Organizers

Jeff Jackson, The University of Alabama, USA
Peter Marwedel, University of Dortmund, Germany
Kenneth Ricks, The University of Alabama, USA

Program Committee

J. W. Bruce, Mississippi State University, USA
Alex Dean, North Carolina State University, USA
Martin Grimheden, Royal Institute of Technology, Sweden
Yann-Hang Lee, Arizona State University, USA
Sin Ming Loo, Boise State University, USA
Jogesh K. Muppala, The Hong Kong University of Science and Technology, Hong Kong
Victor Nelson, Auburn University, USA
Falk Salewski, Aachen University, Germany
Stewart Tansley, Microsoft Corporation
Martin Törngren, Royal Institute of Technology, Sweden
Chi-Sheng (Daniel) Shih, National Taiwan University, Taiwan
Shiao-Li Tsao, National Chiao Tung University, Taiwan
Frank Vahid, University of California Riverside, USA
Wayne Wolf, Georgia Tech, USA

2008 Workshop on Embedded Systems Education

WESE2008

October 23, 2008, Atlanta, Georgia, USA

October 23, 2008

9:00	**Opening**
9:10	**Keynote Address**

Frank Vahid: *Timing is Everything – Embedded Systems Demand Early Teaching of Structured Time-Oriented Programming*

10:00	**Coffee Break**
10:30	**Session I**

Dennis Brylow: *Nexos: A Next Generation Embedded Systems Laboratory*
Patrick Schaumont: *Hardware/Software Co-design is a starting point in Embedded Systems Architecture Education*
James R Vallino: *Interdisciplinary Teaming as an Effective Method to Teach Real-Time and Embedded Systems Courses*

12:00	**Lunch**
13:20	**Session II**

Andreas Hansson: *Multi-Processor Programming in the Embedded System Curriculum*
Sin Ming Loo: *Use of Discrete and Soft Processors in Introductory Microprocessors and Embedded Systems Curriculum*

14:20	**Minibreak**
14:30	**Session III**

Susan Lysecky: *eBlocks – Embedded Systems Building Blocks to Enable Project-Based Learning*
Scott Sirowy: *Virtual Microcontrollers*

15:30	**Coffee Break**
16:00	**Roundtable Discussion I**

Workshop attendee roundtable discussion: *What can be done to expand collaborative efforts in Embedded Systems Education?*

17:00	**Close**

Table of Contents

Organizers' Message ii
Organizing and Program Committee Members iii
Program iv

Timing is Everything – Embedded Systems Demand Early Teaching of Structured Time-Oriented Programming 1
Frank Vahid, University of California, Riverside, USA; *Tony Givargis, University of California, Irvine, USA*

Nexos: A Next Generation Embedded Systems Laboratory 10
Dennis Brylow, Marquette University, USA; *Bina Ramamurthy, SUNY Buffalo, USA*

Hardware/Software Co-design is a starting point in Embedded Systems Architecture Education 18
*Patrick Schaumont***,** *Virginia Polytechnic Institute and State University, USA*

Interdisciplinary Teaming as an Effective Method to Teach Real-Time and Embedded Systems Courses 25
James R Vallino and Roy S Czernikowski, Rochester Institute of Technology, USA

Multi-Processor Programming in the Embedded System Curriculum 33
Andreas Hansson, Benny Akesson, and Jef van Meerbergen, Eindhoven University of Technology, The Netherlands

Use of Discrete and Soft Processors in Introductory Microprocessors and Embedded Systems Curriculum 41
Sin Ming Loo and C. Arlen Planting, Boise State University, USA

eBlocks – Embedded Systems Building Blocks to Enable Project-Based Learning 49
Anuradha Phalke and Susan Lysecky, University Of Arizona, USA

Virtual Microcontrollers 57
Scott Sirowy and David Sheldon; *University of California, Riverside, USA*; *Tony Givargis, University of California, Irvine, USA*; *Frank Vahid, University of California, Riverside, USA*

Timing is Everything – Embedded Systems Demand Early Teaching of Structured Time-Oriented Programming

Frank Vahid
Department of Computer Science and Engineering
University of California, Riverside, USA
Also with the Center for Embedded Computer Systems, UC Irvine
vahid@cs.ucr.edu

Tony Givargis
Center for Embedded Computer Systems
University of California, Irvine, USA
givargis@uci.edu

ABSTRACT

Computing was originally dominated by desktop and hence data-oriented systems. However, embedded and hence time-oriented systems, which must measure input events or generate output events of specified time durations, or must execute at regular time intervals, are increasingly commonplace. Blinking a light on and off for 1 second represents a "Hello World" example of a time-oriented system. Time-oriented programming differs significantly from the more common data-oriented programming, and developing correct maintainable time-oriented programs is challenging. The current situation of embedded courses being senior-level courses hampers effective teaching of time-oriented programming, as early-learned programming habits can be hard to break. Early freshmen or sophomore-level introduction of time-oriented programming, involving the right balance between abstractions and resource awareness, may provide a better foundation. A clean microcontroller with a timer, coupled with the synchronous state machine computation model, can provide such a balance.

1. INTRODUCTION

The Rise of Embedded Systems

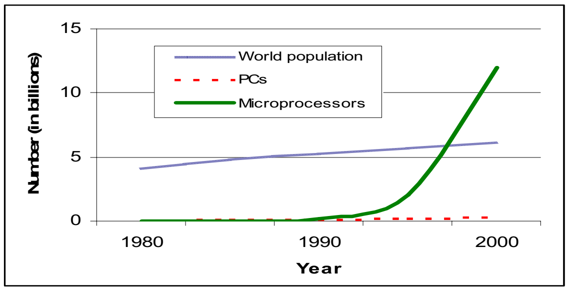
Embedded systems include computing systems that interact extensively with physical real-world devices. Examples are consumer electronics (cameras, cell phones, portable games), automotive electronics (cruise control, navigation), communications equipment (base stations, network routers), factory automation equipment (robotics, sensors, inventory control systems), office automation equipment (scanners, copiers, printers), medical devices (pacemakers, ventilators, ultrasound machines), home automation (security systems, temperature control, smart appliances), and much more. The decreasing cost and size and the increasing performance of computing chips, following Moore's Law, have led to a dramatic proliferation of embedded systems in recent decades, as indicated by the tremendous growth in numbers of microprocessors worldwide shown in Figure 1. Novel embedded systems applications are introduced at a rapid rate, including items like smart ingestible pills, household robots, and bodily-worn health monitoring networks. Of the approximately 150,000 U.S. patents granted per year, roughly 10,000-20,000 are embedded systems related [14].

Current Courses: Details, Details, Details

The teaching of embedded system programming, however, has progressed only moderately in the past two decades, and lacks a solid discipline. Many courses from the 1980s were oriented around an embedded processor chip known as a microcontroller, which has a small low-cost microprocessor coupled with key embedded processing features including program-accessible input/output pins, timers, serial communication devices (UARTs), analog-digital converters, and other peripherals. These small resource-constrained devices required detailed assembly programming and electronics knowledge to set up and use the microprocessor chip and its various peripherals. Many such courses have since replaced assembly programming by C programming and use some libraries to elevate the programmer's focus, and many have adopted the "embedded systems" name. However, most such courses continue to emphasize microcontroller-specific programming and electronics details necessary to get a basic embedded system working, with little attention paid to higher-level embedded system design concepts. This is akin to driver education emphasizing how to add gasoline to a car, check the tires, adjust the mirrors, start the car, and go forward and stop, rather than emphasizing higher-level concepts like how to maintain defensive distances, approach intersections, or plan routes.

Instructors are not to blame for the low-level emphasis. Setting up and running embedded systems courses is *hard*. Unlike desktop computing courses in which fairly standard and stable platforms and tools exist, embedded systems courses must deal with a rather chaotic technical landscape. Dozens of widely-used microcontroller families exist, including 8-bit devices like the Intel 8051, Motorola 68HC05, 68HC08, 68HC11, Microchip PIC, Atmel AVR, Zilog Z80, and much more. For each family, dozens and sometimes hundreds of variations exist (different numbers of pins, size of on-chip RAM, support for external memory, etc.), produced by tens of different companies. 16-bit and 32-bit devices are also available and have similar variety. For a chosen device, a physical programming device (which downloads machine code into the physical chip) must be found. Assemblers and cross-compilers must be found that target the particular device. Development boards must be made or purchased that support the device, such that the device can interface with buttons, switches, LEDs, displays, and other

Figure 1: Growth in number of microprocessors worldwide in past decade due to growth in embedded systems market.



components. Simulators or debuggers may be incorporated. Lab assignments must then be developed around this multitude of items, working around the various items' bugs, sensitivities, or other imperfections (of which many exist). Hardware parts suffer damage (e.g., a microcontroller chip may burn out just by inserting it backwards into a programmer device, or an ageing programmer device may fail to consistently program chips correctly), requiring continual troubleshooting and correction. Furthermore, microcontrollers are tough devices to work with – due to historical artifacts or mass-production needs, the devices tend to require extensive configuration for a particular purpose. Due to each device or tool having a smaller audience than a PC or Windows-based C compiler, documentation is usually scarce (and is sometimes wrong), and simulation and debug tools are scant and often with bugs too. New chips, platforms, operating systems, compilers, debuggers, and even companies providing such items, come and go every few years.

Given the challenging nature of building working embedded systems, instructors often focus primarily on teaching the student the details of how to use the myriad software and hardware tools, to configure and use the microcontroller and each of its key peripherals, to understand how interrupt service routines interact with a main program, and other details necessary to build functioning systems. Figure 2 gives some idea of the challenge, showing the initialization code required to set up a particular microcontroller for a particular usage; note the many distinct items that must be configured (and one small mistake may cause the system to fail). After that teaching process, which may take months, a course may have just enough time left for a student to build an interesting project, with little or no time for teaching a discipline of embedded systems programming. If a second embedded systems course does exist, the course is typically a project course rather than a course that teaches a disciplined embedded programming approach.

Figure 2: Sample C initialization code for a particular microcontroller – extensive knowledge of details is necessary to properly configure a microcontroller for a particular use.

```
// ------------------------------
// configure output ports
// ------------------------------
ADCON0 = 0x00; // disable A/D converter
CM1CON0 = 0x00;
CM2CON0 = 0x00;//disable comparators */
ANSELH = 0x00;
ANSEL = 0x00; // configure pins as digital channels
TRISA = 0x08; // all bits output except RA3
TRISB = 0xF0; // Port B inputs
RABPU = 1;
WPUB4 = 1;    // enable weak pull ups on RB4
 IOCB4 = 1;   // enable interrupt on change for RB4
TRISC = 0x00; // PORTC all set to outputs
PORTA = 0x00;
PORTB = 0x00;
PORTC = 0x00; // initialize ports
// --------------------------------
// Timer0 setup
// --------------------------------
CLRWDT();   // turn off watch dog timer
OPTION = 0x07;    // setup prescaler
TMR0 = PRELOAD; // preload timer
T0IE = 1; //enable timer0 interrupts
// ---------------------------------
// Setup button interrupts
// ---------------------------------
RABIE = 1; //Enable change on PORTB interrupts
GIE = 1;       //global interupts enabled
```

As evidence of the low-level focus on modern embedded systems courses, consider the top-selling books in the embedded systems textbook market of 16,000 books per year, as reported by John Wiley and Sons: (1) The HCS12/9S12, An Introduction to Hardware and Software Interfacing, Huang, Delmar Cengage, 2005 – 12%; (2) The 68HC12 Microcontroller: Theory and Application (2nd edition of earlier book: Embedded Systems Design and Applications with the 68HC12 and HCS12), Barret and Pack, Prentice hall, 2004 – 9%; (3) Software and Hardware Engineering: Motorola M68HC12, Cady and Sibigtroth, Oxford University Press – 7%.; (4) Embedded Microcomputer Systems: Real Time Interfacing, Valvano, Int. Thomson Publishers, 2006 – 7%; (5) Microcomputer Engineering, Miller, Prentice Hall, 2003 – 6%; (6) Computers as Components: Principles of Embedded Computing System Design, Wolf, Morgan Kaufman, 2005 – 6%; (7) Embedded System Design: A Unified Hardware/Software Introduction, Vahid and Givargis, John Wiley and Sons, 2001 – 4%. Books 1-5 all focus on the details of particular microcontrollers. Some do address higher-level concepts, but typically do so late and rather lightly. Books 6 and 7 (the latter authored by ourselves) sought to introduce a higher-level discipline to embedded system design (in contrast to emphasizing programming).

Because learning the myriad details of microcontrollers, interfacing, troubleshooting, etc., require rather sophisticated students, embedded systems courses are typically taught at the senior level, as illustrated in Figure 3. All of the above books are typically used in senior-level courses, and items 6 and 7 sometimes in a second embedded systems course or even graduate course. We performed a Google search for "embedded systems" in .edu sites; the first 20 courses we found were all upper-division or graduate level, with typical names being "Real Time Embedded Systems" or "Introduction to Microcontrollers."

Thus, a student of a modern embedded systems course may develop a rather myopic view of embedded programming, viewing it as a collection of low-level details and methods necessary to configure and use a microcontroller. "High-level"

Figure 3: Typical embedded system courses are senior-level, emphasizing myriad details. Adding time-oriented programming *early* in the training can provide a better foundation for such courses.

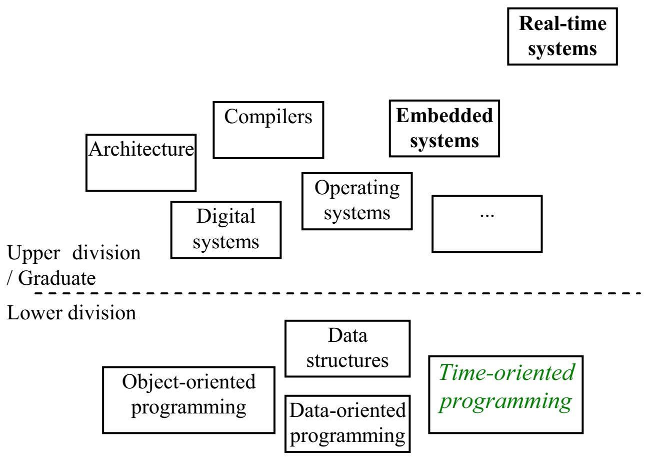
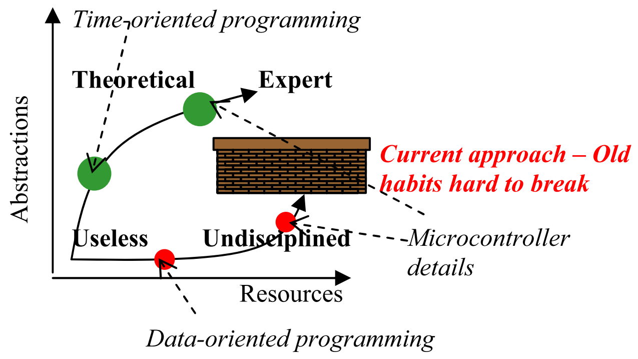


Figure 4: Current curricula teach years of data-oriented programming and then microcontroller details – leading to "undisciplined" embedded programmers having a hard time overcoming the wall of data-oriented programming habits. First teaching time-oriented abstractions establishes a theoretical foundation that may lead to better embedded programmers.


Time-oriented programming
Theoretical
Expert
Abstractions
Current approach – Old habits hard to break
Useless
Undisciplined
Microcontroller details
Resources
Data-oriented programming

may merely mean using C rather than assembly language. The net result is that today's professional embedded systems programmers create code that is amazingly ad hoc, being exceptionally difficult to maintain and often taking a long time to develop. The lack of discipline may explain in part why the majority of embedded systems project are completed late, on average 4 months late for the typical 14 month project [10]. Various companies with whom we interact, including Cisco, Broadcom, Western Digital, Pulmonetics (medical ventilators), Qualcomm, Freescale, Microsoft, and others, have commonly indicated that computing graduates, even those with computer engineering degrees and/or with embedded systems course and project backgrounds, lack the ability to program embedded systems due to "unawareness of resources," "no programming discipline," "inability to deal with time," "a habit of hacking," or even "excessive focus on objects or libraries," requiring "long periods of training" before such graduates can write good embedded code.

2. STRUCTURED TIME-ORIENTED PROGRAMMING

Old Habits are Hard to Break

Given the increasing importance and complexity of modern embedded systems, a more disciplined view of embedded programming is becoming essential (our definition of "disciplined" will be explained subsequently). We have been experimenting with the introduction of disciplined embedded programming methods for many years, and have concluded that an approach that attempts to introduce disciplined methods after an initial low-level microcontroller-details introduction, which is better of course than no introduction of disciplined methods at all, nevertheless is a sub-optimal approach. The reason is because students have spent several years learning data-oriented programming, leading to a perspective and a set of habits that can be hard to change, as illustrated in Figure 4. Instead, we claim that an approach that first introduces disciplined embedded programming methods and later teaches necessary low-level microcontroller details will lead to embedded system designers developing better programs, by creating an improved foundational perspective within the student.

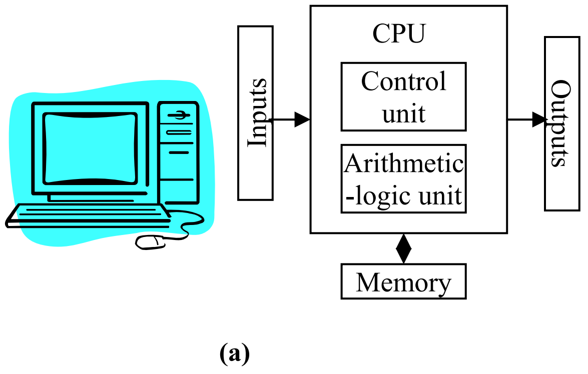
After over a decade of teaching embedded systems in various ways and interacting with dozens of embedded systems teachers and courses worldwide, the authors view *structured time-oriented programming* as one of the key features of a disciplined embedded programming approach. A characteristic of embedded systems, which distinguishes them from traditional desktop (including server) computing systems, is the orientation around the notion of *time*. For example, an embedded system equivalent of a "Hello World" program might repeatedly blink an LED (light-emitting diode) on for 1 second and then off for 1 second, requiring an explicit notion of real time ("1 second"). In contrast, desktop computing has focused on data-oriented programming emphasizing data transformation—reading input data, transforming the data, and outputting new data, with no notion of real time—even since the design of early computers, which was driven by data-oriented applications like processing census data and computing bomb trajectories. Whereas time is a behavioral consequence of desktop programs, time is part of the explicit functionality of embedded systems. Explicit functionality of embedded systems also involves the related notion of *events* – external actions that can occur at any time and to which the system responds. For example, an alternative "Hello World" equivalent might turn on an LED for 1 second every time that a button is pressed, the button press forming an input event. Some desktop programming does incorporate time and event concepts (e.g., blinking a cursor in a graphical display, or responding to mouse click events), but to a lesser extent than embedded programming. For simplicity, rather than always referring to both time and events, we will in this paper take the liberty of using the term time to refer to both concepts, although we realize there are distinctions between the two concepts.

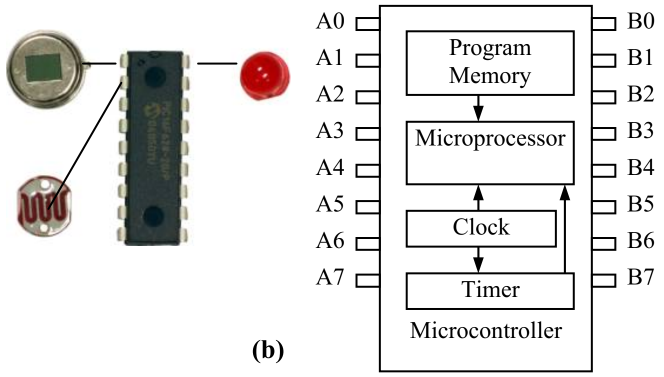
While some time-oriented programming courses exist today, they tend to build on advanced concepts of real-time operating systems (RTOSes) or of parallel programming languages, using extensive abstraction to hide many lower-level details. For our purposes, the appropriate balance must be found between abstracting away low-level details to enable focus on higher-level issues, while also exposing enough low-level details to ensure that programmers have resource awareness and can build systems using today's embedded microprocessors, which may or may not be running RTOSes. Indeed, our interactions with companies that produce embedded systems have revealed an intense desire from those companies for more computing professionals that have a much stronger understanding of underlying computing resources; embedded programmers who only know abstractions and never learned the details of underlying resources may produce grossly inefficient code and be unable to hammer out the details often necessary to get real systems completely working.

Virtual Microcontroller

We propose the use of a *virtual microcontroller* as a step towards achieving the appropriate balance between abstraction and resource awareness. Fundamental resources in time-oriented embedded programming of a microcontroller include a *microprocessor*, a *timer*, and an *interrupt service routine*. A main program can initialize and activate a timer, which in turn automatically calls at specified intervals an interrupt service

Figure 5: "Clean" computing platforms: (a) A simplified view of a computer provides the foundation for most desktop-programming courses, (b) likewise, a simplified view of a microcontroller can provide a good foundation for a time-oriented embedded programming course.





routine (ISR), pausing the main program's execution during such calls. Creating time-oriented programs using just those basic resources, without the aid of an RTOS, represents a fundamental embedded programming skill, akin to a surgeon learning to perform surgery with a scalpel but without the aid of modern robotic surgical tools. The programmer (or surgeon) develops an intuitive understanding, which not only may help when using the more advanced tools, but also enables competence even when the more advanced tools are unavailable.

The virtual microcontroller therefore consists of a simple microprocessor and a timer (along with required program memory and general-purpose I/O), as illustrated in Figure 5(b). All items are present in very straightforward form. For example, the I/O consists of eight inputs A0-A7 and eight outputs B0-B7 accessible by name in C or assembly (microcontrollers often have I/O that can be configured for either input, output, or both, or can serve as memory address/data lines instead, and thus require configuration). The timer is set in C by calling a predefined function called TimerSet(T) where T specifies the interrupt interval in milliseconds (most microcontrollers instead require extensive configuration of various registers, such as mode registers, frequency registers, prescaler registers, and more, to obtain a particular interrupt rate). This simplified, or "clean," computing platform is akin to the simplified platform commonly used in data-oriented programming courses, shown in Figure 4(a). The clean platforms provide a sufficient abstraction on which to program, introducing just enough resource concepts for solid understanding, but without overexposing the student to resource details.

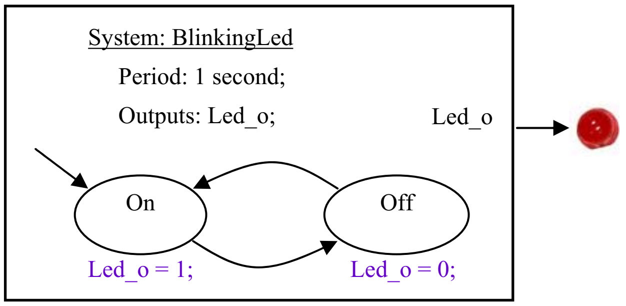
Synchronous State Machines – SynchSMs

While a virtual microcontroller provides a clean platform for developing and executing time-oriented code, the C language (which is used in a majority of embedded systems [10], though our discussion applies equally to C++, Java, or assembly language) is not particularly well-suited for time-oriented programming. Left alone to figure out how to implement timed behavior using a microcontroller with a timer and ISR, students develop an "impressive" variety of methods, with some putting most code in the main function and others putting most code in the ISR, some using sequenced code and others used heavily iterated code and global status variables, and so on.

Instead, we propose the use of *synchronous state machines*. Good computation models for data-oriented desktop processing include sequential programming or object-oriented programming models. For embedded processing, however, other computation models are common. In particular, state machines excel at modeling basic sequential control behavior, as well as more complex behavior. State machines (SMs) allow not only straightforward description of processing of events, but also provide an elegant basis for specifying timing. For programming purposes, a state machine can utilize declared variables and possess arithmetic operations and conditions, as in the FSMD model of [15]. In fact, each state can have a sequence of associated actions described using sequential programming constructs such as assignment statements, if-then-else statements, loops with constant bounds, and even (non-recursive) subroutines having such statements as described above. Such SMs are commonplace in hardware descriptions targeting synthesis to circuits.

State machines provide a solid basis for a disciplined approach to time-oriented embedded system programming. The authors have developed and utilized such approaches in their courses during the past several years. A simple approach involves using a synchronous SM, or *synchSM*, whereby all states last for a defined time interval known as a tick rate or period, as in Figure 6. Thus, if the synchSM's period is 1 second, then the shown synchSM will set output Led_o to 1 for 1 second, then to 0 for 1 second, then to 1 for 1 second, and so on, causing

Figure 6: SynchSM for the Blinking LED (time-oriented "Hello World" equivalent) example.



the LED to blink as desired.

SynchSMs mesh well with the basic microcontroller time elements. A programmer can set the timer's interrupt rate to the desired tick rate, and program the timer's ISR to raise a global flag. The programmer's main code, implementing the synchSM, can detect the raised flag (and reset it) and then proceed to the next state.

Students may be well served by being encouraged to work with the synchSM abstraction extensively on a wide variety of examples, to develop a solid perspective of describing timed behavior using a timed computation model.

Implementing SynchSMs in C – Models Matter, Not Languages

Of course, programs are typically written using C code (or similar), bringing us to another important concept in structured time-oriented programming. The concept is that of conceptualizing system behavior using a *computation model* (e.g., synchSMs), and then implementing that model in C using a straightforward template-based approach. A straightforward template approach is shown in Figure 7, wherein the main program waits for a tick, and then processes the state machine's transitions and state actions using switch statements. In contrast, today every programmer combines main code, timers, and timer ISRs in unique and creative ways. While creativity in some endeavors is a wonderful thing, creativity in programming makes programs harder to understand, debug, and maintain. A synchSM approach represents a highly-structured approach, akin to structured programming guidelines for data-oriented programming [12] but instead focused on time.

An additional advantage of the state machine approach is that multiple concurrent state machines can be handled via straightforward round-robin processing. Straightforward counter methods can handle synchSMs with different periods. Priority schemes among the synchSMs could be introduced too, leading towards the basic idea of RTOSes.

A similar approach can be used for other computation models common in embedded systems. Other common computation models in embedded systems include dataflow models, such as synchronous dataflow. With care, different models can even be implemented in a single C program on a single microcontroller.

Figure 7: Template approach for implementing a synchSM in C.

```
#define Led_o B0

int BL_Clk=0;
void TimerISR()
{
   BL_Clk = 1;
}

void main(void)
{
   enum BL_StateType {BL_On, BL_Off}
      BL_State;
   B=0;//Init outputs
   TimerSet(1000); // 1 second
   TimerOn();

   BL_State = BL_On;
   while (1) {
      switch (BL_State) {// State actions
         case BL_On:
            Led_o = 1;
            break;
         case BL_Off:
            Led_o = 0;
            break;
      } // State actions

      while (!BL_Clk); BL_Clk = 0;

      switch (BL_State) { // Transitions
         case BL_On:
            BL_State = BL_Off;
            break;
         case BL_Off:
            BL_State = BL_On;
            break;
      } // Transitions
   } // while(1)
} // main
```

3. TOOLS

Microcontroller Simulator

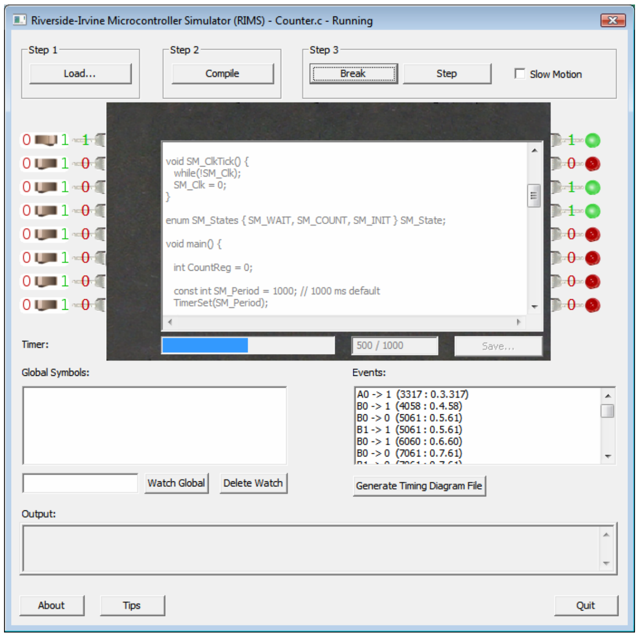
Appropriate tools are needed to support the proposed approach to teaching structured time-oriented programming. We are developing a tool, the Riverside-Irvine Microcontroller Simulator (RIMS), to support programming of the proposed virtual microcontroller, seen in Figure 8.

The tool graphically depicts a microcontroller with 8 input and 8 output pins. Concreteness is emphasized to ease the comprehension task for young students. Each input pin is connected to a switch that the user can move to the "0" or "1" position, and each output pin to an LED that illuminates when its pin is 1. The microcontroller's C program is shown inside the microcontroller. The user is shown a 3-step process: (1) Load a C program (or write one directly in the C window and save it), (2) Compile the program, (3) Run the program. The compiler is built into the tool (using the open-source LCC compiler [28]) and the generated assembly code is then executed by an instruction set simulator, which we wrote and built into the tool. The running program may be "break'ed" at any time, executed step-by-step, and have any symbol values viewed, as with a standard debugger. All events on input and output pins, in addition to being graphically shown, are printed to an event text display also.

To help teach the concept of the timer and timer ISR, a "status" bar shows the current value of the timer. Thus, as a program executes, the status bar fills, the ISR is called, and the status bar starts over. The student thus visually sees the "ticking" of the microcontroller's timer and the associated ISR call.

To further support the teaching of timing concepts, the textual event printout can automatically be converted to timing diagrams. Such conversion is accomplished by the RIMS tool generating standard VCD files, which can then be viewed using any of several VCD to timing diagram viewing tools, such as the freely-available Wave tool [44]. All of the above functionality (excluding the VCD viewer) exists as a single Windows executable, making installation and operation easy.

Figure 8: Screenshot of the virtual microcontroller simulation tool, RIMS.

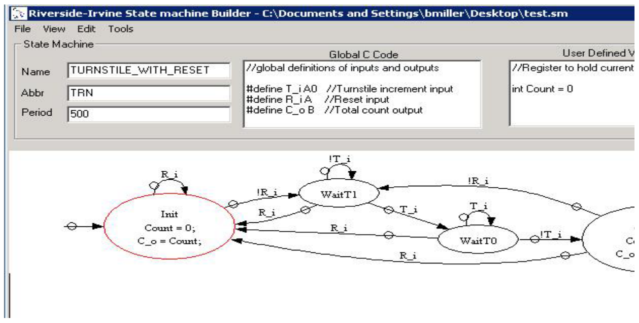


```
void SM_ClkTick() {
  while(!SM_Clk);
  SM_Clk = 0;
}

enum SM_States { SM_WAIT, SM_COUNT, SM_INIT } SM_State;

void main() {

  int CountReg = 0;

  const int SM_Period = 1000; // 1000 ms default
  TimerSet(SM_Period);
```

500 / 1000

```
A0 -> 1  (3317 : 0.3.317)
B0 -> 1  (4058 : 0.4.58)
B0 -> 0  (5061 : 0.5.61)
B1 -> 1  (5061 : 0.5.61)
B0 -> 1  (6060 : 0.6.60)
B0 -> 0  (7061 : 0.7.61)
```

Figure 9: Screenshot of the state machine builder tool, RISB.



Name: TURNSTILE_WITH_RESET
Abbr: TRN
Period: 500

```
//global definitions of inputs and outputs

#define T_i A0   //Turnstile increment input
#define R_i A    //Reset input
#define C_o B    //Total count output
```

```
//Register to hold current

int Count = 0
```

State machine builder

Because describing behavior as synchSMs is so central to our proposed approach, we are also developing a graphical synchSM capture tool – the Riverside-Irvine State machine Builder (RISB), a screenshot of which is shown in Figure 9. The tool allows a user to add states and transitions between states, to type actions for states and to type conditions for transitions, and to declare the state machine's name, period, and any variables. Although the user works with a graphical depiction of the state machine, all graphical placement and routing of the state machine is automatically handled by the open source GraphViz tool [16] that we encapsulated within RISB, allowing the user to focus on the state machine's functionality and not details related to the state machine's graphical display. The tool can automatically translate a state machine to C code. The C code can then be executed on RIMC, with the current state being automatically highlighted in RISB, to help students visualize the execution of the state machine. Such automatic translation to C emphasizes the fact that writing the C code is not a creative endeavor, and instead should follow strict, automatable, guidelines. As such, structured time-oriented programs result.

Physical prototypes

Although the above described framework allows for learning the structured time-oriented embedded programming approach, creating working physical implementations can help crystallize embedded concepts. Thus, mapping the virtual microcontroller to physical platforms may be desired by many instructors of courses having lab components. We are therefore mapping the virtual microcontroller to various physical platforms, and plan to publish instructions for such mapping that instructors and their teaching assistants can follow to create numerous instances for their labs[1]. We have thus far mapped our virtual microcontroller onto three physical platforms – an Atmel AVR microcontroller, a Xilinx Spartan FPGA board, and a desktop PC using a USB interface to an external breadboard for the general-purpose I/O pins. Figure 10(a) shows a mapping onto the AVR microcontroller using a breadboard approach. Students could add input and output components, perhaps using a second board, to interact with the I/O. Figure 10(b) shows the same mapping but in a self-contained box where all inputs are connected to switches and outputs to LEDs, allowing for standalone operation of simple embedded programs having 8 input switches and 8 output LEDs – the same as the RIMS PC-based visual simulator. Figure 10(c) shows mapping to an FPGA board (at the bottom of the picture). For any of these implementations, additional display functionality can be added using the VMC's serial UART, which can be connected through a PC's serial port to output on a serial terminal program, or connected through a custom serial-to-VGA device (which we have designed) to output directly to a monitor.

To simplify the process of mapping programs to these physical implementations, we have developed an approach whereby a textual assembly file can be downloaded onto a USB stick, and that USB stick then plugged into the physical implementation for uploading to the VMC, as illustrated in Figure 10(b). Such an approach makes clear to the student what program is being executed, and enables easy migration of different programs to a physical platform or from one platform to another. We implemented the necessary USB read functionality and the just-in-time assembler for this textual assembly USB file approach, resulting in fully-functioning physical implementations.

4. RELATED WORK

An earlier section described the main existing embedded systems courses that are the target of the project. Several other items can be considered related to this project also.

Another category of courses emphasizes real-time systems, typically based on an RTOS. Jian [23] presents a survey of the state of real-time systems education, showing that very few universities offer courses in the topics. Schwarz [35] observes

[1] Note: We do not plan to provide physical prototypes to other universities, but rather descriptions of how to build them for a few common physical platforms, which other universities can use or adapt.

