Experiences with a Hardware Description Language for a CS-major's Computer Organization Course

Charles Reiss
Department of Computer Science
University of Virginia
Charlottesville, Virginia
Email: creiss@virginia.edu

Luther Tychonievich
Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, Illinois
Email: luthert@illinois.edu

Abstract—This article presents a novel hardware description language (HDL) and associated 4-week assignment sequence for a computer architecture course, with discussion of our experience developing and using these tools.

At our institution, CS majors take a different computer architecture course than computer engineering majors. The CS majors' course aims to providestudents with an understanding of how nontrivial processors can be built out of simple hardware components. To leverage students' existing familiarity with programming languages, we wanted students to manipulate processor designs using a text-based hardware description language (HDL). We did not, however, want to devote instructional time to the low level complexities like clocking choices or arithmetic logic design.

We developed an instructional hardware description language and associated assignments based on Bryant & O'Halleron's HCL (and designed to be compatible with their text) with some inspiration from Verilog. Unlike HCL, our HDL allows students with flexibility to build and simulate different processor organizations — for example, pipeline designs with different pipeline stages as well as non-pipelined designs — without the differences being hidden in the internal components of the simulator. By specializing for building a processor, our tooling prominently to students and our testing infrastructure how their simulated processors executed programs. Also, using simple enforcement of signal widths and avoiding floating or undefined values, our HDL design helped catch many students errors while still seeming to treat all values as bits.

We also discuss our experience teaching a course using these tools to several hundred students a semester for the last eight years.We evaluate our experience through a review of student and instructor feedback, which suggest increased student and instructor satisfaction and that the HDL and assignments provide a solid basis for explaining later topics such as superscalar processors and processor-aware software optimization.

I. Introduction

Computer organization courses teach about the design of computing machinery, including processor design. Levels of detail in such courses vary greatly; for example, the CS2013 curriculum guidelines suggest programs could teach anywhere from 16 hours to 3 full courses on computer organization [1].

Frequently, computer organization courses target a variety of audiences. Some teach computer organization and architecture to prepare students to engage in processor design [2]. Some focus on systems programming, teaching the parts of computer organization needed to understand the programming tasks [3], [4], [5]. Some focus on the impacts of hardware design on software performance and security [6].

At our institution (a large R1 university in the eastern United States), we switched from having a single course shared by computer engineering majors and computer science majors to having separate courses for the two degree programs. In this article we provide an experience report about certain aspects of how we designed the computer science majors course. Since the students taking our course are generally not those likely to be interested in the details of circuit design and synthesis, we aimed to provide a good understanding of the processor design strategies and tradeoffs without being distracted by the sort of practical details that our computer engineering majors would be concerned about.

Our course is required of all CS majors (roughly 650 students per year) and is typically taken in their third year of the major. Prior to this course students work at a very substantial level of abstraction. From this abstraction, processors may seem 'magical'. Though we teach computer organization from a software-oriented perspective, we want students to understand that they could build a processor — even if our course does not include all of the details.

Beyond understanding how processors are built, our students should understand modern processor performance, including processor design optimizations like pipelining, other forms of instruction-level parallelism, and speculative execution. To give this broader overview of modern processor designs and their implications, we included less detail on some aspects of processor hardware that would be included in a course focused more on building processors.

To achieve these goals, we wanted to spend 4–5 weeks on processor design, enough to give them a thorough understanding of their design while still leaving time to apply this understanding to later topics like caching and optimizations. We also wanted students to experience two different designs the same instruction-set architecture (ISA) so they would understand how processor architecture can influence performance. These aims resulted in

our extending a teaching hardware description language (HDL) with new features and tooling and designing a 4-week assignment sequence using it.

The remainder of this paper is an experience report. We describe our hardware design assignments, the tools we designed to support those, and our experiences with them. Because this was undertaken over several years as one of several changes to the associated course, the effectiveness of these course components are primarily shown via discussion of student and instructor experiences.

II. Prior Approaches

Undergraduate computer organization course use a variety of approaches to teach how processors work. Frequently courses use a combination of lectures along with worksheets and quizzes to help students practice what they learned. Like many instructors, we wanted a more hands-on approach that should build lasting knowledge and be more engaging to students.

To support a hands-on approach, many processor simulators that have been used [7], [8], [9], [10], [11], [12], [13], [14], [15]. Many of these provide visibility into the internal operation of the simulated processor, which is useful for learning how a given processor design operates but presents the processor as a fait accompli. We wanted to prepare student to consider multiple designs, including some of their own creation, so we opted to have students build their own processor simulators.

