Helping Students Understand the Datapath with Simulators and Crazy Models

Michael B. Gousie
Department of Math & Computer Science
Wheaton College
26 E. Main Street
Norton, MA 02766
mgousie@wheatoncollege.edu

James D. Teresco
Department of Computer Science
The College of Saint Rose
432 Western Ave.
Albany, NY 12203
terescoj@strose.edu

ABSTRACT

Undergraduate computer science programs at many small colleges often include only one course focused on hardware. Many important concepts are covered in such a course, including the basics of computer architecture. By the end of such a course, students should have a good understanding of how a binary machine instruction is executed in hardware. Unfortunately, even a simplified diagram of a datapath is often difficult for students to master. We present two approaches that use lab exercises to help to address this problem. In one, students build a working model of the datapath out of ordinary materials; in the other, a software simulator is designed and implemented. These approaches are described and their merits discussed.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; C.0 [**Computer Systems Organization**]: General

Keywords

Computer organization pedagogy, laboratory assignments, modeling computer architecture, simulating computer architecture

1. INTRODUCTION

Many small colleges require and/or offer only one course focused on hardware/computer organization. As such, this course could be the only place students will be exposed to a variety of important topics from circuits and binary representations to assembly language programming. Ultimately, we would like students in such a course to understand how a circuit can be constructed that is capable of executing a machine language program for a (relatively) simple instruction set architecture. This means being able to look at a complex diagram of the datapath and control of an architecture implementation and understand how it can possibly execute machine language programs.

How can we best maximize student understanding of this fundamental but challenging concept, working within the constraints of a semester course? We can present diagrams in class, examine individual components, and show how instructions work their way through the architecture. Students often grasp the concepts abstractly from this approach. However, they are not gaining concrete understanding that would allow them, for example, to be able to complete an exam question that asks what bits are found on each line given a particular binary instruction, or to be able to augment an architecture to implement a new instruction.

Staring at a diagram and/or watching a simulation run can aid in understanding, but not to the level that "getting your hands dirty" and examining the underlying details would permit. Constructing a circuit (*e.g.*, as in [10]) on breadboards, using a logic simulator or even with a hardware definition language, would certainly qualify as getting your hands dirty, but any of these is likely to be too large a project for our context. In the era where many of the instruction set architectures were heavily microcoded, a microcode design project (*e.g.*, [6]) could serve this purpose well. However, many architectures studied in classes today (we use MIPS) do not lend themselves well to a microcoding approach.

We present two approaches to improving the teaching and the student understanding of the single-cycle MIPS datapath, a summary of which is found in Section 2, and described in detail in Patterson & Hennessy [8]. In each case, one or more closed lab sessions are used to provide additional student hands-on experience compared to lecture-only classes. In Gousie's course, developed at Wheaton College, students are given a project to create a simulation of the single-cycle MIPS datapath. The simulation must show the movement of data through the various parts of the hardware for (ideally) an R-format instruction. The novel idea is that the simulation consists of a working model made out of any material *except* the usual datapath diagram(s). Electronic simulations are acceptable if they are sufficiently unique; that is, different from commonly available software.

In Teresco's course, which has been taught at Mount Holyoke College, Rensselaer Polytechnic Institute, and Siena College, students develop their own software simulator for the single-cycle MIPS datapath. Previous class and lab work in this course focuses on both digital logic and MIPS assembly language programming leading up to a unit on datapath and control that brings these two extremes together. The goal is to ensure that students understand how the components used in the example datapath can be constructed using the building blocks and techniques they have used to construct similar components using breadboards and/or a logic simulator (such as Logisim [3]). They should also be able to see that while the datapath and control we study only implements a subset of the MIPS instruction set, they understand how they would implement remaining instructions. By developing their own simulator, not just to simulate the effect of MIPS instructions (as done in tools such as [5, 11]), but to simulate them by modeling the underlying datapath and control, they can gain a much deeper insight into how the datapath and control components work together to achieve the desired effect for each instruction.