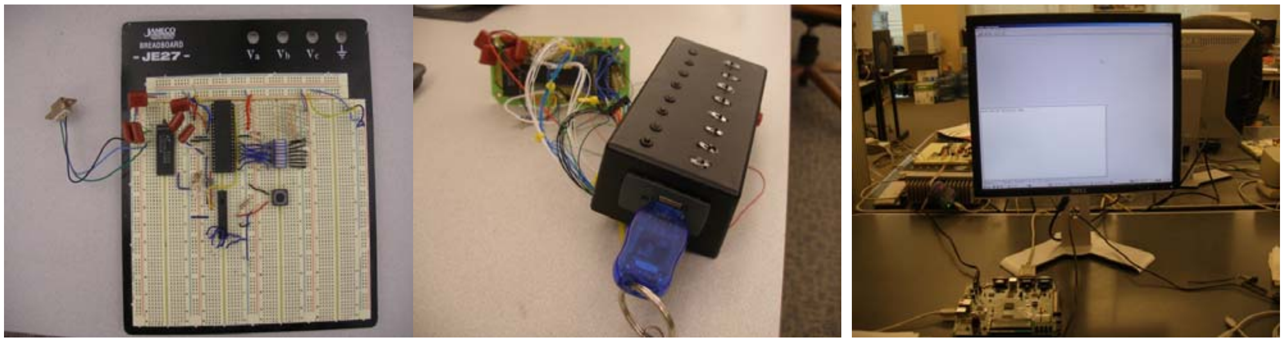


Figure 10: Virtual microcontroller implementations: (a) on an AVR microcontroller breadboard, with input/output wires that can be connected to other circuits, (b) in a black-box, with internal AVR-microcontroller-based circuitry exposed, and (c) on a Xilinx Spartan 3E FPGA, which happens to also use a connection to provide additional output to a terminal. All three can execute the same virtual microcontroller program identically.

(a) (b) (c)

that undergraduate real-time course offerings vary and lack any accepted standard of contents or discipline. The textbook by Burns [8] presents real-time systems and the programming languages that work well with them, used primarily at the senior or graduate level. Kornecki [26] argues that universities do not pay enough attention to practical software development in the field of time-oriented reactive programming. ACM's curriculum guidelines make only short mention of the subject [1]. A number of real time operating systems have been introduced to provide a higher level of abstraction between the application software and embedded hardware, including the open source eCos [13], and VxVorks and RTLinux from WindRiver [45].

Much progress has been made in the past two decades on capturing embedded system functionality. These include several event-based languages. Esterel [6] is a synchronous programming language for the development of complex reactive systems. The notion of time is replaced with the notion of order, called the multiform notion of time, which means only simultaneity and presence of events are considered. Statecharts [17][18] is a visual formalism for complex systems. Statecharts extend traditional state diagram transition diagrams with notions of hierarchy, concurrency, and communication. They also include some timing mechanism (e.g., "timeout" events). Numerous other languages can be viewed as supporting event/time based descriptions, including VHDL [21], Verilog [41], Rapide [32], LOTOS [7], CSP [19], Ada [39], and more.

Several approaches focus extensively on time. Kopetz [25] presents the time-triggered architecture, which is a computing infrastructure for the design and implementation of dependable distributed embedded systems. Applications are decomposed into autonomous clusters each with a fault-tolerant global time base. The global time base simplifies communication and guarantees timeliness of real-time applications. Commercial companies like TTTech [40] stemmed from this work. Lamport [27] discusses time-oriented programming in the context of events in a distributed system and how to synchronize items. Ouimet [33][34] introduces timed abstract state machines. ASMs consist of a set of mutually exclusive rules each guarded by a condition of variables. A rule whose condition at a given execution step will update external and internal variables. Timed ASMs involve timing extensions to ASMs, such as specifying the time duration of a rule's execution (which may be a range; a specific value in the range will be randomly chosen during runtime).

Other related approaches emphasize models of computation. The Polis framework [4] defines codesign finite state machines (CFSMs) as a formal model for communicating FSMs to which a system captured in a language like Esterel can be translated for analysis and synthesis. Lee [31] presents a framework for comparing models of computation, including Kahn process networks, dataflow, sequential processes, concurrent sequential processes with rendezvous, Petri nets, and discrete-event systems. Other works [24][38] have focused on quantitatively comparing various computation models, specifically targeting parallel models. Jeukens [22] investigates how best to use various computation models to design increasingly complex systems. Andrews [3] investigates how best to leverage computation models for hybrid CPU-FPGA platforms. Lee [29] discusses various requirements of future embedded programming, including time concepts.

Most of the above involve advanced concepts and thus not ideal for introductory undergraduate courses. Increasing attention is being paid to embedded systems education, through workshops (e.g., the Workshop on Embedded Systems Education, WESE, held since 2005), special issues of journals, and numerous special sessions and papers appearing in mainstream research venues. Numerous research projects attempt to improve engineering education. Hodge [20] introduces the concept of a Virtual Circuit Laboratory, a virtual environment for a beginning electrical engineering course that mimics failure modes in order to aid students in developing solid debugging techniques. The environment not only provides a convenient test environment, but also allows an instructor to concentrate more on teaching. Butler [9] developed a web-based microprocessor fundamental course, which includes a Fundamental Computer that provides students in a first year engineering course a less threatening introduction to microprocessors and how to program.

Other researchers have concentrated on developing or evaluating computing architectures for beginning students or non-engineers. Benjamin [5] describes the BlackFin architecture, a hybrid microcontroller and digital signal processor. The architecture provides a rich instruction set based on MIPS with variable width data, and parallel processing support. Ricks [9] evaluates the VME Architecture in the context of addressing the need for better embedded system education. The Eblocks project [11] concentrated on developing sensor blocks that people without programming or electronics knowledge could connect to build basic customized sensor-based embedded systems.

Much research has involved virtualization [30][36], with several commercial products developed in response to the need for portable virtual machines. VMware [43] and the open source product Xen [46] concentrate on developing virtual machines that allow the end-user to run multiple operating systems concurrently. The Java Virtual Machine [37] allows the programmer to write operating system independent code, and tools like DOS Box and console emulators allow the user to run legacy applications in modern operating systems.

5. CONCLUSIONS

Embedded systems have experienced massive growth in the past two decades. Embedded system education has improved, but exists largely as a late add-on to a traditional data-oriented computing curriculum, leading to ad hoc modified data-oriented programming methods rather than a disciplined method targeted to the time-oriented nature of embedded systems. Early introduction of a structured time-oriented programming approach may help. Key features include a virtual microcontroller that exposes just enough low-level resources along with higher-level abstractions, a synchronous state machine (synchSM) computation model for explicit time-oriented programming and a clear method for capturing synchSMs in the prevailing C language, a set of easy to use tools that support synchSM and C capture, compilation, simulation, and debugging, and the ability to readily develop physical prototypes of a programmed virtual microcontroller. These items may catalyze adoption of early teaching of time-oriented programming, which we hope to see become part of a standard computing curriculum in the coming decade.

6. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation (CNS-0614957).

References

[1] Association for Computing Machinery. Computing Curricula 2005. http://www.acm.org/education/curricula-recommendations

[2] Alice Programming Environment, http://www.alice.org.

[3] ANDREWS, D., NIEHAUS, D., JIDIN, R., FINLEY, M., PECK, W., FRISBIE, M., ORTIZ, J., ED KOMP. AND ASHENDEN, P. Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link. IEEE Micro. On page(s): 42- 53, Volume: 24, Issue: 4, July-Aug. 2004

[4] BELARIN, F., M. CHIODO, H. HSIEH, A. JURESKA, L. LAVAGNO, C. PASSERONE, A. SANGIOVANNI-VINCENTELLI, E. SENTOVICH, K. SUZUKI, AND B. TABBARA, Hardware-Software Co-Design of Embedded System: The POLIS Approach Norwell, MA: Kluwer, 1997..

[5] BENJAMIN, M., KAELI, D., AND PLATCOW, R. Experiences with the Blackfin Architecture in an Embedded Systems Lab.. WCAE '06

[6] BERRY, G. AND GONTHIER, G. The ESTEREL Synchronous Programming Language: design, semantics, implementation. *Sci. Comput. Program.* 19, 2 (Nov. 1992), 87-152.

[7] BOLOGNESI, T. AND BRINKSMA, E. Introduction to the IS0 specification Language LOTOS.. Amsterdam: North-Holland, 1989, pp. 23-73.

[8] BURNS. A. AND WELLINGS, A. Real-Time systems and Programming Languages. Third Edition. Pearson Education Limited. 2001.

[9] BUTLER, J. AND BROCKMAN, J. Web-based Learning Tools on Microprocessor Fundamentals for a First-Year Engineering Course. 2003. Proceedings of the American Society for Engineering Education.

[10] CMP. Embedded Systems Design State of Embedded Market Survey, 2006, http://www.embedded.com/columns/survey.

[11] COTTRELL, S. AND F. VAHID. A Logic Enabling Configuration by Non-Experts in Sensor Networks. Conference on Human Factors in Computing. 2005

[12] DIJKSTRA, E. Notes on Structured Programming. T.H.-Report 70-WSK-03, Second Edition, April 1970.

[13] Ecos. http://ecos.sourceware.org/

[14] FAST GmbH. Study of worldwide trends and R&D programmes in embedded systems, 2005. ftp://ftp.cordis.europa.eu/pub/ist/docs/embedded/final-study-181105_en.pdf

[15] GAJSKI, D., DUTT, N., WU, A. AND LIN, S. High-Level Synthesis: Introduction to Chip and System Design. Springer 1992.

[16] Graphviz – Graph Visualization Software, http://www.graphviz.org.

[17] HAREL, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (Jun. 1987), 231-274.JIAN, K. Constructing a Solid Real-Time Operating Systems Course in Computer Science Major. *J. Comput. Small Coll.* 22, 4 (Apr. 2007), 65-74

[18] HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., AND SHTUL-TRAURING, A. 1988. Statemate: A Working Environment for the Development of Complex Reactive Systems.. International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 396-406..

[19] HOARE, C.A.R. Communicating Sequential Processes. Comm. of the ACM, vol. 21, no. 8, pp. 666-677, Aug. 1978.

[20] HODGE, H. HINTON, H.S, AND LIGHTNER, M. Virtual Circuit Laboratory. ASEE. American Society for Engineering Education. 2000

[21] IEEE, INC., IEEE Standard VHDL Language Reference Manual, IEEE Standard 1076-1987. Los Alamitos, Calif.: IEEE CS Press, 1987.

[22] JEUKENS, I. AND STRUM, M. On the Choice of Models of Computation for Writing Executable Specifications of System Level Designs. In Proceedings of the 13th Symposium on integrated Circuits and Systems Design.2000.

[23] JIAN, K. 2007. Constructing A Solid Real-Time Operating Systems Course in Computer Science Major. *J. Comput. Small Coll.* 22, 4 (Apr. 2007), 65-74

[24] JUURLINK, B. H. AND WIJSHOFF, H. A. A Quantitative Comparison of Parallel Computation Models. *ACM Trans. Comput. Syst.* 16, 3 (Aug. 1998), 271-318

[25] KOPETZ, H. AND BAUER, G. The Time-Triggered Architecture. Proceedings of the IEEE. On page(s): 112- 126, Volume: 91, Issue: 1, Jan 2003

[26] KORNECKI, A. Real-Time System Course in Undergraduate CS/CE Programs. IEEE Transactions on Education. Volume 40. Number 4. November 1997.

[27] LAMPORT,L. Time, Clocks, and the Ordering of Events in a Distributed System. Comm. ACM, July 1978, pp. 558-565.

[28] LCC, A RETARGETABLE COMPILER. http://www.cs.princeton.edu/software/lcc/

[29] LEE, E.A. What's Ahead for Embedded Software? IEEE Computer, vol. 33, no. 9, pp. 18-26, Sept. 2000.

[30] LEVIS, P. AND CULLER, D. 2002. Maté: a tiny virtual machine for sensor networks. SIGOPS Oper. Syst. Rev. 36, 5 (Dec. 2002), 85-95.

[31] LEE, E.A AND SANGIOVANNI-VINCENTELLI, A. A Framework for Comparing Models of Computation IEEE Trans. CAD Integrated Circuits and Systems, Dec. 1998, pp. 1217-1229

[32] LUCKHAM, D. C. AND VERA, J. An Event-Based Architecture Definition Language. *IEEE Trans. Softw. Eng.* 21, 9 (Sep. 1995), 717-734

[33] OUIMET, M. AND LUNDQVIST, K. Incorporating Time in the Modeling of Hardware and Software Systems: Concepts, Paradigms, and Paradoxes. In Proceedings of the international Workshop on Modeling in Software Engineering (May 20 - 26, 2007). International Conference on Software Engineering.

[34] OUIMET, M. AND LUNDQVIST, K. The TASM Language and the Hi-Five Framework: Specification, Validation, and Verification of Embedded Real-Time Systems. APSEC 2007

[35] SCHWARZ J.J., SKUBICH J., MARANZANA M., AND AUBRY R., Graphical Programming and Real-Time Design Instruction, in Real-Time Systems Education, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 20-25.

[36] SMITH, J. AND NAIR, R. VIRTUAL MACHINES: Versatile Platforms for Systems and Processes. Morgan-Kaufman Publishers. 2005.

[37] STARK, R., SCHMID, J, AND BORGER, E. Java and the Virtual Machine- Definition, Verification, and Validation. 2001.

[38] SKILLICORN, D.B., AND TALIA, D. Models and Languages for Parallel Computation. *ACM Comput. Surv.* 30, 2 (Jun. 1998), 123-169

[39] TAFT S.. Ada 2005 Reference Manual, LNCS 4348, Springer-Verlag. 2006

[40] TIME-TRIGGERED TECHNOLOGY. http://www.tttech.com/

[41] THOMAS, D.E AND MOORBY, P.R., The Verilog Hardware Description Language. Kluwer Academic Publishers, 1991

[42] VAHID, F AND GIVARGIS, T. Embedded System Design: A Unified Hardware/Software Introduction. John Wiley and Sons, 2001.

[43] VMWARE. http://www.vmware.com/

[44] WAVE VCD Viewer, http://www.iss-us.com/wavevcd/, 2008.

[45] WINDRIVER Systems. http://www.windriver.com/

[46] XEN. http://www.xen.org

Nexos: A Next Generation Embedded Systems Laboratory

Dennis Brylow
MSCS Department
Marquette University
1313 W. Wisconsin Ave.,
Milwaukee, WI 53226
brylow@mscs.mu.edu

Bina Ramamurthy
CS&E Department
SUNY at Buffalo
201 Bell Hall
Buffalo, NY 14260-2000
bina@cse.buffalo.edu

ABSTRACT

The Nexos Project is a joint effort at Marquette University (MU) and University of Buffalo (UB) to build curriculum materials and a supporting experimental laboratory for hands-on projects in embedded systems courses. Our approach focuses on inexpensive, flexible, commodity embedded hardware, (the Linksys WRT54GL wireless router,) freely available development and debugging tools, and a fresh implementation of a classic operating system that is now ideal for embedded system exploration. The prototype laboratory environment is being used in multiple courses at our respective Universities, with excellent results. We report on the infrastructure we have developed, our initial course offerings at both schools, and an evaluation of our success thus far.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; K.3.2 [**Computer and Info Science Education**]: Curriculum

Keywords

Embedded systems education, Nexos, Embedded Xinu

1. INTRODUCTION

Embedded systems comprise a large and growing segment of the computing sphere, but efforts to prepare students for work on design and implementation in an embedded context face a number of important obstacles. First, embedded systems are by their nature quite diverse, both in scope and in function. Typical embedded systems can range from 4- and 8-bit microcontrollers, with hundreds or a few thousand bytes of storage, to full-fledged high-end processors with multiple cores and gigabytes of storage. System requirements can range from low-power, event-driven operation, to gigabit-speed hard real-time deadlines, and can include everything in between. In short, the diversity of embedded systems dwarfs the variation found in desktop and server computing, and this enormous divergence can impact every aspect of system development.

Second, embedded systems traditionally cannot make use of the most powerful compilation, debugging and automated testing tools available for production desktop systems. Their input/output channels are comparatively narrow, they often lack sufficient spare resources to support costly profiling or instrumentation, and their design budgets cannot support large virtual machines or elaborate software runtimes. Third, computer science and engineering degree programs are already well-established at many institutions, and often do not have room for entirely new courses in the core.

Finally, many schools lack the financial resources, laboratory space, or faculty expertise to commit to dedicated embedded instructional equipment. With a few notable exceptions, many of the most popular commercial embedded platforms do not provide inexpensive evaluation boards, are supported only by proprietary development tools, and lack the kinds of useful peripherals that would naturally lead to a body of general purpose laboratory assignments.

While it is certainly daunting to face all of these problems, the need for inexpensive, flexible, hands-on laboratory experiences with embedded systems is most pressing. This paper describes initial results from joint efforts by Marquette University (MU) and University of Buffalo (UB) to develop a curriculum and support infrastructure to address several of these pressing challenges.

The Nexos project aims to provide effective, duplicable, modern curriculum assistance to schools looking to incorporate embedded systems components into core computer science and engineering courses throughout the curriculum. Our approach focuses on inexpensive, commodity hardware — the Linksys WRT54GL wireless router family—readily available to both students and faculty. We are developing hardware and software to support both small- and medium-scale laboratory installations centered around the WRT54GL. Our curriculum development efforts have concentrated on developing laboratory assignments, teaching materials, and a supporting web portal to assist other departments interested in adoption. A textbook / laboratory manual is in progress, and two additional schools have begun adapting our laboratory environment for their own courses.

The remainder of this paper is organized as follows: a brief outline of prior and related work; a description of the Embedded Xinu operating system at the core of our laboratory environment and its related tools; a description of the content in prototype courses taught at both UB and MU; and evaluations of our efforts thus far.

Prior and Related Work

There has been much recent work on embedded systems education. Koopman et al. of Carnegie Mellon University (CMU)

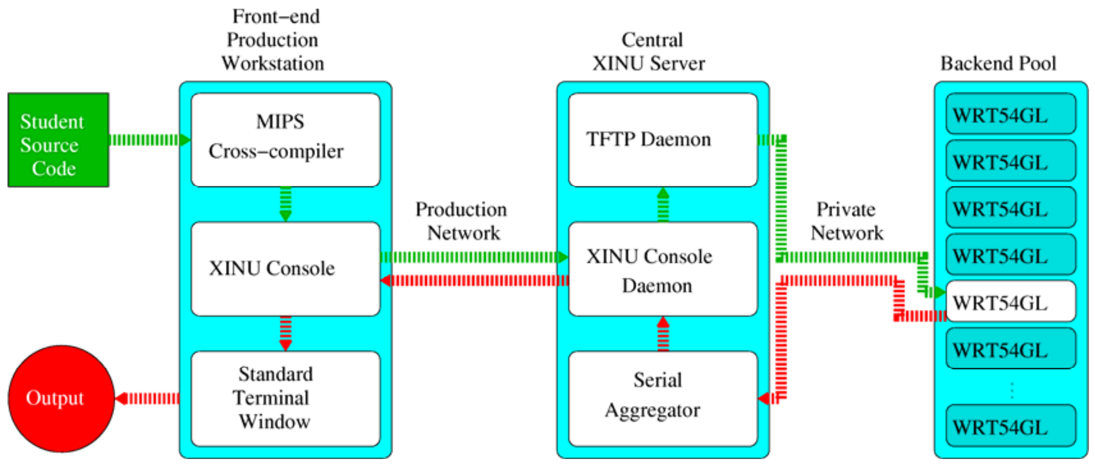


Figure 1: From Student Source Code to Operating System Output

presents an extensive study of the status of embedded system education in [11]. The CMU group discusses the diverse approaches in twelve different embedded application areas, including cross-cutting embedded skills areas such as real-time systems and energy-aware computing, and concludes with a list of lesson learned. The guiding principles for Project Nexos closely follow this list of lessons learned.

Wolf and Madsen [16] discuss embedded systems education for the future and provide a detailed description of their experiences at Princeton University. This paper clearly brings out the importance of hardware-software co-design concepts, recommending C-like languages over an assembly language approach. We concur with Wolf and Madsen that, "next-generation courses in embedded computing should move away from the discussion of components and toward the discussion of analysis and design of systems."

Hamrita et al from University of Georgia [9] reports an interdisciplinary approach that comprises four different courses in robotics, embedded systems, and two courses on microcontrollers. This is an intensive curriculum in embedded systems and would be very hard to adopt for a liberal arts degree in computer science. A summary of the 2005 Workshop on Embedded System Education (WESE 2005) [10] covers a list of presentations on existing embedded systemscourses from all over the world. Most of these presentations refer to quite traditional courses.

A variety of platforms have been proposed for embedded systems curricula. Ricks et. al [15] use an aerospace processor to pursue substantially similar teaching goals to our own. Others have focused on robotics platforms [9], dedicated digital signal processors [1, 8], and more familiar microcontroller systems [6]. We believe that a consumer-grade networking applicance like the WRT54GL presents a platform of comparable or better flexibility at a lower cost than much of this prior work, and with a clear path toward reuse for its intended purpose.

Other previous work has emphasized the use of test-driven development (TDD) in embedded systems courses [13], or have presented tools for facilitating embedded system testing [12]. Our project includes automated testing tools both to assist student development and faculty assessment.

The Embedded Xinu kernel is based upon groundbreaking experimental operating system work by Comer in the 1980's [7].

The Nexos Project is based upon MU's own prior work on integrating embedded systems components into existing core courses [5, 3], and UB's prior development of an embedded systems course.

2. LABORATORY ENVIRONMENT

This section describes the Embedded Xinu laboratory environment we have developed to support the experimental emphasis on embedded systems in the new curricula at MU and UB.

2.1 Hardware

As our target platform, we have chosen the ubiquitous Linksys WRT54GL family of wireless routers. These routers devices are readily available at inexpensive prices, and can already be found in homes, small businesses, and college dorm rooms. They contain a little-endian embedded MIPS 32 processor, operating in the 200-300 MHz range, with 16 MB of RAM, and 4MB FlashROM. Several versions of the router are available, but all have easily accessible serial port connections on the main board that allow direct access to the device firmware.

With only minor modifications to the casing and the addition of an RS-232/TTL serial transceiver circuit, the serial console of the WRT54GL becomes accessible to the student. From the serial console, a user can interrupt the normal boot sequence from FlashROM, and substitute an uploaded kernel image from over the local area network (LAN) port. Thus, students can quickly find themselves running their own embedded O/S kernel on the raw hardware, with no emulation or simulation involved. An overview of this process is shown in Figure 1.

WRT54GL routers are on the upper end of the enormous spectrum of embedded devices available to us; their specifications are roughly equivalent to desktop PCs from the early 1990's. Yet, while they are not as resource-constrained as 4- and 8-bit embedded microcontrollers, they are true embedded systems in the modern context. Processor speed and RAM size are orders of magnitude less than typical desktop PCs, and input/output channels to the processor are narrow. They are missing major components that would be found in non-embedded contexts – no hard disk or optical storage, no video adapter, mouse, or keyboard. They provide a variety of interesting peripherals (serial ports, wireless and wired network interfaces, Flash ROM storage, LED and general purpose I/O pins,) but present realistic obstacles to traditional debugging techniques. The highly resource constrained operating systems they contain are event-driven systems expected to run indefinitely with no direct interaction from users or administrators. In short, we feel that they are an excellent choice for prolonged experimentation by students with an interest in embedded computing. As an added benefit, they are a RISC architecture commonly taught in lower division computer organization courses, so many students will already have some familiarity with the instruction set.

As a consumer network device, the WRT54GL is particularly well-suited to work with embedded operating systems and embedded networking; however, the high speed serial links and the general-purpose I/O pins on the board allow for expansion in many other possible directions, including analog and digital devices sensors and other peripherals. Hobbyists in the embedded Linux community have used the GPIO pins to add a wide variety of devices to their routers.

The modifications made to the platform are non-destructive, and the WRT54GL once again becomes a fully-functional wireless router running its stock system upon reboot. The Embedded Xinu site [2] provides parts lists, diagrams, directions and pictures for making the simple serial modifications required.

This embedded platform can be used in a stand-alone configuration in which a single student computer with a network card and a serial port manages a dedicated WRT54GL router. The advantage of this configuration is simplicity, and it can be realized in both a laboratory setting, or in many cases at home and in dorm rooms. Students in the initial offering of UB's CSE 321 course (described below) followed this route, with many students choosing to purchase their own routers for use in the course. The cost of the hardware, with modifications, is less than a typical textbook, and the platform can be used later for either continued embedded system exploration, or for its original purpose as a home networking applicance. (Or both, with sufficient effort.)

As shown in Figure 1, the platform can also be used in a dedicated pool configuration, where a collection of routers are made available for "checkout" on the network, and students can remotely power, upload, and interact with their embedded operating system kernels from any front-end machine on the network with appropriate connection tools. In the pool configuration, students can make use of the platform without requiring dedicated lab space, and the pool can be used by several different courses simultaneously.

While we focus our discussion in this paper on the 54GL platform, it is always risky to depend too much on a single consumer product. We have also worked with three other router models, including the next generation router slated to ultimately replace the 54GL, to insure that we are not completely tied to a single model or manufacturer.

The next section describes the software infrastructure we have built to support the pool configuration at MU.

2.2 Software

We have ported the venerable Xinu operating system [7] to the embedded MIPS32 processor, and have built appropriate device drivers for several of the key peripherals on the WRT54GL platform. The source code is freely available under a BSD-style license from the Embedded Xinu web portal [2].

Our embedded O/S kernel is small but elegant, and designed to be completely fathomable in a short time by undergraduates working for the first time on embedded systems. The layers of the Embedded Xinu kernel are shown in Figure 2, and include all of the components normally found in an embedded operating system.

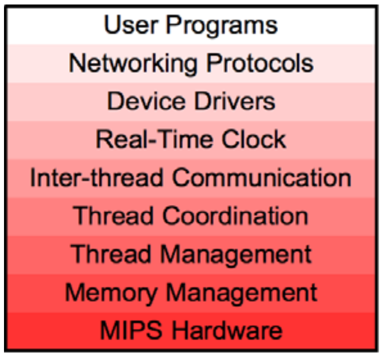


Figure 2: Layers of the Embedded Xinu Kernel

Our software can be compiled using freely-available crosscompiler tools that are readily available from the web and mainstream Linux distributions. In the stand-alone configuration, all that is required for software is a cross-compiler and a TFTP server, another component that is freely-available for all of the major desktop operating systems.

For a dedicated pool of backends, additional connection software is required to allocate routers to remote users and to manage special-purpose hardware for remote power control and the large number of serial console connections required. The Embedded Xinu portal supplies parts lists and configuration information for remote power control and network attached serial annexes; we also supply open source tools for managing the remote connections and controlling the dedicated backend pool.

The MU Systems Laboratory contains a dedicated pool of two dozen backend target routers used for our embedded systems courses and others. The high-level overview of the pool is shown in Figure 1; additional details are available online and in [5].

3. CURRICULUM

This section describes two Nexos Project courses that have been developed in tandem at UB and MU, leveraging the Embedded Xinu infrastructure. The first course, UB's CSE 321, is an embedded systems course normally taken by sophomores as part of the core curriculum. The second course, MU's COSC 198, is an upper-division elective taken by students who have already taken core operating system and hardware system courses with embedded components. While CSE 321 and COSC 198 are two very different courses at two very different institutions, they both leverage a common laboratory infrastructure, and common curriculum elements. The two taken together show the diversity of assignments and course focus that are possible within the Nexos framework. We describe both courses in detail below.

3.1 Embedded Systems Course at UB

The Computer Science and Engineering department (CSE) at University of Buffalo has decided to address the need for a new course to bridge the gap between lower level courses in data structures and computer organization and higher level courses such as operating systems. A new course in embedded systems has been created to bridge this gap, and to strengthen the operating systems curriculum. While the traditional operating systems course is offered at the senior level, the new embedded system course will be taught at the sophomore level. This model enables students to draw ideas from modernized lower level courses, and also gives them ample time to research, explore and apply the embedded operating systems concepts in their research and internship efforts in ways that were not possible with a single senior level course. Prerequisite for the course is Data Structures and Algorithms (CSE 250) or an equivalent. The topics covered in this course (CSE 321 [14]) begin at the boundary of application level software and extend to applications in the real world.

CSE 321 was offered for the first time in fall of 2007 as a pilot. The curriculum for the course is strongly founded on fundamentals that are often overlooked in the modernization of courses. At the same time, the curriculum also exposes current trends and concepts enabling modern innovations and devices such as sensors, heart pace makers, digital music players, and communication devices. The course pedagogy includes four components: 1) A core component that provides coverage for fundamental concepts; 2) An extension that allows room for applied concepts; 3) Field visit(s) to a local industry; and 4) Class visit(s) featuring Q&A sessions with a distinguished scientist in a relevant area.

The core component deals with coverage of fundamental concepts that are expected to be common among the course offerings at various institutions. However the extension component can be customized to meet curricular needs and to reflect the areas of specialization of the host department.

Course topics include resource management, concurrency, secure coding practices, memory management, timeline design and analysis using metrics and schedulability tests, hardware interfaces, device driver programming, memory maps and boot kernels, firmware and ROM-resident system code, communications and networking, and debugging live systems. Intellectually, this course advances considerations of computer system architecture, multi-threaded control, fault tolerance, the translation of requirements into a well-partitioned software architecture and practical design, the subsequent translation into code, the documentation of technical ideas (promoting writing skills), and strategies for system configurability, hardware state tracking, and safety. Class visits by an expert are recommended by Koopman et al in their survey paper [11] to add real impact to concepts like requirements and tradeoffs discussed in class. Field visits are modeled after the field-trips that K-12 curricula have implemented for many years with great success.

A high level description of the curriculum is given in Figure 3. Each of the four components are described in greater detail below. The second column of the table shows a list of hands-on laboratory exercises that are carried out by students on the Embedded Xinu / Linksys WRT54GL environment. These laboratory exercises correspond closely with MU's operating systems course [5]. Thus the implementation in Figure 3 illustrates the feasibility of retrofitting an existing course curriculum with an Embedded Xinu-based lab environment and exercises. Educators may choose to customize this framework to suit their curricular needs.

Component I: Core
The core covers the fundamental topics we expect to be common among most Embedded Operating Systems courses. The objective of this component is to lay down a strong foundation for the students, and to sharpen their C programming and basic hardware-level competencies. It covers nuances of memory allocation, deallocation, memory leaks and implications, buffer management and overflow implications, basic device driver design, concurrency and event driven programming. This component also offers an opportunity to discuss open source software development and the GNU GPL (General Public License). The history of the Linksys WRT54GL wireless router itself yields a very interesting case study for this topic.

Component II: Extension
The second component of the course is intended to be a variable and applied section that can be customized to suit the needs of a particular instructor or curriculum. At a minimum, it covers kernel level aspects such as bootstrapping, ROM resident firmware strategies and debugging aspects. For CSE 321, real-time systems concepts were introduced; another institution could, for example, replace this material with an emphasis on sensor networks, or with material on ubiquitous computing.

Component III: Field Trip
Student participation in a college-level field trip begins well before the actual date; students start by writing a report on their personal objectives for the trip and their expectations, and complete the report after the visit. While Embedded Xinu-based laboratory exercises provide hands-on experience, a field trip to an industry that applies the concepts students learned in the course can be a valuable tool to inspire and motivate them toward further study of embedded systems.

Component IV: Class Visit
This final component gives an opportunity for the students to meet an expert in the field of embedded systems, a pedagogical pattern that seems to be quite popular with the students. Question and answer (Q&A) format is used for this component to allow wider student participation and to avoid the monotony of the lecture format.

Topics	Laboratory Exercises/Demonstrations
I. Core	
Operating system fundamentals; Secure coding : pointers, memory management	1: String operations and memory allocation

Buffer overruns: string vulnerabilities	2: Experimenting with buffer overflow
Device drivers	3: Writing and testing a basic device driver
Context switch and scheduling	Introduce Linksys WRT54GL
Interrupt handlers	4: Context switch and scheduling
Open source OS and GPL	Demo: Embedded Linux
Concurrency and process control	5: Process control
Event driven and asynchronous programming	6: Preemption and synchronization
Analysis and optimization for performance, space and power	Demo: Analysis and optimization
II. Extension	
Embedded systems: Memory maps and boot kernels	7: Heap management
Firmware and ROM resident execution strategies	Demo: Show and tell of sample firmware
Wireless devices; Deeply embedded systems: sensor motes and introduction to sensor networks	8: Implement and test multi-level device driver
Digital signal processors	
Handheld devices: end-use applications such as Internet applicances, smart devices, GPS instruments	9: Design and implementation of basic file system
Debugging live systems	Demo: debugging
Realtime systems; predictability, deadlines and schedulability	Demo: Cardiac pace-maker
III. Field Trip	
To local industry: Atto Tech Inc. and/or Wilson Greatbatch Inc.; Moog Controls Inc.	Students write a report about their visit
IV. Class Visit	
Prominent researcher in the area; industrial practitioner	Students ask questions

Figure 3: CSE 321 Course Topics and Supporting Laboratory Exercises

The combination of modernized curriculum, hands-on laboratory experiments, an industrial field-trip and "ask the expert" Q&A sessions is intended to fill the interest bandwidth of "net generation" students, keeping them motivated to perform well, and ultimately helping them to benefit from the course.

3.1.1 CSE 321 Lab Exercises and Demos

The first three lab exercises (Figure 3) aim to review and reinforce the C language skills of the students. Many students do not have adequate background in some important concepts such as memory allocation and memory leaks, particularly when lower level courses have relied on garbage collected languages like Java as the first language. UB CSE department uses Java for its introductory courses and moves on to other languages (C, C++, and Lisp) in higher level classes. Lectures and labs are supported by in-class demonstrations of concepts and devices. For example, one of the demos involves Linux, the GPL, embedded Linux, and the history of the Linksys WRT54GL wireless router as a popular device for embedded Linux hobbyists. This is followed by a comparison of embedded Linux and Embedded Xinu that brings out the need for analysis and optimization for performance and size. This demonstration partially fulfills the need for qualitative requirements that are critical for many embedded applications.

3.2 Embedded Systems Course at MU

MU's COSC 198: Embedded Systems was offered for the first time in spring of 2008. The course was offered as an elective, with our embedded operating systems course [5] as a prerequisite. As a result, COSC 198 [4] could be considered the advanced level course in a sequence of systems courses that revolve around embedded laboratory experiences.

The target population of the course was upper division undergraduates and junior graduate students in computer science, computer engineering, and biomedical engineering. In addition to the embedded operating systems prerequisite, most students also had prior coursework in hardware systems, digital logic or computer architecture. Twelve students enrolled in this first offering, a mix of third- and fourth-year undergraduates from the target majors noted above, and a couple of first- and second-year computer science graduate students.

COSC 198 was comprised of three main components: first, a lecture component covering new material on embedded and real-time systems, following roughly the chapters in a recommended textbook; second, a seminar component in which students read, presented, and discussed research publications from venues in embedded systems and related areas; and third, a laboratory component in which student teams designed, implemented, and tested their own new subsystems in the Embedded Xinu framework. Each of these components will be discussed below.

The stated outcomes for this course were that upon completion students would be able to: 1) Read, understand, and present current research papers in the area of embedded systems; 2) Design, implement, and test their own embedded system components for integration into a larger system; and 3) Document complexities of hardware/software interaction in their embedded system components in sufficient detail that the work can be understood and replicated by others.

Lecture Topics: Lectures topics in COSC 198 were chosen primarily to introduce new material that would be directly relevant to subsequent research paper assignments, and to laboratory projects. It corresponds roughly to the "Core" component of CSE 321 developed at UB. Some intentional overlap with prior courses was necessary because of the diverse backgrounds of the students. Engineering majors had already worked with feedback control systems, but computer science majors had not. Conversely, computer scientists had dealt more extensively with asynchrony than the engineers. Some of these distinctions are particular to the respective curricula at MU, and would not be the rule at other institutions.

Given that the primary Embedded Xinu laboratory platform is a consumer networking appliance, student interest naturally turned toward embedded networking technology as laboratory projects were brainstormed. In the final third of the term, much of the lecture material was devoted to intermediate-level networking topics, to support this direction. This took the spot of the "Extension" segment of the curriculum developed at UB.

Lecture topics in the term included: characteristics of embedded systems; characteristics of real-time systems; soft vs. hard real-time; cyclic, asynchronous, and unpredictable systems; control systems and feedback; digital sampling, error, and resolution; scheduling and schedulability analysis; worst-case execution time analysis; case studies of real embedded/real-time systems; and networking (Ethernet, IP, ICMP, UDP, DHCP, TCP).