We needed to choose what development environment students would use to build a simulator. Perhaps the most straightforward is to use a hardware description language used in industry or processor design research, such as Verilog, VHDL, or Chisel [16], [17], [18]. This approach helps students with future work in processor design and can allow students to deploy their design on 'real' hardware using FPGAs and similar [19], [20], [21], [22]. But teaching students how to work with these industrially focused tools would be time consuming, in part because of details that are mostly irrelevant to our course's more software-focused perspective.

To avoid these difficulties, a number of educational hardware programming languages exist, both graphical and text-based.

Graphical programming languages allow students to draw a graphical representation of a circuit [23], [24], [25]. Previously, our course used one of these tools, Logisim [24]. Though Logisim's visual design helps visualize circuits and avoids teaching syntax, students had difficulty making designs modular in the graphical format. Logisim has tools to enable this, but students did not use these effectively when the interface was so different from the programming abstractions they used in prior courses.

To capitalize on student's familiarity with text-based programming, we chose a text-based register transfer language [26], [27] for our course. In particular, we found that the HCL, as introduced by the textbook Computer Systems: A Programmer's Perspective (CS:APP), supported our goals well [27]. This language seems inspired by Verilog, but lacks many of its complexities. In adapting this language, we made a new version, which we describe later in the paper.

A different common approach that we did not select is having students build a simulator from scratch using a general-purpose programming language they already know [28], [29]. This avoids the necessity to teach a new environment. However, in our experience students using a general-purpose language have trouble seeing how their code can represent hardware and may not realize when they use designs that would work poorly in hardware, such as doing loop-like operations in a single-cycle.

III. Our Assignments

Our assignments have students build a processor, following the textbook Computer Systems: A Programmer's Perspective's design and instruction set. Students did so in two phases, first building a processor that completes every instruction in one cycle, then a pipelined processor, both similar to designs presented in the textbook.

Our assignments are split into 8 parts: 4 staffed, loosely-graded in-person labs each introducing a following week-long homework that is graded more completely. The first two weeks of labs and homeworks have them build a single-cycle processor, the next two a pipelined processor for the same ISA.

For the single-cycle processor, we first introduced students to our HCL language variant by having them implement the logic for incrementing the program counter for non-control-flow instructions. Since we use a variable-length instruction set, this requires using the instruction opcode to compute how to increment the program counter. In the following assignments, students implement the rest of instruction logic, instruction-by-instruction. The first homework focuses on instructions requiring basic access to registers; the second lab introduces instructions with arithmetic, conditional evaluation (through conditional moves) and memory accesses; and the final homework requires handling the control flow instructions and the stack manipulation instructions in the ISA.

For the pipelined processor, we choose not to follow CS:APP's approach of starting with a full single-cycle processor and converting it to a full pipelined processor. Instead, for all but our final pipelining assignment students implement only subsets of the ISA. This choice lowers the complexity of each assignment and allows for easier incremental testing.

To introduce pipelining in lab, we had students convert a single-cycle processor supporting only register-to-register and immediate-to-register moves to a two-stage pipeline. One stage includes the instruction fetch and register read operation, and the second includes the register write operation. With this pipelining assignment, we had two goals: first, we want students to learn how to pass values

between pipeline stages; and, second, we want to show a simple example of pipeline hazards and their resolution with forwarding.[1]

In the first pipelining homework, students extend their two-stage design from lab into a five-stage pipelined design with support for some instructions involving condition code handling. As students implemented this subset of the ISA, we encouraged students to add one instruction at a time, using their complete single-cycle processor as a reference.

The final lab and homework for the pipelined processor focus primarily on handling pipeline hazards that cannot be resolved through forwarding. The lab includes only load and store instructions to keep the processor simple while guiding the students to implement stalling to resolve hazards. The final homework has students combine the previous lab and homework and implement the rest of the ISA, notably including branch prediction with an always-taken strategy (following the design in CS:APP).

Our full set of assignments can be downloaded from https://github.com/charlesreiss/hclrs-assignments

IV. CS:APP HCL language

The CS:APP textbook describes the operation of the processor circuit by introducing a language they call HCL (hardware control language). Though the authors of CS:APP did use this educational language to make a complete processor that they formally verified [30], [31], they did not seem to use it in processor construction assignments. Instead, their instructor materials include assignments where students make limited modifications to existing designs. We revised the language to better support our processor construction, as described in section V. To provide context, we first describe the original HCL language.

