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.

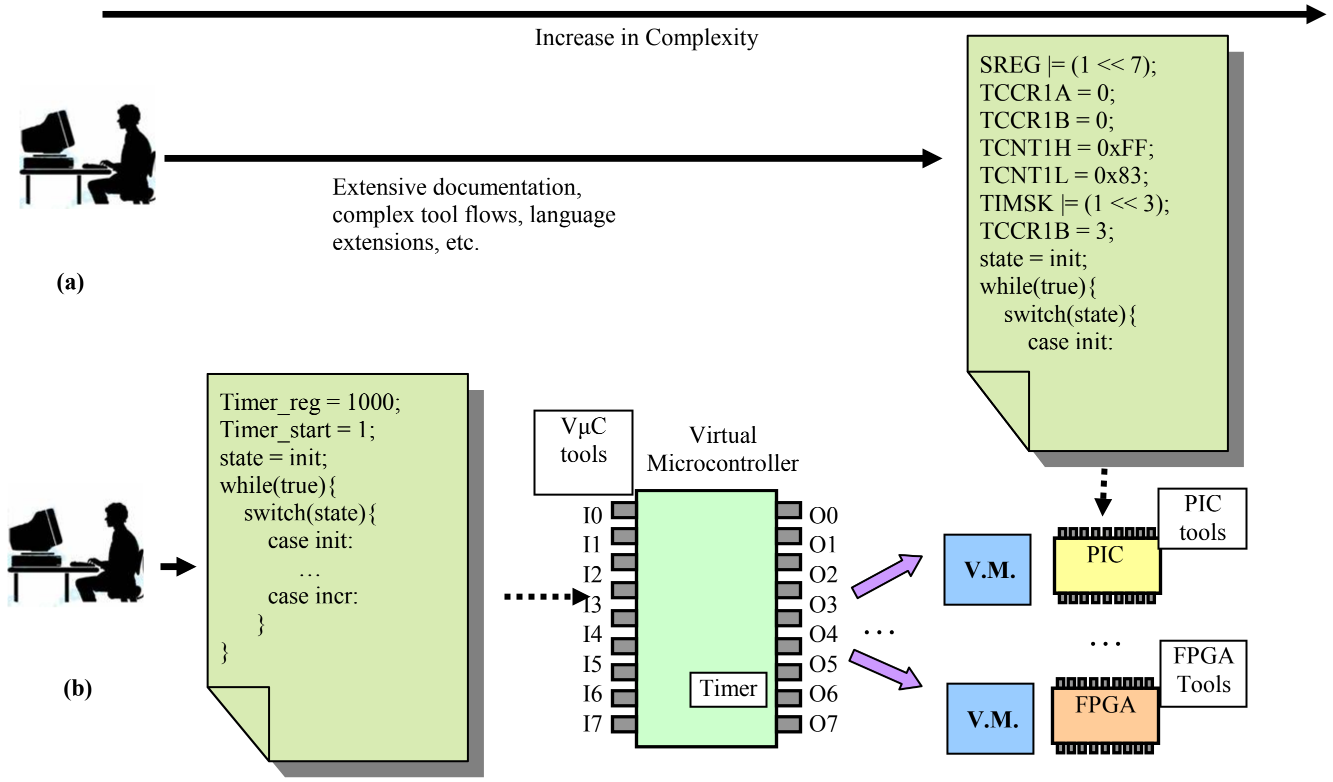
Two approaches are commonplace today for elevating the level of programming. A bottom-up approach first introduces low-level detailed programming, and then introduces higher-level concepts in a second course. While practical in the sense of teaching technical skills enabling physical implementation, 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
2. ADDI $1 $2 imm
3. ADDIU $1 $2 imm
4. AND $1 $2 $3
5. ANDI $1 $2 imm
6. BEQ $1 $2 [Label]
7. J [Label]
8. JR $1
9. LW $1 0($2)
10. NOOP
11. OR $1 $2 $3
12. ORI $1 $2 imm
13. ***RETI***
14. SLL $1 $2 $3
15. SLT $1 $2 $3
16. SW $1 0($2)
17. SUB $1 $2 $3
18. SUBI $1 $2 imm
19. XOR $1 $2 $3
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.

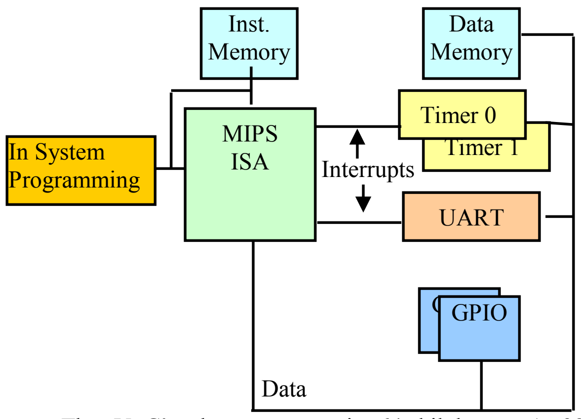
Learning the complete MIPS ISA might overwhelm students. We thus chose to use a twenty-instruction subset, shown in Figure 2, chosen as a representative mix of the entire MIPS ISA. Using the subset allows for easier learning, at the expense of larger code size and slower performance, which are less important 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.