Seminar Topics: Interwoven with the lecture component of the course, students worked alone and in pairs to read, present, and lead class discussions of current research in embedded systems and related areas. Working from an initial list of possiblepapers, and then branching out on their own later in the term, students drew papers from such venues as PLDI and SOSP, LCTES, ASPLOS, PODC, USENIX, and SIGCSE. Topics covered included: embedded, real-time operating system kernels; fine-grained locking and lock-free synchronization; designing against interrupt-driven overload; memory protection and isolation in embedded processors; general purpose computing with GPUs; optimizing embedded networking protocol stacks; resource bound and program analysis in embedded systems; and other embedded system laboratories and curricula.

While some of these topics proved to be too ambitious to cover thoroughly in this level of course, most provoked lively class discussion of the challenges facing modern embedded system designers.

Laboratory Projects: For the practical laboratory component of COSC 198, students split into teams of six. In the first portion of the term, the teams worked in parallel on competing implementations of the same prescribed embedded system software components, and then presented their work to the rest of the class in a "bake-off" format. In the latter half of the term, the teams were allowed (in consultation with the professor) to brainstorm their own project goals, and then pursue an assigned subset.

Team work throughout the term emphasized best practice software design principles, including revision control and collaboration tools, testsuite validation, regular design reviews, and group code reviews.

Laboratory projects included: 1) Ethernet device driver. Completion of an asynchronous device driver on the Embedded Xinu / Linksys WRT54GL platform, including DMA pool management and O/S buffering; 2) Packet sniffer for receiving, classifying, and displaying various types of network traffic; 3) ARP (Address Resolution Protocol) and ICMP (Internet Control Message Protocol), two key underlying pieces of the modern TCP/IP Internet. Teams implemented enough of these protocols to support ping and traceroute primitives from their embedded operating system kernels; 4) UDP (User Datagram Protocol), the work horse of unreliable network communication. Students implemented enough of UDP to support their own UDP service clients, including DHCP and rdate.

At this point, the teams went their separate ways based upon their interests. One group proceeded to implement a working subset of TCP (Transmission Control Protocol), the reliability layer of Internet communication, allowing their embedded operating system to establish connections and exchange data with many other external services. Another group pursued memory protection, a mechanism using the virtual memory hardware of the underlying architecture to guarantee memory isolation between processes in the embedded operating system. While neither team were entirely successful in achieving their advanced goals, each of the teams met key benchmarks and ended the semester with substantial accomplishment. Both of these final projects are of the level typically found in second-year graduate-level systems courses at some other Universities.

4. EVALUATION

This section discusses the results of various assessment mechanisms used at both UB and MU to weigh the educational value of the new curricula in conjunction with the embedded laboratory enviroment.

4.1 Embedded Systems Course at UB

The prototype offering of CSE 321 was small and comprised only 6 students. UB had ample opportunity to get feedback by closely observing the small number of students, but some assessment instruments lacked both statistical significance and anonymity with such a small enrollment. Next fall's offering is likely to have a normal enrollment of 20 to 30 students, and assessment will continue.

The pilot offering of CSE 321 at UB used the stand-alone configuration for the WRT54GL router platform; each student had a router and associated connectors. Besides the low cost of the hardware, we had some significant observations that are really impressive about the environment: 1) time to set up was small (about 2 hours of student lab time per unit), plus the cross compiler configuration time; 2) portability was high (2 of the 6 students decided to buy their own hardware to work at home since it was less than the cost of many texts) and students could carry their equipment around in a kit bag; and 3) the set-up was self-administered, meaning that we did not need to depend on department system administrators to install and configure the environment for us. This is especially important for the adoptability of Nexos in both large research schools where priorities are elsewhere and in small, remote schools that lack technical expertise to maintain specialized systems.

Figure 4 contains a summary of aggregate results from the special-purpose assessment instrument section relevant to the technical objectives of the course. Responses are reported as averages over a five point scale, ranging from *Strongly Agree* (1) to *Strongly Disagree* (5).

Evaluation Question	Avg
1. Understand the fundamental components and operation of an embedded system	1.5
2. Can design and implement an embedded system	2.0
3. Understand the fundamental components and	

	operation of a realtime system	2.25
4.	Can design and implement a realtime system	2.0
5.	Are able to program using Xinu environment	1.25
6.	Understand the interface between hardware and software	1.0
7.	Understand the interaction of hardware and software	1.25
8.	Are able to get a complete picture of an O/S	2.0
9.	Are able to embed the system you programmed in real hardware	1.75

Figure 4: Student evaluation of course objectives for CSE 321

Figure 5 contains a summary from the special-purpose assessment instrument section relevant to the Embedded Xinu laboratory resources used for the term projects. Responses are reported as averages over a five point scale, ranging from *Strongly Agree* (1) to *Strongly Disagree* (5).

	Evaluation Question	Avg
1.	Hardware resources appropriate	2.0
2.	Software resources appropriate	1.25
3.	Computer resources adequate for assignments	1.5
4.	Computer resources available and accessible	1.5
5.	Enabled "hands on" embedded system experience	1.25
6.	Enabled "hands on" realtime system experience	2.25
7.	Able to work with Embedded Xinu	1.0
8.	Able to work with WRT54GL platform	1.0
9.	Xinu wiki useful	1.25

Figure 5: Student evaluation of Embedded Xinu in CSE 321

Overall, UB successfully duplicated the basic Embedded Xinu environment to teach a course on embedded and realtime systems. Student suggestions for improvement included complaints about the difficulty of soldering wires for the serial transceiver. By the start of the spring term, MU had provided a solution in the form of a printer circuit board design for easy transceiver assembly. UB students frequently consulted the Embedded Xinu Wiki [2] maintained by MU and sent in requests for more information. One of the updates UB requested was a wiki page for shell programming; the prompt MU response is at http://xinu.mscs.mu.edu/Shell.

Students were excited by the use of knowledge from their earlier courses. For example, they designed a UML class diagram for Embedded Xinu, and experienced the effects of single threaded kernel through the errors they encountered. Students explored the hardware on their own by buying parts such as serial-to-USB cables from Ebay and USB drivers from Radioshack.

UB plans an offering of the course for fall 2008, with normal enrollment (20-30 students). This will be supported by an extended, scaled-up version of Embedded Xinu, and a dedicated pool of backend platforms.

4.2 Embedded Systems Course at MU

COSC 198: Embedded Systems was the first offering of its kind at MU, so it cannot be quantitatively compared with predecessor courses. Instead, we rely on three other measures to assess the student learning resulting from the course. First, we have quantitative, direct measure of the practical subsystem deliverables produced from the laboratory component of the course. Second, we have the standard MU course evaluation forms completed by students at the end of the term. Finally, we have the results of a specially produced assessment instrument, jointly developed for the Nexos project by UB and MU, and administered by an independent evaluator. Student participation in the assessment was anonymous and voluntary, and the protocol was approved by both the UB and MU Institutional Review Board (IRB) for research involving human subjects.

The systems produced by the teams in the first group of laboratory projects were either very good or excellent in each case. For each major functionality milestone, teams passed all or all but one of the key testing criteria. In the second, advanced phase of laboratory projects, the teams were not able to pass the majority of testing criteria – however, they had set highly ambitious goals for themselves.

Evaluation question	Median	Decile
The course as a whole was:	5.0	9
The course content was:	4.9	9

Figure 6: Student evaluation of COSC 198

Figure 6 summarizes anonymous student evaluation of the course using the standard MU assessment instrument. Answers were given in a scale from *Excellent* (5) to *Very Poor* (0), and the aggregate score is reported as a median. The decile rank compares the median score of an item in this course with the median scores for all courses at MU in the past two years; a decile of 9 indicates a median score in the top 10% of all courses.

	Evaluation question	Avg
1.	Understand the fundamental components and operation of an embedded system	1.6
2.	Can design and implement an embedded system	1.8
3.	Understand the fundamental components and operation of a realtime system	1.5
4.	Can design and implement a realtime system	1.8
5.	Are able to program using Xinu environment	1.3
6.	Understand the interface between hardware and software	1.4
7.	Understand the interaction of hardware and software	1.3
8.	Are able to get a complete picture of an O/S	1.4
9.	Are able to embed the system you programmed in real hardware	1.5

Figure 7: Student evaluation of COSC 198

Figure 7 contains a summary of the special-purpose assessment instrument section relevant to the technical objectives of the course. Responses are reported as averages over a five point scale, ranging from *Strongly Agree* (1) to *Strongly Disagree* (5). Note that this scale is reversed from the previous assessment tool, and lower numbers are better.

	Evaluation question	Avg
1.	Hardware resources appropriate	1.3
2.	Software resources appropriate	1.3
3.	Computer resources adequate for assignments	1.3
4.	Computer resources available and accessible	1.3
5.	Enabled "hands on" embedded system experience	1.2
6.	Enabled "hands on" realtime system experience	1.6
7.	Able to work with Embedded Xinu	1.3
8.	Able to work with WRT54GL platform	1.6
9.	Xinu wiki usefu	1.9

Figure 8: Student evals of Embedded Xinu in COSC 198

Finally, Figure 8 contains a summary of the special-purpose assessment instrument section relevant to the Embedded Xinu laboratory resources used for the term projects. Responses are reported as averages over a five point scale, ranging from *Strongly Agree* (1) to *Strongly Disagree* (5).

4.3 Discussion

Both the direct measure and the two indirect measures assessing the Embedded Systems course at MU are uniformly excellent, and indicate that the Embedded Xinu platform enabled an engaging and rewarding learning experience. Students' written comments were similarly glowing, and suggested a greater focus on case studies and application-centered papers for the next iteration of the course.

The indirect measures at UB were similarly impressive, although it is difficult to draw strong conclusions before we have a larger sample. Assessment our curriculum materials and infrastructure will continue with future offerings of these courses, with evaluation of student performance in subsequent courses, and with experiences reported by other schools currently adopting the fruits of the Nexos Project.

5. CONCLUSIONS

We have presented the results of inaugural offerings of embedded systems courses at two Universities, based upon a common laboratory environment to support hands-on experimentation. Our infrastructure focuses on flexible, inexpensive, ubiquitous consumer hardware, and open-source tools that are widely available. The Nexos Project comprises our curriculum materials, hardware and software tools, and a community web portal to support widespread adoption of the same. Our initial assessments have been extremely promising, and we believe that our approach can help bring embedded systems education to many schools that would not otherwise have the resources, space, or expertise to initiate such an undertaking.

6. ACKNOWLEDGMENTS

The authors are grateful for the students in MU's COSC 198 and UB's CSE 321 courses who persevered through our experimentation in the 2007-2008 school year. This work is supported in part by NSF grant DUE-CCLI-0737476.

7. REFERENCES

[1] Benjamin, Kaeli, and Platcow. Experiences with the blackfin architecture in an embedded systems lab. In *WCAE 2006: Wkshp on Computer Architecture Education*, 2006.

[2] Brylow. Embedded XINU project, 2007. http://www.mscs.mu.edu/~brylow/xinu/.

[3] Brylow. An experimental laboratory environment for teaching embedded hardware systems. In *WCAE 2007: Wkshp on Computer Architecture Education*, pages 44–51. ACM Press, June 2007. ISBN: 978-1-59593-797-1.

[4] Brylow. COSC 198: Embedded systems, 2008. http://www.mscs.mu.edu/~brylow/cosc198/Spring2008/.

[5] Brylow. An experimental laboratory environment for teaching embedded operating systems. In *SIGCSE 2008: 39th SIGCSE technical symposium on Computer science education*, volume 40, pages 192–196, New York, 2008.

[6] Cheng, Rover, and Mutka. A multi-pronged approach to bringing embedded systems into undergraduate education. In *Proceedings of ASEE 98: American Society for Engineering Education Annual Conference*, 1998.

[7] Comer. *Operating System Design: The XINU Approach*. Prentice Hall, 1984.

[8] Franklin and Seng. Experiences with the blackfin architecture for embedded systems education. In *WCAE 2005: Wkshp on Computer Architecture Education*, 2005.

[9] Hamrita, Potter, and Bishop. Robotics, microcontroller and embedded systems education initiatives: An interdisciplinary approach. *International Journal of Engineering Education*, 21(4):730–738, 2005.

[10] Jackson and Caspi. Embedded systems education: future directions, initiatives, and cooperation. *SIGBED Review*, 2(4):1–4, 2005.

[11] Koopman, Choset, Gandhi, et.al. Undergraduate embedded system education at Carnegie Mellon. *Transactions on Embedded Computing Systems*, 4(3):500–528, 2005.

[12] Legourski, Trödhandl, and Weiss. A system for automatic testing of embedded software in undergraduate study exercises. *SIGBED Review*, 2(4):48–55, 2005.

[13] J. Miller and M. Smith. A TDD approach to introducing students to embedded programming. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 33–37, New York, NY, USA, 2007. ACM Press.

[14] Ramamurthy. CSE 321: Realtime and embedded systems, 2007. http://www.cse.buffalo.edu/~bina/cse321/fall2007/.

[15] Ricks, Stapleton, and Jackson. An embedded systems course and course sequence. In *WCAE 2005: Workshop on Computer Architecture Education*, 2005.

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

Hardware/Software Co-design is a starting point in Embedded Systems Architecture Education

Patrick Schaumont
Bradley Department of Electrical and Computer Engineering
Virginia Tech
schaum@vt.edu

ABSTRACT
Embedded Systems Architectures are hard to design, and there is no generally accepted method of doing it. In recent years, this problem has become even harder because of the wide variety of programmable components (FPGA, ASIP, DSP, …). We propose hardware/software codesign as a starting point for teaching the topic. Codesign helps designer-students to think about architecture design in terms of a trade-off between performance and flexibility. Our senior-level undergraduate course in hardware/software codesign includes a hands-on project that requires students to optimize embedded system architecture across the traditional boundaries of hardware and software. We describe a lab series that combines system modeling with refinement on an FPGA board, and that concludes with a class-wide hardware/software codesign contest. The results of the contest clearly illustrate the strengths of 'systems thinking' over 'component thinking'.

1. INTRODUCTION

Embedded Systems Architecture design is the task of selecting and programming a suitable configuration of components for a given system application. Programmable chip companies, with the help of Moore's law, are providing us with amazing selection of components to do this. Traditionally, the creation of embedded system architectures used to be relatively straightforward: use a microcontroller for flexibility and add hardware peripherals for specialized functions. Nowadays, designers can apply multiple component types (e.g. Field Programmable Gate Arrays, Digital Signal Processors, and Application-Specific Instruction-set Processors) to find the optimum over multiple design objectives, including system flexibility, power consumption, design cost, and design time.

Building embedded system architectures is not an easy task. Each programmable component comes with its own design flow and tools, and with its own programming model. Each one presents a separate learning curve to the designer.

This contribution considers how embedded-system educators can help future engineers to prepare for this complex architecture design space. Obviously, it is not feasible to train students in each possible programmable technology – there are too few hours in a semester to do that. In current practice, educators select a single component type (e.g. FPGA), and then teach students how to map and optimize an example application for this component. Educators thus use a thematic, application-driven approach to train students [1]. In order to cover a broader problem space (more component types or more applications), a structured approach to teaching embedded systems architecture may be preferable. This is an important motivation for developing a structured introduction to hardware/software codesign [2].

A central idea in hardware/software codesign is to merge two design processes: hardware design uses spatial decomposition and is well suited for performance, while software design uses temporal decomposition and is well suited for flexibility. A successful combination of hardware and software enables designers to obtain solutions that are the right combination of flexibility and performance. Thus, we think of hardware/software co-design as a simplified version of the more complex trade-off that needs to be made during embedded systems architecture design, namely the partitioning between platform architecture and platform function. For this reason, we think that hardware/software co-design is the proper starting point for education in this area.

Among programmable components, FPGA platforms have been very successful in providing a target that equally suits software design and hardware design. Several courses have explored this in the context of codesign [3] [4]. We also note that there is a complementary view to embedded systems design which starts from a software-centric system view (rather than a hardware-centric system view). In that case, the problem being addressed is how to teach architecture-specific software. The Embedded Software consortium in Taiwan, for example, has defined a software curriculum because of the high add-on value that software can bring to hardware design [5]. Vanderbilt University has defined and embedded-software and systems concentration in their engineering curriculum to address the specific needs of embedded software that interacts with electrical, mechanical and other hybrid systems [6].

The rest of this paper elaborates on the need for - and our approach to - embedded systems architecture education. The following section enumerates some of the difficulties for 'newbies' in hardware/software codesign, and we point out possible causes. Section 3 discusses the approach we have followed. Section 4 explains the hands-on project we used in an undergraduate course on hardware/software codesign. The project demonstrated the importance of system-level thinking in embedded system architecture design, and the role that hardware/software codesign plays in it. We conclude the paper with a few open challenges and a positive note.

2. NEWBIE CODESIGN PROBLEMS

A modern undergraduate curriculum in computer engineering tends to use a rather strict partitioning in hardware-oriented and software-oriented topics. Figure 1 shows a typical example. After an introductory programming course, students take courses with a hardware focus or a software focus. It is not until the final-year capstone project that students will experience the full problem space of embedded systems architecture design. This may be too late. In our experience, senior computer-engineering students that follow a curriculum as shown in Figure 1 tend to develop a 'bias' towards hardware design or software design, and this hampers the development of good systems-architecture thinking. The following subsections illustrate some of the difficulties faced by aspiring codesign students.

2.1 Combining Modeling Languages

Although there has been a tremendous research effort in system-level languages over the past decade, these efforts have not yet made their full impact on the curriculum. Therefore, the hardware-branch and software-branch in Figure 1 use different, incompatible modeling – and programming languages (for example, C and VHDL). The semantic gap between these design languages is very large, and it reflects fundamental differences in thinking about design. Consider the following illustrations of the difference between writing C and modeling hardware using RTL (VHDL or Verilog).

- The concept of *time* in RTL and C differs enormously. Software designers write untimed C and hardware designers write event-driven RTL. A common ground between hardware timing and software timing would be to count time based on clock cycles or maybe instructions. Instead, RTL designers insist on event-driven modeling for the occasional asynchronous gate, and C programmers don't want to add timing details that destroy portability. Students are forced to choose their camp.
- The notion of *model* and *implementation* is very different in C and RTL. For all practical purposes, a program in C *is* an implementation. An RTL program on the other hand is a simulation artifact, and the implementation is only available after logic synthesis. In RTL, what you write is not necessarily what you get, which is hard to grasp for designers with a software mindset.
- Nearly identical syntax in C and RTL may mean very different things. A `for`-loop in C is a control-flow construct. A `for`-loop in RTL, on the other hand, is syntactical sugar that is unrelated to the control-flow in the implementation. In fact, if we discount the modeling of state machines using `case` statements, RTL does not offer a good means to model control.

The language differences between C and RTL do not stop designers from excelling in either hardware design or else software design. However, the differences make a combined mastering of C and RTL very hard.

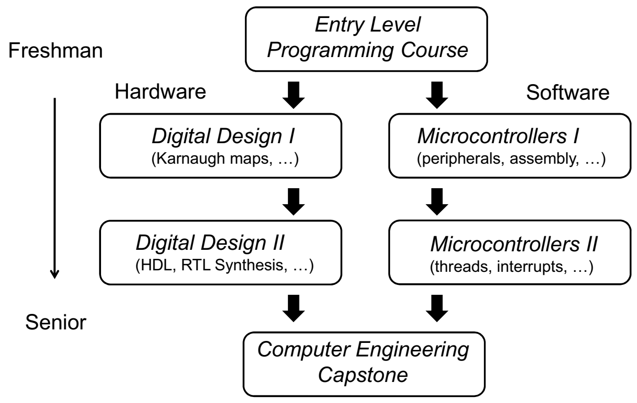


Figure 1: Separate Software and Hardware Tracks in Typical Computer Engineering Curricula

2.2 Designing Interfaces

A second hurdle for students in embedded systems architecture design is the design of interfaces. The efficiency of an embedded system is critically dependent on the efficiency of interfaces (between hardware and software, between coprocessors and processors, etc). Thus, embedded system architects should think of these interfaces as essential design features. Instead, the curriculum in Figure 1 puts the focus on the individual domains, and not on the links that interconnect the domains. As a result, interfaces become a second-class citizen in embedded architecture design.

As a contrasting example, computer science students study the interface between instruction-set architecture and micro-architecture during an entire introductory course ('Introduction to Computer Architecture') [7]. This prepares them to deal with many computer-architecture issues such as pipeline-stalls, memory-bottleneck, etc. For embedded-system engineering students no similar introductory and structured interfacing-course exists.

2.3 Design Tools versus Design Methodology

A final issue for students in embedded system architecture design is the myriad of design tools they need to use. Each programmable component comes with its own design environment, often incompatible with others. The curriculum in Figure 1 promotes the use of such specialized tools, but it does not teach how to combine their use. A second issue is that many aspects of modern design cannot be covered with design tools alone. Some examples include good debugging practice, defensive programming, design for observability, version control, and divide-and-conquer problem solving. These issues, as well as many others, can be collected under the common theme of design *methodology*: the recipe to transform design ideas into design implementation.

Therefore we believe that an embedded systems architecture student will benefit more from a sound design methodology using simple tools then from fancy and automated design tools that operate stand-alone that that abstract out crucial details of the problem.

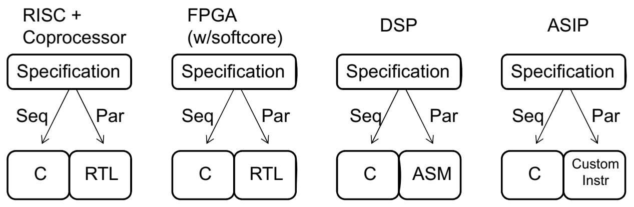


Figure 2: Hardware/Software Codesign is a generic formulation of a problem with multiple instances

3. CODESIGN AS BASIS FOR EMBEDDED ARCHITECTURE DESIGN

In this section, we briefly motivate and outline our course, which is targeted to seniors and first-year graduate students. We present hardware/software codesign to students as a generic solution for a design problem that re-appears in many different forms during the design of embedded system architectures. Specifically, hardware/software codesign targets the combination of a generic processing engine and a specialized processing engine. A designer then maps a specification so as to optimize the efficiency (power, preformance, utilization ...) of the overall architecture.

In all of the cases shown in Figure 2, the generic processing engine is a machine that runs C. The specialized processing engine is, depending on the case, a coprocessor, an FPGA (-coprocessor), a DSP instruction-set, or an ASIP instruction. Although the target platform is different in each case, the underlying design concepts are strongly related, and it makes sense to address them in the context of a structured introduction to hardware/software codesign. The next subsection describes the course topics, while the subsection after that addresses the issue of design tools.

3.1 Course Topics

The hardware/software codesign course contains three parts.

Fundamentals: The first part introduces fundamental ideas in embedded system architectures. On top is a discussion on concurrent specifications and parallel implementation. We use Synchronous Dataflow (SDF, [8]) as an introduction to concurrent specifications. SDF semantics are very well suited for this because of their formal properties in combination with their practical applications (signal processing). Besides a discussion of SDF, we also teach the students how to analyze the control flow and the data flow in a C program. This analysis is very useful when considering architectural alternatives for a C program.

Custom Architecture Design Space: A second part in the course is an in-breadth discussion of the custom architecture space. As illustrated in Figure 3, we start with finite-state-machine-with-datapath models, and gradually proceed to System-on-Chip architectures. The in-breadth discussion of architectures starts with typical 'hardware' targets and proceeds to typical 'software' targets. This way, students learn that there is only a single design space. It is easy to show that each step in the sequence of Figure 3

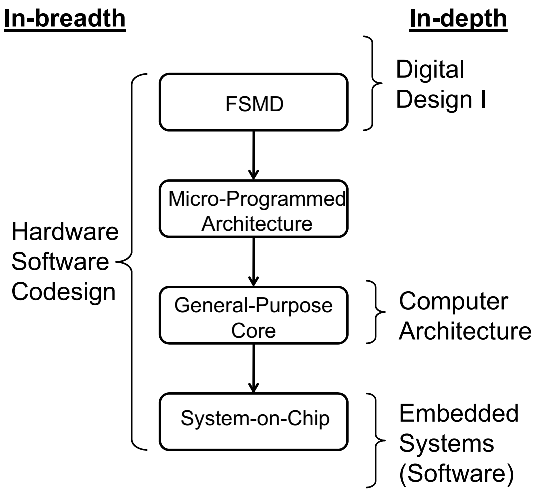


Figure 3: Covering the custom architecture-space in-breadth rather than in-depth.

is an improvement over earlier targets in terms of flexibility. Obviously, in-breadth discussions imply that we cannot address the full details of each architecture. However, our discussion is complementary to the in-depth approach of existing courses: FSMD, general-purpose cores and system-on-chip may be addressed in detail in a digital-design course, a computer architecture course, and an embedded systems software course respectively.

Hardware/Software Interfaces: The final part of the course is a discussion on hardware/software interfaces. We describe how C communicates with a specialized processing component such as a coprocessor or a custom datapath. This is a broad topic, and it includes a description of data communication from C to hardware (interfaces and buses), synchronization methods (handshakes, mailboxes, queues), encapsulation of software (custom instructions, API's) and encapsulation of hardware modules.

3.2 Tools and Design Flow

The course topics are augmented with an intensive hands-on part. Because of the difficulties with a mixed-language approach based on C and RTL, we are using GEZEL, a codesign modeling - and cosimulation environment [9]. GEZEL combines a design language for FSMD with custom interfaces to instruction-set simulators. GEZEL provides cycle-based co-simulation and a path to implementation by converting the FSMD models into VHDL. Students write co-designed models by combining C and GEZEL language. In comparison with the use of RTL and C for codesign, we address the following issues:

- Hardware is expressed using cycle-based, single-clock FSMD modeling. These models result in a compact syntax, close to implementation. In GEZEL, a `reg` variable is really a flip-flop. In our experience, a simple mechanism to express hardware models is very important to enable students to concentrate on methodology and design. Some students enter our class without previous knowledge on RTL programming (Verilog is only covered in an elective course).

```
1.  //-----------------------------------
2.  // ARM core with FSL interface in GEZEL
3.  ipblock arm1 {
4.    iptype "armsystem";
5.    ipparm "exec = exec.elf";
6.  }
7.
8.  ipblock fsl1(out data   : ns(32);
9.               out exists : ns(1);
10.              in  read   : ns(1)) {
11.   iptype "armfslslave";
12.   ipparm "core=arm1";
13.   ipparm "write=0x80000000";
14.   ipparm "status=0x80000004";
15. }
16.
17. //------------------------------------
18. // GEZEL FSMD, reads from FSL, accumulates
19. dp avg_fsmd    (in  rdata   : tc(32);
20.                 in  exists  : ns(1);
21.                 out read    : ns(1)) {
22.
23. reg rexists : ns(1);
24. reg acc : tc(32);
25.
26.   always        { rexists = exists; }
27.   sfg doread    { read  = 1;
28.                   acc  = acc + rdata; }
29.   sfg dontread  { read  = 0; }
30. }
31.
32. fsm fsm_avg_fsmd(avg_fsmd) {
33.   initial s0;
34.   @s0 if (rexists) then (doread)   -> s0;
35.                    else (dontread) -> s0;
36. }
37.
38. //-------------------------------------
39. // C driver for ARM core with FSL interface
40. void sendarray (int *in,
41.                 unsigned length) {
42.   volatile unsigned int
43.     *wchannel_data   = (int *) 0x80000000;
44.   volatile unsigned int
45.     *wchannel_status = (int *) 0x80000004;
46.   int i;
47.
48.   // send content of in[]  to FSL link
49.   for (i=0; i<length; i++) {
50.     while (*wchannel_status == 1) ;
51.     *channel_data = in[i];
52.   }
53. }
```

Listing 1: This HW/SW model accumulates a data stream.

- GEZEL offers access to HW/SW interfaces as library blocks in the language. These library blocks expose the pin-out of interfaces without burdening the model with unneeded internals. The library blocks reflect actual interfaces from prototyping environments; their use guarantees that the GEZEL FSMD can be easily connected to the prototype once they are converted to VHDL. Listing 1 illustrates one type of HW/SW interface (Fast Simplex Link), connected to a software driver and an FSMD. The overall operation of this model is to send an array of integers from software to hardware over an FSL link, and to accumulate the sum of these integers in the hardware model. Some features of the model are as follows. Lines 8-15 show the FSL link. This interface is modeled after the tightly-coupled Fast Simplex Link found in Xilinx' MicroBlaze processor, and has a data port and two handshake signals `exists` and `read`. The interface is attached to a core `arm1` (line 12), and this core will control the interface through memory locations `0x80000000` and `0x80000004` (a memory-based emulation of the FSL protocol is needed since the ARM simulation model used by the cosimulator does not have dedicated instructions for the FSL interface). The FSMD module that connects to this hardware-software interface is shown on lines 19-36. The datapath has an `always` instruction that executes every clock cycle (line 26), as well as two instructions `doread` and `dontread` (lines 27-29) that will only execute when told so by the FSM controller on line 32-36. The top-level hardware module, which interconnects the FSMD and the cosimulation interface, is not shown in Listing 1. Finally, a software driver that communicates with the FSMD through the hardware/software interface is shown on lines 40-53. The use of `volatile int` pointers ensures that the C compiler does not optimize the apparently redundant memory-read and memory-write operations.
- The cosimulation environment is interactive, and command-line driven. The model in Listing 1 would be captured in two files (e.g. `sw.c` and `hw.fdl`) and would be simulated by cross-compiling the embedded software, and next by running the cosimulator:

```
> arm-linux-gcc sw.c -o exec.elf
> gplatform hw.fdl
```

The design environment used by the students contains the following elements:

- A KNOPPIX CDROM with a pre-installed GEZEL-based codesign environment, including all cross-compilation and simulation tools.
- A Xilinx-based EDK+ISE environment, which is used as a backend for code created in the codesign environment.
- A Spartan-3E Starter Kit with a baseline configuration including Microblaze, on-chip timer, off-chip DDR RAM memory.

4. A LAB SERIES TO INTRODUCE CODESIGN

As described in the previous section, the students in the codesign course make use of a KNOPPIX cdrom (for modeling and cosimulation) and an FPGA board with design software (for prototyping) [10]. Figure 4 shows the organization of the hands-on experiments based on these tools. An initial lab series familiarizes students with the use of the tools. That experience then converges into a *Codesign Challenge*, a competition that challenges students into building the fastest possible implementation of a given C program onto an FPGA.

4.1 Lab Series

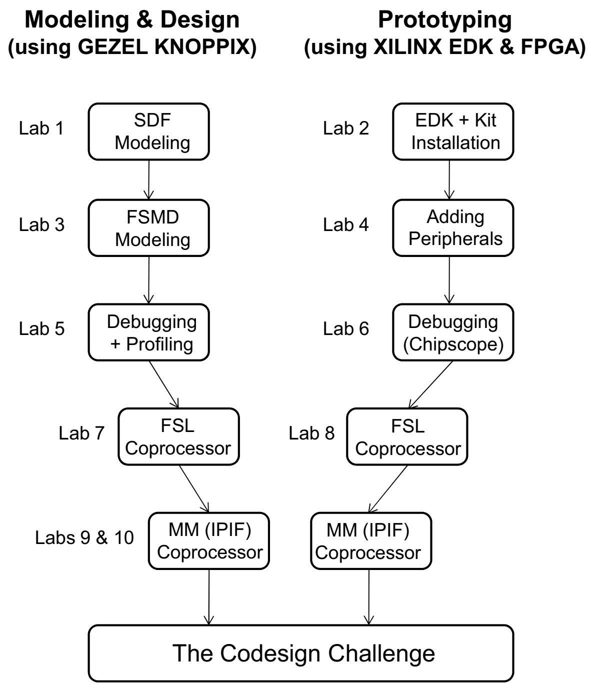


Figure 4: The Lab Series combines modeling and prototyping and merges into a 'Codesign Challenge'

A set of 10 lab projects prepare students for the codesign challenge. Five lab projects cover modeling and design; five additional projects cover the use of the FPGA prototyping environment. The two tracks of lab projects gradually converge, so that the codesign environment is coupled to the FPGA design flow. Students then are able to complete the following tasks: convert a single C program into a combination of a C program and a hardware coprocessor; verify the resulting design using cosimulation; port the coprocessor and C program to the FPGA platform; and verify the performance of the design in the resulting prototype.

The modeling assignments include SDF & FSMD modeling, profiling of embedded software using an instruction-set simulator, and two coprocessor designs. The coprocessors use a tightly-coupled Fast Simplex Link and a general-purpose Memory-mapped interface.

In the complementary prototyping assignments, students learn to take the output of the modeling flow (software driver and generated coprocessor VHDL) and connect that into the FPGA environment. They also familiarize themselves with the numerous available platform architecture parameters (location and size of memories, configuration of buses, optimization during software compilation and hardware synthesis, etc).
The combined use of GEZEL cosimulation and EDK synthesis is done for several reasons. First, EDK has a steep learning curve

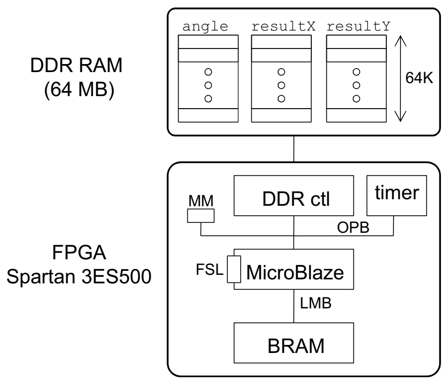


Figure 5: The 'Codesign Challenge' in fall 2007 was to optimize 64K CORDIC rotations.

compared to the simplicity of GEZEL. Second, even though EDK can hide many of the details of the underlying machine, the cosimulation mechanism is still quite cumbersome as it requires elaboration of the hardware model. Finally, the students that take the codesign course may not have taken a course on VHDL or Verilog yet, since the latter is an elective at the authors' institution. However, it should also be clarified that the author has not yet attempted to organize this lab sequence solely using EDK.

4.2 Codesign Challenge

The Codesign Challenge is deliberately set up as an open-ended assignment to improve the performance of a given C specification as much as possible. The constraints are the reference specification, the resulting platform, and the design time (two weeks). The initial ranking of results is based on absolute performance of the student designs, even though the in-class discussion of the results is based on Pareto-optimality of performance and resource usage. In the following, we describe the results of the Codesign Challenge we ran in the fall semester of 2007. In that class, 28 students participated in the final project.

Figure 5 shows the target platform for the Codesign Challenge. A DDR RAM memory contains a vector `angle` with 65536 values. The design in the FPGA must read this vector, and transform each element in the vector to an (X,Y) tuple representing the sine and cosine of that angle. The result is stored in the DDR RAM. A CORDIC algorithm with 20 successive rotations is used for this transformation [11]. The reference design in the FPGA includes a MicroBlaze processor with local memory and a peripheral bus system with timer. The students also received a software implementation of the CORDIC design for this reference platform. The performance of their design is measured as the wall-clock time needed to rotate 64K vectors stored in off-chip memory.

