A MIPS DATAPATH SIMULATOR FOR ENHANCING VISUAL LEARNING OF COMPUTER ARCHITECTURE

BY

CHRISTIAAN D. HAZLETT

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Teaching Associate Professor Geoffrey Herman
ABSTRACT

In this thesis, I introduce the Datapath Simulator—a tool for visually teaching the fundamentals of hardware architecture. Built over the 2019-2020 academic year, the Simulator allows course staff and students to virtually build and explore hardware architecture circuits. The Datapath Simulator is built around the MIPS architecture and supports running MIPS32 instructions on the simulated circuits. Users can view and change values on busses, components, and memory across clock cycles to support building a visual mental model of an operational datapath. I also present the results of a survey of University of Illinois hardware architecture students on their reception of adding the Datapath Simulator to our course, CS233. I find that the majority of students would support using this visual tool to augment the current course materials, which include static datapath diagrams and non-visual simulation of the MIPS datapath through Verilog.
To Spaceballs, for being the best post-original trilogy Star Wars movie.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>RELATED WORK</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Previous MIPS Simulators</td>
<td>2</td>
</tr>
<tr>
<td>2.2</td>
<td>Digital Logic Simulators</td>
<td>3</td>
</tr>
<tr>
<td>2.3</td>
<td>Digital Manipulatives</td>
<td>3</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td>DEFINITIONS</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Hardware Architecture Components</td>
<td>7</td>
</tr>
<tr>
<td>3.2</td>
<td>Labels</td>
<td>10</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td>MIPS DATAPATH MODULES</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>MIPS Register File</td>
<td>11</td>
</tr>
<tr>
<td>4.2</td>
<td>MIPS Arithmetic Logic Unit</td>
<td>12</td>
</tr>
<tr>
<td>4.3</td>
<td>The MIPS Decoder</td>
<td>12</td>
</tr>
<tr>
<td>4.4</td>
<td>Busses, Mappers, and Sign and Zero Extensions</td>
<td>16</td>
</tr>
<tr>
<td><strong>5</strong></td>
<td>PROJECT ARCHITECTURE</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Datapath Simulator</td>
<td>18</td>
</tr>
<tr>
<td>5.2</td>
<td>Back-End Server</td>
<td>26</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>CIRCUIT DESIGN</td>
<td>27</td>
</tr>
<tr>
<td>6.1</td>
<td>Linkable Circuits</td>
<td>27</td>
</tr>
<tr>
<td>6.2</td>
<td>Design Paradigms</td>
<td>27</td>
</tr>
<tr>
<td>6.3</td>
<td>Example Design Strategy</td>
<td>29</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td>CS233 CIRCUITS</td>
<td>30</td>
</tr>
<tr>
<td>7.1</td>
<td>Arithmetic Machine</td>
<td>30</td>
</tr>
<tr>
<td>7.2</td>
<td>Full MIPS Machine</td>
<td>31</td>
</tr>
<tr>
<td>7.3</td>
<td>Full Pipelined Datapath</td>
<td>32</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>USE CASES</td>
<td>34</td>
</tr>
<tr>
<td>8.1</td>
<td>During Lectures as a Visual Aid</td>
<td>34</td>
</tr>
<tr>
<td>8.2</td>
<td>In Office Hours</td>
<td>35</td>
</tr>
<tr>
<td>8.3</td>
<td>As a Study Tool</td>
<td>35</td>
</tr>
<tr>
<td>8.4</td>
<td>As an Exam Resource in the Computer Based Testing Facility</td>
<td>35</td>
</tr>
<tr>
<td>8.5</td>
<td>For Collaboratively Building Circuits in Class</td>
<td>36</td>
</tr>
<tr>
<td><strong>9</strong></td>
<td>USER TESTS AND RESULTS</td>
<td>37</td>
</tr>
<tr>
<td>9.1</td>
<td>User Survey</td>
<td>37</td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>CONCLUSION</td>
<td>46</td>
</tr>
<tr>
<td>10.1</td>
<td>Future Work</td>
<td>46</td>
</tr>
<tr>
<td><strong>REFERENCES</strong></td>
<td></td>
<td>48</td>
</tr>
</tbody>
</table>
CHAPTER 1: INTRODUCTION

In teaching the undergraduate hardware architecture course, CS233, at the University of Illinois at Urbana-Champaign, the course staff routinely makes use of teaching aids such as whiteboards, Powerpoint slides, and paper handouts that visually show key diagrams from the course, like the MIPS datapath. In CS233, we teach a simplified version of the MIPS architecture, and we have noticed that students may understand the class concepts better when the course staff explicitly traces the path that values take on these diagrams using a pen or a finger while explaining how the datapath works as opposed to just providing the students with a static image of the datapath and asking them to figure it out [1].

Ongoing work in the department is looking at the difference in understanding made by giving students manipulable versions of the datapath for use in our hardware architecture class—Herman et al. has found giving students visual cues to be one of the primary ways to help them recall previously formed knowledge [2]. This thesis outlines the creation of a digital datapath manipulative—the Datapath Simulator—for use in the class to visually aid teaching and learning. I also present the results of a survey conducted during the Spring semester of 2020 to help gauge students’ receptivity to the Simulator.

Students in our hardware architecture class today use a variety of materials to learn the class concepts, including lecture slides and handouts, Verilog (in labs), and Prarielearn problems (for online homework). With this survey, I aim to find out whether adding a visual datapath simulation aid to this mix (or replacing one of the current tools) appeals to students and if so, where they think that such a tool would be most useful in their education.

The Datapath Simulator is a browser-based program written in Javascript that allows students to create interactive circuits using MIPS hardware architecture components. Then, students are able to simulate MIPS instructions executing on their datapath in real-time and trace the values of component inputs and outputs, busses, the register file, and data memory across clock cycles.

I found student reception to the Datapath Simulator to be overwhelmingly positive, and I determined that students would most like to see a visual aid used in the class in addition to the existing tools, as well as made available to help think though datapath problems on exams–which we have found that students currently feel hampered on because of a lack of a manipulative in the testing center. Adding a visual datapath simulation tool to CS233 would, according to students, help develop their understanding of course concepts and their ability to put them into practice on assessments.
CHAPTER 2: RELATED WORK

The Computer Science Department at the University of Illinois has been looking at the potential benefits of giving students manipulable versions of the MIPS Datapath to aid their learning. By reducing the “cognitive load” that students experience from the need to visualize the datapath in their minds while learning, our aim is to free mental resources that can be better put toward understanding a concept rather than simply memorizing it. Previous work by Herman et al. found that students often rely on memorized processes or past homework solutions when solving new problems, rather than reasoning from a foundational mental concept \[3\]. By enabling students to learn visually and intuitively with a manipulable datapath model, we aim to encourage the building of a foundational concept of how datapath components work together rather than have students memorize formulas and strategies to obtain correct answers on class problems.

2.1 PREVIOUS MIPS SIMULATORS

Previous MIPS simulators have been developed at various universities and as hobbyist tools. Dr. K. G. Smitha, a lecturer at Nanyang Technological University, created an online MIPS Simulator in 2016 that shows an instruction entry box similar to mine (see section 5.1.6) \[4\]. It displays the register values after running a single instruction, and has static explanation images of the MIPS Datapath for students to view. In addition, Dr. Smitha’s simulator displays animated pre-drawn images of data moving through the datapath, although it does not allow the user to choose instructions or values to simulate. There are many such extremely simple MIPS simulators in existence, however their purpose is primarily to show the results of instructions—they don’t help students visualize the actual datapath in a working machine.

QtSPIM is an open-source MIPS emulator that is used in CS233 labs. While it supports a wide variety of features for running and debugging MIPS programs, it does not show intermediate component values or allow users to graphically view the MIPS datapath. QtSPIM does, however, have the most complete implementation of MIPS instructions of any of the simulators I have found. While QtSpim is extremely useful for students in hardware architecture courses, it does not address system visualization at all. It is more suitable for running complete MIPS32 programs and debugging MIPS assembly than for use as a hardware architecture teaching tool \[5\].

