PIPS: An Instruction Set Architecture for Teaching Computer Organization

Charlie Curtsinger
curtsinger@grinnell.edu
Grinnell College
Grinnell, Iowa, USA

Jerod Weinman
jerod@acm.org
Grinnell College
Grinnell, Iowa, USA

ABSTRACT

CSC 211: Computer Organization and Architecture at Grinnell College introduces computer science students to the basics of digital circuits, logic design, and computer organization/architecture. This course is designed to help students develop a deeper understanding of how processors function, and how their design can impact the code they write. During the course, students build components like adders, multiplexors, ALUs, and registers with real circuits, and larger components in a digital logic simulator (Logisim). This progression culminates in a four-week lab sequence where students create an assembler and datapath for the *PIPS Instruction Set Architecture*, which we have designed specifically for this course.

In this paper we describe the design and specific learning goals of the PIPS architecture, the four-week lab sequence where students implement a working PIPS assembler and datapath, and our experiences using this lab sequence for the past three years. All student starter materials and instructions for these labs are available at DOI:11084/10426, with solutions and grading infrastructure available to instructors upon request.

CCS CONCEPTS

• **Computer systems organization** → **Reduced instruction set computing**; • **Applied computing** → *Education.*

KEYWORDS

instruction set architecture, computer organization, CS education

ACM Reference Format:
Charlie Curtsinger and Jerod Weinman. 2021. PIPS: An Instruction Set Architecture for Teaching Computer Organization. In *WCAE '21: Workshop on Computer Architecture Education, June 17, 2021.* ACM, New York, NY, USA, 8 pages. https://doi.org/11084/10437

1 INTRODUCTION

Computer science majors at Grinnell College take a single course on computer organization and architecture, CSC 211. This course is intended to give students an understanding of how the computer executes the programs they write, from assembly language down to the transistor level. One important component of this course is a sequence of datapath labs, where students build a working processor implementation in Logisim and an assembler to generate machine code for their datapath. Our goal with this lab sequence is to cement students' understanding of how all the components of a datapath work together to execute instructions. These labs give students the experience of writing and executing programs on a complete system they built themselves. To achieve this in just a few weeks of a single architecture course, we have designed a purpose-built instruction set architecture (ISA) we call PIPS.

PIPS is a RISC architecture inspired by MIPS. We chose MIPS as a starting point because this architecture is the focus of our course textbook [22]. While PIPS assembly bears a striking resemblance to MIPS, the instruction set, machine code representation, and architectural details are distinct from MIPS. We have designed PIPS to be easy to *implement*, while retaining many of the interesting challenges in building a working datapath and assembler for a more full-featured ISA. These simplifications allow students to complete this lab sequence during 3–4 weeks toward the end of the semester.

The six-part lab sequence described in this paper asks students to implement a single-cycle PIPS datapath in Logisim [8], along with an assembler to generate machine code for that datapath. The completed datapath and assembler are capable of handling arbitrarily complex programs (provided they fit in the 16-bit address space) including programs with procedure calls, stack allocation, and terminal input and output using memory-mapped I/O. With the exception of three instructions provided as examples, students write the translation and syntax rules for every PIPS instruction their datapath executes. Students make all of the connections between datapath components and design their own simple LUT-based microcode control scheme for the datapath.

This paper describes the design of the PIPS instruction set architecture, the six-part lab sequence students follow to implement a PIPS datapath and assembler, and our experience using these labs over three iterations of CSC 211. The starter circuit, assembler framework, and all lab instructions are available at DOI:11084/10426; solutions and grading infrastructure available to instructors upon request.

2 RELATED WORK

Early work in computer architecture education presented hardware design labs guiding students for building a working computer on real hardware [10, 29].

Several works over the last few decades have created and described basic or simplified machines for teaching primary ideas in computer architecture. These include the ANT processor for learning about assembly code and data representation [12], the rudimentary machine "RM" [20], and ESCAPE [28], among others.

WCAE '21, June 17, 2021,

https://doi.org/11084/10437

Of course several simulators have been developed around pedagogical or otherwise relatively simple RISC ISAs. These include LC-3 [7, 21], DLX [5, 16], RISC-V [2, 14, 18], and MIPS [17, 25].

As an alternative to processor design and simulation requiring a graphical circuit layout, some have proposed pedagogical hardware description languages (HDLs) [19, 24]. Students design the DINO CPU [18] for RISC-V using the full-featured Chisel language [3].

A related approach allows students to expand variations for exploring design trade-offs in FPGA [9, 15] or simulation [26].

Some combine approaches. WRAMP provides both a simulation and custom hardware board implementation for a simple MIPS-like ISA; student work targets the platform [23]. DOP [6] and ANT-32 [13] are similar, if slightly more complex. S3 uses a given ISA, and students build the components from a brick-like toolkit and flash to an FPGA using mostly schematics with only basic VHDL [11]. At the other end of the spectrum, some allow students free reign on an FPGA to design the (best) processor for a given ISA [27].

Our approach provides a simulation-only option for institutions where hardware is not an option. Because the PIPS ISA and datapath eschew the more complicated features (i.e., pipelining, traps, interrupts, etc.) of even textbook ISAs (e.g., DLX and LC-3) students can easily and quickly build most all of the simulated hardware and software stack themselves. By comparison, one independent student project reports building a user-controlled LC-3 datapath in Logisim [4], but the instruction decoding proved too difficult to complete in the time available.