2. THE SINGLE-CYCLE MIPS DATAPATH

Patterson & Hennessy [8] present the single-cycle datapath (Figure 1) for a subset of 9 MIPS instructions as their introduction to an implementation scheme for a MIPS processor[1]. This representative set of instructions includes load (`lw`) and store (`sw`), 5 arithmetic and logical instructions (`add`, `sub`, `and`, `or`, and `slt`), and a conditional (`beq`) and an unconditional (`j`) branch. This includes 5 R-format instructions, which take the form

`op`	`rs`	`rt`	`rd`	`shmat`	`funct`
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

3 I-format instructions, which take the form

`op`	`rs`	`rt`	`address`
6 bits	5 bits	5 bits	16 bits

and one J-format instruction, which takes the form

`op`	`address`
6 bits	26 bits

Instructions outside the subset require similar implementation techniques and are omitted for simplicity. Some of these, such as immediate mode arithmetic operators and bit shifts, are later considered on homework or exam questions that ask students to show how they would augment the architecture to implement these instructions.

3. TWO WAYS TO AID UNDERSTANDING OF THE DATAPATH

Of course, there are many approaches to presenting the datapath. While students have seen and should be able to understand every component in the diagram in Figure 1 by the time they encounter it, its complexity is intimidating if not overwhelming for many students. In our experience, students have tremendous difficulty understanding the way data flows from an instruction through the various hardware devices to finally achieve a (partial) result. We have found that building up the datapath in parts, as is done in [8] is not effective, nor is tracing the datapath in lecture. Nevertheless, a Google search will show that this is the approach that is taken by many instructors. Other authors of similar textbooks include datapath simulators, such as [7]. In Gousie's

[1]This figure was published in [8] on Page 329, Copyright Elsevier (2009). Used with permission.

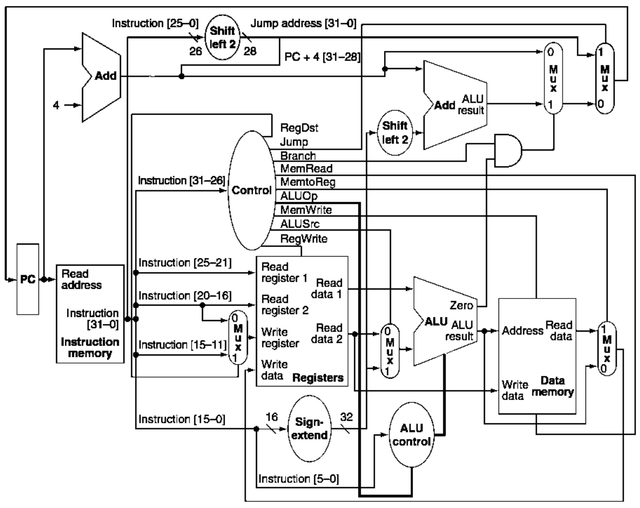


Figure 1: The single-cycle MIPS datapath of [8].

course, the Knob & Switch computer [2] has been used for many years; students seem to understand the broader workings of the hardware but then still can not show the value of individual bits on a given line in a comprehensive datapath diagram. What follows is not meant to supplant current best practices, but rather to add a hands-on layer to augment what is currently being taught.

3.1 Computer Organization Model Expo

Gousie's course includes a lab that attempts to teach the datapath using a new method called the Comp Org *Crazy* Model Expo. The intent of this lab is to have student groups present a working model showing how data flows through the CPU for a given R-format instruction.

3.1.1 Model Expo Rules

Students are given an assignment with the following parameters:

1. Students can form groups of up to three.

2. Given an R-format instruction, the group must build a model simulating the flow of data through the CPU, using whatever material that is convenient. Normally, this means that the model will be made of physical materials. Crazy ideas are encouraged, as long as the resulting model adequately shows the datapath.