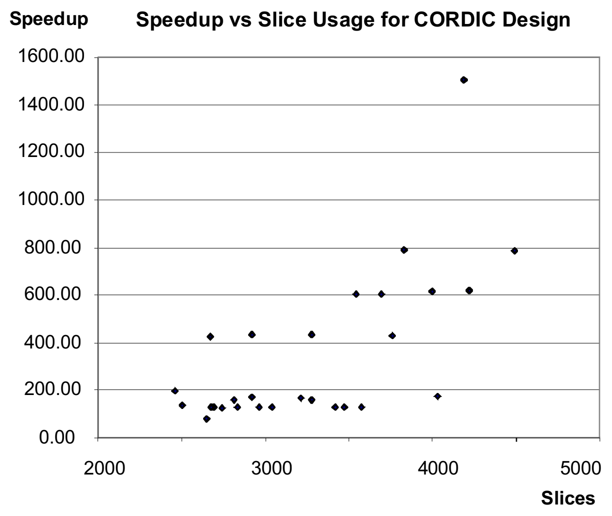


Figure 6: The results of the Codesign Challenge. Design performance is measured as a speedup over the intiial reference implementation. Each dot is one student.

After two weeks of design time, the students turned in their results. Figure 6 reflects the speedup of their design (over the reference software implementation) versus resource usage. A wide variation can be observed for each parameter - as could be expected in such an open-ended assignment. Figure 6 shows that the student designs can be partitioned into in two groups: those with a speedup lower than 200, and those with a better speedup. All of the students implemented a hardware accelerator for the CORDIC computations, with a typical design computing a rotation in 20 clock cycles. However, as can be observed from the system architecture in Figure 5, each rotation requires three off-chip memory accesses, and the performance of the reference design was around 20 cycles per memory access. The designs bounded at a speedup of 200 are those that did not take this bottleneck into account.

About one third of the class figured that hardware acceleration alone would not save the day and that further optimization of the system architecture was necessary. The results obtained in their designs illustrate the case that embedded system architecture design goes beyond hardware/software partitioning. Among the optimizations performed by this group of students are the following:

- Allocate multiple coprocessors, and exploit the parallelism available in 65536 independent CORDIC rotations;
- Write driver software that overlaps input/output communication with coprocessor computation;
- Move the `.text` segment of the code to on-chip memory and free up the system communication bus.

Two students from those with previous Verilog experience decided to develop hardware directly in Verilog (rather than in GEZEL) for compactness and performance; however, the resulting system was harder to debug because cosimulation at RTL is inadequate and time-consuming.

Another interesting issue is how students allocated design time under a limited time budget. The best designs were those that focused (almost exclusively) on the embedded system architecture and the implementation of system-level data streams. Those top students were also able to decide how much effort a given optimization was worth, because they started by estimating the performance limits in their platform.

5. CONCLUSIONS & FUTURE WORK

The breakneck speed of technological development enables senior students to complete in a class project what was considered a high-end design 10 years ago. The technologies and tools are available, and they are cheap. However, the computer engineering curriculum is lagging. The idea that digital design education should start with digital gates is losing its relevance when designers are no longer concerned with individual gates. Educators are in need of more effective design abstractions. We have used hardware/software codesign as a step towards structured thinking about embedded systems architecture design. It is not possible to cover each programmable technology that is being proposed today. Hence, our objective is to produce engineers that can quickly adapt to new programmable systems architectures.

As a positive note, the embedded systems design space has never been more interesting and offered more opportunities, for students and educators alike. There are significant opportunities available in the computer engineering curriculum for new tools, subjects, and course projects.

6. REFERENCES

[1] M. Grimheden, M. Torngren, ”What is Embedded Systems and How Should It Be Taught?” ACM Trans. on Embedded Computing Systems (TECS), 4(3): 633–651, ACM Press, NY, 2005.

[2] P. Schaumont, "A Senior-level Course in Hardware/sofwtare Codesign,", MSE 07, 7-8, June 2007.

[3] R. Chamberlain, J. Lockwood, S. Gayen, R. Hough, and P. Jones, "Use of a soft-core processor in a hardware/software codesign laboratory," in Proc. IEEE Int. Conf. MicroElectronics System Design Education, Anaheim, CA, 97-98, June 2005.

[4] C. Bieser, K.D. Muller-Glaser, and J. Becker, "Hardware/software co-training lab: From VHDL bit-level coding up to case-tool based system modeling," in Proc. IEEE Int. Conf. MicroElectronics System Education, Anaheim, CA, 134-135, June 2005.

[5] S. Tsao, T. Huang, C. King, "The Development and Deployment of Embedded Software Curricula in Taiwan," ACM SIGBED Review, 64-72, 2007.

[6] J. Sztipanovits, G. Biswas, K. Frampton, A. Gokhale, L. Howard, G. Karsai, J. Koo, X. Koutsoukos, D. Schmidt, "Introducing Embedded Software and Systems Education and Advanced Learning in an Engineering Curriculum," ACM Trans. on Embedded Computing Systems, 4(3):549-568.

[7] D. Patterson, J. Hennessy, "Computer Organization and Design: The Harwdare/Software Interface," MKP Publishers.

[8] E. Lee, D. Messerschmitt, "Synchronous Data Flow," Proc. IEEE, September 1987.

[9] GEZEL homepage, `http://rijndael.ece.vt.edu/gezel2`

[10] P. Athanas, C. Patterson, "A Holistic Approach towards a Unified CpE Laboratory Platfrom", MSE 07, 73-64, Juen 2007.

[11] R. Andraka, "A Survey of CORDIC Algorithms for FPGA's," Proc. FPGA 1998, 191-200, 1998

7. APPENDIX A: Course Topics

The following enumerates the lecture topics of the course. This material is covered in approximately 22 lectures, excluding reviews, midterms, and exam.

Fundamentals

- What is hardware-software codesign?
    - Flexibility versus performance
    - Concurrent and sequential specifications, parallel implementations
    - Modeling abstraction levels
- Synchronous Data-flow Modeling (as an example of concurrent specification)
    - Semantics of SDF
    - Analysis of SDF
    - Implementing SDF in Software
    - Implementing SDF in Hardware
- Control Edges and Data Edges (of a C program)

The Custom Architecture Design Space

- FSMD: a systematic but inflexible model for hardware
- Micro-programmed Architectures: Flexible, scalable control, but hard to program and optimize (e.g. pipelining)
- General-purpose Embedded Cores: Flexible, scalable control, easy to program, but hard to specialize
- System-on-Chip: Flexible, scalable control, easy to program, feasible to specialize

The Custom-Hardware/Software Interface

- On-chip busses
- Memory-mapped interfaces
- Coprocessor (dedicated processor-HW) interfaces
- Control Design in Co-processors
- Intellectual-property Interfaces
- Advanced solutions
    - ASIPs
    - Using C for Hardware Design

Interdisciplinary Teaming as an Effective Method to Teach Real-Time and Embedded Systems Courses

James R Vallino
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY 14623, USA
+1.585.475.2991
J.Vallino@se.rit.edu

Roy S Czernikowski
Department of Computer Engineering
Rochester Institute of Technology
Rochester, NY 14623, USA
+1.585.475.5292
rsceec@rit.edu

ABSTRACT

The body of knowledge for engineering real-time and embedded systems spans multiple computing disciplines. To effectively prepare students to work in these areas requires coursework that uses an interdisciplinary approach. This paper describes the approach that Rochester Institute of Technology's Departments of Computer Engineering and Software Engineering developed. This approach uses a cluster of three courses which cover a range of topics in real-time and embedded systems engineering. Students in each discipline take the courses, and teams of two, with one student from each discipline, work on all course projects. The paper describes the cluster of courses, their evolution over the last five years, and the laboratory in which the classes are taught. We present evaluation data to show the courses' effectiveness increasing student interest in real-time and embedded systems, and helping them obtain employment in the area.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**] – *real-time and embedded systems,* K.3.2 [**Computers and Education**] Computer and Information Science Education – *computer science education*.

General Terms

Design

Keywords

Real-time and embedded systems education, real-time and embedded systems courses.

1. INTRODUCTION

The standard computing curricula concentrate primarily on general-purpose desktop applications. The demands and requirements for these systems are notably different than those for real-time and embedded systems. Without coursework that specifically addresses the special requirements for these systems, students will not have the opportunity to gain the necessary skills for engineering software in real-time and embedded systems. Additionally, the current interdisciplinary nature of the real-time and embedded systems profession intertwines intimate knowledge of both the hardware and the software operating the components. Many traditional courses have worked exclusively with small microcontroller projects. This unfortunately does not reflect the breadth of the current field. We set out to develop our approach so that our course cluster and laboratory facilities encapsulated this reality and provided our students with exposure to a broader range of skills needed for entry-level engineering of real-time and embedded systems.

We presented our original work including detailed course objectives, course projects and initial evaluation of the first two courses in [4, 16]. Since that work, we have made significant changes to those two courses, and delivered the third course several times. This paper provides background on our lab, and describes the current syllabi for all the courses. We detail the lessons we learned and improvements we made to the courses as they evolved in the three years since our previous reporting. This includes the complete development of the third course. The paper concludes with our most recent evaluation data, and future directions for our work. We also have information about these courses, including password protected areas for faculty, on our real-time and embedded systems website [15].

2. REAL-TIME AND EMBEDDED SYSTEMS AT RIT

2.1 Background

In the computer engineering program at Rochester Institute of Technology, senior projects often focus on real-time and embedded systems, but there was no formal instruction in the engineering of the software for 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 addressed issues in developing real-time or embedded software; they had been chosen because they were the closest courses relevant to the domain.

We decided that the best way for us to address these shortcomings in the real-time and embedded domain in both the computer engineering and the software engineering curricula was to develop an interdisciplinary approach. The presence of students from both departments created a unique opportunity for synergy. The computer engineering students possess significant 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. They possess skills focused on the engineering of software that are more fully developed than for a student in the typical computer science program. Developing software for real-time and embedded systems is where the skills of computer engineering and software engineering students intersect.

It should be noted that all undergraduate engineering students at Rochester Institute of Technology are required to have a year of cooperative work experience before being awarded a baccalaureate degree in the five year engineering programs. These "coop" work periods are interspersed with academic quarters of study in the last three years of study. The typical students in these courses have had about nine months of work experience before entering these courses. To date, we have offered the first two courses in the sequence multiple times with some consistency; the third course has been something of an ongoing experiment that seems to be stabilizing.

2.2 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 [5] pc-104 board with timers, A/D converters, D/A converters, and digital I/O as our target systems.

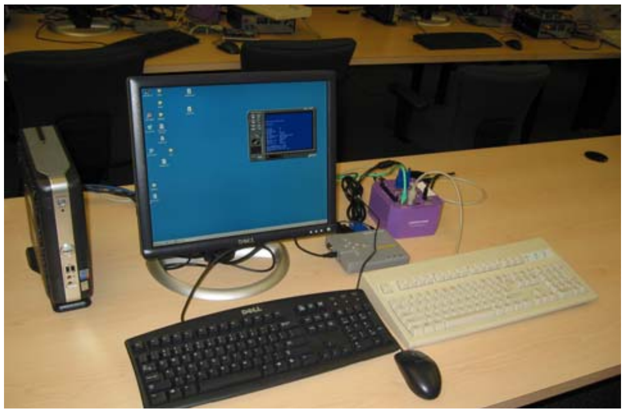


Figure 1. Basic lab station

Figure 1 above shows the basic lab work area for each student group including the development workstation and the embedded "purplebox" target system. 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. With the converter's zoom and panning capabilities students see the VGA output. 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, shown in Figure 2, is 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. The laboratory currently has two oscilloscopes that are moved from station to station, as needed.

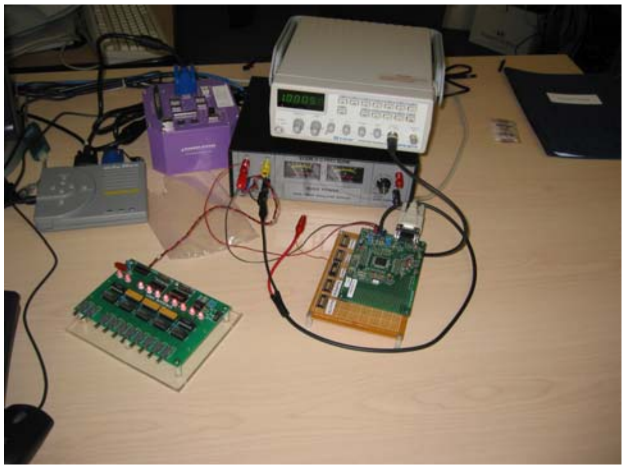


Figure 2. Microcontroller and peripherals

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 [11]. We selected their inverted pendulum and ball-and-beam systems. The last component of equipment in the laboratory is a Digilent Spartan 3 FPGA board [6]. Also in the third course, the students experiment with hardware/software co-design using this FPGA board. Each student station has one of these boards.

2.3 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 [9] supports assembly language programming on the 68HC12 microcontroller. We received a software grant from Wind River Systems [17] allowing the use of VxWorks and the Tornado development system. We are currently considering the use of the QNX Neutrino operating system environment through a grant from QNX Systems. These are the commercial real-time operating systems that the students use in the laboratory. Matlab and Simulink from The MathWorks [14] are used for simulating and controlling the Quanser experiments. We also received software grants from IBM [8] for the Rational Rose development suite and Rational Rose Real-Time as UML modeling tools. Finally, the students use Rhapsody from Telelogic [13] as a UML modeling tool. Rhapsody's statechart modeling and code generation features are used heavily in the second course in the sequence.

3. AN INTERDISCIPLINARY COURSE CLUSTER

Three courses compose our cluster in real-time and embedded systems. In July 2003, we started work on the laboratory and the development of this course cluster. Each of these upper-division undergraduate courses is four academic quarter credit hours and meets for ten weeks of classes having a pair of two-hour studio sessions per week. We offer each course once in our three-quarter academic calendar.

Each of the courses is cross-listed in computer engineering, and software engineering. The course curricula are delivered in a studio-lab environment where we mix lecture material with hands-on exercises. 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. Typically, we have not had a difference of more than two between the student registrations in the two disciplines. Depending on the course project, this is handled by creating one or two teams of three, or having one team of students from a single discipline. For some projects, we will provide additional assistance to a non-interdisciplinary team. These courses are also available to students in the computer science and electrical engineering programs, but we have had only a small number of these students registering in the courses.

3.1 Real-Time and Embedded Systems

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 (RTOS) and using it for programming projects on development systems and embedded target systems. Some fundamental material on real-time operating systems is also presented. Example topics include: scheduling algorithms, priority inversion, and configuration of a real-time operating system for a target platform and host development system. The textbook for the course is Real-Time Systems and Software by Shaw [12]. This course requires as a prerequisite either a standard Operating Systems course or the software engineering program's course Principles of Concurrent Software 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
- Language Support for Real-Time
- Real-Time and Embedded Systems Taxonomy
- Safety Critical Systems.

The project assignments for this course are:

<u>Microcontroller programming</u>: Students program the 68HC12 microcontroller to act as an interval timer. This assembly language program measures the inter-arrival time of a series of 1000 pulses using the hardware timers available on the processor. Using these timers the students see how to measure with microsecond resolution.

<u>Real-Time Operating System multi-tasking primitives</u>: 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 had been done within a simulated target system running on the development station.

<u>Real-Time Operating System performance measurements</u>: There are two smaller projects that fall into this category and are 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 periodic pulses. The second project measures the interrupt response time of the system by having the microcontroller measure the time between generating an interrupt signal to the target and receiving a response from the target.

<u>Final project</u>: There is a final programming project. This project is usually 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.

3.2 Modeling of Real-Time Systems

The second course is titled Modeling of Real-Time Systems. The course takes an engineering approach to the design of these systems by describing the system characteristics via UML models before beginning implementation. This course has the same operating systems course or concurrent systems course prerequisite. Students who take the first course prior to this course have a small advantage, but we have worked to provide sufficient resource materials for students who have not taken it.

The textbook for the course is <u>Doing Hard Time</u> by Douglass [7]. The course covers the following topics:

- 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. Initially, the projects progressed 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 modeling practice using the UML, similar to what they do in all the courses in their software engineering 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 took the lead on most projects. Many upper-division computer engineering students have not done any modeling in the UML since their second-year software engineering course.

The strong software engineering emphasis in this course has caused some problems with maintaining computer engineering student enrollment. We do not want to have the situation where the computer engineering students feel that their expertise is not required for most of the projects in this course. With each offering of this course, we have worked to shift the content more toward the computer engineering program. The first course project was a requirements analysis and design assignment. The students received a copy of the user manual for a consumer device. The devices we used included a blood pressure monitor, pedometer, and combined binocular/digital camera. Using the manual, the students identified the actors and use case requirements for the product, and then did a class-level design for an object-oriented implementation of the system. No implementation was done in this project. Our experience is that doing software only requirements moved too far from the computer engineers' expertise. In a recent offering of the course, we dropped the class-level design from this project. We want to keep some aspect of working with requirements because it is crucial to get the requirements correct on any project [2], but this project will require further modifications to bring it more in line with the computer engineering students' interests.

In our experience, the computer science, computer engineering, and software engineering students all gravitate to static class-level modeling with ease. Most of the students feel comfortable creating a design and drawing a UML class structure diagram. Capturing the dynamic behavior of the system is much more difficult for the students. Many real-time and embedded systems have state-based behavior. A significant part of this course discusses UML statecharts as a mechanism for capturing dynamic system behavior.

The second project is a design and implementation project with a major emphasis on manual implementation of the statechart that describes the system behavior. We have used a cyclometer, and a chilled water controller for this project. With the last offering of the course, this project has moved from a standalone application to an application running on our target systems under an RTOS. The program must interface with the FPGA board which provides access to it 7-segment displays, LEDs, buttons and switches. The FPGA development board acts as the user interface controller.

The Modeling course makes extensive use of on-line discussion areas. This is a place where we use "low-stakes" grading of writing assignments that do not carry much course credit. Through multiple offerings of this course, we have also increased the number of in-class exercises that students do. These help reinforce the lecture material that is covered.

3.3 Performance Engineering of Real-Time and Embedded Systems

The third course is titled Performance Engineering of Real-Time and Embedded Systems. The objectives for this course are for students to explore aspects of real-time and embedded systems with an emphasis on measuring their performance. The Performance Engineering course has the Real-Time and Embedded Systems course as a prerequisite. Students without that course who have taken operating systems or concurrent systems have been successful in this course. Based on that experience, we are considering changing the course prerequisites to be the same as for the other two courses. Topics covered by this 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 list above is an unusual combination of topics that is not covered in any single textbook. We cover the course topics with class discussions and exercises, handouts, and references to on-line resources for the students. The course splits approximately in half between real-time and embedded topics. We have offered this course three times, and with each offering we have made major modifications trying to achieve the course's objectives. At this point, most of the material from the course's first offering has been replaced with new class material and projects.

The real-time part of the course comes first in the syllabus. Real-time scheduling algorithms are discussed in detail in the first course and briefly in the Modeling course. In this third class, the discussion and exercises review real-time scheduler theory and algorithms including rate-monotonic, earliest deadline first, and least slack time. For their first project, students design and implement a testbed in which they can experiment with several scheduling algorithms. The testbed executes on our target machines running under the real-time operating system (RTOS) environment. We have a class exercise which introduces the RTOS environment and a paper and pencil exercise determining task execution timing for different scheduling algorithms. The students implement their scheduler outside of the operating system kernel because the learning curve for replacing the RTOS's scheduler would be too steep. We have an extensive class discussion about how to structure their testbed to accommodate both fixed priority and dynamic priority scheduling algorithms. This requires careful consideration of the testbed's synchronization mechanism. The students design experiments to test the schedulability limits for several scheduling algorithms comparing their results to the theoretic limits.

The next real-time systems topic is a basic discussion of control systems. The computer engineering students have seen Z-transforms, though only a small number have taken a digital controls course. The software engineering students have none of that background. Obviously, our coverage of real-time control cannot be from a deep control engineering perspective. Instead, we try to provide an intuitive perspective, and have students concentrate on the issues surrounding the implementation of linear control algorithms. A lecture introduces Z-transform notation and students work on a class exercise to implement a simple transform. This is a new experience even for the computer engineers who had a digital controls course which only dealt with the transforms as mathematical entities within Matlab. The project requires students to implement a standard proportional-integral-derivative (PID) controller on the target systems under the RTOS. The plant they are controlling is a simulation running on the development workstation using the Control System Plant Simulator (CSPS) [3]. We have extensive class discussion about how to structure their controller including issues of the timing of analog input and output conversions, and identification of control algorithm values which can be calculated prior to the next time interval. The project requires students to measure controller performance and tune the PID parameters for best performance against the project's stated control goals.

The embedded systems part of the course starts with an introductory lecture on VHDL. A class exercise gets students familiar with the FPGA development environment and walks through a simple VHDL development project. All students, including the software engineers who have done no prior VHDL work, have an individual VHDL development exercise to complete. This is a simple exercise that should take the computer engineering students, who have experience with VHDL, no more than a few hours to complete. We intend the remaining embedded systems work to concentrate on hardware-software co-design, and this is the area where we have had the most difficulty achieving our conceptual goal for this course.

In the first course offering, the students performed 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. This project requires the strong VHDL experience of the computer engineering students. The image data exchange was through the parallel ports on both the target system and the FPGA development board. We intended that students would make a hardware-software co-design tradeoff by placing more device control functionality in the FPGA. At each step, the students would measure the change in system performance as the boundary between hardware and software was moved. It became clear that the largest challenge was getting reliable communication between the target system and the FPGA. Workarounds for unreliable communication overwhelmed gains made by the hardware implementation of algorithm elements.

We next tried to incorporate hardware-software co-design tools such as System C and Impulse C. We chose Impulse C and received significant support from the tool vendor during the course. The problem was again the communication between the target and FPGA, this time in the form of no Impulse C board support for the parallel port connection that we wanted to use. We asked the students to implement the necessary board support. Even with the extensive vendor support we received, partway through this exercise we realized that it was too daunting a challenge for the students. We quickly assessed the situation, and offered options to the students. One group of four students continued through the remainder of the term to work in the Impulse C environment. Another pair continued working on the project similar to the initial course offering, i.e. outside of the Impulse C environment. The remainder of the students moved onto other projects that we hurriedly created.

We learned two things from these failed attempts at achieving our course concept for hardware-software co-design. First, we needed to eliminate the parallel port's loose coupling between the software processor and the FPGA hardware implementation of algorithm elements. The industry trend, and what is supported by the hardware-software co-design tool vendors, is to embed processor cores in the FPGA. These processors will execute the algorithm elements that remain in software. Second, the students were very motivated to work on projects that were open-ended and where they had some choice in the project details and scope.

For the third offering of this course, we created an undergraduate research-style project which ran through the last four weeks of the term. We defined several topic areas all of which had open-ended project statements that would need further defining. Each student was given an opportunity to state a preference for a project area. The instructor created the final team pairs based on these preferences. Because each project was open-ended, the team's initial task was to define the exact scope, goals, and milestones for their project. While each topic area touched on material covered in class, all the projects had areas that required the students to do extensive investigation beyond that coverage. A project might entail one or more investigations, such as, learning: how to work in a different development environment, how to model a physical system, or how to work with material done by a previous project team. We describe the topic areas next, and include information about what teams accomplished.

Control of physical devices: one student pair was assigned to the Quanser inverted pendulum, and another team to the ball-and-beam system. The project goal was to control these physical devices through the Quanser-supplied interface board, but not through the high-level Simulink interface that Quanser used. The two teams decided to use different approaches. The ball and beam team started with a previous student's project work which used the RTX Real-time Extension for Control of Windows [1]. The team calibrated the sensors and motors, and developed two different control algorithms. The inverted pendulum team installed QNX Neutrino to develop an interface to the sensors and motors, and control the device. Their work with QNX Neutrino will be very useful when we move the entire lab to that RTOS.

PicoBlaze embedded processor: three team pairs chose to explore the PicoBlaze processor core. This low block-count core can be embedded in our small Spartan 3 FPGA devices. Two teams developed an interface to a magnetic card reader, and one team measured distances with two ultrasound distance sensors. The magnetic card reader teams needed to do a lot of research work to understand how to interpret data from the card strip. Neither team was able to reliably read and interpret data, even when one team's member went to the security people at his job and asked them to code several cards with known data for testing purposes. The ultrasound team built an accurate distance measuring peripheral. These projects provided a level of hardware-software co-design experience using embedded processor cores which is an important direction to achieve our objectives for this course.

Hardware-Software Co-Design: As discussed above, our previous attempts at hardware-software co-design, which used a parallel port connection for loose coupling of the software processor and the hardware elements of the algorithm, had mixed success due to problems with communications. One student team took on the challenge to fix these communication problems. They were able to increase the error-free byte transfer rate through the parallel port by a factor of almost 10, and built a framework in the FPGA for easy switching between several different hardware implementations of an algorithm. This will be an improved base to test performance enhancements for a co-design algorithm using the hardware that we currently have in the laboratory.

Real-Time System Simulation and Control: This project asked a team to select a physical system that they could model using the CSPS system [3]. The team would need to provide a graphical user interface for the model simulation which would run on the development station. The team would implement a controller for the plant on the target systems. We could then use this work as the real-time control project in future course offerings. Unfortunately, no students selected to work in this topic area.

Microsoft Robotics Studio: This project area asked a team to explore the capabilities of Microsoft's Robotics Studio [10] as a simulation engine for physical systems. The Robotics Studio has a full physics engine embedded in it. An interface to our data acquisition system could provide a bridge between the physics engine and a target system controller. One team selected this project. They ran into a number of complications just getting the Robotics Studio to run on a development workstation. Once past those problems, the team successfully modeled an inverted pendulum system with characteristics corresponding to our Quanser inverted pendulum, and built a rudimentary controller for it. The team did not have time to do a careful validation of the operation of their simulation to that of the physical device. This work shows promise for future use in the lab if we can overcome many of the installation problems that the team encountered.

We are very happy with the results the students obtained when working on their final undergraduate research projects. This was a win-win project for students and faculty. The students had some control over the choice of their project for the last four weeks of the term. The students were motivated to do the extra investigation work, and did not complain about the vague project requirements. There are two aspects of the assignment that we feel were essential to engage the students. First, the students defined their own project direction under the guidance and final approval of the course instructor. This gave them "buy-in" to the project. Second, we made it clear that as a "research" project it was not known exactly what could be accomplished in the timeframe given for the project. Grading was not going to be strictly based on achievement of specific goals. Each team did specify initial goals for their project, and could still receive a good grade if those goals were not achieved provided that the team demonstrated "due diligence" working on the project. To monitor project work on a weekly basis, each student tracked his or her time on the project along with the tasks accomplished. During class time, each team gave a weekly 5 minute project status presentation to the class describing progress made, problems encountered, and the expected accomplishments for the upcoming week. In addition, the instructor held a private 15 minute meeting with each team once a week to discuss further details of the project progress. It was in these meetings that we were able to assess the due diligence of the team and provide feedback.

The win aspect for the faculty is twofold. By purposely stating the projects as open-ended, we escape the problem, often seen with new projects, of having to guess a reasonable project scope, and provide associated deliverable expectations. The results achieved by teams that met the due diligence requirement show us reasonable expectations. We can then use that information to rework a project into a more traditional close-ended form. Secondly, with judicious selection of the topic areas, we placed some of the burden for exploring new areas on the students. With permission to use their work, we can let future teams build upon it to develop robust project frameworks that we faculty do not have the time to implement ourselves.

4. EVALUATION

We have evaluated the effect of our course cluster using student surveys. Increasing student interest in real-time and embedded systems, and aiding students in finding employment in the area were two goals for our work. The results of all our surveys to-date have consistently indicated success in meeting these goals. We presented evaluation data from the initial offerings of these courses in [4, 16].

As is the case in many organizations, crossing organizational boundaries can create unique problems. We have had our share of them running these interdisciplinary courses. RIT's departments gain credit-hours-generated credit based on the course number. Independent of whether a software engineering or computer engineering faculty member is teaching the course, credit-hours-generated are split between both departments because of the cross-listed numbers. Our department chairs have assumed that this evens out since we have also balanced the responsibility for teaching the courses between the two departments. While the aggregate number of students registered for each course is sufficient, because the courses are cross-listed under multiple numbers, individually, they often set off the course audit warnings for low enrollment in the individual courses. Our department chairs have had to provide explanations for allowing the low enrollments. Finally, it took us a bit of time to get the scheduling coordinated between the departments in separate colleges so that in each term all sections of the course were offered at the same time, in the same room, and with the correct registration limits. Similar coordination problems existed for scheduling concurrent final exams for the courses.

The latest survey has data from fifteen students who took at least one of the three courses. This data represents about a 50% response rate from the students in each course during the 2007-2008 academic year. The numbers of students taking the courses in this cluster were: all three courses – 2; two courses – 6; only one course – 7. Six computer engineering and nine software engineering students responded to the survey. The courses helped to increase student interest in real-time and embedded systems with 87% of the respondents replying Agree or Strongly Agree to the question "These courses increased my interest in real-time and embedded systems." A smaller percentage, 47%, Agree or Strongly Agree that they plan to seek employment in the real-time or embedded systems area. Six students, 40%, Agree or Strongly Agree that taking one of the courses in the cluster assisted the student in getting a co-op or full time position.

We were also interested in the students' perception of the three individual courses, and, particularly, whether the students' found value from the interdisciplinary teaming. These most recent results are shown in Table 1 below. The table rows show data for individual questions asked about each of the courses which are organized in the columns. Each data entry is the number of students who responded Agree or Strongly Agree to the question.

Table 1. Results of survey of student perceptions

	R-T&E Sys.	Modeling	Perf. Eng.
# of students	8	8	9
Amount learned was worth the time	7	5	8
Recommend to a friend	8	6	9
Adequate preparation	8	7	8
Benefit from teaming	5	5	8

This data shows that the student perception of these courses is positive particularly for the Real-Time and Embedded Systems, and Performance Engineering of Real-Time and Embedded Systems courses. The first course has been well-received by the students from its inception. As described previously, we have made significant changes to the third course to improve the ties of its content to the real-time and embedded systems area. The students particularly liked the final "research" project which allowed them to choose their individual topic area and gave a wide range of flexibility in the direction for each project.

The results for Modeling of Real-Time Systems show some weakness in our opinion. We attribute this to the heavy software engineering emphasis of the modeling aspects of the course which the computer engineering students view as too abstract. Even some software engineering students expressed dissatisfaction because too much was a repeat of modeling that they do in several other software engineering courses. We believe that this indicates the need to make modifications to the basic content, and the nature of the projects. Our move to implement the second project on the target systems running under the real-time operating system was a step in the right direction. We present other ideas for this course in the next section.

The students were almost unanimously of the opinion that the interdisciplinary teams were beneficial in the Performance Engineering course. We attribute that to the very open-ended nature of the project work, particularly the final project which had topic areas that required thinking across the hardware-software boundary. For the first introductory course and the Modeling course, a majority of students still Agree or Strongly Agree that the interdisciplinary teaming is beneficial. The negative student comments that we received center on spending too much time instructing a partner in the other discipline. The direction of the instruction depends on whether the project has a stronger software engineering or computer engineering content. Our retort is that when you assist someone else's learning of a topic it helps solidify your own learning of the material. This does not sway those who may be focused on the amount of work that must be done to get a particular grade. It remains a problem to find projects for all three courses that provide an equal challenge to students from each discipline, and benefit from our interdisciplinary teaming.

5. FUTURE DIRECTIONS

There are two changes that we plan to make in the laboratory's hardware and software infrastructure. A major change will be switching the principal real-time operating system from the Wind River Systems' VxWorks to QNX Neutrino. We made several unsuccessful attempts to create a VxWorks board support package for our new target systems with Pentium-based processor boards. Recently, QNX Software Systems released QNX and their entire development suite in open licensing. The better support provided for QNX on our Diamond Systems targets made it easier to get QNX running on our new processor boards. We now need to retarget/redevelop all our course exercises and projects to this new operating systems environment. The second change we anticipate is moving to new development boards with larger FPGA devices.

There are a number of areas where we feel these courses need improvement. We feel that we got the first course with the right content early on, and it has seen the fewest changes. For now, our own plans for modification are to introduce another physical device control project, such as, controlling the position of a model airplane servo motor with pulse-width modulation.

The Modeling of Real-Time Systems course continues to suffer from too strong an emphasis on the more abstract software engineering modeling techniques. Requirements analysis is one topic that computer engineering students have identified as too abstract. The content of this course still needs to move closer to the hardware. We plan to remove discussion of actors and use cases for requirements analysis. We think that substituting discussion and practice in the specification of specific, well-formed requirements statements for real-time systems will better serve both the software engineering and computer engineering students. Currently, the requirements project uses a consumer device. We think that this project will have more meaning for the students, if we connect the work to the design and implementation second project. Removing some discussion of requirements analysis will provide time for more discussion of real-time design patterns which are very lightly covered now. The third project, which uses the code-generation capabilities of Rhapsody, was executed as a small standalone application. A four-function calculator and a garage door opener are two applications that we have used in the past. We will consider using the same application through the entire term by having the students autocode the application that was used for the requirements, and design/implementation projects.

The Performance Engineering course has undergone the most significant changes from its initial offering. With our latest successes, we are more willing to believe that the concept of splitting the course into two parts, real-time control of physical systems and hardware-software co-design, is sound. However, the course has not yet fully realized that concept. Before adopting a widespread physical systems control project, we need to develop simulations of the inverted pendulum, and ball-and-beam systems. This will allow students to test their controllers in simulation before moving to the physical systems, which are resource constrained.

For the hardware-software co-design aspect of the Performance Engineering course, we also must take major steps to move forward. One student project from last year did solve the problems that we had in communicating via the parallel port between a target system and the Spartan 3 FPGA board. This work should allow a hardware-software co-design project enabling migration of some software components into VHDL implementations on the FPGA. Our goal is still to use a tool such as Impulse C to do high-level algorithm development in C followed by automatic VHDL generation for the components migrated to hardware. We would, however, abandon our initial approach of loosely coupling the software processor and hardware components, in favor of a processor core embedded in the FPGA alongside the components migrated to hardware. This is the trend in industry, and the approach that the design tool vendors support. Our PicoBlaze final projects gave us positive results from students working with this approach, but to move forward we need to acquire development boards with larger FPGAs that can hold more powerful embedded processor cores.

The final project in the Performance Engineering course was well-received by the students, and we will continue these open-ended undergraduate research style projects in future course offerings. We want to improve the projects' focus by reducing the number of topic areas from which students select, and placing additional constraints on some areas to ensure that the new projects build on results of previous work.

This interdisciplinary technique for course delivery has worked so well that we hope to apply it in areas other than real-time and embedded systems. The next area under consideration is an interdisciplinary course cluster in cryptography. This cluster would have three courses and involve the computer engineering, computer science, and software engineering programs. The first course would be a modified existing introductory cryptography course offered by computer science. A second course would concentrate on secure design and implementation of software systems. The third course in the cluster would study hardware-software co-design techniques. These last two courses are new, and would use cryptographic algorithms as the motivation for all course exercises and projects. As we do in our real-time and embedded systems cluster, we would control registration to balance the number of software- and hardware-oriented students.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. DUE-0311269. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] Ardence. RTX - Real-time Extension for Control of Windows. http://www.ardence.com/embedded/products.aspx?id=70. Accessed 12 July 2008.
[2] Brooks, F. P., Jr. *No Silver Bullet - Essense and Accident*. Addison-Wesley, City, 1995.
[3] Chandler, D. and Vallino, J. Control System Plant Simulator: A Framework for Hardware-In-The-Loop Simulation. In *Proceedings of the ASEE 2008 Annual Conference* (Pittsburgh, PA, June, 2008).
[4] Czernikowski, R. S. and Vallino, J. R. Embedded systems courses at RIT. In *Proceedings of the 2005 Workshop on Computer Architecture Education: held in conjunction with the 32nd International Symposium on Computer Architecture* (Madison, Wisconsin, 2005). ACM.
[5] Diamond Systems. http://www.diamondsystems.com.
[6] Digilent. http://www.digilentinc.com.
[7] Douglass, B. P. *Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison Wesley, Reading, 1999.
[8] IBM Rational Software. http://www-306.ibm.com/software/rational/.
[9] MGTEK. http://www.mgtwk.com/miniide.
[10] Microsoft Corporation. Microsoft Robotics Developer Center. http://msdn.microsoft.com/en-us/robotics/default.aspx. Accessed 11 July 2008.
[11] Quanser Systems. http://www.quanser.com.
[12] Shaw, A. C. *Real-Time Systems and Software*. John Wiley & Sons, New York, 2001.
[13] Telelogic. Telelogic Rhapsody. http://www.telelogic.com/products/rhapsody/index.cfm.
[14] The Math Works. http://www.mathworks.com.
[15] Vallino, J. Real-Time and Embedded Systems Course Sequence. http://www.se.rit.edu/~rtembed. Accessed 11 July 2008.
[16] Vallino, J. R. and Czernikowski, R. S. Thinking inside the box: a multi-disciplinary real-time and embedded systems course sequence. In *Proceedings of the Frontiers in Education Conference* (Indianapolis, IN, October, 2005). FIE '05. T3G-12-17.
[17] Wind River Systems. http://www.windriver.com.