Moreover, the use of a logic simulator based on circuit layout—rather than a hardware description language—forces computer science students more accustomed to imperative programming to instead confront the complexities peculiar to circuits. As Lowe-Power and Nitta [18] have pointed out, while HDLs make it easy to declare circuit connections, they can mask misconceptions carried by a programmer's mental model.

Contrary to their experience, our students find Logisim a generally intuitive, reliable, and easy-to-use platform for logic design. Because it is free/libre, open source, and implemented in Java, the program is widely available and portable. Unfortunately, active development on the original Logisim project ceased in 2011, spawning several forks. We use a fork of Logisim (v2.7.2) created for the CS3410 course at Cornell[1] because it supports test vectors for autograding. While that feature has since been merged into Logisim evolution,[2] which seems presently to be the most active fork, we have not yet attempted a migration.

Though they might not appear in the published literature, others have certainly asked students to build a datapath in Logisim. By packaging the novel pedagogical ISA description and labs here, we hope to make it easy for many others to adopt this practice and give their students the benefit of building a complete compute system.

3 CONTEXT

We briefly review the institutional context of the host major and course, the curricular role of the course itself, and where the assignment sits among the various course learning goals and activities.

[1] http://www.cs.cornell.edu/courses/cs3410/2015sp
[2] https://github.com/reds-heig/logisim-evolution

3.1 Institutional Context

Grinnell College is a small, private, highly-selective liberal arts college of about 1700 students. Students ordinarily declare their major in the fourth semester, just before registering for courses to begin their junior year. With no explicit distribution requirements, students typically spend their first and second college years exploring possible majors and areas of interest. Hence, students may come to this material in their second, third, or fourth year. The semester runs fourteen weeks (plus one week for final exams), and a normal load is four courses of four credits each. Students do not earn additional course credit for lab components.

3.2 Curricular Context

The CS major at Grinnell features an introductory course in functional problem solving using Scheme, with a follow-on course in the imperative paradigm with C. The first course naturally includes extensive coverage of recursion, and the second course covers pointers and memory management in addition to binary data representation and bitwise operations. These two introductory courses form the minimal prerequisite chain for the course in which this material is presented. However, some students take the course after having completed several other required and/or elective CS courses as well.

Computer Science majors are required to take either this course or a course on Operating Systems and Parallel Algorithms to satisfy a systems requirement for the major. In consideration of the current ACM/IEEE computing curriculum guidelines [1], we recommend students take both; institutional limitations on the number of credits required by a major keep us from requiring both courses.

Graduates obtain a Bachelor of Arts in Computer Science.

3.3 Course Context

The course is taught in a workshop style, with several topics covered by laboratory work integrated with course meetings, rather than a separate lab section. While the PIPS labs are done collaboratively (in pairs) and mostly during class with the assistance and supervision of the course instructor and a lab assistant, they could also easily function as a traditional multi-week homework assignment. Each section of the course is limited to 24 students, due to the hardware constraints in other portions of the course.

Although work in the current course includes physical circuit design, it is not a required prerequisite for PIPS. Our students start by building logic gates from transistors. They then use TTL logic gate ICs to construct a simple 1-bit full adder, a 2-bit decoder and 4-way multiplexor leading to a 1-bit ALU. They finish by building a TTL gate-based SR latch, D latch, and D flip-flop leading to a 2x1 register file. In addition to these physical circuits, before coming to PIPS, they build wider versions of an ALU and a register file in simulation from higher-level components (i.e., adders, decoders, multiplexors, D flip-flops). Building on top of the digital logic, they also program a PIC32 microprocessor in MIPS assembly, writing loops, calling functions, and using I/O pins to read switches, light LEDs, and play a song. Later in the term (*after* completing PIPS) they also build a small memory cache controller in simulation.

The course's supporting textbook is Patterson and Hennessy's MIPS-based computer organization and architecture book [22], including pipelined processor design and control.

4 PIPS OVERVIEW

PIPS is a 16-bit ISA with 15 general-purpose registers, 16 instruction opcodes, and a fixed-width 32-bit instruction encoding. Students implement PIPS using a single-cycle, microcode-controlled datapath with memory-mapped I/O for keyboard input and terminal output. The human-readable PIPS assembly language students write is modeled on MIPS. However, PIPS uses an entirely different machine code format and is more restricted both in the number of registers and single-instruction operations than MIPS. These simplifications are deliberate; reducing the complexity in encoding and decoding PIPS instructions makes it possible for students to implement a working PIPS assembler and datapath in four weeks of the course.

The following sections describe the available PIPS instructions, the components and general structure of the PIPS datapath, and the PIPS assembler.

4.1 Instructions and Encoding

The PIPS instruction set closely mirrors MIPS, with some changes that simplify implementation. The PIPS instruction set is encoded using two possible instruction formats: i-format and r-format (shown in Figure 1). The i-format instructions encode operations that take a register and 16-bit immediate value as parameters, while the r-format instructions take two register parameters. Unlike MIPS, PIPS does not use a different format for jump instructions.

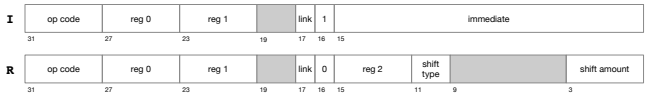

Figure 1: Two PIPS instruction formats: immediate (I) and register (R).

The four-bit register fields in PIPS instructions encode one of the 16 available registers. While PIPS has fewer registers than MIPS, they follow the same naming scheme and calling conventions.