3. A formal poster must be created that explains the workings of the model in the context of one of the diagrams from Patterson & Hennessy.

4. The group must present the model in no more than ten minutes at the Comp Org *Crazy* Model Expo. The event takes place during one of the regularly scheduled lab sessions.

Before embarking on the project, students must gain approval of the topic from the instructor; that is, which R-format instruction they are going to simulate so that multiple groups do not simulate the same instruction. Students can also amend the rules somewhat through negotiation; for

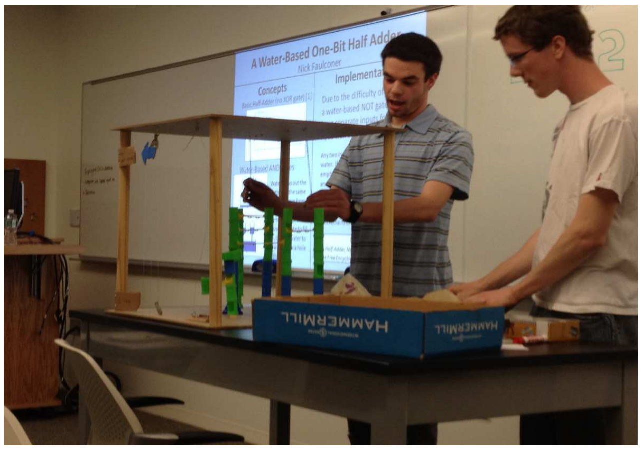


Figure 2: The Rube Goldberg contraption that simulates the jump instruction.

example, one group used Alice for the simulation, which obviously is not a physical model, and another group used a J-format instruction.

The idea behind this project is that students must think carefully about how the datapath actually works so that they can translate their understanding to some sort of physical device that models how the electrons (bits) are moving through the hardware. By leaving the type of materials completely open (indeed, crazy is encouraged), and stressing that this should be a fun and not arduous exercise (as well as promising pizza during the actual lab), most students seem to take on the challenge with aplomb.

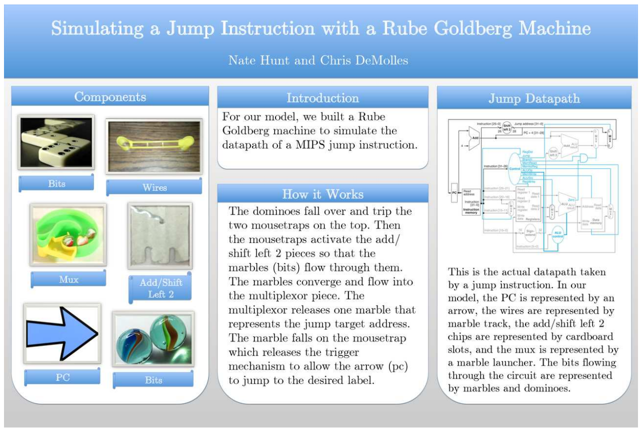
The class during the spring of 2012 had 23 students, divided into 10 groups. Many groups simulated the datapath in somewhat predictable ways. For example, several groups used water (similar to [1]) or marbles and tubing to simulate data flowing through components. One group built a small cardboard town with roads; the flow of data was simulated by cars driving through the town. Three of the more successful and original models/presentations are described in the following sections.

3.1.2 *Model One: Rube Goldberg Device*

In this model, a working machine made of dowels, string, pulleys, and other assorted parts (see Figure 2), showed how the MIPS jump instruction (`j`) works. The user triggers the Instruction Decode phase by touching the first of a series of dominoes; these fall until the shift left component is hit. Marbles represent a subsequent portion of the datapath; these flow until an arrow representing the instruction address is moved from its initial position signifying the original instruction to its new position. This, then, completes the instruction. Figure 3 shows the poster that accompanied the project.

