INTEGRATING HARDWARE EXPERIENCES INTO A COMPUTER ARCHITECTURE CORE COURSE

Fred G. Martin
University of Massachusetts Lowell
Department of Computer Science
1 University Avenue
Lowell, MA 01854
fredm@cs.uml.edu

ABSTRACT

A core curriculum computer architecture course is designed with a significant hardware component. The course builds upon prior student coursework in logic design and organization, revisiting some of this material from a fresh perspective as it builds toward simulation-based presentation of central architecture topics. The course introduces an inexpensive, take-home lab kit, which students use for project work on their own PC. The lab kit supports introductory, hardware-based activities in digital logic and microprocessor programming. The goal of the work is to provide students with accessible, hands-on learning experiences as a component of a traditional computer architecture course, while not requiring the space, equipment, and personnel resources of a dedicated hardware laboratory. This paper presents the design of the course, including both hardware and simulation-based assignments, and observations after 4 iterations of the course.

1 INTRODUCTION

There is a clear trend toward increased use of simulated hardware and software systems for pedagogical purposes in computer science education. The reasons behind this are clear: it is easier for a student to build a virtual system than a physical one. This allows instructors to facilitate the design of more complex systems given an allotted time [6]. Also, students can gain experience interacting with systems for which physical hardware is not readily available. For example, students can create a software-based implementation of a pedagogical processor design, or write code for an obsolete but otherwise exemplary CPU chip.

Those of us who enjoy working with hardware are actively opposing this trend. Our motivations are two-fold. First, we appreciate the expressive power of designing and bringing new physical devices into the world, and we desire to pass along this passion to our students. Second, we believe that in order for students to deeply

understand computing—particularly, the organization and architecture topics—they simply must get their hands on actual hardware. These two aspects go hand in hand; we often see students get "bitten by the hardware bug" after they have had a rewarding experience with it.

This paper describes a core curriculum computer architecture course that has a significant hardware component. A take-home lab kit was developed for the course to support the hardware work. The paper is organized as follows. The remainder of this introduction describes the author's institutional context and educational goals in creating the course and its associated assignments. The following section presents the design of the kit. Next presented is the course design, including student assignments in detail. The whole course is discussed, in order to illustrate how the author's hardware-based material connects to simulation- and programming-based assignments. The paper ends with some concluding remarks.

1.1 Institutional Context

The author's institution has separate Computer Science and Electrical and Computer Engineering departments, which are housed in different colleges. Both departments have faculty whose interests straddle the boundaries between the disciplines (this author included). While the two departments have a cordial relationship, and there is some faculty collaboration, only a few students enroll in a dual major with both departments.

The Computer Science department (where this author is located) has about 20 faculty members. About half are senior faculty; of these, many have been with the department since its inception. Despite the existence of the separate ECE program in the university, these faculty gave CS department a strong practical bent, including the hiring of a several hardware-oriented junior faculty.

The Computer Science program has a mandatory three-semester sequence in architecture topics, with separate courses in Logic Design, Computer Organization, and Computer Architecture. As has been noted (e.g., [3]), given the reduction of architecture topics in the CC2001 curriculum to a requirement of just one semester [1], having three semesters for these topics is an enviable situation.

Nevertheless, the author was motivated to include a strong hardware component when he was assigned the Computer Architecture course. In fact, the Logic Design course, despite being in taught the ECE department, is one of the many that has evolved from being based on breadboards, chips, and wiring to a version that is now primarily taught using simulation tools. Further, our Computer Organization course is "hardware-free," as it is based on tools that simulate the MIPS processor.

Thus, students reach their junior year and the required Architecture course, with its core content being microprocessor design internals, having never directly programmed one. As observed by Ivanov [4], "Without sufficiently emphasizing hardware issues, students gain a one-sided view of Computer Science, never reaching a solid understanding of all the issues involved in hardware-software interaction, and the intimate interdependency between computer organization, operating systems, and programming languages." Clearly, our department needed a hands-on lab at some portion of the program, and the opportunity existed in the architecture course.

1.2 Additional Motivations

The author had several additional considerations in designing the course. Primarily, the hardware experience should provide a basis and context for the study of architecture topics—which will be encountered later in the same course, but would be treated using software simulation. As noted by Pilgrim [5], even small-scale integrated digital logic is new to students, and skipping up too many levels of abstraction may be "at the expense of a deeper understanding of the underlying principles."