In HCL, the storage and computation components are fixed (chosen by the authors to support their processor design) and therefore not specified within an HCL program. Statements in an HCL program focus on selecting inputs to those components based on the current instruction being executed. For example, a statement like:

```
word dstE = [
    icode in { IRRMOVQ, IOPQ } : rB;
    icode in { IPOPQ } : RRSP;
    1 : RNONE
];
```

would choose the register file destination dstE based on the instruction opcode icode – choosing between the rB field of the instruction and the register identifier constants RRSP and RNONE.

[1]In some pipelined processor designs, this design would not require an explicit forwarding implementation because, to support pipelining, the register file would read new values while they are being currently written. We provide the same register file for both pipeline and single-cycle assignments, so students implement this functionality outside of the register file.

A. Signals

The HCL language operates on a fixed set of signals, each of which holds one value during each cycle and is declared as either bool or int. Some signal values are controlled by functionality 'built-in' to the simulator, including memories, the register file, other registers, the ALU, and the program counter incrementing logic.

Other signal values are assigned in statements in the HCL file, using an expression written in terms of other signals. These expressions are intended to correspond to some combinatorial circuit. For example foo = bar makes the value of the signal foo the same as the signal bar. These expression also support bitwise operations (such as foo = bar & baz, where & represents bitwise AND, and bar and baz are other signals), inequalities (foo = bar < baz), checking membership in a list of values (foo = baz in {1,3,5,8} to set foo to 1 if and only if the signal baz is one of 1, 3, 5, or 8), and case expressions that are intended to correspond to multiplexers. Case expressions represent an if/else if/else if/...expression. For example, foo = [bar == 1 : a ; baz == 2 : b] sets foo to a if bar is 1, to b is bar is not 1 and baz 2.

B. Simulation and Timing

To simulate the circuit, the original implementation parsed and converted HCL assignment statements into either C code or Verilog code and combined this with code implementing the built-in functionality. The built-in functionality code was different depending on whether a five-stage pipelined processor or a non-pipelined processor was being simulated.

The C implementation seems to be the primary one intended for student use. In this implementation, the assignment operations run in a fixed order, interleaved with the simulation of the built-in components. The order in which signal assignments are run is hard-coded. This avoids the need to determine an evaluation order for the simulation, but requires the available signals to be fixed.

In the Verilog version, assignments are converted to Verilog assign operations and combined with a Verilog implementation of the built-in functionality.

C. Storage Components

Various built-in circuits store and pass data between simulated clock cyles. This includes a main memory, with two combinatorial read ports and one write port, which is presented to students as an instruction memory and data memory. It also includes a register file with two read and two write ports, and several independent registers for the program counter and registers added for pipelining.

All the storage logic provided in the simulation is triggered by a clock signal, which updates values at the end of each simulated cycle.

In the pipelined versions of the simulator, some of the interactions between pipeline registers and the storage components were hard-coded: the simulated register file

and simulated data memory both output into a pipeline register.

D. Simulator Control and Lack of I/O

The built-in functionality includes features to control the simulation itself. Most notable of these is a Stat signal which determines whether the simulated processor should continue, halt, or halt with an error. For simplicity, no other I/O mechanisms are provided.

V. expanding the simulator

We liked CS:APP's presentation of processor design and wanted to use an HCL-like language to be compatible with that presentation. However we found that HCL was not well suited to having students build up a processor from basic components. With so many signal's effects and connections predefined, students seemed to have difficulty connecting their code to the overall operation of the processor. Course staff reported spending most of their time explaining why students' seemingly-valid designs were incompatible with the predefined parts of the simulator.

We decided to amend HCL and build our own simulator to put the internal signals and registers of the processor more under students' control. In our simulator, signals are called wires[2], and these signals can be either predefined or defined by the students' input file.

One result of this is that students would use the same interpreter for the sequential and pipelined processor design. Though we still had built-in functionality it was not as tied to a particular microarchitecture. This gave us the freedom to experiment with different number of pipeline stages, converging on the assignments described in Section III.

Our full resulting simulator can be downloaded from https://github.com/charlesreiss/hclrs.

A. Requiring Wire Widths

Instead of having a distinction between bools and ints, we gave each wire has a particular bit width, similar to Verilog. Expressions and constants can also have bit widths. Generally, the bit width of an expression assigned to a wire must be equal the bit width of the wire. Similarly, the bit width of the operands to an operator like bitwise AND must be equal, and the width of the result is equal to the width of the operands. In a case expression, each possible result of the case expression must have an equal width.