3.1.3 *Model Two: R-Format Carnage*

Alice [9] was used as a way to quickly build a model of the datapath. The various hardware parts were modeled as buildings in a city. A suite of avatars represent the portions of instructions or data moving about the city between various buildings. In this case, the R-format instructions were modeled. Running the simulation for the MIPS instruction `add $t0, $t1, $t2` results in the actions shown



Simulating a Jump Instruction with a Rube Goldberg Machine

Nate Hunt and Chris DeMolles

Components

Introduction

For our model, we built a Rube Goldberg machine to simulate the datapath of a MIPS jump instruction.

How it Works

The dominoes fall over and trip the two mousetraps on the top. Then the mousetraps activate the add/shift left 2 pieces so that the marbles (bits) flow through them. The marbles converge and flow into the multiplexor piece. The multiplexor releases one marble that represents the jump target address. The marble falls on the mousetrap which releases the trigger mechanism to allow the arrow (pc) to jump to the desired label.

Jump Datapath

This is the actual datapath taken by a jump instruction. In our model, the PC is represented by an arrow, the wires are represented by marble track, the add/shift left 2 chips are represented by cardboard slots, and the mux is represented by a marble launcher. The bits flowing through the circuit are represented by marbles and dominoes.

Figure 3: Poster that explains the workings of the Rube Goldberg contraption.

below. Only the data portions are covered here; see Figure 4 for information on all parts of the datapath.

1. An airplane is sent to City Hall (instruction memory) to fetch the instruction.

2. The 32-bit instruction is split into its constituent parts. Two Eskimos represent the address of the two read registers (`$t1` and `$t2`), Prince Charming represents the address of the write register (`$t0`), a toy soldier represents the funct code, and a teacher represents the opcode. Parts that are not used in the instruction (for example, the shamt code) die off.

3. The two Eskimos (`$t1` and `$t2`) move to the register file (a townhouse) and transform into a shark and penguin (two data values).

4. The shark and penguin (two data values) move to the ALU (a farmhouse), where the sum is computed and stored in a knight.

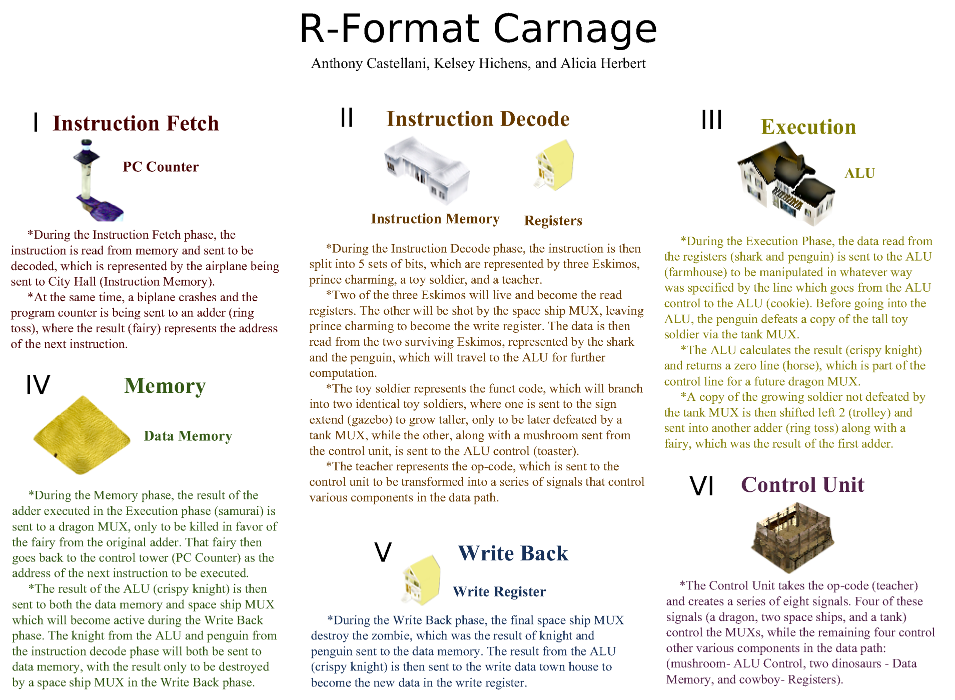
5. The knight moves to the register file (townhouse) and its data is written into the write register addressed by Prince Charming (`$t0`; see Step 1) completing the instruction.