The hardware activities should be at an introductory level, since students will likely have minimal prior hardware experience. Students who took our Logic Design course would have learned with simulation tools; transfer students would likely have similar backgrounds.

Aside from pedagogical goals, if students became excited and enthusiastic about the course content, the course would be considered a success. In this regard, the course was an opportunity to groom students for subsequent project and research work.

1.3 Constraints

When the course was developed, the department did not have laboratory space that would be suitable for establishing a traditional hardware lab. Further, it was not practical to modify the course schedule and credit allotment to support an additional meeting for laboratory work.

Therefore, a kit was designed that could be provided to students on loan for the duration of the semester. Most of our students have their own computers in their dorm rooms or homes that they regularly use for coursework. For those who did not, permission was obtained for them to connect the kit apparatus to our departmental lab computers.

To support as many students as possible, Java-based tools were developed. These tools allow students to download code to the kit microprocessor, and are functional on Windows, Mac OS X, and Linux-based computers.

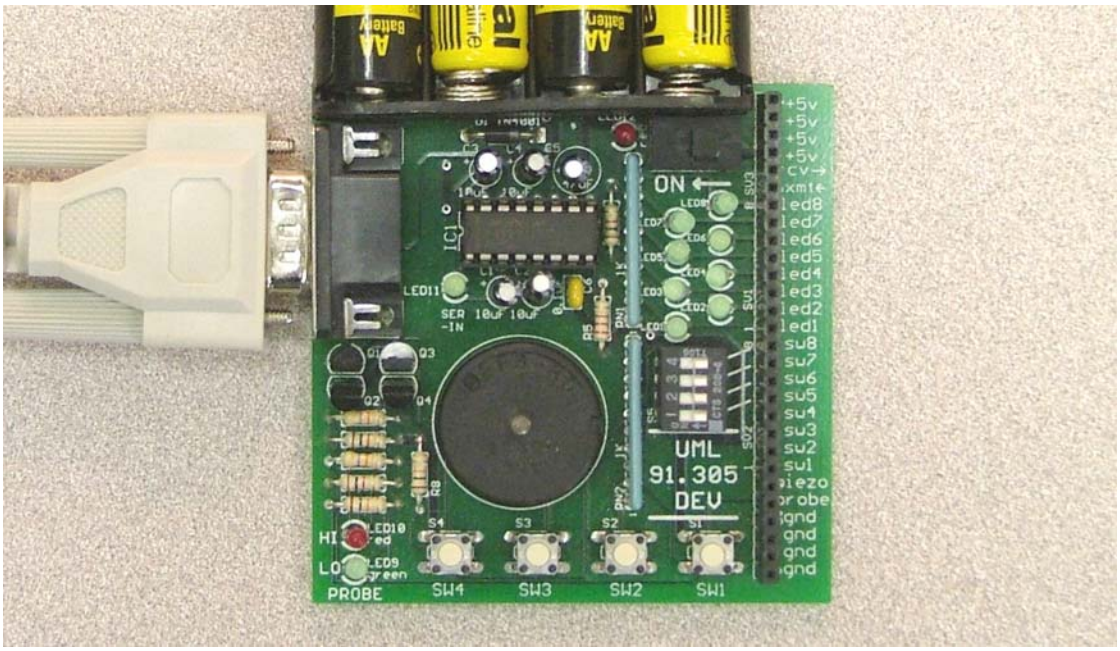

Figure 1: Custom "305 DEV" Digital Development Board

2 A PORTABLE, LOW-COST DIGITAL DESIGN KIT

Because of the constraint of developing a kit that students could bring home, it was not feasible to purchase commercial powered breadboards for digital design. These usually cost $200 or more and are bulky. Instead, the author designed a small printed circuit board (PCB) that included key circuits necessary for basic digital prototyping. The board is shown in Figure 1.