Table 1: PIPS registers

Name	Number	Use	Call Preserved
`$zero`	0	Constant (16'b0)	read-only
`$v0`	1	Function return value	No
`$a0`–`$a3`	2–5	Function arguments	No
`$t0`–`$t3`	6–9	Temporaries	No
`$s0`–`$s3`	10–13	Saved registers	Yes
`$sp`	14	Stack pointer	Yes
`$ra`	15	Return address	Yes

Both r-format and i-format instructions use the same four-bit opcode field, which selects one of the 16 available PIPS opcodes: add, sub, and, or, nand, nor, xor, slt, sltu, lb, lw, sb, sw, beq, bne, and j. Every opcode is usable in both r-format and i-format instructions; ALU operations like add perform register–register addition when encoded in r-format, and register–immediate addition in i-format. The only operation limited to a single format is bit shifting, which is only available for r-format instructions because the shift type and amount are encoded in space reserved for the immediate field of i-format instructions. PIPS does not need unsigned variants of most ALU operations (slt is the one exception) because the 32-bit instruction leaves space for a full 16-bit immediate value, making sign extension unnecessary. Table 2 shows the complete list of instructions available on PIPS, and Table 3 shows the pseudo-instructions composed of multiple PIPS instructions.

Table 2: PIPS instructions

Instr	Fmt	Semantics	Op
`add`	R	`R[R0] = R[R1] + R[R2]`	0
`addi`	I	`R[R0] = R[R1] + immediate`	0
`sll`	R	`R[R0] = R[R2] << shamt`	0
`srl`	R	`R[R0] = R[R2] >>> shamt`	0
`sra`	R	`R[R0] = R[R2] >> shamt`	0
`sub`	R	`R[R0] = R[R1] - R[R2]`	1
`subi`	I	`R[R0] = R[R1] - immediate`	1
`and`	R	`R[R0] = R[R1] & R[R2]`	2
`andi`	I	`R[R0] = R[R1] & immediate`	2
`or`	R	`R[R0] = R[R1] | R[R2]`	3
`ori`	I	`R[R0] = R[R1] | immediate`	3
`nand`	R	`R[R0] = ~(R[R1] & R[R2])`	4
`nandi`	I	`R[R0] = ~(R[R1] & immediate)`	4
`nor`	R	`R[R0] = ~(R[R1] | R[R2])`	5
`nori`	I	`R[R0] = ~(R[R1] | immediate)`	5
`xor`	R	`R[R0] = R[R1] ^ R[R2]`	6
`xori`	I	`R[R0] = R[R1] ^ immediate`	6
`slt`	R	`R[R0] = R[R1] < R[R2]`	7
`slti`	I	`R[R0] = R[R1] < immediate`	7
`sltu`	R	`R[R0] = R[R1] < R[R2]`	8
`sltiu`	I	`R[R0] = R[R1] < immediate`	8
`lb`	I	`R[R0] = {8'b0,M[R[R1]+ immediate](7:0)}`	9
`lw`	I	`R[R0] = M[R[R1]+immediate]`	10
`sb`	I	`M[R[R0]+immediate](7:0) = R[R1](7:0)`	11
`sw`	I	`M[R[R0]+immediate] = R[R1]`	12
`beq`	I	`if (R[R0] == R[R1]) PC = immediate`	13
`bne`	I	`if (R[R0] != R[R1]) PC = immediate`	14
`j`	I	`PC = immediate`	15
`jal`	I	`PC = immediate; R[15]=PC+4`	15
`jr`	R	PC = R[R2]	15

PIPS supports byte and word (16-bit) memory accesses with the `sb`, `sw`, `lb`, and `lw` instructions. These instructions allow address computation using a register with an immediate offset. There are no alignment requirements for byte accesses, but word memory accesses are silently aligned to two-byte boundaries by rounding the address down to the nearest multiple of two.

Another notable difference of PIPS and MIPS is in the encoding of jump and conditional branch instructions. Unlike MIPS, all branch and jump destinations in PIPS are absolute addresses, encoded entirely in either the immediate field or a register operand. This encoding scheme makes it possible to support the `j` and `jr` (jump and indirect jump) instructions with the same opcode. An additional *link* field in the instruction encoding adds support for procedure

Table 3: PIPS assembler pseudo-instructions

Instr	Semantics
nop	
li	R[R0] = immediate
not	R[R0] = ~R[R1]
push	R[14] = R[14] - 2; M[R[14]] = R[R0]
pop	R[R0] = M[R[14]]; R[14] = R[14] + 2
blt!	R[R0] = R[R0] < R[R1]; if (R[R0] != 16'b0) PC = immediate
ble!	R[R0] = R[R1] < R[R0]; if (R[R0] == 16'b0) PC = immediate
bgt!	R[R0] = R[R1] < R[R0]; if (R[R0] != 16'b0) PC = immediate
bge!	R[R0] = R[R0] < R[R1]; if (R[R0] == 16'b0) PC = immediate

calls; when this field is set, jump instructions store the next program counter value in register `$ra` so the called procedure can return.

The PIPS instruction set maintains many of the familiar idioms in MIPS assembly programming, which aligns well with students' prior experience writing MIPS assembly in the course. Other than the omission of multiplication, division, and synchronization instructions, PIPS's primary difference from MIPS is in its encoding. PIPS's less compact encoding simplifies the instruction decoding process while retaining many of the opportunities for clever datapath design available in MIPS.