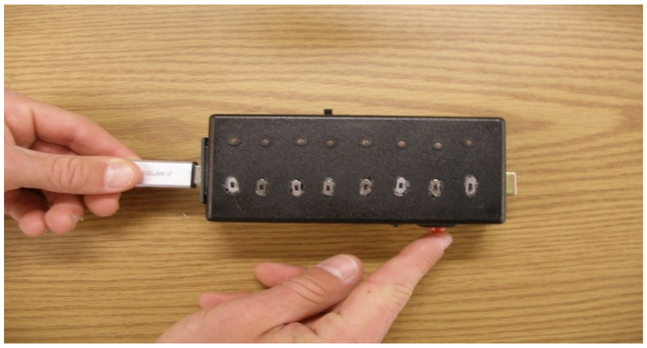
The VµC includes a UART (Universal Asynchronous Receiver/Transmitter), which allows a student to learn how to interface to serial devices, including a PC, for input, display, or debugging purposes. The UART can be programmed and configured using three intuitive memory-mapped registers, the *UART status 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

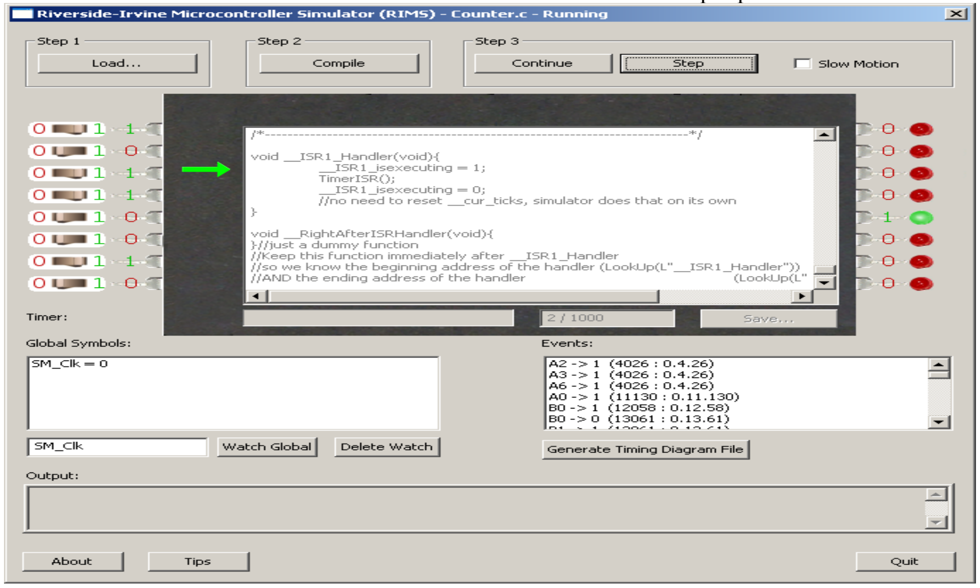
We have also developed a graphical VμC simulator. The simulator supports standard microcontroller simulator/debugger functions, such as steps, breakpoints, run for X simulated seconds, input/output value writes/reads, observation of internal registers including memory-mapped peripheral registers, etc. 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.



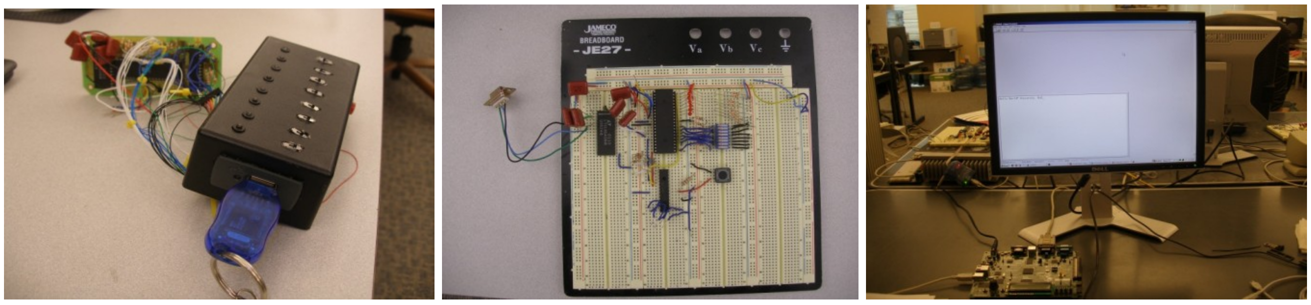
```
/*------------------------------------------------------------------*/

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"
```

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