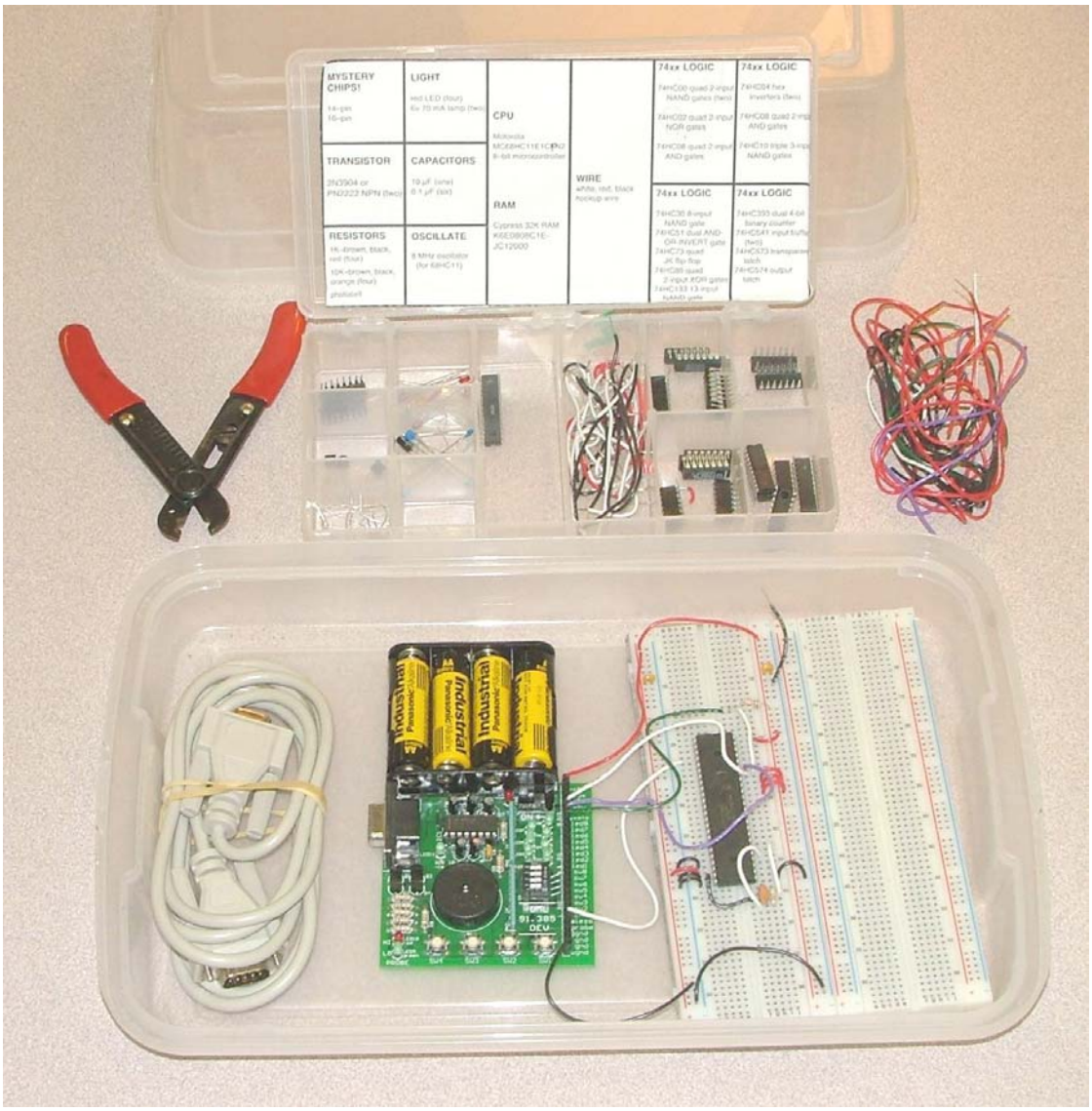

Figure 2: The Entire Digital Development Kit

Clockwise from the bottom left corner, the "305 DEV" board (named after the course number) includes: 4 pushbuttons for generating momentary digital signals, a high/low/open logic probe circuit, an RS232-TTL level shifter for serial interfacing, a +5v power supply based on 4 AA cells (with short-circuit protection), 8 LEDs for observing digital signals, 4 slide switches for generating fixed digital signals, and a piezo beeper element.

The development board itself and all of its components were purchased for about $20 per kit. The first group of students to use the board were taught how to solder and assembled their dev boards themselves. Subsequent classes used the boards soldered by the first class. This worked surprisingly well; even though few of the students had soldered before, most of the boards were assembled properly and have worked without problems.