Multi-Processor Programming in the Embedded System Curriculum

Andreas Hansson[1], Benny Akesson[1] and Jef van Meerbergen[1,2]
[1]Eindhoven University of Technology, Eindhoven, The Netherlands
[2]Philips Research Laboratories, Eindhoven, The Netherlands
m.a.hansson@tue.nl

ABSTRACT

Teaching embedded system design is challenging, as the subject covers a wide range of aspects, and also involves skills that students do not learn from a text book. As a result, hands-on projects, with varying degree of complexity, are the most common approach in existing courses. Traditionally, the projects are limited to uni-processor systems, and do not address the complications involved in parallelising applications and mapping them to multi-processor systems.

In this paper, we describe a two-year-old embedded systems design course given at Eindhoven University of Technology. In the course, the students, divided in groups of four, are faced with the problem of putting an embedded JPEG decoder on the market within one semester. The starting point is a decoder written in sequential C and an embedded multi-processor system, running on an FPGA. We describe the ideas and organisation of the course, and give examples of what challenges the students are faced with. We exemplify results and give suggestions to instructors wishing to teach embedded multi-processor programming elsewhere.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer Science Education*

General Terms

Design, Algorithms, Experimentation

Keywords

Embedded System Design, Education, Multi-Processor System on Chip

1. INTRODUCTION

Embedded systems are characterised by the importance of non-functional requirements, i.e. hard or soft real-time constraints, a limited power budget and limited resources, such as memory footprint [17]. Furthermore, architectures must be programmable to deal with changes in applications [3]. Architectures for embedded systems are the result of a compromise between efficiency and programmability. To limit the design effort a platform-based approach is used, integrating many Intellectual Property (IP) blocks, with multiple processor cores of different types and distributed memories [12]. Examples of those platforms are Nexperia [3] and OMAP [10]. Those are developed in large industrial projects, sometimes involving hundreds of man-years with the bottleneck moving more and more towards software.

The question can be asked how to teach this at university. What should students know and how can they learn it? Are hands-on exercises possible? PC-like systems are supported by good classes (mainly in the CS dept) and excellent material, but this is not so obvious for embedded platforms. In this paper, we describe the approach followed at Eindhoven University of Technology. In the context of a master program on embedded systems, which is a joint program of EE and CS, there is a coherent set of four courses tackling the aforementioned questions. Following a bottom-up approach, the first course focuses on the lower levels of design, i.e. logic and register-transfer level synthesis used to develop IP blocks like ALUs, multipliers, memories etc. The focus is on FPGA implementation. The second course uses those IP blocks to build a wide range of processor cores spanning the whole spectrum from fully programmable microprocessors to digital signal processors and application-specific instruction set processors. The third course discusses the communication between those cores and Network on Chips (NoC) play a central role. The fourth course, Embedded Systems Laboratory, is devoted to 5 ECTS credits (120 hours distributed across 12 weeks) of hands-on design exercise, integrating the previous courses and applying the lessons learnt in those courses. This paper describes this fourth course.

The Embedded Systems Laboratory is similar to project-based courses given at other universities [2, 4, 13, 15, 18] in that it integrates skills from a diverse set of subjects, e.g. programming, processor architecture, computer organisation and NoCs. Most students have experience in these subjects, but few have any experience integrating the skills [4]. We share the observation that hands-on sessions are indispensable to acquire the necessary skills [2], with a good balance between practical knowledge and fundamental understanding [4]. The course therefore consists of one large project, focusing on the process of mapping a particular behaviour, in this case a JPEG decoder, onto an embedded multi-processor platform. The assignment emphasises the growing importance of software in embedded systems [13, 15], and resource-limited performance-oriented design [13, 19], but also involves challenges in areas like personal time management and teamwork, similar to [4]. In contrast to the aforementioned works, the emphasis is on the challenges involved in going from uni- to multi- processor systems, and the importance of communication and synchronisation.

The remainder of this paper is organised as follows. In Section 2 we discuss the starting point of the assignment, being the given application, the hardware platform, and the associated tooling and development environment. Section 3 focuses on the assignment itself, explaining what the students have to do and how they are organised. We also discus our interaction with the student groups and give a structured overview of the course. Next, we elaborate on what challenges are involved in porting the application to the hardware platform in Section 4 and discuss the parallelisation in Section 5. Section 6 highlights the performance evaluation, followed by a discussion in Section 7. Finally, conclusions are presented in Section 8.

2. ASSIGNMENT STARTING POINT

In this section, we discuss the starting point of the assignment, which is to map a JPEG decoder onto a multi-processor platform. We start by giving a brief introduction to the concepts of JPEG decoding in Section 2.1. We then proceed by presenting the hardware platform and its building blocks in Section 2.2. We end this section with an overview of the development environment in Section 2.3.

2.1 Application

The application used in the course is a fully functional JPEG decoder written in ANSI C [7]. Decoding a JPEG image is a non-trivial task involving similar steps as many other media codecs, such as MP3, AAC, and H264. The core of all the aforementioned standards is a discrete cosine transform, that transforms data into the frequency domain. To achieve a good compression ratio, the transformation into the frequency domain is combined with other techniques, like quantisation and run-length encoding.

The JPEG decoder can be generalised into three main decoding stages, shown in Figure 1: Variable Length Decoding (VLD), Inverse Discrete Cosine Transform (IDCT) and Colour Conversion (CC). The VLD decodes the variable-length encoded JPEG data, dequantises it and arranges it into blocks of 8x8 values, referred to as minimum coded units (MCU). The MCUs initially contain frequency data, but are transformed from the frequency domain to the pixel domain, and up-scaled if necessary, by the IDCT step. After this transformation, the blocks contain image data in the YCbCr colour format that the CC step converts to RGB.

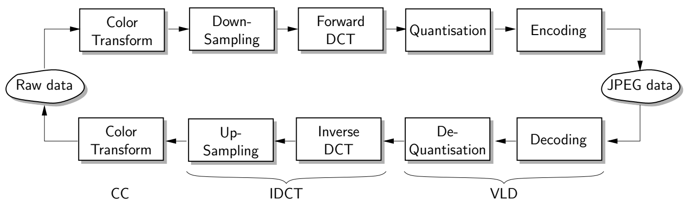


Figure 1: JPEG encoding and decoding.

The JPEG decoder is suitable for the course for a number of reasons: 1) It offers a reasonable amount of code to familiarise with. Some of the students have never encountered C before, and the JPEG decoder proves to be manageable also for those students. 2) The original decoder makes use of dynamic memory allocation and file I/O, something that is not natively supported by the target embedded platform. 3) The simpler JPEG decoder retains the technical complexities of its audio and video counterparts, thus giving the application good educational value. 4) The decoding is data dependent, in that the different decoding steps require a varying amount of computation (and communication) for different images. 5) JPEG decoding is not trivially parallel. The VLD is inherently sequential, whereas the IDCT and CC are easy to parallelise. This leads to a wide variety of parallelisations, trying to overcome the restrictions imposed by the JPEG format. 6) The decoder is small enough to fit in the local instruction memory of one processor. Thereby, it is possible to break the assignment into several steps, where the decoder is first ported, then optimised and parallelised. Solving one problem at a time also gives intermediate deliverables and deadlines, something we have found necessary in tracking the progress. 7) The decoder has the benefit of being familiar to the students and fun to work with, as the results can be presented on a screen attached to the actual hardware platform. As also observed by Edwards [4], video is visually satisfying when it works, and it can be debugged by inspecting the displayed images. 8) By decoding sequential JPEG images, the decoder is turned into a primitive video player, adding real-time aspects to the assignment, without causing faulty behaviour if the requirements are not satisfied. Again, this helps in splitting the assignment into manageable parts.

2.2 Hardware platform

The platform used in the course builds on the concept of using multiple distributed computational and storage resources, interconnected by a scalable NoC. Using this type of programmable embedded platform in the course is representative for signal-processing architectures where low power, and support for many features and standards is imperative.

Like Edwards [4], we want the students to experience real hardware, and not only simulation or modelling. This is made possible thanks to Silicon Hive [20], providing their low-cost, low-power domain-specific Very Large Instruction Word (VLIW) processor cores, and NXP, providing the Æthereal NoC interconnect fabric [6]. The students hence work with industrially relevant IP components and tools. Using an actual hardware instance of the platform increases the amount of issues that the students have to solve, with more tools to manage and more things that can go wrong. It also increases the amount of work (and risk) involved in teaching the course. The reward, however, is that the students can see tangible results of their efforts, something that has shown to be a great motivator.

The architecture we present the students with, depicted in Figure 2, consists of three uniform Silicon Hive cores, a large off-chip SRAM, and a frame buffer for video output. In

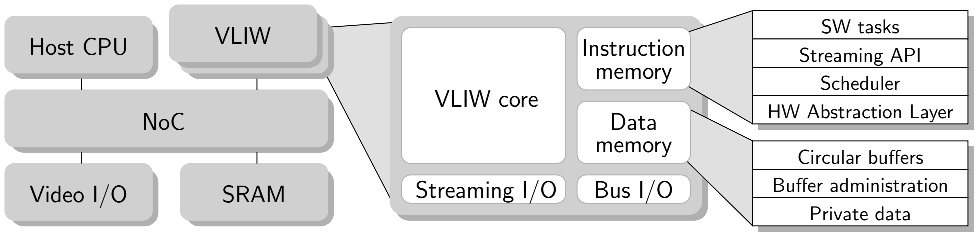


Figure 2: Architecture template.

addition, a general-purpose host CPU is attached to the system. The different components are interconnected by an instance of the Æthereal NoC [6]. A brief discussion on the different building blocks follow.

The Silicon Hive VLIW cores are customisable, making it possible to adapt the costs and the performances of the various computation nodes to a given application. For the course, we use a simple, three-issue-slot architecture, without a floating-point unit. The cores use a memory-mapped architecture and have a master interface to enable reads and writes to memories external to the processor. As shown in Figure 2, every processor also has its own private instruction and data memory. The memories are also accessible through a slave port on the processors bus interfaces, forming a distributed memory together with the dedicated memory tiles. The challenges that arise due to the selected processor architecture are: 1) the application must use fixed-point arithmetic, 2) multiplication and load/store operations compete for the same instruction slot, 3) the application must fit in a local memory of 32 kbyte (which is chosen to just barely fit the complete JPEG decoder), 4) the processor core has no caches and requires explicit memory management, 5) the core has no operating system, thus leaving any task scheduling to the programmer.

In addition to the 32 kbyte of data memory in each core, a central memory tile, here after referred to as *external memory*, provides 8 Mbyte of SRAM. While being significantly larger, the external memory has an access time an order of magnitude larger than the local memories. This is due to the traversal of the NoC, and the sharing of the memory read and write port. The platform instance has only one external memory, as is commonly the case, either for cost reasons or due to a limited number of pins [14]. In addition to the background memory, the cores can also write to a frame buffer, where a designated display controller presents the contents on a DVI output port. This functionality is used during the laboratory sessions to get immediate visual feedback of the results. Note that in contrast to [4], the students do not have to implement any low-level drivers to interface with e.g. DVI and USB interfaces. We believe that earlier courses are sufficient in teaching the lower levels of design, and leaving these elements out gives more time to the higher-level issues we want to emphasize.

As seen in Figure 2, the system contains not only VLIW processing cores, but also a host CPU. Like in the IBM Cell [11], the host is a general-purpose processor that is responsible for the initialisation and orchestration of the hardware resources, e.g. loading the binaries to the VLIWs and configuring the NoC connections [8] and memory arbiters. Although the host processor is typically a part of the SoC, we choose to map the host interface on the NoC to an external PC. This enables the students to use their own PCs, running Linux, as host CPUs during the labs.

All the ports of the aforementioned hardware blocks are physically interconnected by the Æthereal NoC [6]. In Æthereal, the different master and slave interfaces are logically interconnected by *connections*. A connection can be seen as virtual wires, offering a certain throughput and latency guarantee. Together, a set of connections forms a *use-case*, which acts as a virtual on-chip infrastructure. Network resources are pre-allocated for a number of given use-cases, using the UMARS tool [9], and it is left as an exercise for the students to choose an appropriate use-case for their specific JPEG decoder implementations. The hardware is thus fixed, but the programming of the NoC is chosen from a set of pre-computed use-cases [8]. The students select a use-case based on which master and slave ports that need to be interconnected for their specific parallelisation, and what throughput and latency the desire between the different connections.

A complete architecture instance is mapped to a Xilinx Virtex4 LX-160 FPGA [21], fitted on an Agility RC340 board [1]. In contrast to [4] that uses many inexpensive boards, we are using a single board, which sells for about $10.000. Since we are targeting multi-processor systems, a fairly large FPGA device is needed (our current system uses more than 50.000 LUTs). Moreover, the RC340 offers a range of peripherals (audio, video, USB, memories) that are particularly attractive, as it enables results to be shown in far more elaborate ways than blinking LEDs. Interfacing with the peripherals is also fairly straight forward thanks to APIs from Agility.

The FPGA board is available in the classroom during the laboratory sessions, connected to a server PC via USB, and the students connect to this PC, in turn, via the local network. Through this network connection, they can upload bitfiles to the FPGA, and also have their PCs act as the host CPU, once the device is configured. By allowing network connections to the server with the board it is also possible to work remotely between sessions, something that is greatly appreciated and extensively used by the students. By these means, having a single board for 30 students has proven to be manageable, although more boards would of course be advantageous. Only during the last week of the course, when all student groups are preparing for the evaluation, did we experience that students lost considerable time just waiting to access the board.

2.3 Development environment

The Silicon Hive cores are supplied together with a retargetable compiler, assembler and linker, as well as a complete simulation environment. In the Silicon Hive development environment, a number of steps are possible, going from fast checking of the functional correctness, to cycle-accurate simulation. First, all code is compiled with `gcc`, to verify that the algorithm is working for a supplied set of reference images. Second, the code that is to be run on the cores is compiled with `hivecc`, but not scheduled to instructions slots, registers, etc. This enables the programmer to generate code with instruction semantics of the specified core. Third, the compiled code is scheduled to maximise Instruction Level Parallelism (ILP), and the programmer thus gets a complete view of the utilisation of the core's resources, i.e. the register files and intra-core interconnect. In this step, the tools also provide feedback about memory usage, instruction slot scheduling, and detailed profiling information. The fourth and last step uses the FPGA with the student's computer acting as a host. The host then loads the microcode to the embedded cores on the FPGA and starts the execution.

The availability of development tools and support libraries removes much tedious and error-prone work, and thus enables the students to focus on the higher-level issues involved in programming multi-processor systems. The one big problem with the proprietary tools is that it complicates the interpretation of any potential error messages. When using `gcc`, for example, Google can most likely help to decipher compiler and linker errors. That knowledge now has to come from the instructors. Teaching assistants should thus be familiar with the tools and the architecture. In our case, all teaching is done by people that are actively using, extending and researching on the platform. We share the view of [15], that integrated development environments shield the students from the compilation process. Indeed, many students express that they develop a new level of understanding after using `make` and a command-line based cross-compilation environment.

To even further reduce the amount of low-level programming, we provide the students with functional example programs (although not optimised) that demonstrate how to read and write to the background memory, closely mimicking the behaviour of `fgetc`, `fseek` and `ftell`. They also get example code that shows how to interface with the frame buffer and the display controller. As a result, already during the first lab session, the students start with a simple example application (adding two numbers), adapt it for the embedded platform, simulate until they achieve the desired behaviour, and run it on the actual FPGA. We have found that, in contrast to our first beliefs, any help on this stage saves a lot of valuable time, without compromising the educational value of the course. In fact, we believe that the relevancy of the course is improved by moving the focus from idiosyncratic APIs and specification formats to higher-level issues.

3. ASSIGNMENT OVERVIEW

In this section we describe how the assignment, to parallelise and map the JPEG application on the presented hardware platform, is actually carried out, from an organisational point of view.

3.1 Course structure

Already from the first laboratory session, the students are divided into groups of four people and presented with a problem: *Put an embedded JPEG decoder on the market in three months*. Similar to [13], we treat each team like a start-up, and effort is made to ensure that all groups are multi disciplinary and multi cultural, and hence contain students with different educational and cultural backgrounds.

During the first week, the students focus on familiarising with the development environment by mapping educational examples to the platform. This is accompanied by lectures introducing the VLIW cores, the NoC, the FPGA, the support libraries etc. The slides, together with a wide range of publications on the architectural building blocks, form the lecture material. Moreover, the students have access to a Wiki, also containing useful information from past year's courses. All of this is available at the course web site [5].

After an initial week of introductory exercises of tutorial nature, the teams assign roles with different responsibilities to their members, much like what is proposed in [13]. The four roles are: 1) application expert, 2) architecture expert, 3) embedded programming expert, and 4) group leader. The task of the application expert involves learning the details of the JPEG decoding algorithm, and to identify the important functions in the code and their interfaces. The architecture expert focuses on the details of the processing cores, NoC and memories. The embedded programming expert learns how to port and upload code to the embedded VLIW core, and how to use the system support libraries. Lastly, the group leader is responsible for dividing the work among the members, reporting the team progress, and helping the team wherever needed.

Experience shows that the workload quite often turns out to be unevenly distributed. Many students suggest to distribute the four people over two sub-groups, focused on exploring different parallelisations. This also gives the benefit of always doing pair programming. After roughly half the course, six weeks, most groups resorted to such an organisation, and in future instances we will recommend such an approach. Furthermore, our experience is that groups should not be larger than four persons. During the last instance of the course, we had a few groups of five people. When asked after the course, all but one such group reported that it would even have been advantageous to have four members instead of five.

Once the roles are assigned, the work on the actual JPEG decoder starts. The work is divided in three distinct phases: 1) porting the application to execute on a single core on the target platform, 2) parallelising the application to use multiple cores, and 3) optimising the solutions to improve performance. The step-by-step arrangement is important as it makes it easier for the students to organise their work into smaller, yet meaningful parts. It also simplifies setting partial deadlines that we follow up on by means of a group page on the course Wiki. Updating this page is one of the most important responsibilities of the group leader. In addition to the Wiki, we also discus the progress of the individual groups during bi-weekly meetings, where each group gets roughly 20-30 minutes time with the teaching assistants.

After two or three weeks of the course, once all groups have gotten sufficiently far to be able to contribute, we also arrange meetings for students with specific roles. In these meetings we discus the difficulties (and any potential solutions) that are unique to the FPGA expert, group leader, etc. This allows the groups to help each other with organisational as well as implementation issues, while still maintaining a competitive atmosphere between the groups. Also in the classroom, we encourage students to ask their peers (also outside their group) for help before consulting a teaching assistant. Our experience is that this approach works well and that the groups still present unique solutions.

3.2 Examination

To pass the course, each team has to present: a demonstration of at least two working solutions, a design document of about six pages, and a group presentation (and demonstration) for the rest of the class. Since students are graded individually on a scale from one to ten, where six and above is a passing grade, each student also gives an individual presentation and present their individual contributions during an oral exam. Thanks to the interaction in the classroom, most of the grading can be done before the individual presentations, but having both is beneficial for the slightly less extrovert students. Moreover, the individual presentations are also an excellent opportunity to get the students' opinions on other fellow students. Often, the students are remarkably honest and not afraid to share their opinions. Similar to [13], grading is based on visible, concrete contributions to the final solutions (based on what role the student had in the group). They are evaluated as application experts, group leaders etc.

4. PORTING THE APPLICATION

The JPEG application is distributed as sequential C code that executes on a normal desktop PC. The first challenge of the design teams is to port the code to execute on a single VLIW core. The major issues to solve involve memory management, and handling of console and file I/O.

No standard library function is provided for dynamic memory allocation, since the memory architecture is non-uniform, creating multiple placement options. Memory allocations are hence done statically, and the programmer determines if a particular variable should be mapped to the limited amount of faster local memory of the core, or to the larger but slower external memory. A challenge in this step is that statically allocating arrays requires algorithmic knowledge from the programmer, since they must be dimensioned for the worst case.

The application, in its original form, makes rich use of the console to print debug information in case there is something wrong in the implementation or the encoded image. The target embedded system has no means of outputting this information, since it does not have a console. Printing this information is, however, very useful to limit debugging time in case errors are introduced in the code during the porting effort. For this reason, these commands are not removed from the code, but rather redefined to empty statements by the pre-processor before compilation for the VLIW core. This allows all debug information to still be printed if the code is compiled for a regular computer to verify its functional correctness.

The original JPEG decoder uses file system I/O to read the encoded bit stream and to write the decoded image. However, the provided architecture does not have a file system. Instead, the core must read the encoded image from the external memory, which is the only memory large enough to store it, and write the decoded image to the frame buffer. The host is used to transfer the encoded image from the file to the external memory, which requires familiarity with the system support libraries for communication between the host application and the FPGA. The decoded image is also written back to external memory during development, allowing the host application to read the output and compare to a reference image that was decoded before porting the code. Automating this procedure allows bugs introduced during porting to be discovered quickly.

During the porting, the students learn to appreciate the different refinement steps of the development environment. It quickly becomes apparent that the FPGA offers tremendous speed, but complicates debugging due to the lack of observability. This is partly a result of how the students access the FPGA, as the network connection makes it difficult if not impossible to use facilities like ChipScope [21]. Most groups resort to using the cycle-level simulation environment, and only use the hardware for the final functional verification. Approximately four weeks into the course, the decoder is ported and working on the FPGA. This is when the parallelisation begins.

5. PARALLELISING THE APPLICATION

After successfully porting the application to the target platform and performing initial benchmarks, the design teams proceed by parallelising the application to make use of multiple cores. As mentioned in Section 3, the assessment criteria require each group to implement and benchmark at least two different parallelisations. The two most common solutions involve exploiting *data parallelism*, by allowing multiple cores to work on different parts of the image, and *functional parallelism*, where the decoding functions are mapped to the different cores. Many variations of these solutions have been explored during the course, including hybrid versions that aim to combine the best of both. In this article, we limit the discussion to the two basic solutions, which are presented in Sections 5.1 and 5.2, respectively.

5.1 Data parallelism

The idea of a data parallel implementation of the application is that multiple cores are assigned to decode different parts of the image. A benefit of this approach is that very few changes are required to the ported code executing on a single core. All cores execute the same program, but use a unique identifier to determine which part of the image to decode. This parallelisation is so simple that some groups even manage to go from a single-core decoder to a first data parallel decoder during one lab session of three hours.

The first question that the students have to answer is how to divide the image among the cores. Different strategies distribute the complexity of the image differently among the cores. This is illustrated in Figure 3 where the image is par-

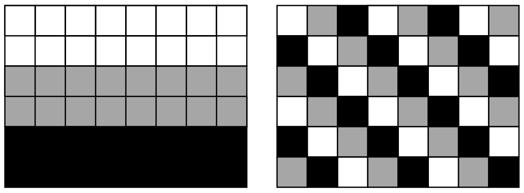


Figure 3: Strategies for data partitioning

titioned among three cores according to the different shades of grey. Dividing the image in three horizontal slices, as done in the left part of the figure, would create an unbalanced load in a scenic picture with a blue sky in the top, since significantly less computation is required by the IDCT for this part of the image. Another strategy that is better in this respect is tiling, shown in the right part of the figure, where a core decodes every third MCU block.

The main drawback with the data parallel JPEG decoder is that the VLD is inherently sequential, and it is not possible to know exactly where a block begins without decoding the previous ones. This implies that all cores must read the encoded image from external memory and perform the VLD, although the IDCT and colour conversion is skipped if the current MCU block does not correspond to its assigned part of the image. This is also seen in the results of the initial parallelisation, where the groups report speed-ups of roughly 1.7 and 2.3 times for 2 and 3 cores, respectively. The students thus get to experience the limited scalability of this approach, and explore solutions to mitigate the effect. Consider, for example, the partitioning strategy to the left in Figure 3, which only requires the first core to read 1/3 of the image from external memory, and the second core 2/3, while the last core must read all of it. This can be compared to the tiling strategy on the right in the figure, which requires all cores to read the entire memory, increasing memory contention. Several groups also develop synchronisation schemes that allow the cores to share information about the parts of the image that are already decoded.

5.2 Functional parallelism

In this solution, the decoding functions are mapped to the different cores, creating a pipeline where an MCU block is processed by all the cores in sequence before decoding is complete and it is written to the frame buffer. An important challenge is to determine how to partition the functions among the different cores to get a balanced load and to minimise inter-core communication. For this purpose, the students use the cycle-accurate simulation model, giving detailed profiling information. A common way to split the decoder is according to the three stages, VLD, IDCT, and CC that were explained in Section 2.1. This partitioning has the benefit of providing clear interfaces between the functions where only frequency blocks and pixel blocks are communicated between the cores.

A difficulty the students are faced with in the functional partitioning is that different pictures place very different computational requirements on the functions in the decoding algorithm. Figure 4 shows the decoding time required for the VLD, IDCT, and CC, respectively, on a single core. The two

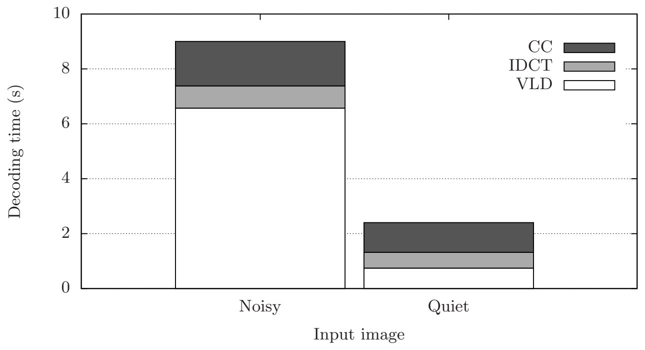


Figure 4: Single-core profiling.

images are both XGA resolution (1024 x 768), but *Noise* (748 kbytes) contains a lot of high frequency information and is dominated by the VLD, whereas *Quiet* (53 kbytes) contains mostly uni-coloured MCUs and is fairly balanced. This shows that the data-dependent behaviour makes it extremely difficult to partition the decoder in a way that creates a good balance between the cores for all pictures.

In addition to the difficulties in splitting the decoder into three equal parts, the students must also implement inter-core communication and synchronisation. The groups typically start by synchronising using simple flags. Eventually, most groups go for a double-buffered approach, or decide to use circular buffers with read and write pointers. An implementation of the C-HEAP protocol [16] is also provided with the hardware platform for reference. The students also evaluate the effects of placement of buffer data and administration, as well as the size of the inter-task buffers. Profiling is also carried out by looking at how often the various tasks stall on full or empty buffers. In implementing the inter-task communication, emphasis is put on hiding the latency of the memory subsystem, i.e. the NoC and memory arbiter and controller. The students thus have to employ concepts like posted writes and burst transfers.

6. ADDRESSING PERFORMANCE

The two times we have given the course, all the student groups have completed the project, meaning that they have at least two parallel JPEG decoders working by the end of the course. To get higher grades, we also require that the students assess the performance of their decoders. Thus, the evaluation is not only functional, but involves performance measures such as execution time, time-to-market and memory footprint.

Once a solution is functionally correct, an iterative optimisation and benchmarking phase begins to improve its quality. In Section 6.1 we elaborate on the benchmarking procedure. We then discuss optimisations for a decoder executing on a single core in Section 6.2 and on multiple cores in Section 6.3.

6.1 Benchmarks

The teams are encouraged to continuously evaluate their designs through quantitative benchmarks throughout the development process. This allows them to directly see the impact of design decisions on quality, and learn about the trade-offs involved. The benchmarking procedure is standardised by a committee, comprised of representatives from all design teams. This ensures that all teams are familiar with the procedure, and that their results are comparable. The standardised benchmarks consider two aspects of embedded systems being performance, in this case decoding time, and memory requirements.

Decoding time is measured by starting a timer on the host after uploading the encoded JPEG image to the external memory. After starting the timer, the host starts the three cores and waits until all of them have completed. A benefit of this benchmarking method is that it is easy to implement, although a drawback of the approach is that the time required for the host to start the cores and to detect that they finished execution is captured by the measurement. Since the host processor is connected via USB, this overhead may add up to a second to the decoding time. For future instances of the course, we are planning to use on on-chip cycle counter for time measurement. Benchmarking the memory requirements of a solution is simple, as the required amount of instruction and data memory is output by the tooling for every core.

We have observed that the students use greatly varying techniques to improve their benchmark results. Some groups simply adopt a trial-and-error approach. Other groups have given example of systematically identifying and addressing bottlenecks. We also see that most students are aware of optimisations for general processors, but are not familiar with the opportunities that present themselves in a multiprocessor system.

6.2 Optimisations for a single-core decoder

The optimisation process is guided by profiling the code using the cycle accurate simulator. Profiling helps identifying functions that are called often or require a lot of time to execute, indicating that they may be good candidates for optimisation.

Optimisations targeting the single core decoder can be categorised as: 1) algorithmic short cuts, 2) adaptations to fit with the computational cores, and 3) adaptations to fit better with the communication infrastructure. The first category involves using knowledge about the JPEG decoding algorithm to speed up decoding, such as throwing away higher frequency components, or exploiting common cases in the image format. The second category concerns making the computation more efficient by adapting it to the processor core architecture to get a more efficient instruction schedule. The third category considers rewriting the code to reduce the number of memory accesses. For the first category, the programmer must have deep insight into the JPEG algorithm. The latter two categories require the programmer to be intimately familiar with the target architecture and tooling.

The most influential algorithmic short cut in JPEG decoding is that of IDCT-bypassing. That is, when an MCU is unicoloured and does not contain any frequency components, the IDCT can be skipped. The short cut does not compromise the result, and in the case of the *Quiet* benchmark image, more than half of the MCUs are skipped. Another important optimisation is that of detecting common colour encodings in the CC. Most JPEGs use only two types of encoding (4:2:2 and 1:1:1), and by implementing special CC functions for these common cases, the indexing in the CC is greatly simplified.

There are many opportunities to improve the JPEG decoding time by exploiting knowledge of the processor core architecture. One of the major adaptations, done by many groups, is to replace the given Loeffler IDCT with Chen-Wang IDCT. The latter uses fewer multiplications and is better matched to the VLIW in question. By further adapting the code to use variables rather than arrays, the load on the register banks increases, but extra transfers to memory are avoided, resulting in a net gain. Another technique that we have seen examples of in the CC is to use look-up tables with precomputed values.

The last category of optimisations targets the memory architecture, aiming to reduce the number of accesses to remote memories, and to use the accesses more efficiently. A significant speed-up is achieved by using local memory rather than the shared external memory (or the memory of another core). The size is, however, very limited, and not all data will fit in the local memories. To use the remote memory accesses more efficiently, the code must be adapted to read/write whole words rather than sub-words, such as characters. The latter optimisation, for example, reduces the time required for the VLD by almost two times.

6.3 Optimisations for parallel decoders

The optimisations used for the single-core decoder are also applicable to the parallel decoders, but it is quickly noted by the students that the speed-ups observed for the single-core solution are not reflected when they are applied to code that runs on multiple cores. This demonstrates the influence that communication and synchronisation has on the decoding time.

There are refinements of data parallel implementations, addressing the memory contention. One such refinement involves ensuring that only one core performs the VLD on a particular line and shares the important results with the other cores through a structure in memory, allowing them to skip the line. This optimisation reduces the decoding time for both the aforementioned images by approximately 15%. A drawback of this refinement is that additional memory (approximately 2 kbyte) is required to store the information shared by the cores. The optimisations for the functionally pipelined version mostly considers moving smaller blocks of code between the cores to improve the load balance, or reducing the amount of data that is communicated between the cores. We have also seen numerous examples of different inter-task communication methodologies, aiming to reduce the cost of data transfers.

7. DISCUSSION

Similar to [13], we focus on the embedded software aspects from a systems perspective. The hardware is given. Still, the platform offers a great amount of mapping decisions, with distributed memory, multiple processors, and several NoC use-cases to choose from. More freedom could of course be given to the students, but we do not deem it feasible to do so with the current 5 ECTS credits of the course. Moreover, we try to minimise the problems involving understanding and complying with idiosyncratic interfaces and I/O devices.

The students appreciate the problem-based nature and report that they specifically learn about the concepts behind JPEG compression, different parallelisations, and the methodology and importance of exploring different architectural mappings. During the course, the student groups evaluate roughly four or five different partitionings, and get to experience the non-linear speed-up. In their presentation they often conclude with the observation that adding more hardware is not the solution. More important than one more core is to optimise the use of the ones that are there.

As in [13], we observe that students retain knowledge better when working through actual implementations that force them to confront the very real limitations and quirks of embedded systems. That said, we feel it is important to reduce the amount of practical knowledge and tool fighting as much as possible. From the first to the second year we also simplified a lot of the lower-level issues, and now help the students to get by the first hurdles in using the system. The amount of tutorial exercises and demonstrative examples is also significantly larger.

We believe that the Embedded System Laboratory delivers a level of realism that helps in both motivating the students, and reinforcing the experiences gained during the course. Based on student evaluations, the course succeeds in bringing together knowledge from other classes and teaches the students skills that they do not learn from a text book [4], e.g. the balance between top-down and bottom-up design, the ability to seek and find the information you need, the ability to debug and reason about observed behaviour, and to understand the different factors affecting the performance of a multi-processor system.

We continue to review and extend the course. With the maturity of the existing tooling and IP we have the opportunity to add more elements to the course and offer even more freedom, e.g. customisation of the processors and NoC instance. This will, however, be optional, as we already find the course sufficiently challenging.

8. CONCLUSIONS

In the Embedded Systems Laboratory, the students get to familiarise with many of the difficulties involved in programming multi-processor embedded systems. By the end of the course, they have successfully ported a JPEG decoder to the target multi-processor platform and evaluated a range of parallelisations on an actual FPGA instance. The assignment presents many challenges, ranging from working in a group to choosing the right compiler directives for a critical piece of an algorithm.

Embedded Systems Laboratory ran for the second time in 2008 with 31 participating master students, both from the Electrical Engineering and International Masters programme on Embedded Systems. The course concepts have furthermore been adopted by the Delft Technical University, where a similar course was given by the Computer Engineering department for the first time this year. Next year, we aim to keep improving the course in line with feedback from students and teaching assistants, and prepare another class of students for the problems we are facing with the wide-spread adoption of multi-processor embedded systems.

9. REFERENCES

[1] Agility DS. http://www.agilityds.com, 2008.

[2] P. Bertels *et al.* Gathering skills for embedded systems design. In *Proc. WESE*, 2007.

[3] S. Dutta *et al.* Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Design and Test of Computers*, 2001.

[4] S. Edwards. Experiences teaching an FPGA-based embedded systems class. *ACM SIGBED Review*, 2(4), 2005.