In 2013, Bruno Nova created a Java-based MIPS simulator, DrMIPS, as a project for
his master’s thesis [6]. This downloadable program displays a dynamically updating MIPS datapath—complete with bus, component, register, and memory values. DrMIPS additionally allows the user to enter MIPS instructions in a text editor which are then simulated. Users can change memory and register values before simulating their program, but while DrMIPS supports both the standard MIPS single-cycle and pipelined machines, it does not allow the user to modify the datapath.

Nova’s work is similar to earlier work done in 2008 by Cambridge University’s Department of Engineering, who also created a downloadable MIPS simulator. Cambridge’s version is limited to simulating only ten instructions, however, and does not allow users to directly modify register values [7].

While previous MIPS simulators have allowed users to trace data values along the datapath and visually simulate the processor’s execution, they do not allow users to modify the datapath at all. The Datapath Simulator encourages students’ learning by allowing them to build the MIPS Datapath in addition to simply simulating it or tracing values between clock cycles.

2.2 DIGITAL LOGIC SIMULATORS

Logisim was created by Dr. Carl Burch at Hendrix College. While it is not strictly a MIPS simulator or a hardware architecture simulator, it does contain the basic digital logic components needed to build up a MIPS system. Logisim supports building and editing a digital logic circuit, and can then visually simulate data flowing on that circuit [8]. Unfortunately, Logisim has not been maintained in over half a decade, and the hurdle of setting up a MIPS system adds a significant amount of extra work that must be done in order to add it to a hardware architecture curriculum based around MIPS. Logisketch is a system that recognizes drawings of digital logic circuits and can transform them into circuit diagrams for simulation [9]. Unfortunately, Logisketch is also not MIPS specific and does not have all the required components for easily building a MIPS system. For this reason, it is not useful for the majority of CS233, where the focus is put on learning the MIPS datapath, rather than digital logic fundamentals.

2.3 DIGITAL MANIPULATIVES

A significant amount of previous work in the area of digital manipulatives has been focused on teaching programming to novices or children through drag-and-drop “actions.” Resnick et
al. have found that their visual programming language, Scratch, has attracted thousands of people since its debut in 2007 [10]. In addition to just programming, researchers at the University of Nebraska have found that using various digital manipulatives to teach STEM subjects in general has shown a great deal of promise [11]. By developing the Datapath Simulator, we are advancing the research on digital manipulatives into the area of hardware architecture as well.
CHAPTER 3: DEFINITIONS

In this section, I present common terms used in this paper.

A **datapath** refers to the collection of components and busses needed to compose a central processing unit’s circuitry. In the context of this paper, the term *datapath* refers to the digital logic pathway used in the MIPS processor architecture.

**Digital Logic** is the processing of binary values using computational circuitry. The basic building blocks of which are logic gates such as *AND*, *OR*, *XOR*, and *NOT*. The gates are combined to form common elements used in computational circuitry, many of which are implemented in the Datapath Simulator and explained in Chapter 4.

A **Reduced Instruction Set Computing** (RISC) architecture is an instruction set comprised of a relatively fewer number of instructions than previously-used Complex Instruction Set Computing (CISC) architectures. Common examples of RISC architectures include MIPS, ARM, and SPARC.

The **MIPS** (Microprocessor without Interlocked Pipeline Stages) architecture is an influential RISC processor architecture developed by the U.S. firm, MIPS Technologies. Its simplicity makes it an ideal architecture for teaching undergraduate students, as it contains most of the elements present in modern-day architectures while remaining relatively easy to conceptually grasp.

**Propagation Delay** refers to the time taken for the inputs of a component or circuit to affect the outputs of the component or circuit. When components are sequentially reliant on one another in a datapath, the propagation delays for the relevant circuits are the sum of the individual delays. In the Datapath Simulator, the propagation delay refers to the calculation time needed to update each component in the datapath and pass the value along to the next component.

**Overflow** refers to the exception that occurs when a digital arithmetic operation results in a value that cannot be represented using the number of digits present in the circuit.

A **bitwise** operation is a digital logic operation at the bit level of a multi-bit value. For example, a bitwise AND of two multi-bit values performs the AND computation at each corresponding bit from both inputs to produce one output that is the same width as the original inputs.

A **port** (also referred to as a connector on Busses) is a place on a module where a Bus can
be attached. In the Datapath Simulator, a port appears as a small black circle on the edge of a module that changes to white when the mouse hovers over it. Clicking the port and dragging creates a new Bus from that port.

An address is an index that specifies an item to select or operate on. In the case of the Register File, an address refers to which register is selected for reading or writing. For the instruction and data memory components, the address refers to which byte or word in memory is selected for reading or writing.

A Register File is a component that contains an array of general-purpose registers which are used to store the computational state of the processor. In modern computer architectures such as MIPS, the Register File is characterized by its separate load and store port configuration.

An Arithmetic Logic Unit is a combinational circuit that is responsible for computing a number of arithmetic and bitwise logical functions, including add, sub, and, or, nor, xor, mul, and div.

The Protoboard is the Datapath Simulator’s virtual implementation of a breadboard, where components and busses can be added to create a prototype circuit.

React is a Javascript front-end framework developed by Facebook. It is used in the Datapath Simulator to keep components’ states updated in the browser display when the backend simulation state changes.

Javascript ES6 is the version of Javascript, a widely-used language for web application development, that the Datapath Simulator is written in. Since ES6 contains features not-yet present in some browsers, I use Webpack plugins to compile Javascript ES6 back to earlier versions for wider compatibility.

A pipeline is a series of processing stages connected together so that the output of one stage is the input to the next stage. It is used in the MIPS architecture as a way of lowering the clock cycle time, which increases instruction throughput.

The Computer Based Testing Facility (CBTF) is a collection of testing rooms on the University of Illinois at Urbana-Champaign’s engineering campus. The CBTF provides courses with the means to schedule computer-based exams for students to take within a several-day window. The CBTF is also staffed by proctors at all times and has a dedicated staff to handle issues with exams or technology.

Prairielearn is the University of Illinois’s online homework and testing platform. It is
used in CS233 to provide daily homework problems, practice exams, and exams taken in the Computer-Based Testing Facility.

**Verilog** is a hardware description language that is used on CS233 labs and exams as a way for students to implement circuits and test their results. It is an entirely textual language with no support for visualizations, and has been standardized by IEEE.

A **Clock** signal is an electrical signal which flips from 0 to 1 at a specific interval. This signal is used to ensure that the components in the processor remain synchronized with one another.

**JSX** is an extension to Javascript that allows inlining HTML components as objects. It is commonly used with React.

### 3.1 HARDWARE ARCHITECTURE COMPONENTS

In this section, I introduce common components used in hardware architecture design. Each of these components has an implementation in the Datapath Simulator, and specific MIPS variants of these components are described in Chapter 4.

#### 3.1.1 Multiplexers and Decoders

A multiplexer allows the circuit designer to choose between two (or more) inputs to the component, the data on which will be sent to the wire connected to the output of the multiplexer component. The desired input is specified by a bus connected to the ctrl input.

A decoder allows the circuit designer to designate an output that should be turned on when the decoder is enabled. The output is specified by a binary number on a bus connected to an n-bit control port. When the decoder enable bit is set to 0, no output busses are turned on.

![Figure 3.1: Both a multiplexer and a decoder](image)

Figure 3.1: Both a multiplexer and a decoder
3.1.2 Logic Gates

The Datapath Simulator includes modules that simulate a number of basic logic gates, which are primarily used in the earlier portions of the course to introduce students to digital logic.

Each logic gate takes two arbitrary-bit inputs (although both inputs must be the same width) and outputs a value of the same width.

![Digital Logic Gates XOR, AND, OR, and NOT](image)

Figure 3.2: The Digital Logic gates XOR, AND, OR, and NOT

Each gate performs a bitwise operation, the truth tables for which are found below:

<table>
<thead>
<tr>
<th>Input X</th>
<th>Input Y</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.1: The truth table for an XOR Gate

<table>
<thead>
<tr>
<th>Input X</th>
<th>Input Y</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2: The truth table for an AND Gate

<table>
<thead>
<tr>
<th>Input X</th>
<th>Input Y</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3: The truth table for an OR Gate

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4: The truth table for a NOT Gate
3.1.3 Inputs and Outputs

One of the most fundamental sets of components available to users of the Datapath Simulator are the input and output components. These are oftentimes the first components added to the Protoboard when designing a circuit, as they define the bounds of what the circuit is designed to accomplish.