Wire widths in effect form a loose type system. In our assignments, generally students would use values of five different widths:

- 1 bit: for boolean signals, like whether to enable data memory writing;
- 3 bits: for the Stat control signal for simulator;
- 4 bits: for register identifiers, primary opcodes, and secondary opcodes;
- 64 bits: for register values and addresses (and intermediate values used to compute them); and
- 80 bits: for the output of the instruction memory

This type system serves to catch certain common errors students made:

- Confusing register identifiers with actual register values. For example, students often try to use the extracted register identifier field from the instruction instead of the output of the register file for an instruction's computation.
- Confusing opcodes with simulator control values. Students would try to use a constant representing the opcode for HALT instead of the constant representing the control signal for the simulation to terminate normally.

Wire widths do not catch all errors that full type annotations could. For example, a common uncaught problem is confusing 64-bit addresses with 64-bit values retrieved from those addresses.

B. Supporting Instruction Decoding

In the CS:APP HCL language, decoding the fields of an instruction (a non-trivial task for the variable-length Y86 instruction set) was done by built-in functionality. We wanted students to understand this process in more detail so we required this functionality to be implemented explicitly in the HCL code.

To enable variable-length decoding in HCL, we provide a bit-subsetting operation – using something like foo[5..8] extracts bits (numbered with 0 being the least significant bit) 5 (inclusive) though 8 (exclusive) of the signal foo as a 4-bit signal.

C. Exploring Dependencies

Unlike the template for students in the original HCL assignments, we permitted students to define their own signals and registers. In addition to requiring us to choose a syntax for register and wire declarations, this also required a more complex simulator. Where previously the simulator could hard-code an order of evaluation, we needed to determine the order of execution based on the actual dependencies between components.

In determining the order of evaluation, we also search for and report any cyclic dependencies. This often happens due to student error: for example, students implementing the jump instruction will sometimes attempt to set the current program counter rather than the next program counter.

Along with determining dependencies for evaluation order, we also detected when built-in functionality could be omitted. For example, this allowed us to support a two write port register file for assignments that required it, but

[2]This terminology is consistent with Verilog. In retrospect, it may be more appropriate to call these signals or bundles or similar, since a typical physical implementation would have multiple actual wires.

not require any extra boilerplate for earlier assignments that only use one write port.

D. Defining Registers

Rather than fixing the set of pipeline registers and the functionality for the program counter, we added the ability to define arbitrary registers. This allowed us to support assignments that used a different set of pipeline stages than in the book. It also allowed us to give students the task of deciding what values needed to be passed within pipeline registers.

Following the conventions of the textbook, aimed primarily at implementing pipelines, registers are defined in groups we called 'register banks', and each register bank was identified by a lowercase and uppercase letter. For example, fD might be a register bank, conventionally the one for pipeline registers between a 'fetch' and 'decode' stage. In each register bank, multiple 'registers' can be defined, each with their own bit-width. For example, following the CS:APP design, a fD would include (among others) a 64-bit valP (PC value) register and a 4-bit rA (first register ID) register. Each of the registers would have their own input and output signals prefixed with the lowercase and uppercase letter from the register name, such as f_valP for input of the register (computed by the students' code) and D_valP for output of the register. In each simulated cycle, the previous cycles values for the input signals would become the output for that cycle.

1) Register Control Signals and Values: Following the design of CS:APP, each register bank also had a control signal that would disable writing, called 'stall', and that would reset the registers to a default value, called 'bubble'. These signals are useful for (and named after their roles in) pipeline control operations, for example to implement a pipeline stall.

2) Non-Pipelined Storage: Students use custom-defined registers for pipeline registers and for the program counter and condition code registers. This allowed us to teach the concept of how registers work within in our first HDL assignment, even though more extensive use of registers would come later.

We elected not to use our custom register functionality for the architectural registers (the registers visible to assembly). For these we supplied a register file. This was primarily due to two practical concerns. First, having readily available register values made it practical to grade and provide debugging output when students only implement simpler register-to-register instructions. Second, our course was not the one where students would learn about register file or memory design, so we would rather spend our time on other topics. Using a supplied register file may have also helped explain the similar built-in data memory component.

VI. Output and Debugging Support

Since the ISA in our assignments do not support any input or output operations, our simulator's output must include some information about the 'internal' state of our simulated machine. The default output shows the contents of storage – memory, the register file, and program-defined registers – and what instructions were fetched. To aid debugging, the default output also disassembled the fetched instruction using the Y86 instruction set.