Note that the above describes only the movement of data; the control lines are also modeled, and work similarly. When the program runs, many avatars, representing the data and the control signals, are moving simultaneously, giving the impression that there is chaos. However, the instructions work themselves out as expected.

Although this implementation may seem somewhat silly, the students working on the project had fun, as did those watching the demo. More importantly, the students must have a good understanding of the underlying mechanism of the datapath in order to implement the program.

3.1.4 *Model Three: Sonification of the Datapath*

In this project, the flow of binary instructions through the datapath is represented as short duration "musical" structures by means of a simple parameter mapping scheme. The datapath is divided into segments that roughly correspond to the stages of a pipeline. Each segment is divided into n



R-Format Carnage

Anthony Castellani, Kelsey Hichens, and Alicia Herbert

I Instruction Fetch

PC Counter

*During the Instruction Fetch phase, the instruction is read from memory and sent to be decoded, which is represented by the airplane being sent to City Hall (Instruction Memory).

*At the same time, a biplane crashes and the program counter is being sent to an adder (ring toss), where the result (fairy) represents the address of the next instruction.

II Instruction Decode

Instruction Memory

Registers

*During the Instruction Decode phase, the instruction is then split into 5 sets of bits, which are represented by three Eskimos, prince charming, a toy soldier, and a teacher.

*Two of the three Eskimos will live and become the read registers. The other will be shot by the space ship MUX, leaving prince charming to become the write register. The data is then read from the two surviving Eskimos, represented by the shark and the penguin, which will travel to the ALU for further computation.

*The toy soldier represents the funct code, which will branch into two identical toy soldiers, where one is sent to the sign extend (gazebo) to grow taller, only to be later defeated by a tank MUX, while the other, along with a mushroom sent from the control unit, is sent to the ALU control (toaster).

*The teacher represents the op-code, which is sent to the control unit to be transformed into a series of signals that control various components in the data path.

III Execution

ALU

*During the Execution Phase, the data read from the registers (shark and penguin) is sent to the ALU (farmhouse) to be manipulated in whatever way was specified by the line which goes from the ALU control to the ALU (cookie). Before going into the ALU, the penguin defeats a copy of the tall toy soldier via the tank MUX.

*The ALU calculates the result (crispy knight) and returns a zero line (horse), which is part of the control line for a future dragon MUX.

*A copy of the growing soldier not defeated by the tank MUX is then shifted left 2 (trolley) and sent into another adder (ring toss) along with a fairy, which was the result of the first adder.

IV Memory

Data Memory

*During the Memory phase, the result of the adder executed in the Execution phase (samurai) is sent to a dragon MUX, only to be killed in favor of the fairy from the original adder. That fairy then goes back to the control tower (PC Counter) as the address of the next instruction to be executed.

*The result of the ALU (crispy knight) is then sent to both the data memory and space ship MUX which will become active during the Write Back phase. The knight from the ALU and penguin from the instruction decode phase will both be sent to data memory, with the result only to be destroyed by a space ship MUX in the Write Back phase.

V Write Back

Write Register

*During the Write Back phase, the final space ship MUX destroy the zombie, which was the result of knight and penguin sent to the data memory. The result from the ALU (crispy knight) is then sent to the write data town house to become the new data in the write register.

VI Control Unit

*The Control Unit takes the op-code (teacher) and creates a series of eight signals. Four of these signals (a dragon, two space ships, and a tank) control the MUXs, while the remaining four control other various components in the data path: (mushroom- ALU Control, two dinosaurs - Data Memory, and cowboy- Registers).

Figure 4: Poster showing the datapath components simulated in Alice.