![Figure 3.3: The Input and Output Components](image)

**Inputs**

The Input component lets the user control the data and options flowing into a circuit. With the exception of information coming from memory or the Register File, all of the information present in a circuit comes from an Input component.

The Input component has one option—the width of the wire that it is attached to. This is set in the Popover view for each component, and any wire connected to the component then will adopt whatever width is set in the Input.

The user can also set a value in decimal, hex, or binary via either the Popover display or by editing the value on the component itself in the Protoboard view. To edit the value that the input sends to a connected wire, the user can click on the value displayed on the component. The value then turns into an editable textbox, and the user can input any values using the prefixes 0b, 0x, or no prefix for binary, hex, and decimal interpretations, respectively.

**Outputs**

Outputs display a value from a connected Bus. The display format for an output can be toggled between decimal, hex, and binary by clicking on the value shown on the component.

If an Output is not connected to a wire, the output will display “N/C” on the component.
3.2 LABELS

A special meta-component in the Datapath Simulator, **Labels** allow the user to name certain areas of the Protoboard or to add text descriptions to busses and components. A label’s width, height, and text value can be edited by the user in the Popover dialog, and a label can become “fixed” to a component on the Protoboard by dragging the Label onto it, so that when the component is moved, the Label will be translated as well.

![Figure 3.4: A Label on the bus between a Bus Subset and a Zero Extender](image)

Figure 3.4: A Label on the bus between a Bus Subset and a Zero Extender
CHAPTER 4: MIPS DATAPATH MODULES

In this section, I describe MIPS-specific modules that have been implemented in the Datapath Simulator:

4.1 MIPS REGISTER FILE

The MIPS Register File contains 32 general-purpose registers, some of which are available to the programmer, while others are reserved for use by the MIPS compiler. Since MIPS is a load/store processor architecture, every machine instruction involving changing data (besides load and store) both reads from and writes results to the register file.

Figure 4.1: The MIPS Register File Component

The Register File is capable of reading two registers at the same time. The number of the MIPS register to read is passed into the port on the register file labeled either rsAddr or rtAddr, which causes the Register File to output the data contained in the specified registers to the ports rsData and rtData, respectively. To write data into the register file, the desired receiving register number is specified at the rdAddr port, and the 1-bit port, rdWriteEnable, is set to HIGH. Any data then present on the wire connected to rdData is stored into the specified register on the next downward clock cycle edge.
4.2 MIPS ARITHMETIC LOGIC UNIT

To perform its operations, the MIPS Arithmetic Logic Unit takes two 32-bit data wires as inputs—one each on ports A and B. Then, a 5-bit control signal is connected to the ctrl port, which specifies the operation to perform. The result is available on the output port after only a small propagation delay.

![Figure 4.2: The MIPS Arithmetic Logic Unit](image)

Additional wires present on the ALU (overflow, zero, and negative) give 1-bit informational signals about the output from the ALU operation.

<table>
<thead>
<tr>
<th>ALU Opcode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Unsigned addition</td>
</tr>
<tr>
<td>010</td>
<td>Two’s-complement addition</td>
</tr>
<tr>
<td>011</td>
<td>Two’s-complement subtraction</td>
</tr>
<tr>
<td>100</td>
<td>bit-wise AND</td>
</tr>
<tr>
<td>101</td>
<td>bit-wise OR</td>
</tr>
<tr>
<td>110</td>
<td>bit-wise NOR</td>
</tr>
<tr>
<td>111</td>
<td>bit-wise XOR</td>
</tr>
</tbody>
</table>

Table 4.1: A table of operations supported by the ALU in the Datapath Simulator

4.3 THE MIPS DECODER

The MIPS architecture implements a special decoder component which is responsible for directing all of the control signals in the processor.
The Datapath Simulator version of the MIPS Decoder implements the following control signals:

<table>
<thead>
<tr>
<th>Decoder Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu_op</td>
<td>3-bit value to specify the operation to be performed by the main MIPS ALU</td>
</tr>
<tr>
<td>write_enable</td>
<td>Specifies whether to enable writing to the Register File</td>
</tr>
<tr>
<td>rd_src</td>
<td>Whether the rt or rd register should be used when writing to the Register File</td>
</tr>
<tr>
<td>alu_src2</td>
<td>Whether the rt_data value or signExtendedImmediate should go to the B ALU input</td>
</tr>
<tr>
<td>control_type</td>
<td>2-bit value that specifies the input for the Program Counter</td>
</tr>
<tr>
<td>lui</td>
<td>Load Upper Immediate control signal</td>
</tr>
<tr>
<td>slt</td>
<td>Set Less Than control signal</td>
</tr>
<tr>
<td>byte_load</td>
<td>Determines whether a whole word or just a byte is loaded from memory</td>
</tr>
<tr>
<td>word_we</td>
<td>Specifies when a word should be written to data memory</td>
</tr>
<tr>
<td>byte_we</td>
<td>Specifies when a single byte should be written to data memory</td>
</tr>
<tr>
<td>mem_read</td>
<td>Specifies when the data should be read from data memory</td>
</tr>
<tr>
<td>except</td>
<td>Is HIGH when the opcode and funct inputs are not a valid instruction</td>
</tr>
</tbody>
</table>

Table 4.2: A table of signals supported by the MIPS Decoder

The MIPS Decoder takes two inputs: opcode and funct. These inputs correspond to the two fields in the MIPS Instruction format diagram below. In the case of a non-R type instruction, the funct input is ignored.

From these two values, the MIPS decoder sets the appropriate signal values on all of the output ports. The signals are used to correctly move the data throughout the processor datapath, and to chose the appropriate operations to perform on the data at each component.
4.3.1 Instruction and Data Memory

In MIPS architecture diagrams, a machine’s memory is split into two separate components for ease-of-explanation. Although there are both data and instruction memory components, they share the same physical memory module in an actual MIPS machine. The Datapath Simulator implements two separate memory components since the way the user interacts with the data in each is different.

Instruction Memory

Instruction memory in the Datapath Simulator gets its values from the instructions entered using the Code Bar (see section 5.1.6). When a user enters an instruction, it is stored in the instruction memory component as a JSON object containing an array of the fields in the instruction. When the Protoboard calls the instruction memory to update the value of the connected wire, the Instruction Memory assembles the instruction into machine code and sends the resulting 32-bit instruction via a wire to a connected Decoder.

The only input to the Instruction Memory, the address field, tells the component which instruction to push to the connected output wire, data. For simplicity in this simulation model, the Instruction Memory starts with the entered program at address 0x0. Addresses in
the Instruction Memory and the Data Memory are allowed to overlap, but each component keeps a separate record of what exists at each overlapping address. (In reality, MIPS programs should not modify their own code, so this shouldn’t have a usability impact)

Data Memory

The Data Memory component in the MIPS architecture represents the portion of memory reserved for an application’s data. In the Datapath Simulator, the Data Memory component stores data saved written to memory in a JSON format, which is editable by the user, either in the Data Memory interface’s editor, or by writing to the Data Memory during simulation of the circuit. When the user inputs an address into the address input, the Data Memory returns the corresponding value stored in memory. Any data accesses to uninitialized memory returns the value 0x0.

The Data Memory component has 5 inputs: the address input specifies the address that the Data Memory component will either read from or write to, data_in specifies the data to be written to memory, and word_we and byte_we specify whether to write either a word or a byte to memory, respectively. The reset signal, of course, resets the data memory to all zero values.

![Figure 4.6: The Data Memory Component](image)

4.3.2 Pipeline Registers

One strategy for increasing the throughput of a processor is designing it in such a way that each part of the processor is busy with part of an instruction at any given time. Pipelining divides the MIPS processor into stages, each of which handles one instruction at a time. Each instruction starts with the first stage, then moves onto the following ones until it has
completed each stage in the pipeline. As one instruction moves to the next stage, another follows behind it by moving into the previous stage.

To keep data and control signals separated for each of the instructions present on the MIPS datapath at once, each intermediate stage (i.e., not the last stage) ends with a Pipeline Register. The Pipeline Register for each stage behaves in the same way as a normal register—at each positive clock edge, the Pipeline Register accepts the resulting values and any needed control signals from the previous stage. Those values are then available in the next stage at the start of that clock cycle.