4.2 Datapath and Control

Figure 2 shows an overview of the completed PIPS datapath. Students are provided with a starting circuit file that includes a program counter register and adder, instruction memory, uninitialized microcode ROM, register file, ALU, data memory, and a memory controller subcircuit to handle PIPS's alignment requirements. Students are responsible for connecting and controlling these components to support all of the PIPS instructions.

Many of the datapath connections in PIPS mirror those of a MIPS datapath students see in their course textbook, but the PIPS datapath control scheme allows students significantly more flexibility. The core of the control scheme is the microcode ROM, which provides space for 16 bits of control signal for each of PIPS's sixteen instruction opcodes. Students are responsible for determining the control bits encoded in their ROM entries, a lookup table indexed by opcode. Typical implementations require control lines to select the ALU operation, the source of the next program counter value, and memory access size, as well as enabling writes to a register or memory with a handful of additional bits to control routing of register or immediate values.

In addition to the data and control connections, students must add support for three special memory-mapped operations. PIPS treats jumps to the special address `0xff00` as a halt, eliminating the need for a special halt instruction. To support terminal input and output, PIPS uses memory-mapped I/O to print to a console and read from a buffered keyboard input. Values stored to the address `0xff10` are sent to the console as output characters, and reads from `0xff20` consume one byte from the keyboard buffer. These basic memory-mapped I/O facilities are sufficient to support a handful of programs students write in PIPS assembly, assemble using their PIPS assembler rules, and then execute on their completed datapath.

4.3 Assembler

As students complete their PIPS datapath they also implement an assembler to generate PIPS machine code from PIPS assembly language. The assembler is written in Python, but the provided code—approximately 250 lines—handles the assignment of symbols to addresses, offers rudimentary linking of multi-source projects, and supports student-defined syntax and translation rules. The output of the assembler is a `.hex` file[3] that can be loaded into the instruction memory in the Logisim datapath. The provided code also includes utility functions to encode the PIPS i-format and r-format instructions using register and opcode names. Students in this course are not expected to be familiar with Python; the provided assembler code makes it easy for students to write rules for translating assembly instructions to machine code with just a few lines of code for each instruction.

As students add support for new instructions to their datapath they must add rules to translate those assembly instructions to machine code. Each rule uses a Python *decorator* to define the syntax rule for the instruction followed by a function that emits the encoded instruction. The starting materials for this lab sequence include assembler rules for `add`, `addi`, and the `li` pseudo-instruction; students write the translation rules for the rest of the PIPS instruction set along with a small number of pseudo-instructions (see Tables 2 and 3 for a complete list). The complete translation rule for the `addi` instruction is included below.

```
1 @assembler.instruction('addi #, #, #', 1)
2 def addi_instr(dest, op1, immediate):
3   return pips.iformat(opcode='add', r0=dest,
4                       r1=op1, imm=immediate)
```

Line 1 is a Python decorator that specifies the syntax rule for `addi` and registers the function on the next line with the assembler. Each instance of the # character in the syntax rule is an operand wildcard. All other characters are treated as literals that must match the input assembly line exactly for the syntax rule to apply. The second parameter to the decorator indicates the number of 32-bit PIPS instructions this assembly rule will emit; pseudo-instructions may emit more than one PIPS instruction for a single line of assembler input. Reporting this number with the syntax rule allows the PIPS assembler to pre-assign addresses for all instructions and labels in the input assembler file.

Line 2 defines the translation function for the `addi` instruction. This function is called with parameters extracted from the input assembly. For example, the assembly line "`addi $v0, $a0, 4`" would be translated via the call `addi_instr('$v0', '$a0', '4')`. Line 3 uses the provided `pips` module to emit an i-format instruction with

[3]This is a human-readable ASCII file, and it includes the original assembly instructions that generated each line as comments.