Figure 2 shows the rest of the digital development kit. In addition to the development board, students were provided with a serial cable, two solderless breadboards, a wire cutter/stripper, hookup wire, and a parts kit. The parts kit included an assortment of 74HC00-series digital logic chips, resistors, capacitors, and LEDs, and a Motorola 68HC11 microprocessor (subsequently abbreviated, "HC11"). In the

photograph, the microprocessor is shown mounted on the breadboard next to the development board. The entire kit was packaged in a small shoebox-sized plastic storage container.

The entire kit, including the 305 DEV board, cost $50 each. At the time when the course was developed, our class enrollments were relatively high (up to 35 students per class). We purchased 40 kits for $2000.

Maintenance costs have been small because the initial materials purchase was done at a time of peak enrollment. Presently the course has about 20 students per semester. If enrollment had stayed at its peak, we would need to budget $100 to $200 per semester for replenishment. One kit has been lost in 4 semesters of use.

3 COURSE DESIGN

The course was developed for a 14-week semester with 6 weeks of content based on 5 hardware project labs. The author developed supporting material for these labs. The labs are: Digital Logic Lab, State Machine Lab, HC11 Boot Lab, HC11 Beep Lab, and HC11 Address Decoding Lab.

In the remainder of the semester, the course focused on what would be considered the traditional architecture topics: the internal design of a CPU, including a thorough discussion of pipelining, and cache memory. The course adopted the CS/APP text from Bryant and O'Hallaron [2], which takes a practical approach in a combined treatment of organization and architecture topics. In addition to being written in a straightforward matter, the text includes an excellent set of software-based laboratories for student work. The course adopted three of these: the Binary Bomb Lab, Architecture Lab, and Performance Lab.

Table 1 shows an overview of the semester. The remainder of this section presents the student assignments, including both the author's hardware assignments and the ones adopted from the CS/APP text.

3.1 Digital Logic Lab

The Digital Logic Lab is the students' first experience with the lab kit. No assumptions are made about students' prior work with digital logic, so the lab introduces solderless breadboards, wiring techniques including power distribution, and how to use the 305 DEV board to generate and observe digital signals.

The highlight of the lab is the "Mystery Chip" assignment. Students are each given one 14-pin and one 16-pin DIP, from the 74HCxx series, which have had their part numbers sanded off. The students' task is to identify the chips. (All students have the same 14- and 16-pin part.)

Students find this portion of the assignment quite interesting. The 14-pin chip is fairly easy to figure out—it's a package of several gates of the same type—but the 16-pin chip is a decoder, and until its 3 select lines are put into the proper state, its outputs are disabled.

Students are instructed to first wire up a chip's expected power and ground, and then use the 305 DEV board's logic probe circuit (which distinguishes between high, low, and floating) to characterize the pins. The Mystery Chip problem succeeds in introducing students to the concept of a floating signal (e.g., an input line or an output that is tri-stated). This is a pretty subtle concept all around, and one that is nearly always lost in simulation-based work.

Table 1: Overview of Course Design

Week	Lecture Topics	Assignments
1–2	Digital logic; voltage, current, and simplified transistor model; edges vs. levels; the 74xx chip series	**Digital Logic Lab:** prototyping, wiring NAND gates & counters, using transistors, "Mystery Chip" lab
2–3	Design of state machines including implementation; DeMorgan's law; logic minimization	**State Machine Lab:** Control a robot to make it follow a black line
4	ISA of practical 8-bit CPU; condition code flags; number formats; memory map; µcontroller register interface	**HC11 Boot Lab:** Build µprocessor circuit and download test program
5	Timing units: µsec, millisec & sec, machine clock & cycles/instruction, loops & total cycle counts	**HC11 Beep Lab:** Write machine code to generate particular pitches
6	Addressing modes; memory bus incl. addr, data, enable; mem-mapped I/O; timing diagrams; stack operations	**HC11 Address Decoding Lab:** Build combo logic to decode addr bus
7, 8	x86 ISA; C-language procedure call stack; use of the frame ptr; other C-to-asm structures	**CS/APP "Bomblab":** reverse-engineer x86 binary application
9, 10	6-stage execution model; HCL code implementation of CPU; Y86 arch	**CS/APP "Archlab A+B":** coding for and extending the "Y86" processor
11, 12	Long vs. short pipeline, data & control hazards, branch prediction, CPI vs. IPC, superscalar performance	**CS/APP "Archlab C":** implementing pipelining into "Y86" processor
13, 14	Direct-mapped vs. set-associative; write-thru vs. write-back; temporal locality, spatial locality, and stride	**CS/APP "Perflab":** studying effect of stride on real-world cache performance