Since the number of values saved after each stage changes depending on the stage and the type of datapath implemented, the Datapath Simulator’s Pipeline Register component has a variable number of inputs, each of which can be renamed by double clicking the input label on the Pipeline Register component. Shift+double clicking a register removes the register from the Pipeline Register.

![Figure 4.7: The Pipeline Register Component](image)

4.4 BUSSES, MAPPERS, AND SIGN AND ZERO EXTENSIONS

4.4.1 Busses

A Bus carries data between components on the processor datapath. In the MIPS architecture, busses range from 1 to 32 bits in width, and can have varying numbers of components connected to them at any given point along the Bus. Additionally, in the Datapath Simulator the Bus component is the primary method of transfer for all data between component classes. Each BoardComponent class can either read from or write to a connected Bus at each of the BoardComponent’s ports. This mechanism is described in section 5.1.2.
4.4.2 Mappers

In circuit design, it is often necessary to either select a few bits from any given bus, or to concatenate two or more busses together into one larger bus. In the Datapath Simulator, this is done with the **Mapper** component. The Mapper class has inputs for one or more busses, and the busses can be configured to output to one or more busses via the Popover display.

![Mapper Diagram](image)

Figure 4.8: The Mapper component shows a visual depiction of the bus concatenation set up in the Popover control.

4.4.3 Sign and Zero Extensions

A **Sign Extension** component allows the circuit designer to expand a **Bus**'s width while preserving the leading bit from the original **Bus**. This is useful in cases such as expanding the 2's-complement 16-bit immediate field for input into the 32-bit ALU of the MIPS Datapath.

Similarly, a **Zero Extension** is likewise used to expand the width of a **Bus**. However, in this case the expanded most significant bits are set to the value 0. This is useful for handling unsigned immediate values.

![Sign and Zero Extenders](image)

Figure 4.9: The Sign and Zero Extenders
CHAPTER 5: PROJECT ARCHITECTURE

The Datapath Simulator codebase is divided into two main components:

1. The front-end Datapath Simulator and display code
2. The back-end Node JS web server and datastore

5.1 DATAPATH SIMULATOR

The Datapath Simulator itself runs locally in the user’s web browser. It is written in Javascript ES6, and uses the React library to display the simulation view. Within the Datapath Simulator front-end, there are several notable components:

1. BoardComponents
2. Busses
3. Protoboard
4. Timeline
5. Popover
6. Code Bar

5.1.1 BoardComponent Subclasses

Each simulatable component (with the exception of the Bus) in the Datapath Simulator is a subclass of BoardComponent. This class, when instantiated directly, provides an example simulation component with trivial logic. It is the prototype class for all of the other objects used in the simulation, and provides example update methods that are overridden by the subclasses to implement their own component logic and information displays.

This class also implements the click handlers for clicking and dragging the objects around the board, which are not overridden by the subclasses, and the handlers for clicking and dragging the component connectors—whose location, name, and function details are provided in the subclasses but are displayed using methods from the BoardComponent class.

Additionally, this class handles the clock progression tracking for each component and provides methods for the subclasses to check whether the clock cycle number has changed.
since the last update (to implement clocked logic, for example). **BoardComponent** also handles serializing the class instance into a JSON string (for storage on the Node back-end server) and restoring the state from a JSON string.

BoardComponent State Storage

Each component that is instantiated in the simulation has some state values stored along with it. These values are wrapped into the React state datastructure, and include nested values and circular references. Although this is not recommended by the React documentation (it makes serializing and copying the state values much more complicated), I believe that this architecture allows the most freedom for properly simulating the components. React does not operate on object-oriented principles, but by cajoling those ideas into React and using React **refs** to get Javascript references to individual instantiated classes, the Datapath Simulator can take advantage of the best of both worlds—components that are treated as ad-hoc representations of their real-life counterparts, and components that update their displays automatically whenever their states have been changed. The React state for a typical **BoardComponent** is shown in figure 5.1:

Each component in the simulation has a **translation** variable associated with the state that is stored relative to the initial offset of the **Protoboard** itself. This value is updated whenever the component is moved by the user, and is stored in pixel increments that are strictly multiples of 10. Each “pin” on the **Protoboard** is 10 pixels apart, and components are constrained to fall evenly onto those pins when dropped.

Every component also has a title associated with it—this is used by the Popover display, and the value can be changed by the user by editing the component’s name in the Popover dialog. Also associated with each component is a single SVG image. These images were designed by the author in Serif Ltd’s Affinity Designer and were exported as SVG assets into the **static** directory of the Datapath Simulator.

The **connectors** array contains Javascript objects representing the connectors attached to the current component. This array is overridden by each subclass of **BoardComponent**, depending on the component being implemented. Methods are provided in the base class for accessing this array by name. So, if a component is setup with a connector named A, the subclass logic functions can call **getConnectorWithName()** to get a reference to that connector object. Additionally, when a connection is made (as determined by the **Protoboard** class and each **Bus** in the simulation), a reference to the instance of the **Bus** that is connected to the connector is stored in the **connector.reference** variable.
Figure 5.1: An abridged state object from the instantiation of the BoardComponent class

When a component updates its logic, it can then set the value on the wire by, for example, calling `this.getConnectorWithName("output").reference.setDecimal(<computed value>). Each connector also stores the x and y offsets for displaying itself, relative to the component top-left corner and in units of Protoboard pin increments.

`infoDisplaySegments` contains a list of React JSX HTML elements that are displayed in the contextual Popover whenever a component is selected. This list is re-generated and pushed onto this variable at each component update by the `updateComponentDisplay()` method. The `BoardComponent` base class takes care of actually displaying these elements on the Popover at the appropriate time. This list can contain any number of `<span>` elements, which are translated into vertically-divided segments of the Popover. Usually, components make use of this list to display the simulation values associated with the component, as well as to offer form inputs for any values that should be user-changeable.

The `simulationState` object stores values that are used in the computation of the component’s logic. This Javascript object is constrained to a single level, as it is copied
and stored in the Protoboard each clock cycle increment. By copying the values from the input wires into the component and storing intermediate state in this object, I ensure that connecting/disconnecting components or mid-clock cycle updates on connected components don’t interfere with the correct result of the simulation.

Finally, the `currentClockSignal` is the clock value that has been broadcast to the component. The base class provides methods to check whether the clock signal has changed, and if so whether the current clock signal value is a rising edge or a falling edge. Whole values signify a rising edge, while half values signify a falling edge. The component logic functions poll this value on each update, and when the clock signal has changed, they can copy the values from the connect busses into the `simulationState` object, then update their output values.

### 5.1.2 Busses

As the carriers for all of the information that exists inside a circuit, busses play an important role in the Datapath Simulator. The `Bus` class contains a variety of methods that facilitate the transfer of data among components, as well as the methods required to make connections between components and store those relationships.

In the Simulator, the `Bus` class is responsible for maintaining the connections between components by asking the Protoboard if there are any components with connectors that exist at the locations where the bus has its own connectors. When the position of a component or the placement of a bus connector is changed on the Protoboard, the Protoboard notifies all affected busses via the `checkForNewBusConnections` method that it should re-check for potential connections to components on the datapath.

When the `checkForNewBusConnections` method is called, the Protoboard releases the bindings that it stores to each of the connected components and re-establishes any new connections that should be made (by checking for corresponding connectors via the Protoboard).

As the only simulation class in the Datapath Simulator that does not inherit from the `BoardComponent` class, the `Bus` component stores information in a different format. The state for each `Bus` on the datapath is stored in this data structure:
```javascript
this.state = {
  width: 32,
  busValue: [],
  anchors: [
    // {x: number, y: number}
  ],
  anchorPairings: [
    // [number, number]
  ],
  connectedComponents: [
    // {
      anchor: (index),
      component: reference to component
    }
  ],
  isInEditMode: false,
  isInHoverMode: false
};
```

Figure 5.2: The initial state object from the Bus class

Each busses’ value is stored as an array of either `binaryValue.ONE` or `binaryValue.ZERO` enum values. These enum values correspond to the integers 1 and 0, respectively. This method of storing binary values ensures that we don’t suffer from precision errors or other related weirdnesses when modeling the wires.