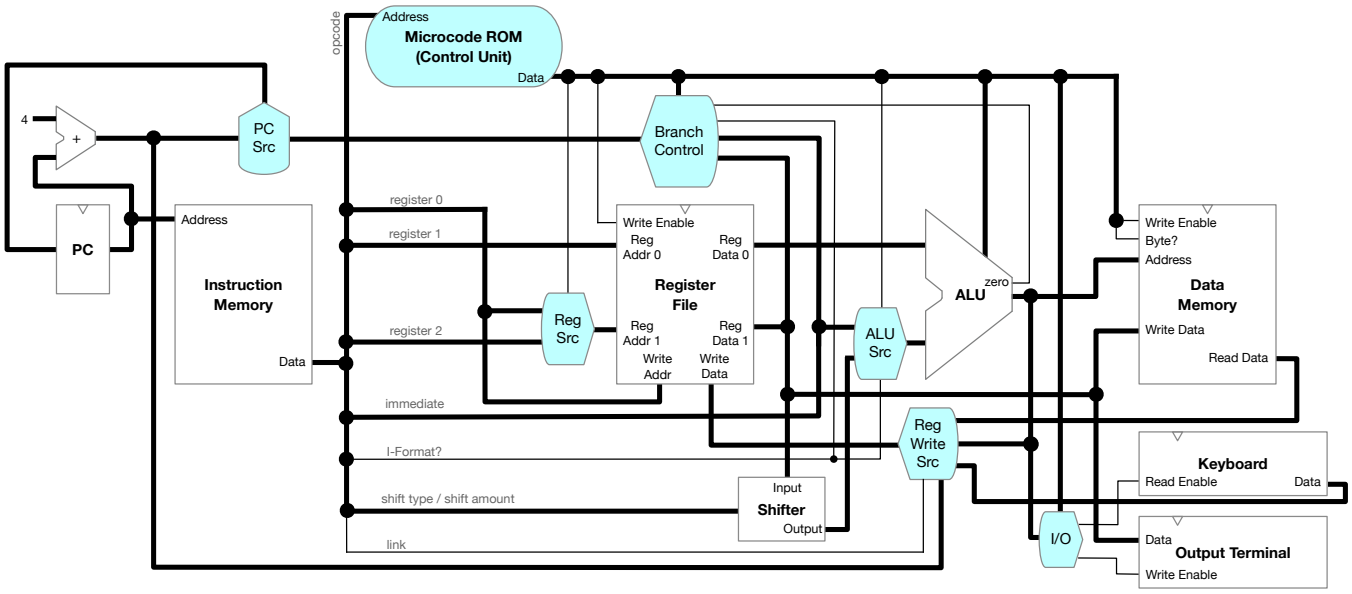

Figure 2: Overview of the final PIPS datapath developed incrementally by students, who must build the control elements and wire all connections. The starter file includes the PC and its adder, both memories, microcode ROM, register file, and ALU.

the requested field values. The `pips` module validates inputs and handles the mapping from register and opcode names to numbers.

Because pseudo-instructions ought to be expressed as a sequence of other instructions, we encourage students implement their pseudo-instructions with calls to their own Python functions, representing a translation to (real) instructions, rather than with direct calls to `pips.iformat` or `pips.rformat` helpers. The example translation for `li` (load immediate) is therefore given in terms of the `addi_instr` function defined above:

```
1 @assembler.instruction('li #, #', 1)
2 def li_instr(dest, immediate):
3   return addi_instr(dest, '$zero', immediate)
```

For pseudo-instructions that emit multiple instructions, the calls in line 3 are simply concatenated with the Python + operator.

We have encountered few issues with the assembler component of this lab sequence. Students' unfamiliarity with Python is not an issue. Nearly all questions about the assembler portion of the lab sequence are related to the encoding of specific PIPS instructions. We find that the assembler portion of these labs avoids some confusion, as students keep instruction encodings in mind as they add to their datapaths.

Writing the assembler rules gives students an opportunity to build a full system stack, from the programs they write in assembly down to the individual control and data lines in their datapath.

5 LAB SEQUENCE

Students implement the PIPS datapath and assembler over the course of six labs, typically completed over three or four weeks. We assume students are comfortable with basic logic gates and common structures like multiplexors and have experience using these components in Logisim. Students are expected to write PIPS assembly programs as part of the lab sequence, so familiarity with MIPS or another RISC ISA would be best.

5.1 Learning Goals

By the end of the lab sequence, we expect that students:

(1) can devise encodings of useful instructions given their functional requirements,
(2) understand the purpose of connections between components in a datapath,
(3) can determine the necessary control line settings in the datapath for their instructions, and therefore
(4) obtain a holistic view by implementing a full system, from the assembly programs they write to the individual connections in the datapath.

Using the novel PIPS ISA for these labs ensures that students *use* their understanding of computer organization to complete their implementations, rather than simply reproducing a reference implementation from diagrams.

5.2 Detailed Overview

The six parts of this lab sequence guide students through the construction of a complete single-cycle PIPS datapath. These labs are typically spread over three or four weeks of a semester.

As noted in Section 3.3, this lab sequence builds on prior labs where students implement an ALU, register file, and other key components of the datapath that are provided at the start of this lab sequence; while this is not necessarily required, it serves to emphasize learning goal 4, as students will already know how to build the components included in the starting materials for this lab sequence. We describe each of the six lab stages below.

Part 1 – Addition: Students begin by wiring their datapath to support the `add` and `addi` instructions. The assembler rules for these instructions are provided. However, using the microcode ROM and instruction fields, students must design and implement a datapath control scheme that performs the addition of either two registers or a register and an immediate value.
Part 2 – ALU Operations: Students complete the remaining basic ALU operations, including subtraction, bitwise operations, and both signed and unsigned integer comparisons. These each require an additional assembler translation rule and ROM entry.
Part 3 – Branches and Jumps: Students add assembler rules, control wires/logic, and microcode for conditional branches and jump instructions. This part requires the most additional logic in the datapath. All jump and branch instructions require logic to select the next program counter value, sometimes based on a (possibly inverted) condition check in the ALU. The `jal` (jump and link) instruction requires students to route `PC+4` to the register file so it can be saved in the return address register `$ra`.

After completing this part, students can write PIPS assembly programs that make simple leaf procedure calls, but without memory instructions there is no way to save a return address or other registers on the stack.
Part 4 – Memory and Terminal Output: Students add assembler rules, control wires/logic, and microcode for memory operations. Implementing the `sb` and `sw` operations, which write to memory, introduces a race condition in most datapaths; these instructions must not modify the contents of a register, but turning off the register file's write-enable input does not necessarily happen before other changes in the datapath. To fix this issue, students change the register file to run on an inverted clock signal. This inversion ensures that all register reads happen in the first half of each cycle, while writes happen after control signals have been updated.