Hardware Labs **Simulation** **Real Code**

3.2 State Machine Lab

The state machine lab presents the traditional approach to implementing a finite state machine, using state transition diagrams, assignment of unique values to states, transition rules based on input values, and implementation using logic equations. In this lab, students build a state machine controller for a mobile robot.

Most students in the course (all except those who are transfer students) will have taken our institution's Logic Design course, where this material does receive thorough treatment. However, as our Logic Design course is based on the use of simulation tools, so while students will have *designed* a state machine in the past, they will not have *built* one.

In this lab, students are given the template for a functional design (e.g., using a 74HC574 latch to hold the state bits) and a complete state transition diagram. They are

required to write the logic equations that generate the next state value and 2 bits of motor outputs (based on the current state and 1 bit of sensor input from the robot), design an implementation of these equations using the chips in their kits, and then build their design. Finally, they test their design by plugging it into the robot, and seeing whether or not the robot is able to follow the line. Students' work is checked at the logic equation level and circuit level (ideally, before they perform the wiring).

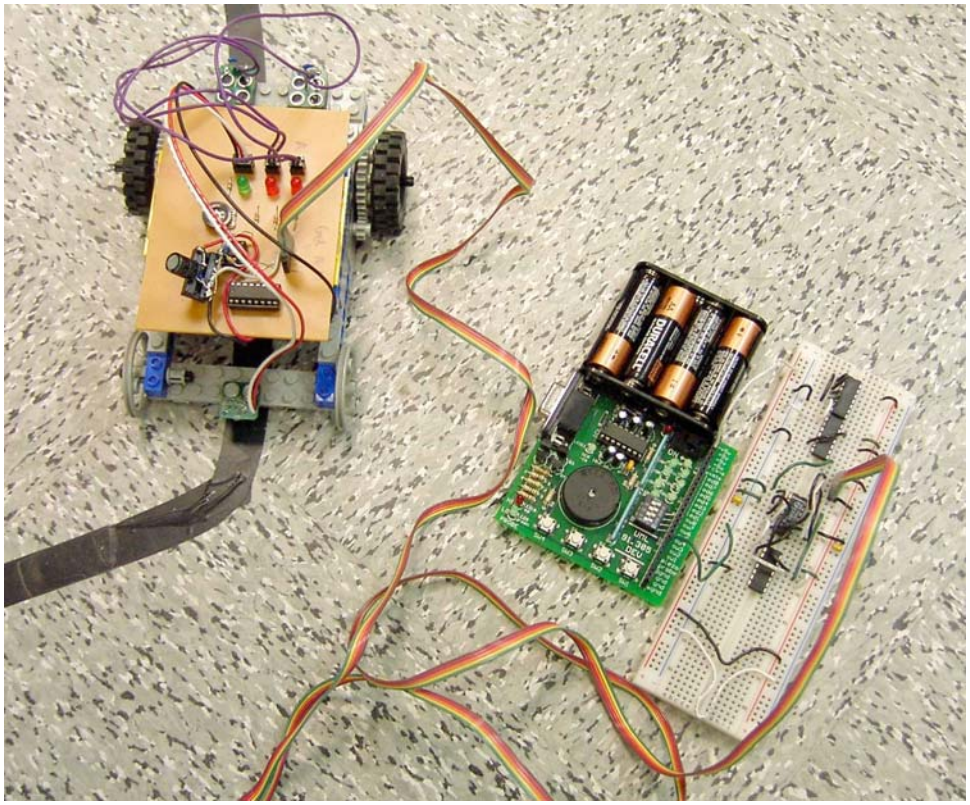

Figure 3: Photograph of Line-Following Robot with State Machine Controller

Figure 3 is a photograph of the actual robot. It is connected via a 5-conductor ribbon cable to the student's implementation. The robot accepts 2 signals: "A," which controls the left motor, and "B," which controls the right motor. The robot produces a signal "L," which is true when the robot is on the line. The other two wires are power and ground.