`anchors` stores `x` and `y` pair values that correspond to the Protoboard-relative locations of the anchor point (or connectors) in the Bus instance.

`anchorPairings` contains the indexes (from the `anchors` array) of two anchor points that are connected via a line in the busses’ visual representation. This data structure allows me to arbitrarily connect anchor points in a graph-like way, which means that wires can be split or re-arranged as necessary.

`connectedComponents` contains a data structure for each of the components that are connected to the Bus. The component’s connecting anchor-point is kept (via index integer value) as well as the reference to the component’s class and connector class. This allows me to provide simple methods in the Bus implementation that return or modify references to any connected components (for broadcasting values, getting adjacent components, etc).

`isInEditMode` and `isInHoverMode` correspond to the display modes of the Bus on the Protoboard. When an event that changes either of these variables is handled, the Bus is
re-drawn via standard React methods with the value of these variables determining the visual state of the Bus.

5.1.3 Protoboard

The Protoboard is the main class that the user interacts with in the Datapath Simulator. It contains all of the required methods needed for creating, displaying, saving, restoring, connecting, and moving components around in the simulation, as well as the methods needed for the actual simulation of the MIPS datapath.

When components are added or modified on the datapath, the Protoboard calls the `updateComponentLogic()` method on each of the `BoardComponent` objects in the simulation. This method is responsible for handling the logic of each component, and is overridden by each subclass of `BoardComponent`.

Figure 5.3: The Datapath Simulator Displaying the protoboard, component drawer, and timeline.
The Protoboard keeps a reference to all the currently-instantiated objects in the simulation (via Javascript pass-by-reference of objects), as well as to all of the busses in the simulation. Whenever a Bus or BoardComponent is moved on the Protoboard by the user, the Protoboard calls a method on each BoardComponent, asking it for the positions (relative to the base of the component) of each of its connectors. These values and the related components are then passed to each Bus, which will make a connection by storing a reference to the Bus object within the component and a reference to the component within the Bus. In this way, I can have two-directional communication, and each component can see which busses it is connected to as well as which components are connected to that Bus. Whenever a component value changes, a method is called to publish the value onto the Bus connected at the given port and the value is therefore available to read by all other connected components on the Protoboard.

5.1.4 Timeline

The simulation Timeline sits along the bottom of the simulation display, and when expanded contains controls for viewing and setting the current clock cycle of the simulation. Whenever a component or value is changed by the user, the clock signal is reset to its initial value of zero. This change is broadcast to all the components in the simulation. As the user increases the clock cycle count by either scrubbing through the timeline or incrementing the counter, the simulator updates each component by calling updateComponentLogic() until the simulation reaches a steady state. Then, the simulator increments the clock cycle by calling clockSignal(<clock value>) on each of the components in the simulation, and continues by updating the logic on each component until the simulation has reached a steady state again. After the simulation has settled at each clock cycle in the simulation, the state of each component is serialized and stored in an array inside the Protoboard. When the user scrubs through the clock cycle timeline, if the values for that clock cycle have already been simulated, the state of each component on the Protoboard is restored, and the user can then view the values at each component for that clock cycle by clicking the component in the Protoboard view. Whenever a component is modified, the simulation state is discarded, and new values are calculated at each clock cycle increment.

5.1.5 Popover

When designing the Datapath Simulator, one consideration was that the interface for interacting with each component should be as consistent as possible. I originally started with
a popup interface that would show over each component, however I quickly realized that this would confuse users since the popups often covered other important parts of the circuit diagrams that users want to see simultaneously. This problem was particularly exaggerated when different components would show popups that were different sizes. I solved this problem by moving to a fixed-height bar for interacting with the components that sits at the top of the display. This bar, which operates much like the menu bar from macOS, allows users to judge the amount of space they need to leave when positioning components on the Protoboard display while interacting with objects. While the Popover can grow and shrink in width, its height remains constant, making interacting with the components a more consistent experience.

![Input component with popup interface](image)

**Figure 5.4:** The Popover display showing options for an Input component.

Each component stores its own display segments for the Popover, and when the component is selected, the Protoboard instructs the component to push its segments onto the Popover display. The Popover display acts like a stack: display segments, denoted as JSX `<span>`s can be pushed onto the Popover’s display array, or the entire array can be cleared when a new component is selected.

5.1.6 Code Bar

When a user adds an Instruction Memory component onto the datapath, it is necessary to specify the instructions to be outputted from the Instruction Memory component. When a user clicks the Instruction Memory, the Code Bar appears on the left side of the screen. This display has been designed to make entering MIPS instructions as easy and error-free as possible.

The Instruction Memory has the ability to currently store twenty MIPS instructions, the first of which starts at address 0x0. The Code Bar allows users to enter common MIPS instruction for each line as well as any register numbers or immediate values that are parameters to the instructions.

When a user clicks in the first textbox for a line in the Code Bar, the MIPS instruction
Figure 5.5: The Code Bar display provides an easy way for users to enter MIPS Instructions using proper syntax.

field autocompletes as the user types out the instruction. Then, depending on the selected instruction, the rest of the line shows inputs for registers, immediate values, or offsets according to the syntax for the selected instruction.

5.2 BACK-END SERVER

The back-end NodeJS server for the Datapath Simulator performs two functions: a) it serves the Datapath Simulator over an http connection to browsers requesting the website and b) it receives saved Protoboard JSON files and serves them to the front-end when a saved circuit layout is requested by the front-end.

Currently, the back-end server stores data in a memory store, with the JSON file saved along with the metadata id, is_template, and secret.

When the /load route is called with the URL parameter id, the back-end server replies with a JSON file corresponding to that id if one exists. If it does not exist, a 404 error is returned instead.

If the Datapath Simulator back-end returns a JSON file with the variable is_template set, the front-end will assign a new id to any modifications to the circuit. If a user attempts to maliciously store data to an id marked with is_template, the back-end server will reject the request. Currently the only way to mark a saved circuit as a template is to manually change the value in the datastore.

When the front-end saves the current state of the Protoboard, it makes a request to /store with the URL parameter id and a body containing the JSON representation of the current state of the Datapath Simulator instance. The Datapath Simulator back-end checks the secret variable sent to the front-end when the circuit was loaded to ensure that it matches, and then stores the JSON representation into its memory store.
CHAPTER 6: CIRCUIT DESIGN

There are several features in the Datapath Simulator that I think will have a significant impact on the way that students and educators interact with the tool. First, circuits created with the Datapath Simulator are “linkable,” meaning that users can share links to circuits they’ve designed and educators can create reference designs for distribution to the class. I have also incorporated certain common interaction paradigms into the Datapath Simulator so that its interface will be familiar to users of modern computer design programs.

Additionally, I have identified an example design strategy using the Simulator that I believe will prove useful for users beginning to design their own circuits.

6.1 LINKABLE CIRCUITS

One of the inspirations for the simulator’s workflow is the website CodeShare [codeshare.io]. When visiting the website’s homepage, it offers users the chance to either create a new document or to open a previously existing document via the document’s ID code. When the user starts a new document, the website provides the unique ID for the document in the browser’s URL. Subsequently, when a user opens the URL on another computer the same document is loaded and is accessible to the user there.

Similar functionality is available in the Datapath Simulator to make its use familiar to computer science students already acquainted with CodeShare. When a user opens the homepage of the Datapath Simulator, the website presents the students with a curated list of available circuit designs. Clicking any of these links opens that circuit as a new document editable by the user. Because of the system’s ability to mark certain circuit design IDs as immutable, the system allows instructors to design circuits and post them publicly. Any changes to the circuit are saved to the new ID generated when that circuit is opened by a user. These changed circuits are not immutable, and therefore are editable by users whenever they are revisited.

6.2 DESIGN PARADIGMS

As part of the simulation process, students and instructors will be required to design new, or edit existing, circuits using the Datapath Simulator. I have incorporated several common user interaction paradigms into the interface for the Datapath Simulator, which I hope will
6.2.1 Drag-and-Drop Actions

Most of the user interaction paradigm for building circuits in the Datapath Simulator is based around drag-and-drop interactions. When the user begins working on a new circuit, the Datapath Simulator presents an empty Protoboard. As the user continues to add components to the Protoboard and space on the screen becomes limited, the user can grab any empty space and drag with his/her mouse to move the Protoboard view infinitely in any direction. This allows the user to allocate more space to their circuit.