Using these memory operations, students add support for memory-mapped terminal output, along with a special halt address where programs should jump to stop execution. Students use these additions to "compile" (by hand) a recursive decimal number printing procedure from C to PIPS assembly, which they then use to write a recursive procedure that prints Fibonacci numbers. Because the print procedure requires integer quotient and modulo operations, students must also devise iterative procedures for these tasks.
Part 5 – Shifting and Pseudo-Instructions: Students add support for shift instructions, along with a small number of useful pseudo-instructions listed in Table 3. Students also modify their Fibonacci program to use `push` and `pop`. One quirk of the PIPS ISA is that it does not include an assembler temporary register; as a result, integer comparison pseudo-instructions like `blt!` (branch if less than) have the side-effect of modifying one of the input registers. We may address this in a future version of PIPS (see §5.5), but this detail has not been a significant source of confusion.
Part 6 – Keyboard Input: Finally, students add support for memory-mapped keyboard input. They conclude by writing additional PIPS assembly programs that use keyboard input, including a basic calculator program.

Of course, the prompts for each of these labs features far more detailed instructions for students, guiding them in what connections they need to make in the datapath and framing the control elements they need to add as well as the control bits for the microcode.

5.3 Grading

The correctness of most lab segments is automatically assessable by an autograder. However, many lab segments ask for the instructor or teaching assistant to inspect and verify correctness before students move on. Having a supervisory human in the loop at these stages is not strictly necessary; incorrect or non-functional implementations will eventually be discovered by students or through the autograder. However, it becomes more difficult for students to uncover the cause(s) of incorrect behavior the longer it has been since they worked on that portion of the stack. Therefore, these inspections provide quick opportunities to correct things that may cause lengthy delays for students in the future.

Student work can also be scored on more subjective elements, such as clear and efficient circuit design, element layout, and wiring as well as assembly code formatting, comments, and logical structure. While these could be skipped (i.e., for large classes with insufficient support staff), we find they provide important opportunities for student reflection on their craft. A well-honed rubric helps streamline the process; the archive material includes an example.

The manual inspection checkpoints provide opportunities for coaching and informal feedback on some of these elements. We also use these checkpoints formally to enforce certain standards before the work is submitted. As a particular example, specifying the microcode for control only requires sixteen lines, each containing four hexadecimal digits. Because this is hardly illuminating on its own, we typically require inline comments explicitly documenting (1) the semantics of each bit column (which accrue as more instructions are supported), (2) the specific bit-level settings for each semantic grouping (e.g., `Binv=1`, `ALUOp=10`, etc.), (3) a rewriting that groups each of these bits into nibbles, and finally (4) the translation of those nibbles into the four hex digit ROM line entry. Although it can seem tedious, we found this process both helpful and necessary to generate correct ROM lines when first implementing a solution ourselves, and we regularly relay this fact to students. If errant behavior should ever be detected in the processor, this documentation enables a step-by-step verification of the design process; the microcode control is a frequent source of such errors.

Parts 1–3 are all manually checked during lab meetings (or later in office hours) allowing for immediate feedback and revision if necessary. These parts could be turned in to a grading portal and marked, but because subsequent parts generally require their predecessors to be correct, it is important to return feedback quickly and ensure it moves students towards correct implementations. Thus we find the promptness of direct, manual feedback and intervention critical to the success of the lab sequence. Students rarely ask to be checked without having done sufficient testing to demonstrate correctness (in part because we ask them to write short PIPS assembly programs to that end), so the process is usually fairly quick.

We have developed terminal output-based autograders that begin by testing the cumulative behaviors of Parts 1–4. To provide low-level assessment, a suite of three tests first checks the basic instructions. The first tests that a simple "Hello" program produces the correct output. A second tests that the branch and jump instructions give the correct output (based on intended PC updates). The final tests all four memory instructions by pushing and popping values on the stack, printing to check that the values are correct.

To isolate possible sources of error and score appropriately, each of these suites runs the test program on a reference assembler and the student's datapath, the student's assembler and a reference datapath, and the student's assembler and datapath, expecting all combinations to agree with the reference assembler and datapath.

We also then easily verify that the student's Fibonacci program produces the intended terminal output when assembled with the student's assembler and run on their datapath.

The autograder for Part 5 uses multiple strategies. For shifting, the shifter subcircuit (cf. Figure 1) is verified with combinational tests (a Logisim feature), the translations produced by students' assembler rules are compared to a reference implementation, and the execution of shift instructions are verified by testing for register equality with the expected results (comparing against expected terminal output, as for Part 4). The pseudo-instructions admit multiple implementation strategies; we try comparison to *expected* reference implementations, but backstop this with a behavior-based output comparison test (taking the larger of the two scores).

Over the years, we have found just enough variation in some students' translations that still obey the specification (i.e., addition is commutative, so argument ordering to ALU lanes is arguably implementation defined) to keep us wary. Although not complicated, manually inspecting the three-line translation sequence of each rule is admittedly tedious for Part 2; removing the implementation-defined aspects of the translation by refining the specification (cf. Table 2) would allow us to incorporate more autograding earlier.

Although theoretically possible, Part 6 presently has no autograder due to a Logisim bug. While `stdin` provides keyboard input in Logisim's TTY mode, *sometimes* the first input character is skipped. Graders thus manually verify the keyboard functionality and circuit design as well as assembly program correctness.

5.4 Design Choices

The PIPS lab sequence omits a number of elements one expects in a more realistic datapath. Performance-focused features such as pipelining, caching, and branch prediction are all covered in the course, but they are not included in this lab sequence. Without timing simulation, Logisim is a poor platform for evaluating datapath performance; the net effect of implementing pipelining in a Logisim datapath is that it is *significantly* more complicated, while programs behave exactly as they did without pipelining. A predecessor to this lab sequence did ask students to pipeline a more limited datapath, but that exercise came at the cost of other more interesting additions like these labs' assembler components.