[5] Embedded systems laboratory. http://www.es.ele.tue.nl/education/EmbeddedSystems.html, 2008.

[6] K. Goossens *et al.* The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. and Test of Comp.*, 2005.

[7] P. Guerrier. *Un Réseau D'Interconnexion pour Systémes Intégrés*. PhD thesis, Université Paris VI, 2000.

[8] A. Hansson and K. Goossens. Trade-offs in the configuration of a network on chip for multiple use-cases. In *Proc. NOCS*, 2007.

[9] A. Hansson *et al.* Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. DATE*, 2007.

[10] J. Helmig. Developing core software technologies for TI's OMAP platform. *Texas Instruments*, 2002.

[11] J. Kahle *et al.* Introduction to the Cell multiprocessor. *IBM Jour. of Research and Develop.*, 49(4/5), 2005.

[12] K. Keutzer *et al.* System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(12), 2000.

[13] P. Koopman *et al.* Undergraduate embedded system education at carnegie mellon. *ACM TECS*, 4(3), 2005.

[14] J. Leijten *et al.* Prophid: a platform-based design method. *Des. Autom. for Emb. Syst.*, 6(1), 2000.

[15] J. Muppala. Bringing embedded software closer to computer science students. *ACM SIGBED Review*, 4(1), 2007.

[16] A. Nieuwland *et al.* C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems. *Des. Autom. for Emb. Syst.*, 7(3), 2002.

[17] C. Rowen and S. Leibson. *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors*. Prentice Hall PTR, 2004.

[18] A. Sangiovanni-Vincentelli and A. Pinto. Embedded system education: a new paradigm for engineering schools? *ACM SIGBED Review*, 2(4), 2005.

[19] C. Shih *et al.* Toward HW/SW integration: A networked embedded system course in taiwan. *ACM SIGBED Review*, 4(1), 2007.

[20] Silicon Hive. http://www.siliconhive.com, 2008.

[21] Xilinx, Inc. http://www.xilinx.com, 2008.

Use of Discrete and Soft Processors in Introductory Microprocessors and Embedded Systems Curriculum

Sin Ming Loo
Electrical and Computer Engineering Department
Boise State University
Boise, Idaho 83725
208-426-5679
smloo@boisestate.edu

C. Arlen Planting
Electrical and Computer Engineering Department
Boise State University
Boise, Idaho 83725
208-426-4826
clarenceplanting@boisestate.edu

ABSTRACT

This paper describes a sequence of two courses, starting with the teaching of introductory microprocessor concepts and extending to advanced embedded system programming. The introductory microprocessor course is taught using a soft processor with a field-programmable gate array as the development platform, a combination which allows the course to undergo continual improvement without being limited by fixed hardware. The second course builds on the foundation of the first course, with an emphasis on working with advanced devices, building complete embedded systems, and developing embedded programming skills with different targets. This paper describes the experiences gained from the first course, and the plan for the second course.

Categories and Subject Descriptors

K.3 [Computers and Education]: Computers and Education - General

General Terms

Experimentation

Keywords

Microprocessors, Soft Processor, Field-Programmable Gate Array, Curriculum

1. INTRODUCTION

Most microprocessors courses have traditionally been taught using a discrete microprocessor, such as Motorola 6800, Intel x86, ARM, or IBM PowerPC series [1]. The x86 platform has historically been the one utilized in the microprocessors course at Boise State University (BSU). The introductory microprocessors course at BSU taught only assembly language programming with little emphasis on other language skills.

The advent of field programmable gate arrays (FPGAs) and access to more powerful embedded processors has made it possible for students to tackle much larger projects than in the past. Increasingly sophisticated projects involving robotics, digital radio communications, MP3 players, video interfacing, and various sensors are more meaningful and exciting to the students, but also require a higher level of proficiency in programming. In our experience most electrical engineering students have learned to design hardware well, but lack the software skills to adequately demonstrate the functionality of that hardware. These skills would not only benefit students in advanced digital courses, but would also increase the students' future value in the workplace.

To address these issues, there has been an ongoing effort at BSU since 2005 to update the computer engineering courses. An integral part of every stage in updating BSU's computer engineering courses involves the use of FPGAs in place of traditional development boards, taking advantage of the fact that the functionality of an FPGA can be changed without requiring physical changes to the board itself.

The endeavor started in the sophomore Digital Systems (EE230) course, with the major change being the introduction of a low-cost FPGA in place of the prototyping board with discrete components. The updated course has been very well received, and provides students an early exposure to reconfigurable hardware concepts. This sets the stage for the introduction of a soft-core processor in the microprocessors course.

Our next updating effort was the junior microprocessors course, with major updates including introduction of the C programming language, stressing the use of structures, unions, and pointers; a soft core microprocessor, and a sizable FPGA as the development target. The updated microprocessors course has also been very well received. Students indicated the heavy workload was worthwhile. In our electrical engineering curriculum, this is the last course where students will interact with processors unless they select a system-level design project involving processors in their capstone senior design sequence.

Students have inquired which course they should take if they want to study advanced topics in microprocessors systems or embedded system design. Our curriculum has elective computer engineering courses in hardware description, system testing, and hardware/software co-design, but none as the natural follow-up to the new microprocessors course.

The remainder of this paper describes the experiences gained from teaching the first course (microprocessors) in the two-course sequence, and the plan for the second course (embedded and portable computing). Section 2 outlines BSU's electrical engineering curriculum and pre-update computer engineering courses. Sections 3 and 4 discuss the goals and resource selection for updating the course sequence. Section 5 provides details of the re-designed microprocessors course, and Section 6 presents our plan for the subsequent course. A summary and conclusions are presented in Sections 7 and 8.

2. EXISTING SITUATION

Boise State University has an ABET-accredited electrical engineering program with computer engineering as an option. One of the core courses offered at Boise State University for students specializing in computer engineering is the Microprocessors course. The students take Microprocessors after they have taken Introduction to Computer Science (basic software skills and object oriented programming with Java) and Digital Systems (sophomore digital logic course).

The Microprocessors course at Boise State University covers microprocessor architecture, software development tools, and low-level hardware interfacing with emphasis on 16-bit and 32-bit microprocessor systems. Machine and assembly language programming, instruction set, addressing modes, programming techniques, memory systems, I/O interfacing, and handling of interrupts are among the topics studied with practical applications in data acquisition, control, and interfacing. This course was reported to be a favorite of many students, largely because of the interesting devices (such as the magnetic card reader) that could be played with by the end of the course. The intent was to retain and potentially enhance this characteristic of the course with the changes implemented.

Since the microprocessors course (lecture and lab) is a requisite for both electrical and computer engineering (ECE) and computer science students, the course must endeavor to address the disparate interests and needs of students in both disciplines. In addition to those specializing in computer engineering, the ECE group includes students interested primarily in other areas such as integrated circuits, communication and signal processing, control systems, power and energy systems, etc. Most computer science students are more interested in hardware with an operating system. Therefore it is important to attempt to achieve a balance in the course that will adequately teach electrical engineering and computer science students the needed fundamentals of microprocessors, while also providing the computer engineering students a solid foundation for advanced courses.

A course titled "Embedded and Portable Computing Systems", specifically addressing embedded design with the PIC microcontroller, has been offered at BSU. This was a primarily hardware-oriented senior/graduate level course utilizing assembly language only. Students taking this class received no C programming instruction.

3. GOALS

The goals of the updated two-course sequence were to more effectively teach the basics of microprocessor programming using updated technology, and to build on the foundation gained in the Microprocessors course to expand what is covered in the Embedded Systems course.

It was decided that the updated Microprocessors course would involve:

- A RISC microprocessor (MIPS-like)
- Simple memory-mapped devices (LEDs, switches, buttons)
- Initial use of assembly language to understand processors
- Transitioning the knowledge of microprocessors to the C language [2]
- Coverage of topics such as polling, time management (delays vs. timer), interrupts and interrupt service routines (ISRs)
- Advanced devices, such as pulse width modulated (PWM) DC motors and A/D conversion

The new follow-on Embedded Systems course would include:

- Advanced time management issues and usage of state machine construct in order to manage time
- Introduction of microcontrollers (specifics of memory and device and how they relate to programming)
- How coding can affect the ease of transferring code to other platforms (retargetability)
- Advanced devices (I^2C, SPI, USB, UART) from hardware and software perspectives

The selection of resources to accomplish these goals is discussed in the following section.

4. SELECTION OF RESOURCES

4.1 Development Board and Tools

The two-course sequence was updated to utilize a soft processor instantiated on an FPGA. A board that had previously been used in graduate level courses at BSU - the Altera DE2 (shown in Figure 1) with Nios II processor - had most of the desired features, including:

- Classic RISC architecture closely approximating MIPS
- Variety of memory types
- Numerous attached devices plus two expansion headers for future add-ons (device support for USB, audio, VGA, Ethernet, UART, PS2, secure digital, and expansion headers)
- Free software integrating industry-standard development and debugging tools (e.g. GNU, Eclipse IDE, GCC compiler and GDB debugger) that students are likely to encounter in their careers
- Instruction set simulator (allowing work to be done at home) provided free with software tools from Altera

Best of all, the board was reconfigurable, allowing a different configuration for each lab and final project. A soft processor is very expandable with new interfacing hardware written in hardware description language. This feature allows the instructor to quickly create different configurations in order to easily meet different needs of various projects, and various courses. In addition, it gives an opportunity to demonstrate hardware/software co-design concepts in subsequent courses.

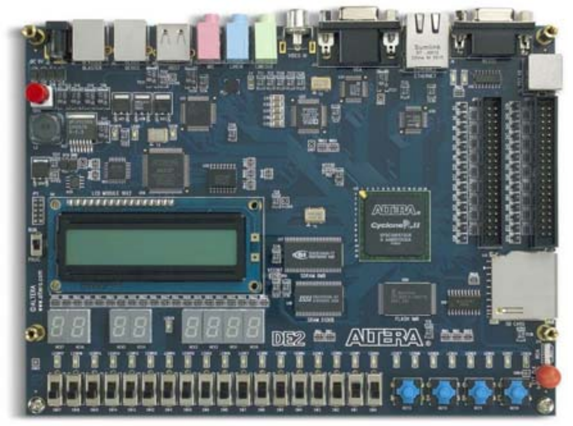


Figure 1. Altera DE2 Development and Education Board [3]

Since a soft processor is a microprocessor core that can be wholly implemented using logic synthesis, this provides the capability to expose the students to numerous different hardware configurations in a single course. Dedicated boards require that a connection resource be permanently allocated for specific purposes, thus limiting usage of the board. Conversely, a soft processor is analogous to a theater where a new stage set can be brought in for each new production. Each device/concept can be introduced with a unique processor configuration. For example, the concept of cache memory can be illustrated by using different configurations to generate a processor with cache and one without cache and comparing instruction performance and results.

As new devices become available or new instructional materials are developed, they can more easily be integrated into the course curriculum with a soft processor. Unlike a discrete processor, all of the work does not need to be done weeks ahead of time and developed on a dedicated board for continued use months/years into the future. Minor changes to labs can be made each year without requiring major redesign of dedicated boards. The configuration of the soft processor can grow or shrink as needs dictate. Simple configurations can be used at the beginning so students can more easily grasp the big picture; more complex configurations can be generated as their understanding increases. If a project requires multiple UARTs, it is easy to add them.

It is conceded that FPGAs are not without disadvantages, including:

- Cost ($495 list, $269 student price; BSU received special academic discount to reduce price to $150)
- Extra instruction to understand reconfigurable aspects (higher abstraction level, need to instantiate every time)
- Creation of processor configurations (can be done by instructor for lower level courses)

However, the capability to change the functionality an unlimited number of times to fit the desired application provides a distinct instructional advantage.

Though the soft processor was considered the best instructional platform, it is acknowledged that it still needs occasional comparison (via demonstration) to traditional discrete microprocessors.

4.2 C Programming Language

Since the C language is the choice for implementation of today's dominant operating systems [4], including Windows and Linux [5], it seemed prudent to follow suit with this precedent. Though C++ has more features than C, these features are not particularly useful in teaching microprocessors and do not offset the fact that it is more difficult to learn than C. In addition, many small microcontrollers (such as the Microchip PIC) are not supported by C++ compilers.

5. MICROPROCESSORS COURSE WITH SOFT PROCESSOR

This course needed revamping to become more representative of what practicing computer engineers deal with on a daily basis, including embedded systems without operating systems. The ultimate objectives are to update the course using a modern development environment with modern debugging capabilities, to teach the basics of microprocessor programming (assembly plus C), and to have the students practice these skills with realistic laboratory assignments and projects.

The Microprocessors course is a one-semester course consisting of two separate parts, the lecture (ECE 332) and the lab (ECE 332L). Though ECE 332 and ECE 332L are co-requisites, each part is individually graded and assigned credits (3 credits for the lecture, and 1 credit for the lab). Lecture classes are taught twice a week in two 1 hour-15 minute sessions, and the lab portion consists of a 3-hour session once a week. The rest of this section contains details of our course update and implementation, including educational methodology, course details, and outcomes.

5.1 Educational Methodology

The primary changes in the updated Microprocessors course involved selection of a MIPS-like processor implementation (RISC architecture vs. the CISC platform previously used), teaching the C programming language in addition to assembly, and utilizing a modern development platform and tools.

Although CISC x86 is the most prevalent microprocessor architecture, this doesn't necessarily equate to the best educational platform for teaching basic microprocessor concepts. The instructions for CISC platforms are variable in length, which complicates learning compared to the fixed length instructions for RISC platforms. As many of the instructions available with the CISC are not applicable to the basic microprocessors course, the significantly smaller set of instructions provided for a RISC platform was considered more appropriate. In addition, the students will also use RISC when they take the Computer Architecture course.

Introduction of the C programming language to the Microprocessors course was proposed because of its ability to enhance productivity and portability with minimal overhead. The course would introduce just enough material from the C programming language that students could work with devices at a low level. This would minimize the overlap for the computer science students, and also give some ECE students their first exposure to the C programming language.

Before updating the Microprocessors course, an experimental course addressing the usage of the C programming language for embedded applications was undertaken (taught in Spring 2007) to investigate methods of incorporating the C language in the electrical engineering curriculum. The experimental course included an accelerated presentation of the C language directed to specific course objectives. When it became apparent that some of the students were struggling with the accelerated approach (basics of the C language in four weeks), the C language instruction was extended for two more weeks. As it turned out, this protracted coverage didn't provide the desired benefits and the better students began to lose interest. However, the exposure to key concepts of the C language did prove to be of value for all the students.

5.2 Resources Utilized

5.2.1 Course References

Textbooks are available for most traditional microprocessor platforms. However, since the concept of teaching soft processors is still evolving, we found no single source text that addressed teaching with the Nios processor. Though there was no unifying document that consolidated the information needed for the updated Microprocessors course, usage of the soft processor was considered valuable enough that this did not change our decision. We accepted the challenge of generating our own instructional materials for the processor-specific (assembly and C) portion of the course.

Numerous references, including textbooks, vendor-supplied handbooks and tutorials, and data sheets were utilized in the course. Reference manuals and tutorials supplied by Altera for the DE2, the Nios II processor, Nios II software developer and Altera Debug Client were used. The textbook "Embedded System Design - A Unified Hardware/Software Introduction" [6] was selected as the text for the lecture portion of the course. "The C Programming Language" (K&R) [7] was the primary reference for the C language portion.

5.2.2 Development Platform and Tools

A microprocessor was considered a more generalized platform better suited for teaching microprocessor basics than microcontrollers, which have a wider variety of implementations (Microcontrollers will be addressed in the Embedded Systems course.). In addition, the Nios II processor has an architecture which, when coupled with memory-mapped I/O, simplifies understanding of the system's address space. The Harvard architecture typically found on microcontrollers requires special C qualifiers to identify data residing in different memory spaces.

The tools provided by Altera include tools to produce hardware configurations, and tools to develop software solutions for soft processors. The tools involving the generation of soft processor configurations were used by the instructors, but were not presented to the students.

For software development, Altera provides an educational development tool (Altera Debug Client) in addition to commercial grade tools (Nios II IDE). Altera Debug Client provides assembly of assembly language programs with no run time support, and compilation of C programs with minimalist run time support (no interrupt service routine or exception handling). The Altera Debug Client also loads resulting code into the DE2 and establishes a debug session with capabilities to see disassembled code, view and modify registers, view and modify memory, and perform other debug functions (e.g. breakpoints). This is ideal for educational purposes.

The Nios II IDE automatically generates software libraries (Hardware Abstraction Language, or HAL) to support most of the devices generated with a particular hardware configuration. While very convenient for advanced users, this hinders the learning process for beginners. For this reason we decided to start with Altera Debug Client for the first part of the course before introducing the Nios II IDE. Using Debug Client eliminates the startup code provided by the C runtime and the exception handling in the Nios II IDE, and more importantly, assures that students have to provide the functionality themselves. This provides a better learning environment for assembly language.

The Nios II IDE development tool was used in the course for development in the C language. However, since one of the learning objectives for the Microprocessors course is interfacing to low level devices and handling of interrupts, Altera's implementation of HAL was disabled for exception processing (interrupt service routines, or ISRs). (Altera's Nios II IDE is based on the popular Eclipse IDE framework with Altera-supplied plug-ins that manage the Nios II projects and the make process. If alternate plug-ins could provide minimalist support then the use of Altera Debug Client could possibly be avoided altogether, and students wouldn't need to learn two development environments.)

5.3 Course Details

5.3.1 Course Approach

A goal-oriented approach was used to present key foundational concepts in both languages, in order to produce a greater level of proficiency more quickly than could be achieved with an exhaustive coverage of either language. With both the assembly and C languages, basic configurations were introduced at the beginning so that students learned to write code as early as possible (in the first lab assignment). Simple example programs were provided in tutorials to promote the learning process. A course outline is presented in Table 1.

Assembly language programming concepts were presented with a mixture of devices to help keep interest in the labs. Introduction of devices began with memory, followed by parallel I/O (PIO) such as LEDs, switches, buttons, and the seven segment display. This took about 4 weeks, and then the focus of the course moved to the C programming language. Based on our belief that it isn't necessary to teach the entire C language to significantly enhance software skills beyond those achieved with assembly language alone, a subset of the C language was introduced after the assembly language portion of the course. Much less time was spent presenting the basic concepts of C language programming than had been spent in the experimental embedded systems programming course.

Table 1: Updated Microprocessors Course Outline

Week	Lecture Topics
1-3	• Nios II Processor System Architecture and Programming • Memory, Registers, Program counter • Assembly Instructions, Memory organization, Addressing modes

	• Assembler Directives, Instruction Set Reference, Instruction encoding/decoding
4-6	• The C Programming Language: K&R Chapters1-4 • C Program structure, Pointers, Structures, Unions, Bit structures • C access to devices • Cache bypass in C • Inline assembly
6-7	• Exceptions
7-8	• Hardware abstraction layer (HAL)
8-9	• Devices: Timers, counter, watchdog timers • UART, PWM
10-15	• Keypad, Keyboard, Analog to digital, Real time clock, LCD controller, Memory controllers • Performance measurement, ISR performance, Simple bus, Communication protocols

The labs developed for the updated Microprocessors Lab course are summarized in Table 2. The first three labs provided the students hands-on experience with microprocessors utilizing the assembly language as covered in the first five weeks of the lecture series. The remaining labs dealt primarily with the C language, with the exception of a portion of the ISR lab.

Table 2: Updated Microprocessors Lab Outline

Week	Topics/Assignment
1	Familiarization with DE2, Nios II, and Debug Client (simplified tutorial)
2	Introduction to memory (assignment: bubble sort routine)
3	Exploration of address space beyond memory, concept of memory mapped I/O. PIO devices: Interfacing to LEDs and switches
4	More advanced PIO (seven-segment). Integration of concepts from previous labs to implement display system using switches, LEDs, buttons and seven segment. Organization of code into modules and directories.
5	C language tutorial. Redo seven segment display in C, continuing use of Debug Client
6	LCD interface routines in C language, continuing use of Debug Client
7	Exceptions: return to assembly language to explore issues of exception processing
8	HAL, introduction to Nios II IDE and related HAL facilities
9	HAL interrupts (abstraction of interrupts provided by Nios-supplied HAL routines)
10	*Spring break*
11	PWM and DC motor and H-bridge: use of PIO core to control direction and speed of DC motor with PWM
12-16	Final Project

5.3.2 Synergy of teaching assembly and C together

Exposure to assembly is required for the Computer Architecture course. However, it was our belief that the addition of the C programming language would provide additional benefits. Both assembly and C can be presented in the same course when taught in the proper balance using a goal-oriented approach. The assembly language was taught first in the course to provide a foundational understanding of processors and platforms that would accelerate the process of teaching C. Assembly language is the best way to understand and learn the foundations of microprocessors, since it is the primary interface to the processor. The C language was added to provide a higher level view of the same processor concepts, further reinforcing the knowledge provided by learning assembly.

The goal-oriented approach utilized involved teaching a directed subset of C from a hardware perspective. The versatility of the C language allows it to be taught at various abstraction levels, beginning as a relatively low-level language and advancing to higher-level concepts as the students gain in understanding. C programming was taught from a hardware-centric perspective using practical examples. Object oriented programming principles were included by example. Topics usually considered as advanced techniques and traditionally presented at the end of a C language course– such as pointers, structures, unions and bit structures – were presented early in the course.

Bit manipulation is one concept that can benefit from the introduction of the C language. The manipulation of bits is generally the realm of hardware devices. The process of bit twiddling using techniques such as bit shifting and masking has traditionally been done in assembly language, and moving that code to C does not yield any benefits. However, with the combined usage of bit structures and unions, this process is reduced to fairly straightforward code. Thus, the introduction of the constructs of pointers, structures and unions can reduce the tedium of dealing with the signals of connected hardware devices. Since bit structures can be platform-dependent, their usage is best restricted to lower platform-dependent layers.

Teaching C *in addition to* assembly provides advantages that would not be provided by simply replacing assembly language with C. In either language, working at the device level requires becoming familiar with the processor and the address space. The concept of pointers must also be learned in either case (pointers in assembly languages may not be recognized as such in the same context as C). Pointers are the most difficult concept to learn in C. Teaching the concepts of pointers in assembly first, observing the instructions involved, and then translating that knowledge to implementation in C made it easier to understand the concept of pointers in C. Once pointers have been learned in assembly, the only differences that need to be learned in C are syntactic. Pointers are the primary reason that C can replace assembly language for device level code.

Other synergies between the assembly and C languages were observed in relation to understanding registers, processor architecture, and processor address space. In the C language, the introduction of the *register* keyword is difficult to understand relative to what usage it could have. After using assembly, it is easier to understand how it can effectively be used. Doing low (device) level microprocessor development in C is difficult to do without a good understanding of the processor architecture and the processors address space. This includes the program, data, stack and devices. In this view it can be argued that understanding the assembler for a processor before trying to do work with C is a definite advantage. This is why we have chosen to overlap the instruction of both the assembly and C languages.

5.4 Course Outcome

Final lab projects were undertaken to consolidate and demonstrate the knowledge gained in the lecture and lab portions of the course. For the final project, there were nine teams ranging from two to six students. The students chose their own teams, which were allowed to propose their own projects and proceed upon approval by the instructor. Teams that did not create their own project were assigned one by the instructor. All final projects were developed in the C language.

Each project was provided with a Nios II processor configuration (*sof* and *ptf* files). Since each project had different requirements and needs, the use of a soft processor allowed the instructor to create different configurations for each team. The teams were required to perform a demonstration of the product for the instructor, and produce a final project report describing their project.

The final projects successfully demonstrated the students' grasp of the knowledge presented in the course. A wide range of devices were utilized in the final projects:

- DE2 version of Space Invaders -- on-board hardware utilized in the implementation included buttons (user input), VGA, LCDs (score display), timers and alarms (movement of aliens and weapons), and the JTAG UART (output for testing purposes),
- Client/Server with IRDA utilized two FPGAs, a keyboard, IR transceiver, and LCD display,
- Motor Speed Detector and Regulator, utilizing PWM and an infrared emitter and detector sensor,
- Audio to LED Display,
- LCD Scrolling Marquee,
- DE2 version of Pong Game,
- Ping Pong utilized VGA, sound and keyboard,
- Etch-A-Sketch, and
- Bomb Squad -- utilized two DE2s, keyboard, wireless modules, motors (remote vehicle), audio and LED display.

All projects were completed within the time frame provided with little help from the instructor. Based on student evaluations of the course, the course update was considered successful.

6. EMBEDDED SYSTEMS COURSE WITH SOFT AND DISCRETE PROCESSORS

6.1 Objectives

The planned second course in the two-course sequence will build upon the introductory (microprocessors) course with advanced topics. The goal is to provide students with equally strong software and hardware backgrounds, such that they can develop systems that run reliably and efficiently.

The first objective is to incorporate both soft and discrete processors in this course. One might ask, if the path of progression is heading toward the use of soft processors in FPGAs, wouldn't the use of a discrete processor be taking a step backwards? That's a reasonable question to ask and one that has no absolute right or wrong answer. Discrete processors typically offer high performance and often exclusive special capabilities that can't be totally replaced or matched by a soft processor counterpart. The other reason to include discrete processors is that it is beneficial for the students to be exposed to another processor in order to learn to write code on one platform that is appropriately layered to port easily to another platform.

The second objective of this course is to teach embedded systems programming considerations, and object oriented programming with C. The course will include teaching layered, modular programming concepts and selected object oriented programming principles applicable to embedded systems [8, 9]. It will also provide exposure to abstraction interfaces of varying quality so the students will gain the finesse to recognize and create an effective hardware abstraction interface. The course will selectively implement object oriented programming principles applicable to small embedded systems.

The final aim is to bring the first two objectives together by providing opportunities to practice using real-world devices. Sensors with different communications interfaces will be brought in as programming assignments and to be used in projects. Sensors that are interfaced using current, voltage and serial protocols such as UART, SPI and I^2C will be part of the curriculum. Since a soft processor is part of the curriculum, hardware/software codesign concepts can also be introduced. If the schedule allows, a small project may be assigned to explore pure software implementation, pure hardware implementation, and an implementation taking advantage of synergism between hardware and software.

6.2 Implementation

The new Embedded Systems course will begin with a few of the more advanced concepts of the C programming language not specifically covered in the Microprocessors course, including object oriented programming in C, layering, UART, and cooperative multiprocessing. Foundational skills developed in the first half of the course will then be employed in the second half to work with I^2C, SPI and USB platforms.

Use of the Altera DE2 for prototyping purposes will continue in this course. Students will tackle similar projects implemented on different target platforms, followed by a review of issues found with each platform. Students will select the platform for their final project early in the course.

This course will target small embedded processors without operating systems, requiring students to develop the code for the services an operating system would normally provide. Concepts typically covered in an operating systems course that are applicable to embedded systems will also be addressed, including time management, solutions to address concurrency issues (race conditions [10]), and communications protocols. Since these issues would typically be handled by an operating system in the case of general purpose computers, different approaches are necessary for small embedded systems [11-16].

In object oriented languages such as Java, data and code are tightly coupled. Conversely, in a structured or procedural language such as C, data and code are uncoupled. To use C in object oriented programming necessitates that data and code be loosely associated (by means of establishing coding and naming conventions) to approximate an object oriented language. Making this transition requires an understanding of the concepts of data abstraction and encapsulation, and C.

Advanced time management issues and usage of state machine construct in order to manage time will be addressed. This approach allows for multiple threads of execution to be accomplished in a cooperative manner. (To allow for accurate measurement of time, this will be used in conjunction with an external crystal and an interrupt service routine.)

A case study of UART will also be presented in the course, incorporating RX/TX, interrupt handling, operating system concepts (issues of concurrency with ISRs), data structures (circular buffers), network protocols in SLIP, and low level device access. Additionally, the solution to the problems encountered in the case study can be structured utilizing layering techniques which go hand-in-hand with encapsulation methods.

The course will be project-oriented, with all projects developed on the Altera DE2 development board and retargeted to other platforms. Code for small embedded systems written on one platform with the intent of porting to another is generally more appropriately layered, which results in well-written code that is inherently retargetable.

The initial project will be development of an ultra-light menu system for embedded applications. This project is intended to reacquaint the students with the Altera DE2 and tools used in the Microprocessors course, and test their understanding and skills using pointers, structures and unions. This should be a very small efficient menu, intended not for dealing with everyday processes but for infrequent updating of configuration items or as a debugging tool. The routine will be a passive component, ready to be used but with no continuous impact on system performance.

The second project will demonstrate a stepwise refinement approach to designing a large project by starting with smaller pieces that are designed and tested separately before being integrated into a final solution. The intent is to illustrate that many pieces of code can coexist and run cooperatively as a whole without needing an operating system to manage the pieces. Components will include those from the Microprocessors lab projects (the majority implemented with minimal amounts of code), the ultra-light menu developed in the first project, and an Altera alarm abstraction that will be provided.

The third project will involve retargeting the comprehensive project developed above to various other platforms. The remaining projects will include a real-time clock (I^2C), other I^2C devices, and SPI devices. The final project will involve a platform containing a USB device.

7. SUMMARY

To date, the updated Microprocessors course has been taught twice and the course has been refined based on experiences in the first semester. For example, the coverage of C programming language concepts has been reduced and the order of presentation revised to facilitate the transition from assembly language to C (pointers, structures, unions and bit structures were taught at the beginning of the C language portion of the course). Tutorials for vendor products were modified to reduce the volume of material beyond the scope and objectives of this course that students were required to sort through. The use of homework and quizzes was increased to reinforce understanding and increase student accountability for learning.

Some material was moved out of the lab portion and into lecture handouts, which adjusted workload to better match the credits for the lecture/lab portions of the course. Lecture materials were classified to emphasize practical (lab-oriented) materials versus text (abstract) materials, allowing introduction of some concepts in a different order than in the text in order to provide the necessary background for the labs to proceed.

The labs were also redesigned to simplify future modification and maximize reusability. The concept is to divide the labs into two parts: one part that is instructional in nature, the other that represents the creative endeavor required of the students. The instructional portion can remain relatively unchanged from semester to semester, requiring updating only to accommodate changes in the hardware and tools used. The creative portion can be changed every semester to insure that students are exposed to new projects.

The upcoming version of the Microprocessors course will also include a refresher section on number system concepts and an early test of programming skills.

With the updated Microprocessors course as a foundation, less time will need to be spent on the C programming language in the embedded systems course. This will leave more time in the new Embedded Systems course for specifically tackling embedded programming for devices and protocols such as the UART, I^2C, SPI.

8. CONCLUSION

A soft processor instantiated on an FPGA with classic RISC architecture was used to provide a modern development environment in the updated Microprocessors course at Boise State University. Industry-standard development and debugging tools (Eclipse IDE, GCC compiler and GDB debugger) that the students are likely to encounter in their careers were also incorporated in the course. A combination of assembly and C language was used to teach the basics of microprocessor programming, and the students learned to practice these skills with realistic laboratory assignments and projects. The planned Embedded Systems course will provide the natural follow-up to the updated microprocessors course.

The update process for the computer engineering courses at Boise State University has been fruitful for students and instructors. Students get to learn modern design techniques with up-to-date tools, beginning with the introductory Microprocessors course and continuing into the Embedded Systems course.

9. REFERENCES

[1] S. M. Loo, "On the Use of a Soft Processor Core in Computer Engineering Education," *Proceedings of 2006 ASEE Annual Conference*, Chicago, IL, June 18-21, 2006.

[2] G. Skelton, "Introducing Software Engineering to Computer Engineering Students," *Proceedings of the 2006 Southeast Conference,* 0-4244-0169-0/062006 IEEE.

[3] http://www.altera.com/education/univ/materials/boards/unv-de2-board.html

[4] G. J. Nutt, 2003. *Operating Systems,* 3rd ed. USA:Addison-Wesley.

[5] A. Silberschatz, P.B. Galvin, and G. Gagne, 2005. *Operating System Concepts,* 7th ed. Hoboken, NJ: John Wiley and Sons, Inc.

[6] F. Vahid and T. Givargis, 2002. *Embedded System Design – A Unified Hardware/Software Introduction,* Hoboken, NJ: John Wiley and Sons, Inc.

[7] B.W. Kernighan and D.M. Ritchie, 1988. *The C Programming Language,* 2nd ed. Upper Saddle River, NJ: Prentice Hall.

[8] M. Curreri, "Object-Oriented C: Creating Foundation Classes Part 1," Available: http://www.embedded.com, *Embedded Systems Design, 9/10/03*.

[9] C. Cantrell, "Embedded Object-Oriented Programming," *Circuit Cellar,* Issue 187, Feb. 2006, pp. 52-59.

[10] D.P. Reed and R.K. Kanodia, "Synchronization with Eventcounts and Sequencers," *Communications of the ACM,* vol. 22, no. 2, Feb. 1979.

[11] K.G. Ricks, W.A. Stapleton, and D.J. Jackson, "An Embedded Systems Course and Course Sequence," *Proceedings of 2005 Workshop on Computer Architecture Education,* Madison, WI June 5, 2005.

[12] D.J. Jackson and P. Caspi, "Embedded Systems Education: Future Directions, Initiatives, and Cooperation," *ACM SIGBED Review,* Volume 2, Issue 4, October 2005.

[13] F. Vahid, "Embedded System Design: UCR's Undergraduate Three-Course Sequence," *2003 IEEE International Conference on Microelectronic Systems Education,* Anaheim, CA, June 1-2, 2003.

[14] J. Conrad, "Introducing Students to the Concept of Embedded Systems," *International Conference on Engineering Education,* Gainesville, FL, October 16-21, 2004.

[15] T.S. Hall, J. Bruckner, and R.L. Halterman, "A Novel Approach to an Embedded Systems Curriculum," *36th ASEE/IEEE Frontiers in Education Conference*, San Diego, CA, October 28-31, 2006.

[16] A. Striegel and D.T. Rover, "Enhancing Student Learning in an Introductory Embedded Systems Laboratory," *32nd ASEE/IEEE Frontiers in Education Conference*, Boston, MA, November 6-9, 2002.

eBlocks – Embedded Systems Building Blocks to Enable Project-Based Learning

Anuradha Phalke
Department of Electrical and Computer Engineering
University Of Arizona
aphalke@email.arizona.edu

Susan Lysecky
Department of Electrical and Computer Engineering
University Of Arizona
slysecky@ece.arizona.edu

ABSTRACT

We present an interactive platform to enhance STEM education through project-based activities that complement the existing curriculum, while drawing on problem solving skills and team work. The addition of hands-on projects is intended not only to reinforce the various concepts learnt through normal class readings and discussion, but also make these topics multi-dimensional by enabling students to apply these principles in their everyday surroundings. The types of courses that can take advantage of the eBlock platform range from middle school math and science classes to university level engineering courses. In this paper we present the eBlocks platform which can be utilized to implement a wide variety of projects without requiring users to have programming or electronics experience. Specifically, we summarize the preliminarily usability results and discuss possible applications of the eBlock platform.

Categories and Subject Descriptors

H5.2 [**User Interfaces**]: User-centered design; Evaluation/methodology, Interaction styles; Prototyping; H5.m [**Miscellaneous**]

General terms

Design, Human Factors

Keywords

eBlocks, user interfaces, STEM education, interactive embedded systems building blocks

1. INTRODUCTION

The math and science skills of K-12 students within the United States are continuously evaluated [1][11][12], with concern that student rankings are dropping with respect to their international counterparts. Beyond secondary schools we see engineering enrollment and retention at the college and university level is adversely effected [22], fewer students are pursuing science and engineering degrees, with only about 6% of undergraduates majoring in engineering [4]. Even further reaching, is the continued debate as to whether there is an "engineering gap" in the United States in that many engineers are retiring but there are not enough engineers to fill these spaces [13][33][35]. The U.S. Bureau of Labor Statistics projects 15 of the 20 fastest growing occupations through 2010 will require significant math and science backgrounds [31]. Given these statistics, we look for possible methods to enhance STEM (Science, Technology, Engineering and Mathematics) performance, as well as increase interest in pursuing STEM careers.