To assist with the implementation, students are given a state transition solution that uses four states. Thus, the current-state information is represented with 2 bits. The equations to generate the next-state and motor control signals require only 3 bits of input each (current state plus L); this makes them fairly simple. They minimize well and the final implementation can be done with just one latch chip and two chips of gates. The physical construction (and debugging) of the circuit does require time, but not unreasonably so.

In practice, this was a challenging lab for most of the students. A number wanted to construct just a simulation-based design using the software tool they had learned in the Logic Design course.

Most students did invest the time required to build the circuit and get it checked off during TA or faculty office hours. Students who brought in non-working designs had the opportunity to learn the most. They were generally willing to spend up to an

hour working with the TA to help debug their design. The most interesting cases were ones where an error in the equations or minimization led to an incorrect implementation. Sometimes, pesky wiring errors were solved with the old standby—rip-up and retry (after the design has been checked over).

Reaction to seeing the robot work varied. Some were ecstatic, and just wanted to watch the robot drive around for a little while. Others were more relieved that the ordeal was over.

3.3 68HC11 Microprocessor Labs

The 68HC11 microprocessor was selected for the microprocessor labs. While the device is an old design, it has a straightforward von Neumann architecture and an external memory bus. I considered and rejected the Microchip PIC. While the series is popular, inexpensive, and easy to use, it has an unusual architecture (e.g., 14-bit instruction word length) and no external memory bus. Despite (or because of?) the HC11's age, it is an excellent chip for pedagogical purposes. Indeed, it still has commercial demand, and has recently been updated with a lead-free version.

Three labs were developed using the Motorola 68HC11 microprocessor: a "boot" lab (building the CPU circuit and demonstrating that you can download code to it), a "beep" lab (generating square wave tones of particular frequencies by cycle-counting), and an address-decoding lab. These are discussed in order.

3.3.1 Boot Lab

In this lab, students build the circuit to operate the HC11 chip, and set up software on their PC to download code to. Verification is accomplished by downloading a test program that flips a bit in one of the HC11's memory-mapped I/O ports, thereby changing an LED on the 305 DEV board from green to red.

Despite the apparently simplicity of this task description, a lot of background knowledge is necessary to understand what is truly going on. Parallel to the lab, lectures introduce the memory space of the HC11 device, including the 512 bytes of internal RAM—the place to which the test program is loaded. The programmer's model of registers is introduced, as is the idea of memory-mapped I/O registers. The actual test program is presented: all three lines of it. This code puts the value 0x10 into register A, stores that value to the "port A" I/O register, and loops endlessly:

```
        ldaa #0x10
        staa 0x1000
loop:   jmp loop
```

Among other issues in understanding, the fact that there are two different types of register named A (the internal machine register and the I/O register) is of much consternation to students.

Finally in this lab students configure their personal computers with Java and serial drivers, and run a small loader program (developed by the author) to stream the boot code out their serial port. The 68HC11 is operated in a serial bootstrap mode, in which it reads 512 bytes from the serial port, loading this data into its internal RAM, after which it jumps and begins executing code at location 0.

3.3.2 Beep Lab

In this lab, students are given an assembly-coded program that produces a square wave on one of the 68HC11 output pins. The square wave's frequency is in the audio range, so when the signal is wired to the piezo element on the 305 DEV board, it produces a strong, continuous beep.

We analyze the program to determine the particular frequency of the beep. This leads into a discussion of timing units, frequency, and the fact that different HC11 instructions take numbers of instruction cycles to execute.

The program itself is a loop around two half-wave countdown delays:

```
          ldx #0x1000             ; setup X to point at PORTA
          clra                    ; 0 into accum A
loop:     bset 0,x,#0b00010000    ; set bit 4 of PORTA
declp1:   deca                    ; A = A - 1
          bne declp1              ; if not 0, keep at it
          bclr 0,x,#0b00010000    ; clear bit 4
declp2:   deca                    ; 1st time here, accum A=0
          bne declp2
          bra loop
```

To count the number of iterations in one of the delay loops, students must understand numeric representations in an 8-bit register (i.e., unsigned integer).

The analysis of the starter program is presented in class. In the lab, students work the problem in the other direction: given a target frequency, how must the program be modified in order to generate that frequency?