Additionally, adding components to the Protoboard uses a drag-and-drop interaction as well. Components are presented to the user in the Component Drawer area of the screen, and the user can drag an item from the drawer to the Protoboard display to add the component to the circuit.

Finally, drag-and-drop interactions are used when connecting components via wires on the Protoboard. A user can click and drag on any unused connector on the Protoboard to begin placing a wire, or the user can click and drag any connector junction to rearrange wires or connect them to a component.

6.2.2 Keyboard Shortcuts

The Datapath Simulator also makes use of several keyboard shortcuts that enable the user to more quickly access certain editor functions. The list of keyboard shortcuts is below:

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Context</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift+Drag</td>
<td>On the Protoboard view</td>
<td>Selects multiple elements</td>
</tr>
<tr>
<td>Double Click</td>
<td>On the Protoboard view</td>
<td>Adds a wire</td>
</tr>
<tr>
<td>'n'</td>
<td>On the Protoboard view</td>
<td>Advances one clock cycle</td>
</tr>
<tr>
<td>'p'</td>
<td>On the Protoboard view</td>
<td>Travels back one clock cycle</td>
</tr>
<tr>
<td>Shift+Double Click</td>
<td>On a component</td>
<td>Removes the component</td>
</tr>
<tr>
<td>Click and Drag</td>
<td>On a component Port</td>
<td>Adds a Bus and starts dragging</td>
</tr>
<tr>
<td>Shift+Double Click</td>
<td>On a Bus connector</td>
<td>Removes the connector</td>
</tr>
<tr>
<td>Double Click</td>
<td>On a Bus connector</td>
<td>Adds another Bus connector</td>
</tr>
</tbody>
</table>

Table 6.1: A table of keyboard shortcuts supported by the Datapath Simulator
6.2.3 Contextual Popover Triggers

When a user clicks a component on the Protoboard, the Popover section of the display changes to show relevant information and controls for the given component. Controls on this display allow the user to specify the number of ports on a component or the control signal that specifies its operation, for example. The Popover section also display relevant information about the selected component, such as the possible input signals for controls, what the values on the connected wires are, and the values of any outputs from the selected component.

6.3 EXAMPLE DESIGN STRATEGY

When designing a circuit using the Datapath Simulator, the user should start with some idea as to what the function of the circuit will be. Generally, the user will know the inputs and outputs for the circuit, as well as have a basic understanding of the major requirements for the circuit. The user can start by dragging the “biggest” components onto the Protoboard first. That is, if the user knows the circuit will require the ability to store values in a Register File, then the user can add the Register File to the datapath first (after creating a new circuit link via the start page of the simulator). Next, the user can add the input and output components to the Protoboard. This is again done by dragging the items from the Component Drawer to the circuit display.

Once the user has created the context for the circuit they intend to implement by adding the large components as well as the starting and ending points, the design process can begin to take an iterative approach where the user “fills in the pieces.” The Datapath Simulator allows users to view the data associated with each component in the simulation during development without implementing the entire datapath from input to output first, so the user can simply add components and continue to view the modified values until they are satisfied with the functionality of the circuit. Certain features, such as being able to view wire and component input and output values, make testing while developing much easier, which helps users avoid bugs and the need to rework their designs continually.
In this chapter, I present three of the datapaths that students build and learn in CS233.

7.1 ARITHMETIC MACHINE

Figure 7.1: The basic MIPS Arithmetic machine performs simple ALU operations on two register values.

By the end of week four of CS233, students have followed along as the instructor has built the MIPS Arithmetic Machine in lecture, and students have completed a Verilog implementation of the machine in the labs. The MIPS Arithmetic Machine is the first practical datapath that students learn, and it has the ability to perform register-to-register arithmetic and logical instructions.

Major parts of this datapath that students learn include the program counter and instruction memory area, the Decoder and Register File area, and the ALU area.

Toward the left of the datapath there is circuitry to control the advancement of the MIPS Program Counter. Since this basic machine does not include the ability to branch or jump, the Program Counter can only advance by four bytes at each clock cycle. An ALU is provided with the add signal and input B hard-coded to the value 4. The output of the Program
Counter is connected to the Instruction Memory, which sends the specified word of instruction memory toward the next part of the datapath.

In the middle portion of the datapath, Bus Subsets (a simpler version of Bus Mappers) divide the 32-bit output from the instruction memory into the fields specified in the R-type and I-type instruction formats. The rsAddr and rtAddr fields are directed to the Register File, while the rdAddr is chosen based on the signal from the MIPS Decoder.

The MIPS Decoder has been connected to the opcode and funct fields, and thus can send the correct control signals to each of the components on the datapath.

Finally, the ALU has been connected to both rsData and either rtData or the sign-extended immediate value, depending on the control signals set by the Decoder. Since this machine simply reads from, performs operations on, and writes back to registers, the ALU output is connected directly to the rdData port on the Register File, allowing it to write the instruction's result data back to the Register File.

7.2 FULL MIPS MACHINE

![Diagram](image)

Figure 7.2: The single-cycle MIPS Datapath implements the full MIPS machine.
By the end of week five in CS233, students have built the entire single-cycle MIPS machine, which is capable of executing R, I, and J type instructions, including loads, stores, branches, and jumps.

Changes to this machine from the Arithmetic Machine include expanding the ability to modify the Program Counter, adding the Data Memory “stage” after the ALU, and having the ability to store memory, ALU results, or an immediate value back to the Register File.

A multiplexer has been added on the top-left of the datapath which allows the datapath to choose between incrementing the program counter by four bytes, incrementing by a branch offset, replacing the program counter value with an immediate value, or replacing the program counter with a value from a register. This change allows the datapath to follow branches, static jumps, and register jumps as well as incrementally move through lines of assembly code.

On the right side of the datapath, we have added the Data Memory component. This component has the ability to read from memory either a byte or a word at an address computed by the ALU or store data from the Register File’s \texttt{rtData} port. Circuitry has also been added to allow the Decoder to choose between saving data to the Register File from the ALU, Data Memory, or an immediate value.

7.3 FULL PIPELINED DATAPATH

![Figure 7.3: The Pipelined MIPS Datapath divides the single-cycle machine into five stages.](image-url)
In the Fully-Pipelined Datapath, students have divided the MIPS Processor up into five stages—the Instruction Fetch, Decode, Execute, Memory, and Writeback stages. Pipeline Registers have been added between each stage to facilitate storing information processed in each stage between clock cycles. Currently, the Datapath Simulator does not support a Forwarding Unit.
CHAPTER 8: USE CASES

For the CS233 Datapath Simulator, I wanted to create an interface that was as familiar to current students as possible. A familiar interface primarily means that the components in the simulator are modeled heavily off of the current class handouts and lecture slides, but also implies that certain user interactions are based on similar paradigms that have emerged as the dominant methods for interaction with digital models.

I foresee the Datapath Simulator being mostly useful in a few different cases in our hardware architecture class.

8.1 DURING LECTURES AS A VISUAL AID

While lecturing, the instructor for CS233 often displays PowerPoint slides with a static version of the current class datapath and annotates them live while speaking. Once a concept has been explained, the instructor flips to the next slide and repeats the process.

One of the issues with this current method is that the slides, when annotated, become crowded and hard to read. For students who view the annotated slides (which are posted online) after the class, it is hard to understand which drawings came first and what the context of the annotation was. Additionally, when the instructor flips to the next slide to advance the datapath diagram, all of the extra annotations disappear from the screen, causing a loss of continuity between portions of the lecture.

To address these issues, I propose using the Datapath Simulator for live-building the datapath during the CS233 lectures. This way, students can see the actual development of the circuit into what becomes a final “production” diagram—one without ink marks over the top of it. Additionally, instead of annotating values on wires, the instructor could simply click on the wires to show the actual values.