The benefits of project based learning have been studied for decades [2][14][15][17][25][28], where students to participate in the learning process rather than passively listening to a lecture, reading a book, or watching a video. Studies have shown that students who are able to investigate the world around them and topics that are relevant to their everyday surrounds not only achieve higher scores than students in traditional classrooms [9][30] but are also more engaged [3][21]. While these learning models have been shown to be effective, traditional STEM courses struggle to provide an active learning environment. Incorporating advanced learning technologies into the curriculum are often times not feasible because test equipment is too expensive and school budgets are already spread thin, while building a custom solution requires a large breadth of programming and engineering expertise. Thus, we are faced with a challenge to develop a platform that is usable by a wide range of teachers and students, cost-effective for schools, and support a variety of STEM learning goals.

We propose to utilize and extend the eBlocks platform [5], a platform composed of numerous fixed-function building blocks, initially developed to enable non-expert users to create customized monitor/control systems. The eBlock platform has the potential to provide an interactive and customizable project-based learning environment that allows students to see how the topics studied in class relate to real-world applications by enabling students to create their own customized systems. In this paper we provide the motivation behind the eBlocks platform, a glimpse of the underlying platform implementation, and initial usability studies to demonstrate the feasibility of the eBlocks platform within STEM education. Lastly, we discuss potential applications of the eBlock platform and future directions.

2. RELATED WORK

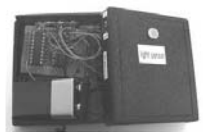
While a great deal of effort has been put forth to improve STEM education, we only highlight a few such projects that seek to include technology as a mechanism to captivate participants. Logiblocs [16] are one such technology consisting of small plastic blocks that users snap together to build various systems and consist of light sensors, buttons, AND, OR, NOT, speakers, bleeps, LEDs, etc. Logiblocs are targeted for primary education and educational toys. Children are able to design various systems and gain insight into the design process by trial and error. While many systems are viable such as flashing warning lights, light breaker alarms, or simplistic robots, the simplistic interfaces limit the possibility to build more advance systems to accompany STEM projects beyond the primary school level. Electronic Blocks [37] also utilize small building blocks consisting of

Figure 1: eBlocks provide a platform which enables non-experts to build a variety of interactive, hands-on projects.

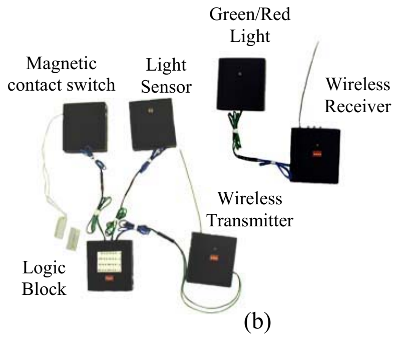
processors incorporated inside of LEGO Duplo Prima blocks. Users simply stack the correct combination of blocks to produce the desired output. Electronic Blocks consist of blocks that detect light, sound, and touch; produce physical output such as illuminating a light, producing sound, or movement; and intermediate blocks alter the expected action by inversion, toggle, or delay. Electronic Blocks are aimed at students between the ages of 3 to 8 years old and are again limited in use for older students due to the simplicity of the blocks.

Perhaps the most closely related project targeting STEM education in primary and middle school is that of MIT Crickets, having evolved from the MIT Programmable Bricks project [18][19]. A Cricket is a tiny computer, powered by a 9-volt battery that can receive information from two sensors and can control two motors. A key principle in Crickets is that people program them to perform a variety of functions. To make programming simple, the Logo language is used [26][27] – a simple, graphical, highly intuitive language. Crickets provided the foundation for the Lego Mindstorm product [36], consisting of numerous sensor and actuator Lego blocks that can be connected to a central microprocessor block to build a variety of small robots, again programmed using a simple graphical language. Crickets provide a general programmable computer, requiring programming by users. While we can learn much from the MIT Cricket approach, we have to consider situations where learning programming languages or having access to computers are not an option. Our goal is to complement the current curriculum without requiring programming or electronics knowledge.

The Infinity Project [10] is an intriguing year long course available to high schools to expose students to, and excite students about, careers in engineering and technology. The course curriculum is designed for students who have completed Algebra II and at least one science laboratory. A graphical software design environment is provided to run on a PC accompanied by a Technology Kit including a central board built around a digital signal processor (DSP), multimedia peripherals, such as digital web camera, speakers, and microphones, and an accompanying text book. Students learn about engineering through the design and experimentation on topics including the Internet, wireless communications, digital imaging, and music and audio processing. The Infinity Project is a fascinating project but mandates prerequisites in math and science. Additionally, we want to involve students before they reach high school. Our goal is for students of all ages and background to be able to easily utilize the eBlock platform to build and configure a variety of systems, providing an opportunity for hands-on projects relating topics discussed in class. While the goals of each project differ,



(a)



(b)

Figure 2: (a) Internals of an eBlock light sensor, (b) implementation of a Garage-open-at-night system using eBlock prototypes.

our proposed project can serve as a complement to the Infinity project. Studies have shown students need to be engaged early. The eBlock project will hopefully give middle school (and eventually elementary school) students the confidence to continue to pursue math and science and participate in programs, such as the Infinity project, throughout their education.

Numerous platforms are available which expose students to technology, with many focusing on the underlying platform. While an introduction to, and an understanding of, the technologies that makeup the digital age is important, we strive to enable students to utilize technology without having expert knowledge of the underlying platform.

3. EBLOCK OVERVIEW

The inspiration for the eBlock platform arose from a simple observation that there exist numerous applications across a variety of domains (residential, commercial, medical, and so on) that can be solved with a handful of sensors and some control logic. For example, a homeowner may want a system to monitor whether a garage door was left open at night. Figure 2(b) illustrates how this system can be implemented using a light sensor, contact switch, a LED (light-emitting diode), and control logic that turns the LED red when no light is sensed (daytime) combined with an open contact switch (indicating the garage door is open). Additionally, if the homeowner wants to place the red/green warning LED away from the sensors, wireless transmit and receive modules can be utilized. As another example, a homeowner might want to set up a system that detects if their child is sleepwalking in the dark. Figure 3(a) illustrates such a system involving a motion sensor block and a light sensor block, feeding into a logic block detecting the motion sensor outputting true and the light sensor outputting false, wirelessly feeding into an LED or buzzer block. Other examples include, but not limited to, a daytime doorbell, detecting motion on the property, detecting if a side gate is open before letting pets into the backyard, monitoring the availability of conference rooms within a company, or monitoring nocturnal activity by turning a recording device on when motion is sensed within a defined area.

With so many application possibilities, why aren't these systems more prevalent? One reason is cost; most applications are too specialized to be cost effective when one considers the many real costs (packaging, marketing, store placement, etc.) of introducing a new consumer product into the market. Thus, very few off-the-

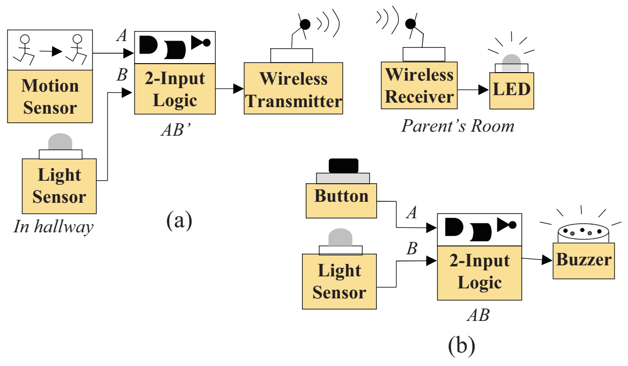


Figure 3: Various applications built with eBlocks, (a) Sleepwalking Detector and (b) Daytime Doorbell.

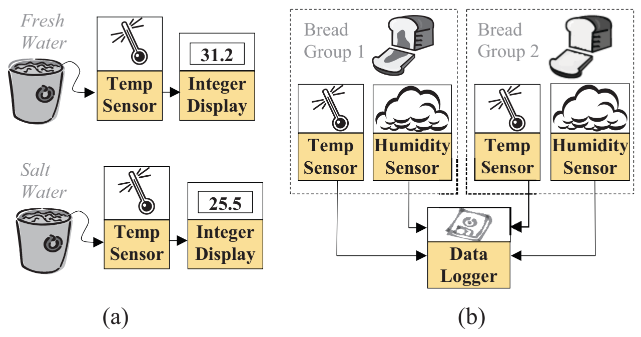


Figure 4: Sample eBlock projects. In (a) eBlocks are used to observe the freezing of fresh and salt water. In (b) eBlock temperature sensors are used to observe the effects of temperature and moisture effect mold on a piece of bread.

shelf applications appear in the market place. Furthermore, while these applications are seemingly simple, building a custom working system from standard electronic components is beyond the skills of ordinary people (homeowners, teachers, office workers, etc). In fact, building these systems is even a challenge for most engineers who haven't been specifically trained in embedded system design. Lastly, hiring an engineer to build a custom solution is possible, but the cost is seldom justifiable.

The eBlocks platform [5][8] emerged from a desire to empower regular people, having no programming or electronics experience, to build custom electronic systems. Rather than creating specialized systems for each application scenario, the eBlock platform is composed of commonly used building blocks that users connect blocks together like Legos™ to form an application. Because the platform consists of building blocks, the development cost of each block can be amortized over a larger volume, yielding lower costs. Furthermore, through the use of hardware wrappers the underlying platform implementation is made transparent to users, alleviating the need for programming or electronics experience.

3.1. EBLOCK PLATFORM

The key to the eBlocks platform is to integrate compute intelligence with previously "dumb" sensors and actuators, as shown in Figure 2(a). In our case, we utilized a PIC16F690 microprocessor [20]. The processor hides low-level interfacing details and communication between nodes so that users simply connect components together to specify the functionality of the application.

A variety of building blocks have been identified, with approximately 20 different blocks physically implemented. These building blocks can be classified into one of the following four categories.

- Sensor blocks - monitor the environment, including motion sensors, light sensors, buttons, contact switches, and so on
- Compute blocks - perform basic logic transformations (e.g. AND, OR, NOT) or basic state functions (e.g. prolong, toggle, trip, pulse).
- Output blocks - provide stimuli, and include light-emitting diodes (LEDs), beepers, electric relays, etc.
- Communication blocks - provide wireless point-to-point communication or replicate a signal to send to multiple blocks.

The initial eBlock platform contains both fixed function and programmable blocks. Fixed function blocks have a specific predefined function that may have slight configurability (i.e. which logical operation to perform, the length of time a signal is prolonged). A generic 2-input, 2-output programmable block is also supported, the programmable block can be configured to perform any user specified function. Using the eBlock simulator, the desired functionality of eBlock system is defined. The simulator interface partitions the desired functionality unto a programmable block (or multiple blocks depending on the desired functionality) and downloads the new function to the programmable block using a PC interface. The programmable block is easily reprogrammed without requiring compilation or device programming by the user. While we have strived to keep the interface and tool chain accessible to novice users, we note that it is more likely that advance users or users with some programming background would opt to utilize the programmable block.

The initial version of eBlocks operate on Boolean data, blocks send and operate on "yes" or "no" packets. Applications are built by snapping blocks together and the order in which blocks connected specifies the functionality of the system. Extensive usability testing has been performed on Boolean blocks interacting with close to 500 participants [5][6][7]. Findings indicate that over 50% of participants, with little or no training, are able to successfully create a variety of applications within 10 minutes of being introduced to the platform. Boolean values however limit the types of projects that can be built, in particular STEM projects require building blocks that operation within the integer domain. Thus, extending the current platform to incorporate integer based eBlocks is essential.

4. EBLOCK EDUCATION KIT

eBlocks have the potential to enhance STEM education by providing students with the opportunity to build customizable and interactive projects that complement the specific topic studied without the burden of first having extensive programming or electronics experience. Course work becomes more interesting, memorable, and other skills such as problem solving and team-

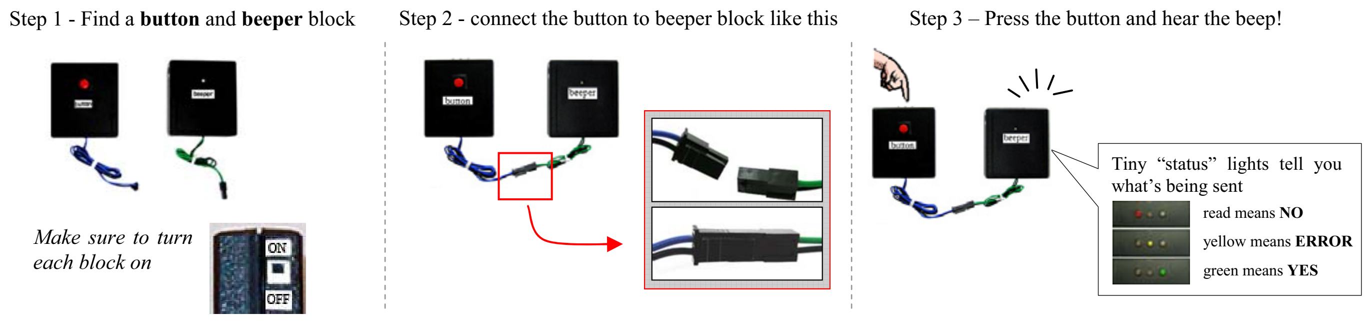

Step 1 - Find a **button** and **beeper** block
Make sure to turn each block on
ON
OFF
Step 2 - connect the button to beeper block like this
Step 3 – Press the button and hear the beep!
Tiny "status" lights tell you what's being sent
read means **NO**
yellow means **ERROR**
green means **YES**

Figure 5: A one page starter guide is provided to participants to quickly illustrate how eBlocks are powered on, connected to one another, and how to determine what the blocks are communicating to one another.

work are utilized and developed. Thus, we set out to define an eBlock Education Kit to target STEM education starting with middle school STEM education with future extensions reaching to the high-school and university level.

To begin we needed to identify which building blocks (i.e. sensor, intermediate, and output blocks) to include in the eBlock Education Kit so that students are able to build a wide variety of applications. Yet, at the same time we must balance the number of components included as not to overwhelm students with a gigantic catalog of eBlocks. A set of interactive projects for middle school students were identified by studying the topics included in the existing STEM curriculum listed on the local school district's webpage [34] as well as several other websites that provide descriptions of middle school science fair projects [23][24][29]. Based on these project ideas, we can determine the types of building blocks required in the initial set.

As we discover the project possibilities, we are also compiling a project guide booklet. The booklet is intended to get teachers and students started by illustrating sample systems that can be implemented with the eBlock Education Kit. Each project begins by posing a question. For example, why don't oceans freeze as quickly as lakes? Next, steps are outlined to help students connect various eBlock components together to build a project that tests the posed question. Figure 4(a) illustrate how temperature sensor and integer display eBlocks can be snapped together to observe the freezing point between containers of fresh and salt water. Students are then asked questions that require interaction with the platform, such as placing both containers into a freezer and recording the temperature of each container every 10 minutes. Additionally a series of follow up questions are provided, requiring students to reflect on the experiment. Figure 4(b) illustrates the setup for another experiment intended to monitor the effects of temperature and moisture on mold. While the application is different we can see that some of the blocks can be reused to create different projects. Furthermore, while the project guide booklet provides numerous projects it serves as a starting point for students and teachers to create their own projects.

As we identify and develop each building block within the platform, we must also ensure all of the resulting physical prototype have intuitive interfaces such that user can readily understand the functionality of that block without extensive training. Thus, as part of the iterative design process, we have also begun testing to determine the effectiveness of the newly proposed interfaces.

5. USABILITY EXPERIMENTS

5.1. Prototype-based Usability Testing

Utilizing the new integer-based eBlock prototypes, we conducted a series of informal experiments to see what aspects of the eBlock Education Kit were readily understood and what aspects were confusing. While we ultimately plan to perform usability testing with the target audience (middle school students), we started usability testing with small groups of college level students to provide an initial glimpse into the interface design and provide an opportunity for refinement before deploying the eBlock platform into middle schools. If an interface is not usable by university students, the interfaces developed would have no hope in a middle school setting. These experiments provide an initial evaluation of the various interfaces and a second testing phase is planned to target middle school students.

Participants were categorized into two groups, non-expert and expert. Non-expert participants are students at the university who have no engineering background, students in non-engineering majors such as pharmacy, accounting, or political science. Expert participants are students who have had at least one semester in an engineering major such as computer engineering, chemical engineering, or hydrology.

All participants were provided with a handout containing a brief introduction to eBlocks, as shown in Figure 5. The flyer quickly illustrates how blocks are powered on, connect to one another, and communicate with one another. Additionally, participants were provided with a set of physical blocks and an eBlock catalog containing descriptions of each block's functionality and interface. These handouts were the only materials provided to participants; we did not provide an oral introduction or answer any questions during testing.

A written survey was also provided that asked participants to build the two systems shown in Figure 6. The first application was not utilized to determine the usability of the integer interfaces (as no integer interfaces were utilized). Rather, this example merely served to indicate if users were at least able to understand how to utilize the eBlock platform. Figure 6(b) illustrates the integer-based application students were asked to construct. Note, the block diagram was not provided for the second system, rather a

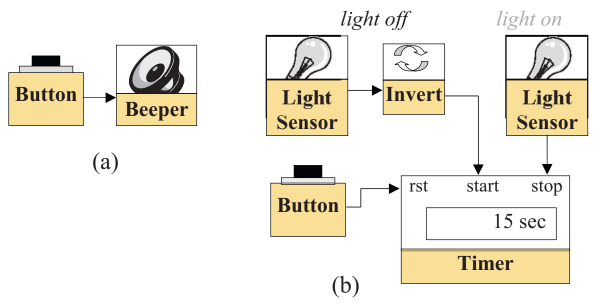


Figure 6: Systems built by participants in prototype-based usability testing, (a) corresponds to doorbell application of System 1 and (b) corresponds to light usage monitoring application of System 2.

textual description of the desired system functionality was provided. In this experiment, we were interested if users would be able to utilize the timer depicted in Figure 6(b). When start = 1 (or "yes") the timer is activated and tracks the number of seconds, when stop = 1 the timer is deactivated, and when rst = 1 the timer display zeros out and the internal counter is reset to zero. Table 1 illustrates the results of these experiments. All participants were easily able to construct the first system, indicating a basic understanding of the eBlock platform. The second system which is more complex, mixing both Boolean and integer blocks, yielded slightly lower success rates. However, 40% of the non-expert participants were able to successfully construct the system, with another 40% close to correct with a minor error in the connection of blocks. It is not surprising to see that expert participants had higher success rates, with all expert participants able to successfully build the corresponding systems. On average experts finished constructing both systems in 8 minutes, while non-experts required 20 minutes.

Because this was the first time participants had encountered eBlocks, we also decided to test if training made any difference in the success rate. In these experiments participants were additionally provided with a training example which included a system description, a block diagram with step-by-step instructions indicating how to connect and configure each block, and a description of how each block contributes to the desired functionality. Many of the blocks used in the training example could then be reused to implement the application illustrated in Figure 6(b). Success rates for non-expert participants increased from 40% to 50%, with another 38% close to constructing a working project. Surprisingly, we found that success rates for expert participants decreased compare to expert participants with no training, as illustrated in Table 2. These results warrant further investigation. The small sample size may have contributed to these irregularities or it may simply be that the training example provided too many details and caused users to second guess themselves or ignore the training phase altogether.

5.2. Simulator-Based Usability Testing

While physical block prototypes are an ideal testing platform, the eBlock simulator shown in Figure 7 was also utilized to enable a larger number of student to participate, as well as enable rapid testing of a numerous interfaces. The simulator is a Java-applet-based graphical user interface (GUI) for the design entry and simulation of various eBlock systems. A pallet located on the right edge of the simulator contains a variety of blocks users can select to utilize within a system simply by dragging blocks into the workspace. The blocks are organized as sensor, output, or computation/communication blocks, and can be viewed by clicking on the corresponding tabs located at the top of the pallet. Users establish connections between blocks by drawing lines between circular representations of blocks' input and outputs. Additionally, users can configure blocks by clicking on the DIP switch image, toggling individual switches between "yes" and "no". eBlocks that sense or interact with their environment, such as light or motion sensors, include accompanying visual representation of their environmental stimuli/interaction to simulate the corresponding environment. For example, each light sensor is accompanied by a light bulb icon that a user clicks to toggle the light on and off. The light bulb indicates the sensor's external environment (e.g., whether the sensor detects the presence or absence of light). The electric relay is connected to a lamp icon that is on or off depending on the block's input. In the bottom left corner of the simulator, a gray text box provides context-sensitive help. If a user places the mouse cursor over blocks in the simulator, the block's description and interface automatically appears in the text box. While the eBlock simulator is behaviorally correct, the simulator does not capture all low-level timing details.

Utilizing the eBlock simulator to gauge usability is expected to

Table 1: Success rates for individuals who did not received training, with participants who were close to correct in parenthesis.

	% Success		Total # Participants
	System 1	System 2	
Non-expert	100 % (0 %)	40 % (40 %)	5
Expert	100 % (0 %)	100 % (0 %)	2

Table 2: Success rates for individuals who received training, with participants who where close to correct in parenthesis.

	% Success		Total # Participants
	System 1	System 2	
Non-expert	100 % (0 %)	50 % (38 %)	8
Expert	100 % (0 %)	57 % (43 %)	7

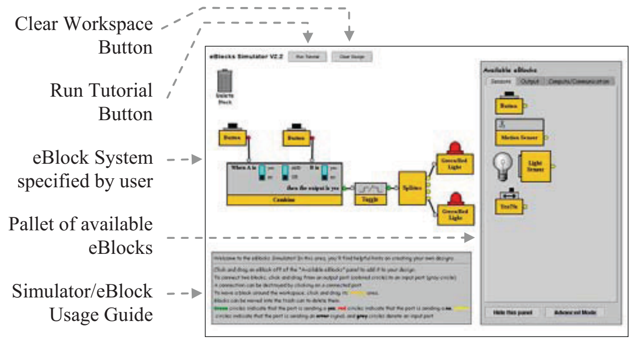

Clear Workspace Button
Run Tutorial Button
eBlock System specified by user
Pallet of available eBlocks
Simulator/eBlock Usage Guide

Figure 7: Java-based eBlock simulator that enables participants to build, test, and refine various eBlock systems without the overhead of physical blocks.

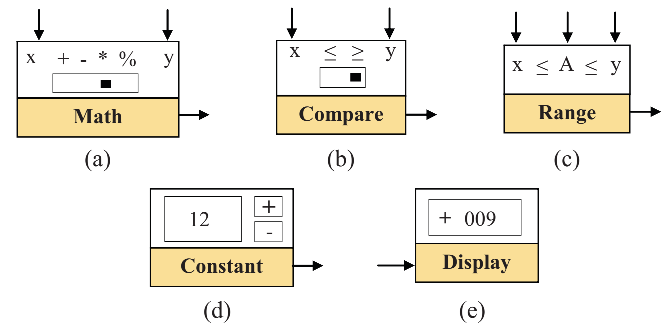


Figure 8: Additional integer-based eBlocks.

Q1. Simple Math (12-3)

Using Constant eBlocks, a Display eBlock, and any other eBlocks you find appropriate, create a system that calculates "12 – 3" and displays the result on a Display eBlock.

Q2. Simple Math (4*3)

Using Constant eBlocks, a Display eBlock, and any other eBlocks you find appropriate, create a system that calculates "4 * 3" and displays the result on a Display eBlock.

Q3. Compare (6>=3)

Using Constant eBlocks, a Green/Red Light eBlock, and any other eBlocks you find appropriate, create a system that lights up a green light if the number 6 is greater than or equal to 3.

Q4. Compare (6<=15)

Using Constant eBlocks, a Green/Red Light eBlock, and any other eBlocks you find appropriate, create a system that lights up a green light if the number 6 is less than or equal to 15.

Q5. Cascaded Math (5+2*4)

Using Constant eBlocks, a Display eBlock, and any other eBlocks you find appropriate, create a system that calculates "5 + 2 * 4" and displays the result on a Display eBlock.

Q6. Cascaded Math (7-(6/3))

Using Constant eBlocks, a Display eBlock, and any other eBlocks you find appropriate, create a system that calculates "7 – (6 / 3)" and displays the result on a Display eBlock.

Q7. Range (2<5<9)

Using Constant eBlocks, a Green/Red Light eBlock, and any other eBlocks you find appropriate, create a system that lights up a green light if the number 5 is greater than 2 and less than 9.

Figure 9: Questions posed to participants to determine usability of newly added integer-based eBlocks.

yield slightly higher success rates than usability experiments conducted with the corresponding physical prototypes. In the initial testing phases we found that users with little or no engineering background did not know how to use components such as DIP switches or slide switches. Additionally, users frequently forget to power on each block, had trouble connecting blocks and attempted to connect two outputs or two inputs together, or crossed the ground and data wires. While these issues are not fundamentally related to the blocks' functionality, they nevertheless reduced the resulting success rates. Most of the initial experiments provided no training for users, thus over time we believe the usability obtained from physical testing would increase to match those obtained in the simulator testing environment.

5.2.1 Additional Integer eBlocks

In addition to the timer block shown in Figure 6(a), we also expanded the eBlock library to include the five new integer based eBlocks shown in Figure 8. The math and compare blocks include a slide switch to enable a user to configure the block to perform various operations. The range block is a multi-function block. If all three inputs are connected, the range block performs the comparison $x \leq A \leq y$. If the x and A inputs are connected, the range block performs the comparison $x \leq A$. Lastly, if the A and y inputs are connected, the range block performs the comparison $A \leq y$. The constant block consists of a display indicating a constant integer value outputted by the block. A "+" button is provided to increase the block's value, and a "-" button is provided to decrease the block's value. Lastly, the display block receives an integer value and displays the corresponding value.

5.2.2 Integer Block Testing

To determine the usability of the various integer-based blocks, we performed experiments with 11 participants who were incoming engineering freshmen and 19 undergraduate students who were in non-engineering majors. Participants utilized the eBlock simulator to build various integer-based systems. Students were asked a total of 4 questions, one question from each of four problem classifications listed in Figure 9. Questions classified as "Simple Math" requires construction of a math formulation such as 5 + 4, requiring a single math block. Questions classified "Compare" requires construction of a comparison such as 4 > 2, the comparison can be implemented utilizing either the compare block or by connecting two out of the three inputs of the range block. Questions classified as "Cascaded Math" is a slightly more difficult math formulation requiring two operators such as 2 * 4 – 5, where the output of one math block is used as the input to a second math block. Questions classified as "Range" requires construction of a range formulation such as 3 < 6 < 8. The range operation can be implemented by either using all three inputs of the range block or by utilizing two compare blocks and a 2-input combine block. One comparison block evaluates 3<6, the second comparison block evaluates 6<8, then both outputs are fed to the 2-input combine block to see if both comparisons are true. Participants were allotted 25 to 30 minutes to complete all 4 questions.

The majority of participants were able to utilize the math block. As shown in Table 3, 82% of users are able to successfully utilize a single math eBlock, while 72% of users able to cascade two math blocks. Within the current integer based block definitions there are two methods to construct a comparison, the comparison block or the range block. Because the comparison and range block have overlapping functionality, we wanted to determine if a single block could be provided to perform both operations. Participants asked to utilize the comparison block to perform a compare operation yielded an average success rate of 81%, as shown in the "Compare using Compare eBlock" entry. Participants asked to utilize the range block to perform a compare operation experienced a significantly lower success rate of 54%, as denoted in the "Compare using Range eBlock" row. Similarly, the range operation can be implemented utilizing either a range or comparison block. Participants using the compare block to perform a range operation yielded a success rate of 13%, as shown in the "Range using Compare eBlock" entry. Participants

Table 3: Success rates for non-expert participants in simulator-based usability experiments given various integer-based interfaces.

	Percent Success	Number of Participants
Simple Math	82 %	49
Cascaded Math	72 %	60
Compare using Compare eBlock	81 %	26
Compare using Range eBlock	54 %	24
Range using Compare eBlock	13 %	15
Range using Range eBlock	53 %	15

provided with a range block yielding an average success rate of 53%, as listed in the "Range using Range eBlock" row.

In both the compare and range experiments the users achieve higher success rates utilizing the block most closely matching the desired functionality. Soloway, et al. performed studies indicating that programming constructs need to "cognitively fit" with the user to be effective [32], thus it is not surprising that we similarly find users achieve higher success rate when a block which closely fit the desired functionality is provided. As part of future work, we plan to conduct additional experiments in which users are provided with both the comparison and range blocks, to determine if higher success rates are actually achieved, or if too many blocks lead to confusion.

6. ROLE OF EBLOCKS IN EDUCATION

While the eBlock Education Kit is presently being developed and tested with middle school STEM courses in mind, we foresee the use of this platform at the high-school and university level. The flexibility of the eBlock platform enables use across a wide range of age groups and disciplines.

At the middle school level the eBlock platform can complement the existing curriculum by enabling students to build complementary monitor/control systems and test out the ideas studied in class. In addition to improving STEM performance, we hope students will begin to think about how technology is integrated into their everyday surroundings and begin to question how it all works. Because the eBlock platform is intended to complement the existing curriculum, rather than redefine the existing curriculum, we do not anticipate any low-level discussion about the underlying implementation or discussion of the actual engineering principles. An afterschool environment would however enable greater flexibility and may provide an opportunity for student to learn more about engineering and high-level engineering concepts.

At the high school level we similarly envision eBlocks complementing the existing curriculum, which would limit discussion of the underlying platform. However, at the high school level courses such as the Infinity Project [10] are intended to expose students to engineering. Depending on the type of courses available, expertise of the instructors, and the availability of resources, the eBlock platform can be similarly be utilized as an introduction to embedded systems. At a higher abstraction, students can practice problem solving by building customized monitor/control systems for various applications, utilize a variety of sensors and actuators, and learn about data collection and analysis. At the lowest level students can utilize eBlocks as a low-cost platform to start investigating communication, interfacing, and usability issues.

At the university level students in engineering disciples can utilize eBlocks in a variety of programming and hardware courses. Many engineering students spend weeks or months learning theory before every being able to build a working system. The eBlock platform provides a quick and easy starting point for engineering students to build a variety of computing systems. Then as students learn about various topics, such as interfacing or interrupt service routines, the platform can be dissected to demonstrate the topic studied within a larger working system. By utilizing a top-down methodology students have a better perspective of how each of the topics studied relate to one another. Similarly, the logic and state blocks can be utilized by students in an introductory programming course as a physical platform or within the simulation environment to setup and test Boolean equations and state-based relationships through a trial and error methodology. Students can relate the functionality of these Boolean and state operators to everyday situations, such as detecting light or button presses to control a doorbell, before applying these operators to programming constructs. Lastly, the eBlock platform has the potential to enhance other university STEM programs. Similar to the middle school application, these students can utilize the eBlock platform to develop complementary monitor/control systems, and illustrate the role of technology within a particular discipline.

7. CONCLUSIONS AND FUTURE WORK

To stimulate interest and improve performance in STEM education we have proposed the use of the eBlock platform to enable the construction of interactive projects that complement the existing curriculum. Initial experiments show that roughly half of the novice participants can readily utilize these building blocks with little or no training. While the initial results are promising, many questions remain unanswered. As future work we must consider a more diverse group of participants, including students at the middle and high school level. We are current working with local middle schools and community based organizations to begin testing the usability and feasibility of the eBlocks platform. Furthermore, the eBlock systems constructed by participants contained only a handful of blocks. We must also investigate the usability of the eBlock platform given increasingly complex applications, as well as the ability for users to define and design their own customized system without the aid of a project booklet.

8. REFERENCES

[1] Arizona Department of Education. Arizona's Instrument to Measure Standards (AIMS) Results, State Report Card, http://www.ade.state.az.us/srcs/statereportcards/StateReportCard06-07.pdf, 2007.

[2] Barron, B., D. Schwartz, N. Vye, A. Moore, A. Petrosino, L. Zech, J. Bransford. Doing with Understanding: Lessons from Research on Problem- and Project-Based Learning. The Journal of the Learning Sciences, 7, pp. 271-311, 1998.

[3] Blumenfeld, P. C., T. Kempler, J. S. Krajcik. Motivation and cognitive engagement in learning environments. In R. K. Sawyer (Ed.), Cambridge handbook of the learning sciences, pp. 475-488, 2006.

[4] Committee Prospering in the Global Economy of the 21st Century: An Agenda for American Science and Technology; Committee on Science, Engineering, and Public Policy. Rising Above the Gathering Storm: Energizing and Employing America for a Brighter Economic Future. Washington DC: National Academy Press, 2007. http://www.nap.edu/catalog/11463.html

[5] Cotterell, S., K. Downey, F. Vahid. Applications and Experiments with eBlocks - Electronic Blocks for Basic Sensor-Based Systems. Sensor and Ad Hoc Communications and Networks, October 2004.

[6] Cotterell, S., F. Vahid. A Logic Block Enabling Logic Configuration by Non-Experts in Sensor Networks.CHI, April 2005.

[7] Cotterell, S., F. Vahid. Usability of State Based Boolean eBlocks. HCII, July 2005.

[8] eBlocks Science Kits. http://www.ece.arizona.edu/~uecs/eb_kits/index.html, 2008.

[9] Geier, R., P., Blumenfeld, R. Marx, J. Krajcik, B. Fishman, E. Soloway. Standardized test outcomes of urban students participating in standards and project based science curricula. Proceedings of the sixth international conference of the learning sciences, 310-317, 2004.

[10] Infinity Project, The. http://www.infinity-project.org/, 2006.

[11] Institute of Education Sciences, U.S. Department of Education. International Outcomes of Learning in Mathematics Literacy and Problem Solving: PISA 2003 Results from the US Perspective. http://nces.ed.gov/surveys/PISA/PISA2003highlights.asp, 2007.

[12] Institute of Education Sciences, U.S. Department of Education. Trends in International Mathematics and Science Study (TIMSS) 2003, http://nces.ed.gov/timss/Results03.asp, 2007.

[13] Jackson, S. The Quiet Crisis: Falling Short in Producing American Science and Technical Talent. Building Engineering and Science Talent (BEST) Report, 2004.

[14] Kovac, J. Student active learning methods in general chemistry. Journal of Chemical Education, Vol. 76, No. 1, pp. 120-124, 1999.

[15] Krajcik, J., C. M. Czerniak. Teaching science in elementary and middle school: A project-based approach (3rd ed.). Mahwah, NJ: Lawrence Erlbaum, 2007.

[16] Logiblocs, http://www.logiblocs.com/, 2007.

[17] Lunsford, B., Herzog, M. Active learning in anatomy and physiology: Student reactions and outcomes in a nontraditional AP course. The American Biology Teacher, Vol. 59, No. 2, pp. 80-84, 1997.

[18] Martin, F., et. al. The MIT Programmable Brick. http://lcs.www.media.mit.edu/groups/el/projects/programmable-brick/, 2006.

[19] Martin, F., et. al. Crickets: Tiny Computers for Big Ideas. http://lcs.www.media.mit.edu/people/fredm/projects/cricket/, 2006.

[20] Microchip Technology Inc. PIC16F631/677/685/687/689/690 Data Sheet. http://ww1.microchip.com/downloads/en/DeviceDoc/41262E.pdf, 2008.

[21] Mistler-Jackson, M., N. B. Songer. Student motivation and internet technology: Are students empowered to learn science? Journal of Research in Science Teaching, 37, pp. 459-479, 2000.