This is a simple problem, but it gets a lot of mileage. My main concern is to provide some concrete basis for an understanding of period and frequency. Later in class, we discuss contemporary CPUs and memory busses, with frequencies in the GHz ranges. I want students to have a referent point for these conversations, and this lab provides practical experience with μsec/MHz and msec/KHz units.

The lab also provides clear evidence that one machine cycle ≠ one instruction. This is explained in subsequent work on instruction execution stages, and ultimately, how superscalar processors can indeed achieve multiple instructions per cycle.

This lab is the most exciting for students in terms of having fun with hardware. They clearly realize that their code is being downloaded from their PC to the chip sitting on the tabletop, and that the chip is executing their program. In a given class, few if any students have done embedded programming before, and this lab provides an easy and enjoyable introduction to it.

3.3.3 Address Decoding Lab

In this lab, students are shown how addresses and data busses work, and how combinational logic can be used to decode a particular address (or range of addresses) to allow the HC11 processor to read or write to an individual 8-bit latch.

The assignment includes an implementation component and a written-design component. In the implementation task, students must design combinational logic to decode 4 address bits along with the HC11's read/write line and E clock line. The decoded signal is routed to the clock input of an 8-bit latch. The latch's outputs are wired to LEDs. Then, students run a program that writes a pattern of bits to the address in question; if their circuit is implemented correctly, they are rewarded with a light-chasing LED pattern.

The lab is difficult to debug; students are not given the tools necessary to properly debug a non-working circuit. Usually, rip-up and retry works, but that is not as satisfying as knowing why the circuit was failing in the first place. The lab is most successful when students have gotten excited about the concepts and are willing to invest some time into producing a working circuit.

In the written design, students design a circuit to map an 8K RAM chip into the memory space of the HC11 processor. The circuit is not constructed.

3.4 CS/APP Binary Bomb Lab

After the Address Decoding lab, we pack away the hardware—but continue to make references to the material during this portion of course.

Our next assignment, the "Bomb Lab," was developed by Bryant and O'Hallaron and is presented in their text [2]. Students are given an x86 executable, compiled for Linux. When run, the program reads in a line of text from stdin. The line of text must match what the program is expecting; otherwise, the executable triggers an "explosion" (i.e., it sends a message to a server and then quits).

Six lines of input must be fed to the program to defuse it; each stage is progressively more complex and exercises different aspects of C-language control and data structures. Furthermore, every student gets his or her own personalized bomb, in which each of the six stages is built from a choice of several related but different possibilities. We use GDB (the command line version) to single-step and debug the bomb executables; this is typically students' first use of this tool.

Lectures that accompany this lab introduce the Pentium/x86 architecture and how the C language is translated into x86 machine code. The C-to-assembly topics include function calls (including stack frames and argument passing), how arrays are implemented, and various control structures. In our Computer Organization class, students have encountered a lot of these same assembly language topics, but have not specifically seen C-to-assembly translation.

Students absolutely love the Bomb Lab—it's positively addicting. More than a few students have reported to me "I should have been doing something else, but I kept working on my bomb."

3.5 CS/APP Architecture Lab

At this point in our degree program, students have seen at least three instruction set architectures: the HC11 and x86, which are CISC designs, and the MIPS design, which is presented in the organization course.

Here, my central presentation is based on Bryant and O'Hallaron's "Y86" processor implementation. This is a pedagogical processor design that is based on the Intel x86 ISA. The Y86 ISA is simplified, containing fewer addressing modes and 32-bit registers exclusively.

Bryant and O'Hallaron provide a comprehensive set of software simulation tools (written in C for Linux) for student work with the Y86 design. The tools include an assembler and a graphical simulator, which shows progress throughout all stages as an instruction is executed.

The Y86 design itself is implemented using "HCL," a custom, C-like hardware description language. Bryant and O'Hallaron provide reference implementations for non-pipelined ("SEQ") and pipelined versions ("PIPE") of their design.

The student lab contains three parts:

- In part A, students write several Y86 assembly language programs and test them using the graphical simulator. Students are encouraged to use GCC, compiling reference solutions that are provided in C, and then hand-translating the resulting x86 assembly into corresponding Y86 code.
- In part B, students implement two new instructions in the SEQ processor by adding the appropriate HCL code to the processor. Tools are provided for recompiling the processor simulator, and students use an automated test suite to verify their solution's correctness.
- In Part C, students optimize the performance of a memory-copy routine by modifying the routine's Y86 code and the design of the PIPE processor. The solution involves implementing branch prediction and re-writing the conditional branches in the assembly code. A test suite is provided to measure code performance.