To further support debugging, we also optionally support detailed output showing the operation of the simulated processor. Broadly, there are two parts to this debugging output: showing how memories and the register file are read and written; and listing the values of every named signal. In initial versions, debugging output wrote signal assignments in the order they were simulated, but that proved confusing for students. To aid students in locating specific signals, later versions show values in alphabetical order after categorization by type (input or output to built-in component, input or output to register, or other).

VII. Compilation Feedback and Common Errors

One of the most serious sources of frustration for students in early versions of our HDL tool was error messages. Frequently students would write code that had syntax errors or semantic errors. In early versions of our tool, we did not spend much effort diagnosing these. Until we started to fix this issues, a great deal of course staff time was spent helping identify syntax errors or helping find likely causes of runtime errors.

A. Syntax Errors and Parsing Issues

The initial implementation used an implementation of a PEG [32] grammar generator [33]. Though this made it fairly simple for us to make an efficient parser, like many parsing libraries, error recovery was not a well-supported feature. This meant that often student errors would appear as a message identifying the line number with no other information, and the line number was often the beginning of the invalid statement instead of the line of the missing semicolon or square bracket within it.

Later, we rewrote our implementation being mindful of error handling supported by our choice of parsing tool. We used an LALR-based parser generator[34] which, when parsing failed, would identify expected possible next tokens that would have continued a valid parse. This allowed us to write a default error message that precisely identified a location in the file, including showing an excerpt of the line identifying the invalid token.

B. Special Patterns Identified

Based on our observations from early versions of the assignments, we identified common syntax errors and extending our language grammar to parse them, such as the cases listed in Figure 1. For these, we elected not to extend our actual language to support these alternative syntaxes; instead, we made our interpreter generate special error messages for each of these cases.

correct syntax	parsed incorrect examples
	wire NAME;
wire NAME : WIDTH;	wire NAME = VALUE;
	wire NMAE : WIDTH = VALUE;
const NAME = VALUE;	const NAME : WIDTH = VALUE;
NAME = [COND1: VALUE1;];	NAME [COND1: VALUE 1;];
	register aB { wire NAME : WIDTH = VALUE; }
register aB { NAME : WIDTH = VALUE; }	register aB { wire NAME = VALUE; }
	register aB { NAME = VALUE; }

Fig. 1. special incorrect syntax that we parsed to produce custom error messages.

We also found it useful to identify several common misuses of case expressions. There were two common errors we found. One was that students would fail to handle some cases in their case expressions. In our initial implementation, case expressions defaulted to 0 when no condition matched, so this error would often be hard to diagnose. Another problematic pattern arose from students treating case expressions as something more akin to a switch statement, so they would write something like:
[FIRST : ...; SECOND : ...; ...]
when they meant to write something like
[signal == FIRST : ...; signal == SECOND : ...; ...].
(Most commonly, the signal in question would represent an opcode.) This would silently fail because the constants FIRST and SECOND would be treated as boolean expressions, and as boolean expressions, they were usually always true.

To help students diagnose these issues we treated 'default' cases in case expressions specially. A default case was one with a constant which was is true. We required the last case and no other cases in each case expression be a default case. This change made a substantial amount of existing code invalid, since it was common for students to write solutions where all possibilities were explicitly handled. We decided that requiring a 'dummy' default case for those solutions was worth avoiding other silent errors.

C. 'Did You Mean?' Messages

In addition to identifying error messages, we also spent effort to include common resolutions in existing error messages. In particular, when a signal that was not defined was used, if there was another signal whose name matched except for capitalization, the error message would ask if students meant to use that. Otherwise, the error message would ask if students meant to declare a signal with that name. We also included some similar suggestions when students did not set signals needed for built-in functionality, mention what other signals they may have meant to set.

VIII. Reflections

A. Good: Better Processor Understanding

Having a more complete hardware description language seemed to help our students better understand processor design. Spending effort on the toolchain's error reporting for assignments meant that our course staff time was used relatively effectively – student questions related to the assignments are mostly be about the processor designs topics being covered. These could either be high-level questions about overall strategies for functionality like branch prediction or stalling, or lower-level questions related to understanding the operation of their simulated processor in order to fix a bug.

After completing the processor development assignments, students showed competence in answering questions about processor design. This included, for example, identifying what type of changes particular new instructions would require to the microarchitecture they learned, or identifying what pipeline hazard handling would be needed in alternate microarchitectures.

Students in course feedback often attributed the assignments with being helpful in their understanding of course material, though some students felt it took too much time. Overall, having students build a pipelined processor in a simulator made us confident that we could support more advanced material, like superscalar processor design, that requires a substantial background in simpler pipelined processor designs. As discussed in the next section, other changes to our course and its assessment unfortunately do not allow us to usefully comparing student scores between semesters to quantitatively confirm or refute our confidence in students' preparation.