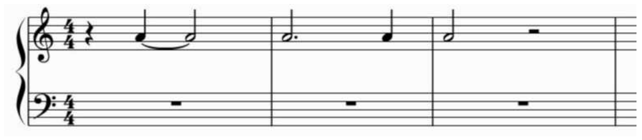


Figure 5: Mapping the binary string 0100100110 to a rhythm.

quarter notes, where n is the maximum number of bits in a binary string passing through that segment. The binary string is mapped to a rhythm in which a one represents the beginning of a note and a zero represents a rest if it is not preceded by any ones in its segment; otherwise, a zero sustains the last note that was played. Figure 5 shows an example of rhythmic mapping.

With a rhythm established, a pitch must be mapped from the data lines. The pitch value of a note is a function of the data line its corresponding binary string is passing through. The individual bits within a binary string together form a chord for that segment.

The program is given a MIPS instruction which is then decoded into its constituent binary strings. These strings are mapped to rhythm and pitch, and the corresponding chords are played. If a pipeline is being simulated, multiple chords will be played for each segment. Thus, choosing the notes for each data line was done so as to produce a pleasing combination of chords when played simultaneously.

3.1.5 *Results*

Every group presented an acceptable model of a datapath (or approved component). As shown in the previous section, several of the projects were quite novel and showed particular ingenuity. Each group prepared a poster of varying quality; this warrants better specifications the next time the project is assigned. Anecdotally, on the final exam that included a major question regarding the datapath, the scores were better than similar final exams (same instructor) in the previous two iterations of the course. Specifically, the mean was 75 with a median of 73 this year, compared to 67/67 and 69/70 in the previous iterations. Furthermore, the average score was a very good 12.3/14 (88%) on the datapath question that typically is problematic for many students. Perhaps of more significance was that the students really took to the project and were interested in not only their own models but their classmates' as well.

3.2 Software Simulation of the Datapath

The implementation of a software simulation of the MIPS datapath and control in Figure 1 serves as a final lab exercise and project for Teresco's course. It is one thing to use an existing simulator, but quite another to have to understand it to the level of detail required to write your own simulator.

3.2.1 *Software Simulation Assignment*

The single-cycle simulator the students are required to implement consists of a number of components.

- A command-line parameter specifies a memory file,

```
0x8c010024 ! 0: lw %1 36(%0) a
0x8c020028 ! 4: lw %2 40(%0) b
0x8c03002c ! 8: lw %3 44(%0) 1
0x00002020 ! 12: add %4 %0 %0 prod = 0
0x00822020 ! 16: add %4 %4 %2 prod += b
0x00230822 ! 20: sub %1 %1 %3 a--
0x10200001 ! 24: beq 1 %1 %0
0x08000004 ! 28: j 16
0xffffffff ! 32: halt
0x00000003 ! 36: a
0x00000004 ! 40: b
0x00000001 ! 44: 1
```

Figure 6: A sample memory file specifying a program to perform multiplication by repeated addition. Only the hexadecimal numbers are relevant – the remainder of each line is treated as a comment by the simulator.

which is simply a list of 32-bit hexadecimal values that are placed into the simulation's memory component. These values may be treated as instructions or as data.

For example, the memory file in Figure 6 loads a program that multiplies two numbers `a` (3, from memory location 36), and `b` (4, from 40), using repeated addition, accumulating the result in register `%4`.

- A simple command line interface is used to control the simulation. This includes commands to execute one or more instructions from memory, print the contents of the registers or a range of memory locations, print the control line values for the current instruction, turn on a debug mode, and quit the simulation. The command-line parser is usually provided in starter code to keep the project more manageable.
- The simulator needs to be able to break down a MIPS instruction into the fields that will be used to drive the simulation of the instruction. In a previous lab assignment, students were required to read in a sequence of MIPS machine instructions, print out the opcode, instruction format, and value of each relevant field for that instruction format.
- The values of each control line in the datapath are computed by a function which shares that control line's name, *e.g.*, the *MemRead* control line's value is obtained as needed (by the data memory unit) by calling the `MemRead()` function in the control unit module. The only permitted use of the opcode field of the instruction is to pass it as a parameter to the control unit module.