The PIPS datapath does not include support for interrupts, virtual memory, system calls, or unified instruction and data memory. While these elements would be interesting additions to PIPS, they are not essential to reach a point where students can write and execute useful programs on a datapath they built themselves. We discuss these features in class but feel it is not necessary to incorporate them in this lab sequence, with its primary goal of enabling a truly comprehensive understanding of the interactions between students' assembly code, the ISA, and their underlying datapath. Moreover, these features are largely available for deeper exploration in other portions of the course (i.e., programming the PIC32 MIPS microprocessor) or in other courses (i.e., Operating Systems).

5.5 Potential Enhancements and Variants

One area for improvement in the current PIPS specification is the lack of an assembler temporary register. Having implemented the `bge!` of PIPS and contrasted the need for it with the `bge` of MIPS, students now likely understand the role of that register better than when we simply say that it exists for the purposes of the assembler in doing its work. One of the other registers could be replaced—most likely `$a3`—and re-designated as the assembler temporary `$at`. However, with two unused bits (18–19) common to both i-format and r-format instructions, and several other unused bits in i-format instructions, it is possible to redefine the formats for five bit register fields, allowing for up to 32 registers.

In the current version, these four-bit fields align to hex nibbles by design. This means students can more easily write machine code by hand—an intentional feature before we'd created the assembler layer of the lab sequence. However, it still means that the hex view of the machine code in instruction memory is more quickly decipherable while observing a program running. Thus, changing to five-bit register fields would not come without some trade-off.

Because an opcode can be used in either instruction format, PIPS could support control flow instructions not available on MIPS. This includes indirect function calls with a `jral` (jump register and link) instruction, or more unusual instructions such as "branch if [not] equal and link" (requiring care to ensure `$ra` is set appropriately).

PIPS instruction encoding also allows greater flexibility in memory addressing than MIPS. Using r-format variants of memory access instructions like `lw` would make it possible to compute an address as the sum of two register values rather than just a register and immediate. Combining this with the shift instruction fields would also allow for address computation of the form `base_addr` + `scale` × `arr_index` for power-of-two `scale` values. This would make array indexing more convenient than in MIPS, eliminating the need for a separate address computation. We expect students would see the value in such instructions, as they have experience implementing array indexing in MIPS assembly, which does not support this addressing mode.

6 EVALUATION AND ASSESSMENTS

This material has been used as an assignment over three iterations of the course (Fall semesters of 2018–2020); there were two sections each year, of roughly 24 students each.

Although we have not conducted a formal assessment of learning outcomes related to the PIPS activities, student comments on end of course evaluations regularly praise the use of Logisim and "the datapath labs". It is frequently cited as the activity that they both enjoy the most and learn the most from. They find that "using Logisim to understand the concepts was extremely helpful" and it is "fulfilling" to essentially build a complete "functioning computer". One student summarized, "it made me feel like I accomplished something" and another that "It's cool how I can picture how a computer works at so many different levels." Another student who noted this was their favorite course activity remarked, "I have a slightly masochistic wish that the course had made us pipeline it, but, for the sake of future students' sanity, please don't act on this suggestion." We agree with this student (see §5.4).

Most students score perfectly on the functional elements of the assignment (datapath, assembler rules, and demonstration programs). Marks are more often lost for more subjective elements such as circuit layout or violating register calling conventions.

Although students gain experience with assembly debuggers in MARS [25] and MPLab, we have no debugger for running PIPS programs. Moreover, when things do not work as expected, the problem(s) could be in multiple places: datapath wiring, microcode control bits, instruction translation, or assembly code logic. Students must work carefully to identify the source(s) of such errors. The lab sequences therefore encourage students to continue running small test programs that verify correct behavior for each instruction, especially as they continue to add support for new functionality. Teaching students to deconstruct and trace the behavior through these levels is a critical part of the experience. This teaching most frequently happens during the in-class lab as errors are encountered, rather than explicitly through any lectures, comments, or written instructions. Although no students commented (negatively or positively) about this aspect of the labs in end-of-course evaluations, a more targeted assessment about what students have learned could shed light on their experiences in this regard.

Several students did comment negatively on the friction induced by the need to wait on manual inspections at several points in the process. However, we view this as a worthwhile trade-off against students (and likely instructors) spending significantly more time tracing back errors and unwinding several new layers of datapath, control, and/or translation to get to the root cause.

7 CONCLUSION

Students with basic knowledge of any modern RISC ISA (e.g., MIPS, ARM, RISC-V, DLX) should be able to understand and program in PIPS with relative ease. The simplified ISA also allows them to build a complete working PIPS CPU in roughly half a day (i.e., six two hour labs). By providing an easy-to-use framework in which students also develop translation rules for PIPS assembly, they gain the experience of building all components of a system on which they can run their own programs (including loops, function calls, and recursion). To do all this, they must integrate and apply their general knowledge of instruction encoding as well as datapath wiring, logic, and control. These are core learning outcomes for any computer organization, architecture, and design course.

By publicly presenting this overview as well as the Creative Commons-licensed lab assignments and infrastructure (including autograders upon request), we hope to make it easily adoptable by others who teach a course covering these ideas, even—perhaps especially—if they are not advanced architecture experts.