After introducing our HDL and its associated assignments, instructors of subsequent courses also reported students were "better prepared" for their courses.

B. Effect on Scores

We did not see direct impacts of our introducing a new HDL and assignments on student scores. Our changes to the HDL assignments took place shortly semesters after the change from a common course for a computer engineering and computer science major to separate courses, so we have limited data from before the new HDL to

Semester	Median HW score	Mean HW score
Fall 2016	70.2	56.3
Spring 2017	75.6	65.4
Fall 2017	98.4	83.3
Spring 2018	96.4	84.6
Fall 2018	96.4	84.6
Spring 2019	98.5*	90.2*
Fall 2020	95.8	82.8
Fall 2021	91.0	81.3
Spring 2022	86.1	97.3
Fall 2022	94.2	81.7
Spring 2023	96.7	86.4

TABLE I
Scores on final pipelined processor design assignment computed using the Fall 2022 test suite, except for the Spring 2019 semester, where the requirements of the assignment omitted several stack instructions.

compare with. In addition, the course underwent other major changes to their curriculum, exams, and instructors, over that time that make score comparisons not useful.

We did see an increase in student scores on the HCL assignment themselves after we improved the implementation as shown in Table I. The improvements in the Fall of 2017 included both changes to error reporting and the supplying students scripts to more easily test their submissions on the supplied test cases. We cannot tell to what extent these two changes improved scores, but the lack of change from later additions intended to further improve error feedback, like better handling of case expressions, suggests that the ease of running tests was likely the primary factor.

Outside of this dramatic change, we saw few changes in overall student scores. This lack of impact was anticipated based on two observations. Especially after the Fall of 2017, the simulation assignments themselves included extensive testing suites the students can run for themselves as often as they wish, allowing them to put in the time needed to get the score they want. Quiz and exam questions were designed each semester to match the level of coverage given to topics that semester; as we refined our HDL and improved the coverage of concepts in the associated assignments, we also increased the level of detail in the related questions on quizzes and exams. For example, our quiz and exam questions on pipelining switched from focusing on understanding the definition of pipelining and the mechanics of registers to understanding the flow and timing of instructions in a pipelined processor.

C. Bad: Growing Pains

We had some difficulties in transitioning to these assignments.

One issue was with staffing. At our institution, most of our teaching assistants (TAs) are undergraduates who work less than 10 hours per week. When TAs had not used our simulator before, they needed to spend a substantial amount of this time familiarizing themselves with the tools rather than helping students. While iterating the tool and assignments improved their quality, each change we made added a new transition cost.

Our initial versions of the assignments did not divide work between them appropriately. Because labs were loosely graded and time on homeworks was not readily visible to us, we relied on TAs to provide feedback on student workload. Several times that resulted in moving tasks between assignments.

It also took us some time to write assignments that provided an appropriate amount of guidance to students. Initially we provided too little practical guidance, resulting in common questions arising during office hours. When we added answers to these to the assignment descriptions, they became unwieldy and students stopped reading them. We converged on descriptions that present the requirements first, with advice on completing them indexed with headers and presented afterwards, which seems to work well.

We also had issues with how students tested their submissions. Initially we provided a few examples of correct behavior and expected students to test their own work, but they were too new to the HDL to feel comfortable doing so, even when we added testing exercises to a lab. To correct this, we now supply example correct outputs and a tool to compare the students' outputs to those correct outputs for each assignment This, as discussed previously, correlated with dramatically increased the scores students earn on those assignments. By adding additional intermediate tests for common errors, it also encourages students continue working on debugging issues when they would not otherwise.

D. Bad: Adaptability for Other Courses

The success of our assignments depended on how much time we devoted to them. Since processor design was one of the most important parts of our course, we spent several weeks of class time explaining processor design. To support our assignments, we also spent approximately one lecture and one lab introducing our hardware description language. To build up to a full pipelined processor, we needed several additional homeworks and labs as well as several weeks of lectures. A course that covered this material in a more terse manner would have difficulty adapting our assignments. Including more 'skeleton' code would limit the amount of time needed, but make it substantially more difficult for students to see the 'big picture' of processor design overall, as was our goal for these assignments. If less time is available, it may be better to use a different style of assignment.

E. Bad: Naming Issues

Some features chosen by CS:APP's HDL we found often caused student confusion, particularly around naming.