The heart of the simulator is the simulation of a single instruction. The simulator models physical components as high-level language constructs. Memory and register components become variables or arrays, multiplexers become conditional statements, and data and control lines become variables.

Each component is then simulated, using only the information that would be available to that component in the circuit, to produce its output or change its state. For example, the instruction memory uses the value presented to it from the program counter, and produces the 32-bit instruction at that location on its output (which becomes a variable, usually named `ir` for the instruction register). Meanwhile, the program counter value is also passed to the adder that is always adding 4 to its input, producing the address of the next instruction that will eventually make its way back into the program counter should that value not be replaced by a branch target later in the instruction simulation. Concurrency is not simulated directly, but part of the exercise is to think carefully about what order things happen in the datapath as an instruction is executed. Students also are forced to realize that these two components are combinational, and will continue to produce their current output values so long as the program counter's value does not change. And they know that will not happen until the final phase of the instruction. This also forces students to realize that the circuit is always computing useless information concurrently with useful information. For example, an arithmetic instruction will never make use of the branch target or jump target ALU results, but those values are computed nonetheless.

The next step is the essential one to implementing a simulation that is true to the datapath. Parts of the instruction (the bit fields determined by the instruction formats) are delivered to several components directly. The opcode is sent only to the control unit, from which all of the control line values can be computed. This simulator does print out the name of the instruction based on the opcode, but this is not used to drive the simulation in any way other than to compute control line values in the control unit. That is, nowhere in the simulation outside of the control unit should there be a place where decisions are made based on opcode. All other decisions are made based on the control line values as computed by the main control or the ALU control.

To ensure that the simulation accurately models the circuit, a debug mode is required. In debug mode, each component reports on its function as the instruction works its way through the circuit. For example, Figure 7 shows what the debug output for the first instruction from the example in Figure 6 above (a memory read, `lw`) might look like. Each component's action is shown, even those whose actions end up not being used by the instruction. Even the ALU, where students may be tempted to implement a subtract instruction using subtraction, the action is modeled as an addition with the subtrahend negated, as this is how the ALU would be implemented in hardware.

Students are encouraged to develop their simulators in C, as one of the goals of the course is to introduce them to that language, but some choose to use Java or another language they know well. We use a standard high-level language without a library like SystemC [4], as the architecture we simulate is simple enough that the additional cost of learning the library is likely more significant than any potential savings in implementation time.

In addition to the simulator, students are required to provide a new MIPS machine code program along the lines of the one shown in Figure 6 to perform some simple but hopefully interesting computation. This helps to reinforce their understanding of the relationships among the MIPS assembly instructions they wish to use, the MIPS machine instructions that represent them, and the actions of the circuit to execute the instructions.

Finally, students are required to describe in detail how the architecture and their simulator would need to be modified

```
Executing instruction: 0x8c010024
op=0x23/35, format=I, rs=0x00/0,
   rt=0x01/1, imm=0x0024/36
PC+4: 0x00000004
Sign Extend: 0x00000024
ReadReg 1: R[0]=0x00000000
ReadReg 2: R[1]=0x00000000
Jump address: 0x00040090
ALU input MUX (from sign extend): 0x00000024
Branch offset: 0x00000090
ALU input A not inverted: 0x00000000
ALU input B not negated: 0x00000024
ALU result (ADD): 0x00000024, Zero=0
Read from memory at 0x00000024, value 0x00000003
Branch target: 0x00000094
Branch selection MUX not taking branch: 0x00000004
Write reg number: 0x01
Write reg data from memory: 0x00000003
New PC no jump: 0x00000004
Write reg: 0x00000003 to R[1]
```

Figure 7: Debug mode output for the execution of the first `lw` instruction in the program in Figure 6.

to implement some of the instructions which are not part of the required MIPS subset. For bonus points, they can augment their simulator to implement those instructions and provide example programs to demonstrate them.