Bryant and O'Hallaron deliver this to their students as a single, huge assignment. I split the assignment into two pieces: Parts A and B and then Part C. It takes us a month to get through it, but this is the meat of the course, and there is ample opportunity to make connections between abstract concepts in processor design and their implementation using HCL code in the Y86 design.

In class, I make explicit connections between the earlier hardware work and the Y86 design. When talking about the CPU registers (which are 32 bits wide), I will say, "This is just like four HC574 chips." I also have students convert simple Y86 code expressions into their logic gate equivalents, to make sure they understand that these are interchangeable representations.

Overall, the Y86 package is extremely well-done, and while the material is complex and challenging, many of my students are able to grasp it. While I do not have conclusive evidence, I believe that the earlier hardware work does motivate and ground the students' understanding of the core architecture content.

3.6 CS/APP Performance Lab

Bryant and O'Hallaron's "Performance Lab" is assigned during the discussion of cache memory systems. In this lab, students optimize the performance of image-processing algorithms by modifying the chunking and stride of memory accesses. The lab provides practical evidence that the way that you write code matters, and helps students translate abstract discussions of caching into practice.

Students are also given a paper-based assignment in which they simulate the operation of tiny direct-mapped and set-associative caches.

4 DISCUSSION AND CONCLUSIONS

The course delivers a blend of topics from logic design, computer organization, and architecture. I often note to students that these are fluid boundaries and that it can be difficult to talk from one perspective without involving the others.

While our students have taken previous semesters of logic and organization, the course presents these topics from a fresh perspective. Students get a more in-depth hardware experience than they do from our Logic Design course. In the organization topics, students see new ISAs (the HC11 and x86) and new ideas (e.g., showing how GCC compiles standard C idioms).

While the course thus does build upon students' prior knowledge, the course content is also modular. Institutions with a single-semester organization/architecture course could also benefit from this approach. Individual labs could readily be integrated into such courses.

The central research question is how the hardware work provides a basis of understanding the core architecture concepts (e.g., processor implementation; cache design). If we did not devote the first 6 weeks of the semester in revisiting material from a hardware perspective, could we get deeper into these topics? The answer might be yes, but I would find it difficult to run this course that way—given how little our students actually know about hardware. On the contrary: in revising the course, I would wish to include more hardware material, such as implementing a processor on an FPGA.

Perhaps the most important measure of success is what students do after the course. A small but crucial group of students continue with our elective undergraduate robotics courses. Of these, a significant number have gone on to graduate school and industry in areas that involve hardware work.

REFERENCES

[1] Association for Computing Machinery and Institute of Electrical and Electronics Engineers Computer Science Joint Task Force, *Computing Curricula 2001*.

[2] R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, Prentice-Hall, 2003. http://csapp.cs.cmu.edu/.

[3] M. Hoffman, "An FPGA-based digital logic lab for computer organization and architecture," *JCSC*, 19(5), pp. 214–227, 2004.

[4] L. Ivanov, "A hardware lab for the computer organization course at small colleges," *JCSC*, 19(2), pp. 185–190, 2003.

[5] R. A. Pilgrim, "Design and construction of the Very Simple Computer (VSC): a laboratory project for an undergraduate computer architecture course," *SIGCSE '93*, pp. 151–154, 1993.

[6] G. S. Wolffe, W. Yurcik, H. Osborne, M. Holliday, "Teaching computer organization/architecture with limited resources using simulators," *SIGCSE '02*, pp. 176–180, 2002.

ACKNOWLEDGMENTS

I would like to thank my students for their candor in engaging with the course material as I developed the course. I would also like to thank Karthik Ramanathan, who worked with me as TA for 3 semesters of the course. Adam Elbirt made helpful comments on a draft of this paper.

OBTAINING THE AUTHOR'S MATERIALS

The 305 DEV board (including electrical schematic, PCB layout, parts listing, assembly manual, and user guide), the laboratory assignments, line-following robot design, and Java-based HC11 tools are available for educational use. Please contact the author for details.