1) Register Bank Control Signals: We copied CS:APP's names of bubble (reset) and stall (write-disable), with the letter of the output of the register bank (for example,

stall_D = 1 write-disables the 'fD' register bank). This led to two confusions. "Stalling" an instruction requires using both stall and bubble signals, but students often assume that only the stall signals should be involved in that operation. Additionally, these signals are processed with the register inputs, not outputs, so the output-based naming caused some students to make off-by-one-cycle errors.

2) Other Signals: The CS:APP authors choose to use names of the form 'valX' for signals representing a 64-bit value. Since these names were very opaque, they were a frequent source of student confusion. It did not help that there were some important 64-bit signals which did not follow this convention (like pc or aluA) and that some of these names were also a built-in read or write port of the register file in the textbook. In our HCL implementation students did not need to use these names, but in lecture we generally followed the textbook's names to avoid more confusion.

The names given to inputs to built-in components also caused confusion. One particular source of confusion is pc, which, consistent with the textbook, is the name we give to the input of the instruction memory. Students commonly confuse this value with the input to or output of a register that stores an instruction address.

Students also would get confused about whether signals were inputs or outputs to built-in components, though it is not clear how to improve this situation. The signals were named from the perspective of the components (for example mem_output for the output of the data memory) when some students expected them to be named from the perspective of the component users. Also, some students did not understand the meaning of various register index and address inputs (especially that they did not represent register or memory values) and so had trouble identifying whether they were outputs or inputs.

IX. Conclusion

We created an educational hardware description language well-suited to supporting processor design assignments. We adapted an existing language from Computer Systems: A Programmer's Perspective [6]. To make it more suitable for our assignments, we made changes to the language. We primarily focused on exposing more of the core high-level functionality of the processor, while still avoiding details of the implementation of storage, arithmetic, and circuit timing that were not part of our course. In the process of deploying our custom language, based on our observations of student difficulties, we paid particular attention to improving error reporting for our interpreter. This tool has successfully supported our hardware design assignments in our computer organization class for the past several years.

References

[1] A. f. C. M. A. Joint Task Force on Computing Curricula and I. C. Society, Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. New York, NY, USA: Association for Computing Machinery, 2013.

[2] J. L. Hennessy and D. A. Patterson, Computer Architecture, Sixth Edition: A Quantitative Approach, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[3] Y. N. Patt and S. Patel, Introduction to Computing Systems: From Bits and Gates to C and Beyond, 1st ed. USA: McGraw-Hill, Inc., 2000.

[4] A. Bloomfield and W. Wulf, "Ibcm: The itty bitty computing machine a one-week module to teach machine language in computing courses," in Proceedings of the 42nd ACM Technical Symposium on Computer Science Education, ser. SIGCSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 371–376. [Online]. Available: https://doi.org/10.1145/1953163.1953273

[5] J. A. Stone, "Using a machine language simulator to teach cs1 concepts," in Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education, ser. ITiCSE-WGR '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 43–45. [Online]. Available: https://doi.org/10.1145/1189215.1189166

[6] R. E. Bryant, O. David Richard, and O. David Richard, Computer systems: a programmer's perspective, third edition. Pearson, 2016.

[7] C. B. Ly and C. Norris, "Ly86-64: A web-based simulator for the y86-64 pipe architecture," in Proceedings of the 2022 ACM Southeast Conference, ser. ACM SE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 31–37. [Online]. Available: https://doi.org/10.1145/3476883.3520224

[8] M. I. Garcia, S. Rodriguez, A. Perez, and A. Garcia, "p88110: A graphical simulator for computer architecture and organization courses," IEEE Transactions on Education, vol. 52, no. 2, pp. 248–256, May 2009.

[9] K. Vollmar and P. Sanderson, "Mars: An education-oriented mips assembly language simulator," SIGCSE Bull., vol. 38, no. 1, p. 239–243, mar 2006. [Online]. Available: https://doi.org/10.1145/1124706.1121415

[10] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," Computer, vol. 35, no. 2, pp. 59–67, Feb 2002.