3.2.2 Results

The project has successfully forced students to think carefully about how each component in the datapath works, what information it has available as input, and what it produces as output. Considering each in isolation (as a small segment of code) and planning the simulation order to ensure proper interaction of components is an intense and fairly painful process for the students, not to mention the instructor who has a class full of students inside and outside his office for the duration of the project.

While there is no meaningful way to compare students who have completed the project with those who have not (the course is typically taught to one section of students once per year), we can examine some grading information. In the most recent course which used this project (Fall 2011), 20 of the 23 students who were still actively engaged in the course submitted a substantially complete simulator. The others were in group of 3 that made an honest effort but continued to have fundamental misunderstandings and did not ask for help until it was too late. The final exam question that was directly related to the project was very successful. The overall score on the question was 84%, and over 90% if we remove the scores of those students who did not complete the project.

The most encouraging results from the project are also those which are harder to quantify. Students coming for help regularly ask the right questions. With many of them, there is that moment when it clicks – they not only understand how they need to implement the simulator, but the datapath diagram and the whole concept of a circuit running a program suddenly makes sense. Also, when class discussion turned to the more complex pipelined datapath, it was clear that their extensive experience with the single-cycle datapath helped them to understand pipelining.

4. CONCLUSION AND FUTURE WORK

We have presented two approaches to laboratory projects that help students grasp the workings of the datapath. In one approach, students build a physical device that simulates the flow of information from one component to another until a particular instruction is completed. Students applauded the lab and anecdotal evidence showed that exam scores improved. The second approach requires students to understand how the datapath circuit executes an instruction by simulating the low-level hardware components in software.

It is extremely difficult to test the efficacy of such labs, making it dangerous to draw any firm conclusions. However, we believe that such hands-on projects help students learn, as has been shown in other areas in general. At the very least, the material is presented in a fresh way that students enjoy much more than a traditional lecture. We will continue our work on these labs and attempt to add more hands-on experiences in the computer organization courses.

5. REFERENCES

[1] D. Berque, I. Serlin, and A. Vlahov. A brief water excursion: introducing computer organization students to a water driven 1-bit half-adder. *SIGCSE Bull.*, 36(2):52–56, June 2004.

[2] G. Braught. Computer organization in the breadth-first course. *J. Comput. Sci. Coll.*, 16(4):182–195, 2001.

[3] C. Burch. Logisim: A graphical system for logic circuit design and simulation. *J. Educ. Resour. Comput.*, 2(1):5–16, 2002.

[4] E. Harcourt. Teaching computer organization and architecture using SystemC. *J. Comput. Sci. Coll.*, 21(2):27–39, Dec. 2005.

[5] J. Larus. 2012. SPIM: A MIPS32 simulator. Retrieved September 6, 2012 from `http://sourceforge.net/projects/spimsimulator/`.

[6] D. Magagnosc. Simulation in computer organization: a goals based study. *SIGCSE Bull.*, 26(1):178–182, Mar. 1994.

[7] L. Null and J. Lobur. MarieSim: The MARIE computer simulator. *J. Educ. Resour. Comput.*, 3(2):1–29, June 2003.

[8] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2009.

[9] R. Pausch, T. Burnette, A. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. Alice: Rapid prototyping system for virtual reality. *IEEE Computer Graphics and Applications*, 15(3):8–11, 1995.

[10] J. Phillips. Simulation of a simple CPU design and its use as an instructional tool in a computer organization course. *J. Comput. Sci. Coll.*, 22(6):140–146, June 2007.

[11] K. Vollmar and P. Sanderson. MARS: an education-oriented MIPS assembly language simulator. *SIGCSE Bull.*, 38(1):239–243, Mar. 2006.