Further, the instructor could distribute links to the starting circuit so that students could follow along in the development of the datapath circuitry. Currently, I have heard from some students who say they trace the datapath diagram out in their notes from the lecture slides as a memory and understanding aid. By giving students the ability to develop the datapath themselves during a lecture, they would be able to follow along with the values on the datapath as they wished and have a fully-developed circuit at the end of the class for future reference.
8.2 IN OFFICE HOURS

During CS233’s office hours, teaching assistants often use paper diagrams of the datapath in the same way as during the lectures to explain concepts to students. Oftentimes, people come to office hours without an understanding of how data moves through the datapath, so teaching assistants will annotate a paper diagram with the data and operations happening in the circuit for a given set of instructions.

One issue with this method, however, is that the paper diagrams can only realistically demonstrate one clock cycle at a time, and they are generally discarded after working with each student since we find that showing the students the process of writing out the information flow is helpful in fostering an understanding of the concepts.

By using the Datapath Simulator as a teaching aid in 233’s office hours, I hope to be able to help show students the data flow on circuits and the construction of the datapath more efficiently and with less waste.

8.3 AS A STUDY TOOL

For the same reasons as above, I believe students could benefit from having access to the Datapath Simulator for use as a study tool. I propose releasing links to pre-built circuits on our class resources page, which students can then explore and modify at their leisure.

8.4 AS AN EXAM RESOURCE IN THE COMPUTER BASED TESTING FACILITY

One complaint we have continuously heard from students regarding CS233 is that we don’t provide a paper copy of the datapath diagram with our Verilog-based datapath exam. Instead, we post a link to a PDF version of the diagram that is available in the Computer Based Testing Facility. This is due to cheating concerns about people who sneak a paper with answers into the exam room, but the result is that students often spend time tracing the PDF from the screen onto a blank sheet of paper before figuring out the correct way to modify the datapath for the exam using the paper version, and then translating that back to Verilog.

However, with the Datapath Simulator, we could provide students with an editable, testable datapath during the exam so that they could spend less time fiddling with copying the PDF to scratch paper and more time working on completing the exam objectives.
8.5 FOR COLLABORATIVELY BUILDING CIRCUITS IN CLASS

One way in which CS233 tries to encourage learning is through working collaboratively in discussion sections to solve simple problems before sending students off to work on a lab assignment. With the Datapath Simulator, it would be possible to facilitate the same types of collaborative learning in larger gatherings, such as during a lecture.

In such a situation, one student could “drive” the Simulator, while one or more other students could talk through the problem and instruct the driver on which components to add and busses to connect. I believe that using the Simulator in this way during lecture would help students engage with the material more quickly and therefore learn it more comprehensively.
CHAPTER 9: USER TESTS AND RESULTS

To better understand students’ desire to use a visual tool for learning the MIPS datapath, I conducted a survey of, and interviewed, 23 CS233 students during the Spring 2020 semester, before CS233’s Exam 3—the exam covering the datapath. I asked a variety of qualitative questions as well as quantitative ranking questions to determine whether or not this tool is useful for enhancing students’ learning.

9.1 USER SURVEY

For the survey, I asked students from CS233 to come into office hours and sit down for 15-20 minutes apiece. First, I asked the students two questions to gauge their current preparedness on the course material covering the MIPS datapath:

- On a scale of 1-10 (10 being the highest), how well do you think you understand the MIPS Datapath?
- On a scale of 1-10, how well prepared for exam 3 do you think you are?

Then, I asked students to complete a task using the Datapath Simulator that was similar in scope to the types of tasks they would be asked to complete on Exam 3. Finally, I asked them some questions about the Datapath Simulator:

- On a scale of 1-10, how much interest do you have in using a similar tool in this class?
- On a scale of 1-10, how much do you think using a visual tool like this one would have helped you better prepare for this exam?
- In which areas of the class do you see a tool such as this being the most useful?
  - Lecture
  - Discussion
  - Labs
  - Prarielearn
  - CBTF
  - On your own (ie, tool is made available to you for studying)
- On a scale of 1-10, how much do you think using a visual tool such as this in CS233 would improve your understanding of the MIPS datapath?
- Would you rather use a visual tool such as this than Verilog in each of these parts of the class? (y/n)
  - Lecture
Discussion

Labs

Prairielearn

CBTF

- Please explain your reasons for preferring either this tool or Verilog above
- What things do you think this tool does well?
- What suggestions do you have to make this tool better?

9.1.1 The Task

Before asking students to tell me about their experiences with the datapath simulator, I asked them to implement the \texttt{srl} instruction on the MIPS datapath. The \texttt{srl} MIPS instruction performs a bitwise shift to the right on a specified register’s value of a specified number of bit positions.

My version of implementing the \texttt{srl} instruction was shorter than what students would have been asked to do on Exam 3 if the \texttt{srl} instruction were a real test case. In the given implementation, the ALU and MIPS Decoder already supported the \texttt{srl} instruction. Students only needed to add a path for the \texttt{shamt} field in the instruction data to the ALU’s \texttt{A} input.

To complete the first part of the task, the students first had to add a new Bus Subset from
the Component Drawer to the Protoboard. Then, the students had to connect the 32-bit MIPS Instruction bus to the input of the Bus Subset component. To do this, the students clicked the bus to enter edit mode, then double clicked a connector and dragged it to the port on the Bus Subset.

Next, students needed to add a 2-to-1 multiplexer to the Protoboard by dragging it from the Component Drawer. The input 0 on the mux was connected to the rsData port on the Register File, and the input 1 on the mux was connected to the output from the new Bus Subset component by clicking and dragging the ports on the components to create a Bus.

Finally, the students had to connect the srl port on the MIPS Decoder to the control input on the multiplexer. This ensured that the original datapath remained intact while the new instruction was able to execute when the Decoder’s srl signal was 1.

I found that most students required assistance with implementing the instruction—primarily in the understanding of the task that they needed to complete on a conceptual level. I helped students along conceptually when needed.

The second part of the task I asked students to complete involved setting register values (of their choice) via the Register File interface and executing the instructions pictured in the Instruction Memory component.

```
add $4, $2, $3
and $7, $4, $5
srl $8, $7, 2
```

To set register values, I asked students to open the Register File’s editor mode. This was done by clicking the Show Values button on the Register File component, however I found that many students failed to notice the button and required some help to find the editing mode. From there, students could simply type values in hex, binary, or decimal into the register value input boxes, which I found was intuitive for students.

Then, I asked students to execute the instructions from the Instruction Memory and view the resulting updated register values. To do this, students had to click the Adopt State button in the Code Bar, then click in the clock cycle diagram to advance the clock at least three cycles. I also found that students needed assistance with this, as the state adoption requirement was unclear to them. However, once the clock cycle diagram appeared after adopting a starting state, students were able to find the resulting register values by clicking the Register File component.
9.1.2 Quantitative Results

Preparedness for the exam

To gauge how well-prepared the students taking the survey were for exam 3, I asked them to rank on a scale from 1-10 (10 being highest) how well they believed they understood the MIPS datapath. 74% of the students responded with a value of 7 or above. Of the responses, 7 was the most common answer (n=11), and 5 was the minimum answer. The full results from this question are presented in figure 9.2.

![Figure 9.2: Responses from 1-10 (10 being highest): How well do you understand the MIPS datapath?](image)

I also asked students directly how well-prepared they believed they were for the exam (figure 9.3) to gauge the confidence they had in their knowledge of the material. The most common response was 6 (n=6), and the least prepared students (n=2) responded with a 3. Nobody responded that they were maximally prepared for exam 3, and two students did not respond to this question since they had already taken exam 3.

![Figure 9.3: Responses from 1-10 (10 being highest): How well-prepared for exam 3 are you?](image)
**Internal Consistency**  To gauge whether the previous two questions accurately measured students’ preparedness for the MIPS datapath exam, I calculated the internal consistency of the results between the two questions. This yielded a Cronbach’s alpha score of 0.818, revealing high reliability.

Students’ perception of the Datapath Simulator

After showing students the Datapath Simulator and asking them to complete the `sr1` implementation task above, I asked students how they felt the datapath simulator could help them in this class.

The first question I asked students was “how much interest do you have in using a tool similar to this in CS233?” Most students (n=14) answered a 9 or a 10. Only 3 students answered less than a 7. These results are presented in figure 9.4.