REFERENCES

[1] ACM/IEEE-CS Joint Task Force on Computing Curricula. 2013. *Computer Science Curricula 2013.* Technical Report. ACM Press and IEEE Computer Society Press. https://doi.org/10.1145/2534860

[2] Rashmi Agrawal, Sahan Bandara, Alan Ehret, Mihailo Isakov, Miguel Mark, and Michel A. Kinsy. 2019. The BRISC-V Platform: A Practical Teaching Approach for Computer Architecture. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/3338698.3338891

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proc. Design Automation Conf.* 1216–1225. https://doi.org/10.1145/2228360.2228584

[4] Golden Barlow, John Hossley, and Timothy Stanley. 2013. Use of interactive logic simulation software to introduce data path and control concepts in introductory architecture courses. *J. Comp. Sci. in Colleges* (December 2013), 4–16. Issue 2.

[5] Miloš Bečvář and Stanislav Kahánek. 2007. VLIW-DLX Simulator for Educational Purposes. In *Proc. Wkshp. on Comp. Arch. Ed.* 8–13. https://doi.org/10.1145/1275633.1275636

[6] Miloš Bečvář, Alois Pluháček, and Jiří Daněček. 2003. DOP: a CPU core for teaching basics of computer architecture. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275521.1275527

[7] Andrew Brownfield and Cindy Norris. 2009. LC3uArch: A Graphical Simulator of the LC-3 Microarchitecture. In *Proc. SIGCSE Tech Symp.* 413–417. https://doi.org/10.1145/1508865.1509013

[8] Carl Burch. 2014. *Logisim.* http://www.cburch.com/logisim/

[9] Daniel Chaver, Yuri Panchul, Enrique Sedano, David M. Harris, Robert Owen, Zubair L. Kakakhel, Bruce Ableidinger, and Sarah L. Harris. 2017. Practical Experiences Based on MIPSfpga. In *Proceedings of the 19th Workshop on Computer Architecture Education.* Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3116214.3116217

[10] James M. Conrad and Luke M. Hassell. 1995. A Senior-Level Computer Hardware Organization Course: Designing a Single Board Computer. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275143.1275147

[11] Jean-Luc Dekeyser and A. Shadi Aljendi. 2015. Adopting New Learning Strategies for Computer Architecture in Higher Education: Case Study: Building the S3 Microprocessor in 24 Hours. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/2795122.2795128

[12] Dan Ellard, Penelope Ellard, James Megquier, and J Bradley Chen. 1998. *The ANT-Architecture–An Architecture for CS1.* Technical Report TR-14-98. Harvard University, Cambridge, MA. http://nrs.harvard.edu/urn-3:HUL.InstRepos:25620471

[13] Daniel Ellard, David Holland, Nicholas Murphy, and Margo Seltzer. 2002. On the design of a new CPU architecture for pedagogical purposes. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275462.1275471

[14] Roberto Giorgi and Gianfranco Mariotti. 2019. WebRISC-V: A Web-Based Education-Oriented RISC-V Pipeline Simulation Environment. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/3338698.3338894

[15] Jan Gray. 2000. Hands-on computer architecture: teaching processor and integrated systems design with FPGAs. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275240.1275262

[16] John L. Hennessey and David A. Patterson. 2017. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufmann.

[17] James Larus. 2018. *SPIM: A MIPS32 Simulator.* http://spimsimulator.sourceforge.net

[18] Jason Lowe-Power and Christopher Nitta. 2019. The Davis In-Order (DINO) CPU: A Teaching-Focused RISC-V CPU Design. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/3338698.3338892

[19] Ethan Miller and Jon Squire. 2000. esim: A structural design language and simulator for computer architecture education. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275240.1275252

[20] Enric Pastor, Fermín Sánchez, and Anna M Del Corral. 1998. A rudimentary machine: Experiences in the design of a pedagogic computer. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275182.1275189

[21] Sanjay J Patel and Yale Patt. 2003. *Introduction to Computing Systems: From Bits and Gates to C and Beyond* (second ed.). McGraw-Hill.

[22] David A Patterson and John L Hennessy. 2013. *Computer Organization and Design: The Hardware/Software Interface (MIPS)* (fifth ed.). Morgan Kaufmann.

[23] Murray Pearson, Dean Armstrong, and Tony McGregor. 2002. Using custom hardware and simulation to support computer systems teaching. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275462.1275470

[24] Sandro Rigo, Marcio Juliato, Rodolfo Azevedo, Guido Araújo, and Paulo Centoducatte. 2004. Teaching computer architecture using an architecture description language. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275571.1275580

[25] Pete Sanderson and Ken Vollmar. 2017. *MARS (MIPS Assembler and Runtime Simulator.* https://courses.missouristate.edu/KenVollmar/MARS

[26] Timothy Daryl Stanley and Mu Wang. 2005. An emulated computer with assembler for teaching undergraduate computer architecture. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275604.1275615

[27] Yutaka Sugawara and Kei Hiraki. 2004. A computer architecture education curriculum through the design and implementation of original processors using FPGAs. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275571.1275576

[28] Jan Van Campenhout, Peter Verplaetse, and Henk Neefs. 1998. ESCAPE: Environment for the Simulation of Computer Architectures for the Purpose of Education. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275182.1275191

[29] Anujan Varma, Lampros Kalampoukas, Dimitrios Stiliadis, and Quinn Jacobson. 1995. CPU design kit: An instructional prototyping platform for teaching processor design. In *Proc. Wkshp. on Comp. Arch. Ed.* https://doi.org/10.1145/1275225.1275226