[11] M. D. Black and P. Komala, "A full system x86 simulator for teaching computer organization," in Proceedings of the 42nd ACM Technical Symposium on Computer Science Education, ser. SIGCSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 365–370. [Online]. Available: https://doi.org/10.1145/1953163.1953272

[12] G. S. Wolffe, W. Yurcik, H. Osborne, and M. A. Holliday, "Teaching computer organization/architecture with limited resources using simulators," SIGCSE Bull., vol. 34, no. 1, p. 176–180, feb 2002. [Online]. Available: https://doi.org/10.1145/563517.563408

[13] M. D. Black and M. Franklin, "Teaching computer architecture with a graphical pc simulator," in Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, ser. ITiCSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 337. [Online]. Available: https://doi.org/10.1145/1999747.1999848

[14] C. Yehezkel, W. Yurcik, M. Pearson, and D. Armstrong, "Three simulator tools for teaching computer architecture: Little man computer, and rtlsim," J. Educ. Resour. Comput., vol. 1, no. 4, p. 60–80, dec 2001. [Online]. Available: https://doi.org/10.1145/514144.514732

[15] J. C. Moure, D. I. Rexachs, and E. Luque, "The kscalar simulator," J. Educ. Resour. Comput., vol. 2, no. 1, p. 73–116, mar 2002. [Online]. Available: https://doi.org/10.1145/545197.545202

[16] "Ieee standard for verilog hardware description language," IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), pp. 1–590, 2006.

[17] "Ieee standards interpretations: Ieee std 1076-1987, ieee standard vhdl language reference manual," IEEE Std 1076/INT-1991, pp. 1–208, 1992.

[18] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in DAC Design automation conference 2012. IEEE, 2012, pp. 1212–1221.
[19] Y. Li and W. Chu, "Using fpga for computer architecture/organization education," in Proceedings of the 1996 Workshop on Computer Architecture Education, ser. WCAE-2 '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 5–es. [Online]. Available: https://doi.org/10.1145/1275152.1275157
[20] M. Hoffman, "An fpga-based digital logic lab for computer organization and architecture," J. Comput. Sci. Coll., vol. 19, no. 5, p. 214–227, may 2004.
[21] K. Zhang, Y. Chang, M. Chen, Y. Bao, and Z. Xu, "Computer organization and design course with fpga cloud," in Proceedings of the 50th ACM Technical Symposium on Computer Science Education, ser. SIGCSE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 927–933. [Online]. Available: https://doi.org/10.1145/3287324.3287475
[22] Y. Zhang, K. Chen, and W. Liu, "Online judge for fpga-based lab projects in computer organization course," in Proceedings of the ACM Turing Celebration Conference - China, ser. ACM TURC'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 15–20. [Online]. Available: https://doi.org/10.1145/3393527.3393531
[23] M. Black, "A hardware simulator for teaching cpu design," in Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education, ser. ITiCSE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 380. [Online]. Available: https://doi.org/10.1145/2325296.2325396
[24] C. Burch, "Logisim: A graphical system for logic circuit design and simulation," J. Educ. Resour. Comput., vol. 2, no. 1, p. 5–16, mar 2002. [Online]. Available: https://doi.org/10.1145/545197.545199
[25] H. Neemann, "Digital," https://github.com/hneemann/Digital.
[26] J. M. Kerridge and N. Willis, "A simulator for teaching computer architecture," SIGCSE Bull., vol. 12, no. 2, p. 65–71, jul 1980. [Online]. Available: https://doi.org/10.1145/989253.989264
[27] R. E. Bryant and D. R. O'Hallaron, "Introducing computer systems from a programmer's perspective," SIGCSE Bull., vol. 33, no. 1, p. 90–94, feb 2001. [Online]. Available: https://doi.org/10.1145/366413.364549
[28] C. Norris and J. Wilkes, "Yess: A y86 pipelined processor simulator," in Proceedings of the 45th Annual Southeast Regional Conference, ser. ACM-SE 45. New York, NY, USA: Association for Computing Machinery, 2007, p. 150–155. [Online]. Available: https://doi.org/10.1145/1233341.1233369
[29] K. L. Pokorny, "Creating a computer simulator as a cs1 student project," in Proceedings of the 46th ACM Technical Symposium on Computer Science Education, ser. SIGCSE '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 42–47. [Online]. Available: https://doi.org/10.1145/2676723.2677210
[30] R. E. Bryant, "Term-level verification of a pipelined cisc microprocessor," Carnegie Mellon University School of Computer Science, Tech. Rep. CMU-CS-05-195, December 2005.
[31] ——, "Formal verification of pipelined y86-64 microprocessors with uclid5," Carnegie Mellon University School of Computer Science, Tech. Rep. CMU-CS-18-122, October 2018.
[32] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 111–122. [Online]. Available: https://doi.org/10.1145/964001.964011
[33] P. Siguad, "PEGGED." [Online]. Available: https://github.com/PhilippeSigaud/Pegged
[34] "LALRPOP." [Online]. Available: http://lalrpop.github.io/lalrpop/index.html