[22] National Science Board. Science and Engineering Indicators 2006, http://www.nsf.gov/statistics/seind06/, 2006.

[23] Newkirk, K. Selah Intermediate School Science Projects, Science Project 2006 Index, http://www.selah.k12.wa.us/SOAR/SciProj2006/index. htm, 2006.

[24] Phillipsburg Christian Academy Science Fair Projects, http://www.fellowshipch.org/pcasciencefair.html, 2007.

[25] Price, S., Y. Rogers. Let's get physical: the learning benefits of interacting in digitally augmented physical spaces. Computers & Education, 43(1-2), August 2004.

[26] Resnick, M. Behavior Construction Kits. Communications of the ACM 36, No. 7, pg. 64-71, 1993.

[27] Resnick, M., S. Ocko, S. Papert, LEGO, Logo, and Design, Children's Environments Quarterly 5(4), pg. 14-18, 1988.

[28] Richards, L., et. al. Promoting active learning with cases and instructional modules. Journal of Engineering Education, Vol. 84, No. 4, pp. 375-381, 1995.

[29] Rubin, J. Science Fair Projects Resources, http://www.juliantrubin.com/schooldirectory/fairresources.html, 2007.

[30] Schneider, R. M., J. Krajcik, R. W. Marx, E. Soloway. Performance of students in project-based science classrooms on a national measure of science achievement. Journal of Research in Science Teaching, 39(5), pp. 410-422, 2002.

[31] Science, Technology, Engineering, and Mathematics Education (STEMEd) Caucus Steering Committee. K-12 STEM ED Report Card: How Arizona Ranks. http://www.stemedcaucus.org, 2006.

[32] Soloway, E., Bonar, J., Ehrlich, K. Cognitive Strategies and Looping Constructs: An Empirical Study. Communications of the ACM, vol. 26, issue 11, pp. 853-860, 1983.

[33] Teitelbaum, M. The U.S. Science and Engineering Workforce: An Unconventional Portrait. National Academies' Government University Industry Research Roundtable (GUIRR) Summit, 2002.

[34] TUSD Curriculum Resource Guide, http://instech.tusd.k12.az.us/standards/index.asp, 2006.

[35] Wallice, K. America's Brain Drain Crisis. Reader's Digest, December 2005.

[36] Wallich, P. Mindstorms Not Just a Kid's Toy. IEEE Spectrum, Vol. 38, No. 9, September 2001.

[37] Wyeth, P., H. Purchase. Tangible Programming Elements for Young Children. Extended Abstract, Conference on Human Factors in Computing Systems, 2002.

Virtual Microcontrollers

Scott Sirowy[†], David Sheldon[†], Tony Givargis[‡], Frank Vahid[†]*

[†]Department of Computer Science and Engineering
University of California, Riverside, USA
*Also with the Center for Embedded Computer Systems at UC Irvine
{ssirowy,dsheldon,vahid }@cs.ucr.edu

[‡]Department of Computer Science
Center for Embedded Computer Systems
University of California, Irvine
givargis@ics.ucr.edu

Abstract

Embedded programming training today commonly involves numerous low-level details of a particular microcontroller. Such details shift focus away from higher-level structured embedded programming concepts. Thus, hard-to-break, unstructured programming habits are commonplace in the field. Yet structured embedded programming is becoming more necessary as embedded systems grow in complexity. We introduce a virtual microcontroller to address this problem. Freed from manufacturing or historical architectural issues, the virtual microcontroller contains the core features to support embedded programming training, and possesses an exceptionally clean interface to low-level features like timers, interrupt service routines, and UARTs. The virtual microcontroller can be mapped onto existing microcontrollers, or even onto FPGAs or a PC, providing more lab and book flexibility, at the expense of performance and size overhead. Most importantly, training can still use a bottom-up resource-aware approach, yet can focus more on structured embedded programming concepts.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education- Time Oriented Programming

General Terms

Design, Human Factors, Languages

Keywords

Embedded Programming, Time Oriented Programming, Education, virtualization, microcontrollers

1. Introduction

Increasingly complex embedded system functionality requires elevation of the introduction to embedded programming from low-level details to higher-level structured programming. Yet the importance of resource aware embedded programmers discourages hiding all low-level details via an operating system.

Present first courses or tutorials on embedded systems often focus on low-level details specific to a particular microcontroller, such as how to configure a particular microcontroller's timers, counters, or UARTs via configuration registers. Due to processor evolution reasons, such details are often convoluted, possibly involving delicate balances between setting of oscillator frequencies, timer registers, interrupt registers, and UART registers, to achieve a serial transmission at a particular baud rate. With hundreds of microcontroller variations, details differ significantly across and even within microcontroller families.

In contrast, embedded system complexity demands elevation of embedded programming to higher-level structured approaches. Such a structured approach may involve using state machine or dataflow computation models captured in a language like C, utilizing clear multi-tasking methods such as round-robin processing of concurrently-executing state machines, and having a clear and consistent methodology for dealing with timed input and output events.

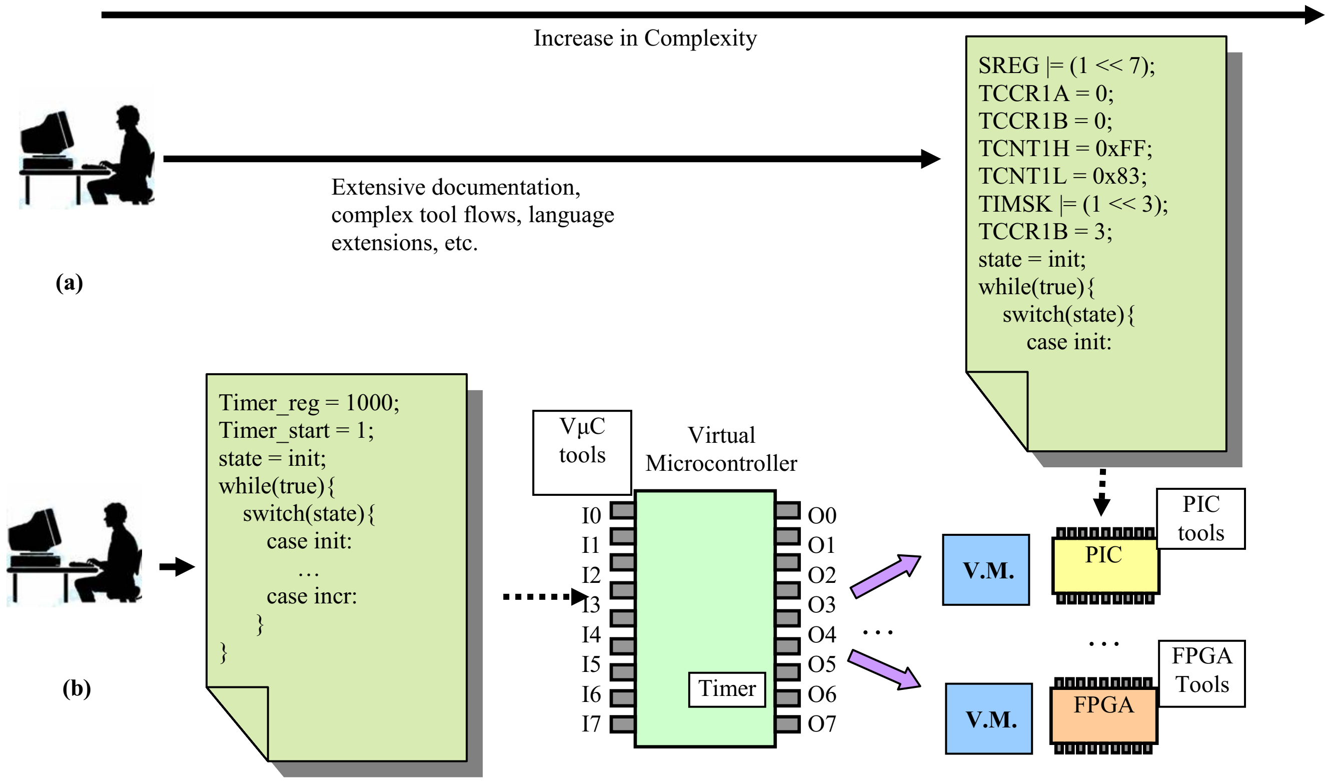
Two approaches are commonplace today for elevating the level of programming. A bottom-up approach first introduces low-level detailed programming, and then introduces higher-level concepts in a second course. While practical in the sense of teaching technical skills enabling physical implementation, this approach has the drawback of allowing undisciplined programming habits to develop, which can be hard to break later. Furthermore, the low-level details may discourage some students from pursuing studies in the area. Also, the second course commonly does not exist (or consists of a capstone project rather than additional training), or students may not take that course. Further, labs and textbooks are highly microcontroller-specific; changes due to obtaining new hardware may require substantial modifications to labs, textbooks, and other materials – and thus are resisted by many instructors.

In contrast, a top-down approach skips the low-level programming and may introduce embedded systems programming using a real-time operating system (RTOS) or other higher-level environment, which provides an abstraction that hides many details. While enabling focus on higher-level issues, this approach has the drawback of not providing students with an intuition of the basic underlying microcontroller mechanisms, and can lead to programmers not cognizant of important resource issues. While elevating programming is important, resource-awareness is also critical for practical embedded development, because many systems do not use RTOSes, and because understanding low-level concepts encourages more effective use of RTOS features.

We propose a compromise approach utilizing a virtual microcontroller, illustrated in Figure 1. The virtual microcontroller exposes fundamental low-level components to the programmer – timers, interrupt service routines, UARTs, general-purpose input/output, etc. – rather than hiding them using an RTOS, yet does so using simple clean structures uncluttered by transient or historical low-level complexities. The virtual microcontroller supports a fixed and non-parameterized architecture with a simple, reduced and C-compatible instruction set. The virtual microcontroller also supports the simplest programming apertures possible, allowing the student to focus on more important embedded programming concepts while still enabling a bottom-up perspective.

Further, the virtual microcontroller can execute on a variety of embedded devices, including various existing microcontrollers,

Figure 1: (a) Programming a real microcontroller often requires a complex flow that is confusing to beginning students and obfuscates crucial embedded systems concepts. (b) The virtual microcontroller, implemented on any number of devices, quickly allows the student to write structured embedded microcontroller code. Instructors must perform a one-time mapping of the virtual microcontroller on their particular device platform.



embedded microprocessors on boards having general-purpose I/O, field-programmable gate arrays (FPGAs), or even on a PC with appropriate general-purpose I/O additions. Instructors must perform a one-time mapping of the virtual microcontroller to their specific device. When changing devices later, instructors perform a remapping, but need not change books or lab materials. The virtual microcontroller also has a graphical simulator, allowing instructors to teach embedded programming even in the sub-optimal case of not having a hardware lab, or supporting additional training by students outside of lab. Even when using different devices, the student continues to use the same virtual microcontroller tools (simulator, debugger, compiler), rather than having to switch to the particular device's own tools.

2. Related Work

Several research projects attempt to improve engineering education. Hodge [8] introduces the concept of a *Virtual Circuit Laboratory*, a virtual environment for a beginning electrical engineering course that mimics failure modes in order to aid students in developing solid debugging techniques. The environment not only provides a convenient test environment, but also allows an instructor to concentrate more on teaching. Butler [2] developed a web-based microprocessor fundamental course, which includes a *Fundamental Computer* that provides students in a first year engineering course a less threatening introduction to microprocessors and how to program.

Other researchers have concentrated on developing or evaluating computing architectures for beginning students or non-engineers. Benjamin [1] describes the *BlackFin* architecture, a hybrid microcontroller and digital signal processor. The architecture provides a rich instruction set based on MIPS with variable width data, and parallel processing support. Ricks [10] evaluates the *VME Architecture* in the context of addressing the need for better embedded system education. The Eblocks project [4] concentrated on developing sensor blocks that people without programming or electronics knowledge could connect to build basic customized sensor-based embedded systems.

Much research has involved virtualization [9][11], with several commercial products developed in response to the need for portable virtual machines. VMware [13] and the open source product Xen [15] concentrate on developing virtual machines that allow the end-user to run multiple operating systems concurrently. The Java Virtual Machine [12] allows the programmer to write operating system independent code, and tools like DOS Box [5] and console emulators allow the user to run legacy applications in modern operating systems.

A number of real time operating systems have been introduced to provide a higher level of abstraction between the application software and embedded hardware, including the open source eCos [6], and VxVorks and RTLinux from WindRiver [14].

To the best of our knowledge, the work described in this paper is the first to describe a virtual microcontroller that can be physically implemented on existing platforms while also supporting programmer access to low-level yet clean, uncluttered microcontroller resources.

Figure 2: Virtual microcontroller MIPS instruction subset. We added RETI to simplify interrupt use.

1. ADD $1 $2 $3	11. OR $1 $2 $3
2. ADDI $1 $2 imm	12. ORI $1 $2 imm
3. ADDIU $1 $2 imm	13. ***RETI***
4. AND $1 $2 $3	14. SLL $1 $2 $3
5. ANDI $1 $2 imm	15. SLT $1 $2 $3
6. BEQ $1 $2 [Label]	16. SW $1 0($2)
7. J [Label]	17. SUB $1 $2 $3
8. JR $1	18. SUBI $1 $2 imm
9. LW $1 0($2)	19. XOR $1 $2 $3
10. NOOP	20. XORI $1 $2 imm

3. Programmer's View

We describe the virtual microcontroller (VµC) from the programmer's point of view. While programmable entirely in C, some instructors may wish to introduce the instruction set too – learning to program and read assembly code is still a common part of training, as assembly code is still written for certain drivers, and is sometimes examined during difficult debugging. We chose an instruction set based on the MIPS ISA (instruction set architecture) in [7].

We considered other choices, including an ARM-like instruction set or Java byte code. The ARM instruction set is similar to many microcontroller instruction sets, and there are already numerous virtual machine implementations built for Java byte code. However, the MIPS ISA provides a more intuitive instruction set, with the additional advantage that the ISA is usually already taught in beginning computer architecture courses.

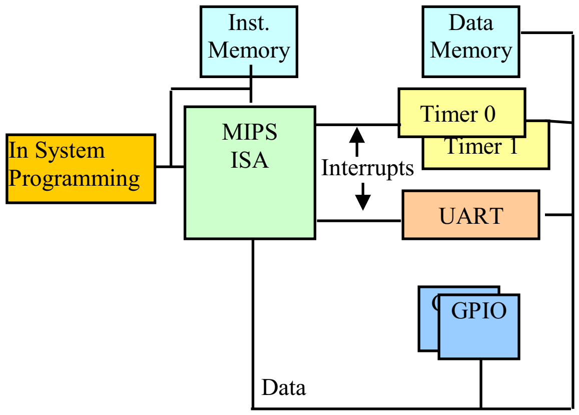
Learning the complete MIPS ISA might overwhelm students. We thus chose to use a twenty-instruction subset, shown in Figure 2, chosen as a representative mix of the entire MIPS ISA. Using the subset allows for easier learning, at the expense of larger code size and slower performance, which are less important in the context of training. The subset also has the drawback of requiring a special C compiler back-end (we are presently developing such a back-end to LCC), and not supporting existing MIPS binaries; again, these are lesser issues in a training setting.

We added a return from interrupt *RETI* instruction, which didn't exist in the original MIPS ISA. Because interrupts are so commonly used in embedded systems, we sought to support interrupts in a clean manner for students. An alternative approach would have been to require the student to use the jump register *JR* instruction to exit interrupts, but such usage distracts from the basic idea of interrupts.

The VµC architecture, shown in Figure 3, is a fixed 32-bit architecture. Microcontrollers used in the beginning classroom are often 8-bit and occasionally 16-bit, but small architectures add additional complexity in moving data between registers and data memory by forcing the student to use an accumulator or a stack, which obfuscate the higher level issues of embedded programming. A 32-bit architecture is both simple to understand and allows easy access to a large register set and memory. Although the virtual microcontroller would have allowed parameterization of the instruction set width for increased flexibility, the functionality was not needed in the context of an embedded systems course.

The VµC uses a four-kilobyte instruction memory, chosen based on off-the-shelf microcontroller memory sizes, and on the size required for several introductory embedded systems labs and exercises that we examined from several embedded systems

Figure 3: Virtual Microcontroller Architecture.



courses. The VµC's data memory is 64 kilobytes. A 32-bit architecture could support a four-gigabyte memory, but supporting such a large space would have made physical mapping to real microcontrollers nearly impossible. The upper half of the 64-kilobyte data memory is devoted to the VµC's memory mapped peripherals and registers. 64 kilobytes of data memory was more than adequate for any of the embedded programs we examined.

The VµC implements a simplified interrupt controller model as viewed by the programmer and the software. The interrupt controller model allowed for easy and intuitive implementation of interrupts with priorities. The interrupt controller consists of two memory-mapped special function registers, an *interrupt status register* and a *interrupt value register*. Together, the two registers act as a simplified interrupt vector table, which is commonly used in off-the-shelf microcontrollers. When the VµC is interrupted, the student simply reads the *interrupt value register* and runs the corresponding interrupt service routine using a programming construct akin to a *case* statement. For convenience, interrupts are automatically turned off by the VµC, so an interrupt routine cannot be interrupted by another interrupt request. Nested interrupts might have confused new students. The *interrupt status register* serves as a software switch to enable and/or disable interrupts, and can easily be written with the value '0' or '1'. Interrupt service routines complete with the *RETI* instruction. The *RETI* instruction will update the VµC's program counter to the last instruction not yet completed, and re-enable interrupts. The interrupt controller is connected to three peripherals: two timers, and a UART. The peripherals have fixed priorities, where the two timers are given top priority followed by the UART. Fixed priorities reduced the complexity of the virtual microcontroller as well as the software being run, allowing the student to concentrate on core embedded programming concepts, at the expense of situations where the priorities need to be different (which are rare in a learning setting).

The VµC interfaces to a basic set of peripherals that enable a variety of embedded systems to be created, from working with general-purpose input/output to timing-oriented programming. The virtual microcontroller separates input and output into two separate memory mapped eight-bit registers, which can be read (input register) or written (output register). Each input and output bit is also accessible individually by name (e.g., I1, O4). Having dedicated input and output eliminates the required step for most microcontrollers of configuring each input/output port's direction. One 8-bit input port and one 8-bit output port was sufficient for most introductory labs we examined. If more ports are needed, external extended parallel I/O techniques can be introduced.

The virtual microcontroller has two timers. At least one timer was required because much of an embedded programming curriculum revolves around timing-based computing models (state machines, interrupts, etc.) The VµC uses two timers because several concepts and applications become more intuitive with the use of two timers. For instance, a student might write an application that mimics two state machines that must transition on every half second, and every two seconds. While the two state machines can be implemented with only one timer, the programming becomes substantially easier with the use of multiple timers. The two timers offer limited configurability via the *Timer 0/1 Control register*. The student can allow or disallow the timers to interrupt the VµC, and can start and stop the timer by writing a few bits. The VµC timer's limited configurability provides a cleaner, concept-oriented interface than ones offered by off-the-shelf microcontrollers. The timers are programmed by writing the memory mapped register *Timer 0/1 Value register* with a millisecond value to time. This millisecond value is in contrast to off-the-shelf microcontrollers, which require writing a value based on that microcontroller's clock frequency. We chose millisecond resolution for the VµC's timers because all labs in the embedded programming course required that granularity or coarser. The millisecond resolution is also an easy time period for students to grasp quickly.

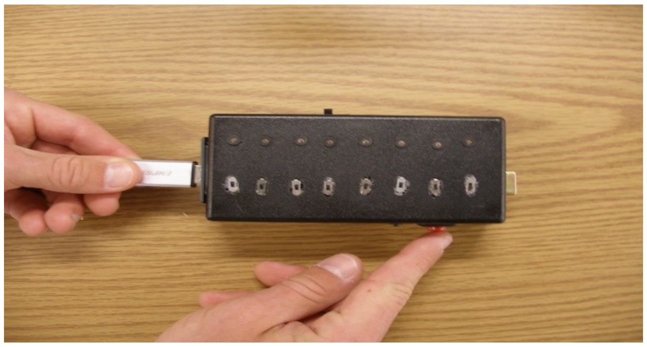
The VµC includes a UART (Universal Asynchronous Receiver/Transmitter), which allows a student to learn how to interface to serial devices, including a PC, for input, display, or debugging purposes. The UART can be programmed and configured using three intuitive memory-mapped registers, the *UART status register*, *UART TX Data register*, and the *UART RX data register*. To write to the UART, the program writes a value to the *UART TX Data register*, and writes a '1' to the *UART Status register* to signal a transmission start. Similarly, the program can read the *UART RX Data register* for valid data once the UART has interrupted the VµC core. As with the VµC's timers, we eliminated several additional features offered by off-the-shelf microcontrollers to ease programming. For instance, the UART baud rate is fixed at 9600, eliminating the need to configure the rate. That rate was chosen based on 9600 baud being the default rate for several off-the-shelf microcontrollers.

4. Portability

As long as a computing platform supports the virtual microcontroller described in Section 3, then code written for the virtual microcontroller will execute identically on different platforms. The need to port code from one platform to another, whether that port is a relatively simple recompilation, or a complete rewrite of the code base, is eliminated. For example, one piece of code that blinks lights every half second running on a virtual microcontroller implemented on a physical microcontroller will also blink the same lights every half second running on a PC-implemented virtual microcontroller.

An advantage of such portability includes the ability for a student to use one implementation at home (e.g., a PC-based implementation) while using a different implementation in a lab (e.g., an FPGA-based implementation). Even the same lab setting may use different implementations based on available physical resources.

Figure 4: The virtual microcontroller is programmed by simply plugging in a USB flash drive with the VµC program and pressing a button.



5. USB Programmability

The virtual microcontroller supports USB programming (here "programming" refers to downloading code into a device) via a USB flash drive, and not a traditional hardware programmer in which a chip is plugged in, programmed, and placed in-system. Such an approach requires non-volatile memory, and requires a removable chip, greatly limiting the ability to implement the virtual microcontroller on various existing devices. Such an approach also requires a separate programmer device, adding to cost, and introducing extra steps for a student. An alternative programming approach is to program a device in-system using a USB cable. While eliminating the need for a programming device, such an approach still requires a PC every time a student wants to change a program.

Instead, we chose a USB flash drive programming approach, illustrated in Figure 4. A student copies the desired program onto a USB drive as a file, plugs the drive into the VµC implementation, and presses a button on the VµC that downloads the program from the flash drive to the VµC instruction memory. The approach eliminates the need for non-volatile memory in the VµC. The approach enables students to load and change programs by inserting and swapping flash drives, enabling more mobility, and ease of examining behavior of each others' program. The approach also matches current usage schemes for popular electronic devices, allowing a beginning student to start programming with minimal effort, and using a familiar paradigm. The cost is that the VµC must contain an internal USB flash drive reader. We use an off-the-shelf reading device, which increases the size and cost of the VµC.

6. VµC Executable Format

The virtual microcontroller uses a human-readable assembly language file as the "executable" format. A traditional binary executable format is more compact, but is unreadable by humans. In contrast, an assembly format is more readable, providing a clearer understanding of what is being executed on the device, reducing the number of files that must be worked with, and possibly enabling comprehension of the program (perhaps via comments in the code). The assembly code is just-in-time (JIT) assembled to machine code inside the VµC. We considered C code as the distribution format, but assembly code enabled simpler JIT tools and also supports assembly coding. A drawback of assembly versus machine code is that unchecked assembly code is more

Figure 5: Virtual microcontroller program *AND* executable format, to increment the value in the general purpose output register every second.

```
--program increments output value on interrupt
J Main
ISR: LW $20 12($10) --load which int. fired
BEQ $20 $0 ISR_zero --branch to ISR 0
RETI
ISR_zero: ADDI $5 $0 1 --r5 holds ISR flag
RETI
Main: ADDI $1 $0 3  --3 is val. to start timer
ADDI $2 $0 500      --incr. 500 ms
ADDI $3 $0 1        --incremented
ADDIU $10 $0 32768  --mem mapped base
ADD $10 $10 $10     --mem mapped base
SW $2 9($10)        --ld timer w/ 1 sec.
SW $1 8($10)        --start timer
Loop: SW $9 2($10)  --r9 to IO output
BEQ $5 $3 update
J Loop
update: ADDI $5 $0 0  --clr interrupt flag
ADD $9 $9 $3          --increment by one
J Loop
```

likely to contain errors (students almost never modify tool-generated machine code, but may modify assembly code). In the VμC, a JIT assembler error causes an error LED to illuminate (a future version may also write assembler errors to an error file on the USB flash drive.) The JIT assembler approach has an additional advantage of requiring no PC-based tools other than a text editor, even allowing assembly code to be developed on a cell-phone or PDA, saved to a USB flash drive, and downloaded to the VμC. Nevertheless, in an environment with a PC-based C compiler or assembler, enforcement of a methodology involving an assembly-code checking tool, or avoidance of changing of compiler-generated assembly code, may be helpful.

Figure 5 shows a sample virtual microcontroller assembly program that increments the value of the general purpose output every half second. Both comments and labels are allowed, to increase the readability of the application. Comments begin with the symbol '--', and continue to end of the current line. Labels are supported as a convenience to the application programmer.

The interrupt vector is at address 1 in the program. When an interrupt occurs, the program code must poll the *interrupt value register* to determine which interrupt should be serviced. In the increment example, only one interrupt could have occurred, but the code still performs the check on the *interrupt value register* to make the code extendable later.

7. Simulator

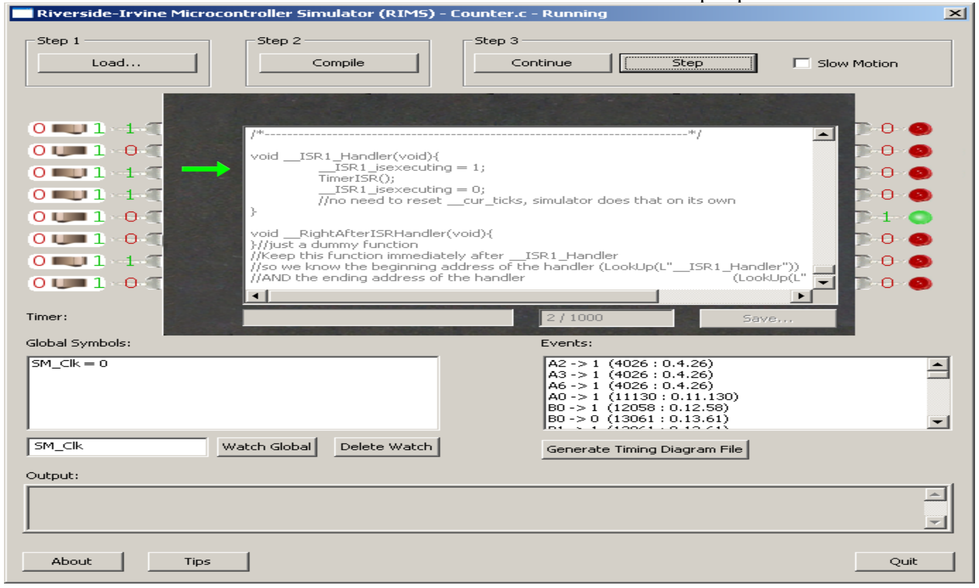
We have also developed a graphical VμC simulator. The simulator supports standard microcontroller simulator/debugger functions, such as steps, breakpoints, run for X simulated seconds, input/output value writes/reads, observation of internal registers including memory-mapped peripheral registers, etc. Furthermore, the simulator provides a graphical view of the timers as they count up to their interrupt time, akin to a "status bar" display ranging from 0% to 100%. Figure 6 shows a screenshot of the simulator. The simulator supports development in the absence of a physical device, and is also useful for instructors when demonstrating new concepts with a projected display.

8. Proof of Concept and Experiments

We implemented the VμC on various physical platforms. Each implementation was based on a core instruction set simulator, which consisted of just under 1,000 lines of C code. The code base is highly modular, allowing further mappings of the VμC to be created with less effort. The differences in each VμC implementation lied in how we mapped the VμC peripherals to physical peripherals.

Figure 7 shows several implementations of the virtual microcontroller. The implementation shown in Figure 7(a) emulates the VμC on a physical Atmel AVR microcontroller, combined with a PIC 18 microcontroller for interfacing with the USB reader device. In this implementation, we physically tied the VμC's general purpose input and output to switches and LEDs,

Figure 6: Virtual microcontroller simulator prototype. The simulator supports standard debugger and register views, as well as a high level view of the virtual microcontroller and connected peripherals.



```
/*----------------------------------------------------------------*/

void __ISR1_Handler(void){
        __ISR1_isexecuting = 1;
        TimerISR();
        __ISR1_isexecuting = 0;
        //no need to reset __cur_ticks, simulator does that on its own
}

void __RightAfterISRHandler(void){
}//just a dummy function
//Keep this function immediately after __ISR1_Handler
//so we know the beginning address of the handler (LookUp(L"__ISR1_Handler"))
//AND the ending address of the handler                          (LookUp(L"
```

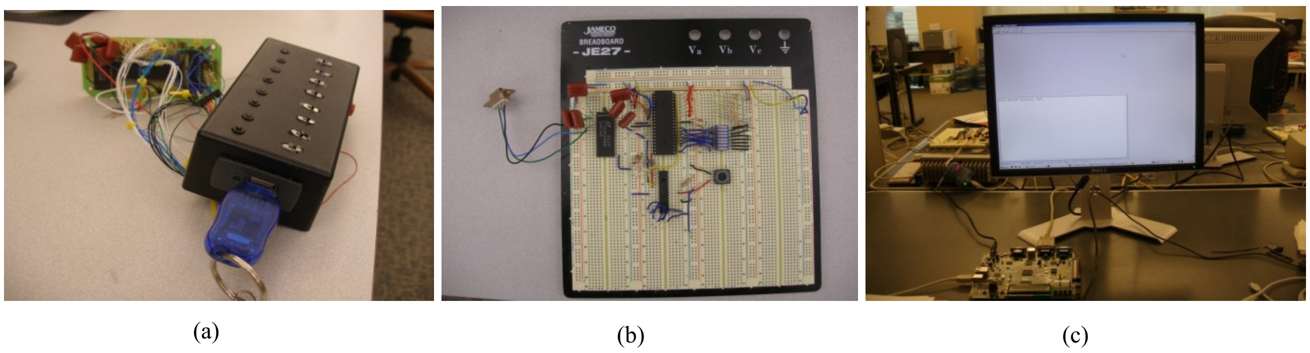
2 / 1000

SM_Clk = 0

SM_Clk

```
A2 -> 1  (4026 : 0.4.26)
A3 -> 1  (4026 : 0.4.26)
A6 -> 1  (4026 : 0.4.26)
A0 -> 1  (11130 : 0.11.130)
B0 -> 1  (12058 : 0.12.58)
B0 -> 0  (13061 : 0.13.61)
```

Figure 7: Virtual microcontroller implementations: (a) in a black-box, with internal AVR-microcontroller-based circuitry exposed, (b) on an AVR microcontroller breadboard, with input/output wires that can be connected to other circuits, and (c) on a Xilinx Spartan 3E FPGA using a serial connection to a PC to output to a serial terminal. All three can execute the same VµC program identically.



providing a standalone device with a simple user interface. An alternative implementation could include both the switches/LEDs plus input/output ports that could be connected to other devices and that could override the switches/LEDs, shown in Figure 7(b). We built an implementation on a Xilinx FPGA, shown in Figure 7(c), by emulating the VµC on a MicroBlaze soft-core processor. We built interface functions on top of the MicroBlaze's physical interrupt controller and timers to communicate with the physical hardware. We built another FPGA implementation, this time describing the VµC in synthesizable VHDL and then synthesizing a circuit onto the FPGA. The ISRs, timers, and UART were created as components that interfaced to the MIPS ISA core, and the FPGAs general purpose input/output. Each implementation required a few days to create. Of course, an instructor may not have to build the implementation from scratch as we did; previous implementations can be described or downloaded from the web.

To test whether the VµC could handle standard embedded systems lab assignments, we redesigned the microcontroller labs from the embedded systems courses at University of California, Riverside, and University of California, Irvine, which have been taught for over 10 years and are similar to numerous microcontroller courses worldwide. The labs introduce a student to basic embedded microcontroller programming concepts, using general purpose input and output, timer-based programming, state machine programming, and interfacing to various peripherals.

The first embedded programming "Hello World" lab involved blinking a light on and off. The code to blink a light on and off in virtual microcontroller code consisted of 16 assembly instructions. The second lab interfaces a microcontroller to seven-segment displays, involving writing to general-purpose outputs, and creating a simple delay loop. The third lab interfaces with a standard keypad by reading general-purpose inputs. The fourth lab introduces interrupts and interrupt service routines. The interrupts are introduced along with the virtual timers, and the students are asked to program a simple decimal counter using interrupts and the concepts used in the previous labs. The fifth lab introduces the serial protocol and interfacing to a microcontroller's UART. The students are asked to read from the serial port, and then output the input with a simple ROT13 encoding. Finally, the last lab brings combines the earlier concepts in design of a reaction timer game. For all of the labs, the input and output ports were sufficient to interface to all of the required external peripherals.

Each lab was redesigned and written in the VµC's assembly language and tested on the implemented platforms. Because the assembly file is also the executable format, the VµC's executable file was 10 times bigger than a traditional binary, due to using ASCII text characters. The VµC implementation internally translates the text file to a traditional binary to reduce internal storage and improve performance.

9. Conclusion

We presented a virtual microcontroller, a clean intuitive microcontroller that allows a beginning embedded programming student to concentrate on structured embedded programming while still learning important low-level resource concepts related to interrupts, timers, and UARTs. We implemented the VµC on several physical devices including an AVR microcontroller and an FPGA, and redesigned a complete introductory course set of labs for the VµC.

10. Acknowledgements

This work was supported in part by the National Science Foundation (CNS-0614957). We also thank Bailey Miller, Jonathan Basseri, and Andrew Becker for their work in developing the VuC simulator and physical prototypes.

References

[1] Benjamin, M., Kaeli, D., and Platcow, R. 2006. Experiences with the Blackfin architecture in an embedded systems lab.. WCAE '06
[2] Butler, J. and Brockman, J. Web-based Learning Tools on Microprocessor Fundamentals for a First-Year Engineering Course. 2003. American Society for Engineering Education.
[3] Celoxica. 2006. DK design suite. http://www.celoxica.com/products/dk/default.asp.
[4] Cottrell, S. and F. Vahid. A Logic Enabling Configuration by Non-Experts in Sensor Networks. HFC. 2005.
[5] DOS Box. http://www.dosbox.com
[6] eCOS. http://ecos.sourceware.org/
[7] Hennessy, J. and Patterson, D. Computer Architecture – A Quantitative Approach. Morgan Kaufman Publishers. 3rd edition. 1996
[8] Hodge, H. Hinton, H.S, and Lightner, M. Virtual Circuit Laboratory. ASEE. American Society for Engineering Education. 2000
[9] Levis, P. and Culler, D. 2002. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.* 36, 5 (Dec. 2002), 85-95.
[10] Ricks, K. G., Jackson, D. J., and Stapleton, W. A. 2005. An evaluation of the VME architecture for use in embedded systems education. SIGBED Rev. 2, 4 (Oct. 2005), 63-69.
[11] Smith, J. and Nair, R. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan-Kaufman Publishers. 2005.
[12] Stark, R., Schmid, J, and Borger, E. Java and the Virtual Machine- Definition, Verificartion, and Validation. 2001.
[13] VMWare. http://www.vmware.com/
[14] WindRiver Systems. http://www.windriver.com/
[15] Xen. http://www.xen.org