Figure 9.4: Responses from 1-10 (10 being highest): How much interest do you have in using a tool like this in this class?

I saw similar responses when asking the question “how much would using this tool have helped you prepare for exam 3?” (figure 9.5) The majority of students again answered either a 7 or above.

For my final question: “how much would this tool help improve your understanding of the MIPS datapath?” (figure 9.6) I also received similar answers from students. Again, the majority chose 7 or higher.

**Internal consistency**  To gauge whether the previous three questions accurately measured students’ perceptions of the Datapath Simulator, I calculated the internal consistency of the results between the three questions. This yielded a Cronbach’s alpha score of 0.818. When considering only the previous two questions (how well would the simulator prepare students
for the exam and help understanding of the datapath, respectively), the Cronbach’s alpha score was 0.832, again showing the reliability of the survey data.

**Interpretation** I found that both students who felt they understood the MIPS datapath better and students who had a higher confidence on exam 3 tended to rank the tool’s usefulness higher than those who did not.

Of students who responded 7 or higher to the first question, the average responses to the latter three questions were 9.05, 8.47, and 8.52 respectively. Those who answered lower than 7 ranked the tool 6.67, 6.83, and 7.33 respectively.

A similar trend exists for students who responded 7 or higher to the second question. Their average responses to the latter three questions were 8.70, 8.90, and 8.40 respectively. Those who answered lower than 7 ranked the tool 8.45, 7.72, and 8.27 respectively.

Overall correlation between whether students felt prepared and that they understood
the MIPS Datapath to whether they thought the Datapath Simulator would be helpful (in preparing for the exam, for learning MIPS, and in general interest in adding it to the course) was relatively low, however. I averaged each student’s answers to the first two questions and their answers to the last three questions, and found the correlation between the two sets of questions to be 0.307.

As a replacement to Verilog

In addition to the ranking questions, I also asked students to tell me whether they would like to see the Datapath Simulator used in various portions of the class instead of Verilog. The results are displayed below in figure 9.7.

![Figure 9.7: Percentage of students who would rather use the Datapath Simulator than Verilog in each portion of the class.](image)

While the data above suggests that students would rather use a visual tool than Verilog, the written feedback tells a more nuanced story.

9.1.3 Written Feedback

As well as the ranking questions presented above, I asked students to give qualitative written feedback on why they preferred (or did not prefer) this tool over Verilog, what they think the tool does well, and what they think could be improved about the tool.

Preference over Verilog

In all, 14 students mentioned in the written feedback that they preferred using a visual tool such as the Datapath Simulator in place of Verilog in general. The most stated reason
for this was they thought that the Datapath Simulator’s visual representation of circuits was clear and easy to understand (n=21). In addition, many students (n=12) also said they thought Verilog was complicated or hard to understand. While many students seemed to dislike Verilog, a majority of students still think Verilog is worth learning in CS233. **Fifteen** respondents explicitly stated that they would prefer to use the Datapath Simulator as a way to double-check their understanding of the concepts before delving into a Verilog implementation, and 6 students thought that replacing key assessments (such as exams) based on Verilog with the Datapath Simulator would make the class too easy.

**Interpretation** Some of the written qualitative results are hard to square with the quantitative responses above. I think that the majority of students would like to see the Datapath Simulator used on exams and in labs as a means of checking students’ understanding before committing to an implementation. A few students mentioned that they would like to see the simulator provided during an exam in place of the PDF copy of the single-cycle MIPS machine that is currently available to download in the CBTF.

Among the reasons for wanting to have access to the Datapath Simulator on exams was that the single-cycle MIPS PDF file was not editable during the exam. This, students said, made it hard for them to keep track of the changes they wanted to make to the datapath visually while completing the exam. Multiple students mentioned using valuable exam time to trace the PDF onto paper so that they could modify the datapath diagram. I see this as a good indication that a more visual approach to education involving the MIPS datapath would be beneficial.

Other comments

In addition to whether students preferred the Datapath Simulator to Verilog, I received a number of other insightful comments from students:

A number of students (n=9) mentioned that they think the Datapath Simulator would be especially useful if it was provided during lecture and discussion times. Currently, many students taking CS233 either trace over paper diagrams during the lectures or use a tablet computer to markup the immutable handouts and slides provided to them. Students feel this is a good way to commit datapath concepts to memory, however, modifying the datapath via pencil or stylus is sloppy and harder to read at a later date.

The most useful features students said they appreciated in the Datapath Simulator were being able to view register data (n=8) and tracing wire value changes at each clock cycle.
(n=6). Students also appreciated that being able to move components around visually and click and drag to connect wires between components was more similar to dealing with physical hardware implementations on a breadboard than Verilog (n=3).

Improvements

Although I showed students an early prototype of the Datapath Simulator that still included many bugs, students were able to point out a few features that would make useful additions to the software.

Six students wanted to see a tutorial implemented for the Datapath Simulator as they were unsure of how to access or use several features. The most significant hurdle I encountered was the understanding of how the “adopt state” button works for determining a starting simulation state from which to simulate clock cycles. Another issue I encountered was students not realizing that buttons on components were clickable. For example, the Register File component contains a “show values” button that when clicked launches a popup for displaying register value. It seems that students were sometimes unaware that the components on the datapath were interactive beyond moving them around and connecting wires.

Four students suggested that I implement color coded wires and components. They thought that the traditional black-and-white diagrams featured in the CS233 handouts (on which the Datapath Simulator is modeled) had the potential to become muddled when modified extensively. Being able to set the color of components, they said, would allow them to more quickly distinguish features of interest on the datapath. (For example, coloring all new additions red would help one student more clearly remember which modifications to make to the Verilog code on exams.)
CHAPTER 10: CONCLUSION

Through the development of the Datapath Simulator, I have demonstrated that it’s possible to build a system that can potentially aid learning in hardware architecture classes by visually allowing students to build and modify a datapath, and by providing the tools necessary to simulate instructions executing on that datapath. The survey shows that the Datapath Simulator is a strong candidate for inclusion in future semesters of CS233, and I hope that by intentionally reducing the cognitive load on students as they learn, and by providing visual exploratory tools for hardware architecture concepts, that I can help students achieve mastery of these computing skills in ways that were not possible previously.

10.1 FUTURE WORK

Although the current Datapath Simulator is a good start on functionality and usability, the system is still in its early stages. I have identified several areas where the Datapath Simulator can improve in the future to offer a more comprehensive set of tools for teaching hardware architecture.

10.1.1 Interface

Zoom One area where the Datapath Simulator can significantly improve in a short amount of time is by adding zooming functionality to the Protoboard View. Currently, the view displays HTML elements for each of the components on the Protoboard. View zooming can be accomplished through a CSS `transform: scale()` style addition. The interface interaction methods would need to be updated to take into account mouse position on the scaled interface.

Adopt State Paradigm Another improvement can be made to the confusing “Adopt State” button paradigm. Students have said they would like to see a tutorial implemented in the Datapath Simulator to explain the need for the button and its actions, but future work may also refactor the interaction paradigm to not require users to set a starting state for forward clock cycle simulation. Instead, the simulation could keep the state values for each component in a data store and could replace the values in the components when needed. Any change to the Datapath would simply start the simulation over from initial values specified by the inputs.
10.1.2 Additional Functionality

**Prairielearn Integration**  Future work could see adding the Datapath Simulator as an assignment mechanism to Prairielearn. This would enable instructors to ask graded questions that involve students setting up specific pathways on the datapath. Work would include migrating the `window` attached methods and changing the CSS styles to fit within the Prairielearn frame, as well as adding a scoring mechanism and exporting final score data to Prairielearn.

**Move to a Verilog Simulation Backend**  In the future, instead of simulating datapath modules natively, the Datapath Simulator could be migrated to a visual extension of Verilog. It could build and run Verilog code as a way of simulating the datapath, then display the outputs from the simulations instead of modeling the components directly.

**Add Subcircuit Support**  Currently, the Datapath Simulator supports only one level of circuit design. Additional functionality could be added which can import previously-implemented circuits as components that can be added to the datapath at a “higher level” of circuit design. At the moment, new component implementation must be done by the software developer in Javascript.
REFERENCES


