# Teaching Out-of-Order Processor Design with the RISC-V ISA

Stephen A. Zekany*
University of Michigan
szekany@umich.edu

Jielun Tan*
University of Michigan
jieltan@umich.edu

James A. Connolly
University of Michigan
conjam@umich.edu

Ronald G. Dreslinski
University of Michigan
rdreslin@umich.edu

## ABSTRACT

We describe our experience teaching an undergraduate capstone (and elective graduate course) in computer architecture with a semester-long project in which teams of five students design and implement an out-of-order (OoO) pipelined processor core using the open-source RISC-V instruction set. The course content includes OoO scheduling algorithms for instructions to exploit instruction-level parallelism (ILP), example designs, caching, prefetching, and virtual memory. The labs and projects help students gain proficiency with the SystemVerilog language.

Students use the concepts learned in class to design processors with the goals of achieving correctness and high performance for a suite of representative test programs. Using RISC-V enables students to validate and benchmark their designs by compiling test programs using GCC with a custom linker. By collaborating as a team, students learn how to write and debug a large code base over the two-month project.

For computer architecture educators, we describe technical aspects of the final project and common advanced features implemented by students. We hope describing our experience serves not only to demonstrate a method of teaching modern computer architecture, but also to inspire other course designs centered around other aspects of modern computer architecture (GPUs, FPGAs, hardware/software codesign, etc). We have open-sourced our lab and project materials to enable others to teach similar courses.

## CCS CONCEPTS

• **Social and professional topics → Computer engineering education**.

## KEYWORDS

teaching computer architecture, computer hardware course, undergraduate design project, computer engineering team project

**ACM Reference Format:**
Stephen A. Zekany, Jielun Tan, James A. Connolly, and Ronald G. Dreslinski. 2021. Teaching Out-of-Order Processor Design with the RISC-V ISA. In *ISCA Workshop on Computer Architecture Education (WCAE '21), June 17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Both authors contributed equally to this research.


*WCAE '21, June 17, 2021, Virtual Event, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Global demand for microprocessors continues to reach new peaks on a yearly basis [2]. Simultaneously, the continued need for high performance in the data center and in mobile and embedded devices is driving increasing complexity and novel designs in computer architecture, especially as Moore's law is slowing [7, 13, 15, 16]. In the past thirty years, state-of-the-art CPUs have progressed from in-order pipelined cores built with roughly a million transistors (e.g. Motorola 68040) to multicore, out-of-order (OoO), deeply-pipelined processors with billions of transistors (e.g. Apple A14). Features such as multi-level caching, cache coherence protocols, and simultaneous multi-threading increase performance but also increase the complexity of the design and validation processes.

However, computer architecture education has not kept pace in a world where even tiny embedded devices generally contain pipelined CPUs [1]. We believe that understanding modern computer architecture is not only a necessary grounding for students interested in hardware design careers, but also highly relevant for those learning to build efficient and reliable software. Our courses should not only teach students how processors are built today and discuss the inherent design trade-offs, but give students hands-on experience with an out-of-order pipeline and multi-level caching.

In this paper, we describe our experience updating and teaching an undergraduate capstone design course with the primary goals of enabling students to develop a deep understanding of how modern processors function and learning to work as a team on a challenging hardware design project. For computer architecture educators, we detail the final project and common advanced features implemented by students, including cache improvements, superscalar design, advanced branch predictors, and multicore/multithread designs. We hope describing our teaching experience serves not only to demonstrate one potential method of teaching modern computer architecture to undergraduate students, but also to inspire other course designs centered around other aspects of modern computer architecture (GPUs, FPGAs, hardware/software codesign, etc).

Our semester-long course includes background in modern computer architecture, case studies of modern hardware, and a hands-on, design-from-scratch project building an OoO processor that supports a subset of the RISC-V [21] instruction set. The lectures cover a review of basic in-order pipelining through the evolution of various OoO microarchitectures to the present, including multiple ways to exploit instruction-level parallelism with OoO execution (as shown in Figure 1).

We updated the course to use the open-source RISC-V instruction set architecture (ISA) (replacing the Alpha ISA [4]) to provide students experience with an emerging, interesting, and relevant architecture, and to make it easier for students to write test cases. While we provide test cases hand-written in assembly, with the RISC-V ecosystem students can now write their own C code and compile it using GCC, reducing the effort required for validation.

The specific contributions of this paper are as follows:

- We describe the necessary structure, schedule, and support to instruct students building synthesizable, out-of-order RISC-V processors from scratch.
- We explain the advantages of RISC-V and the associated tools and logistics.
- We survey students on their experience with the updated course design and reflect on the experience as instructors.

The paper is organized as follows: we first share background about the content of the class (Section 3). We describe the final group project in detail (Section 4) and the logistics necessary to support it (Section 6). Finally we share post-project student survey results and discuss the project experience (Section 7).

In the interest of enabling other instructors to offer a similar course, we have collected current versions of our course assignments and starter code into a public GitHub repository [5]. Access to a private repository with instructor solutions and support code is available by contacting the authors.

## 2 RELATED WORK

Recently, a number of RISC-V implementations of single-cycle and pipelined processors for educational use have been developed. For example, the Davis In-Order (DINO) CPU [10], a Chisel-based five-stage RISC-V pipeline implementing RV32I. DINO includes debugging tools and functional tests, allowing students to understand the underlying architecture instead of implementing helper tools. Similarly, WebRISC-V [8] and the BRISC-V Platform [6] are web-based simulators for RISC-V assembly, with included compilation and debugging tools. These teaching tools are helpful for designing assignments in an introductory class, enabling the instructor to design assignments around a known microarchitecture and system.

Our course project differs from these in that we teach students to design the core of an out-of-order CPU (see Figure 3 and Section 4). Students may have taken a course utilizing these tools as a prerequisite. In fact we have students work with a five-stage RISC-V processor similar to DINO, but the bulk of our class is focused on teaching students to develop a (more complex) out-of-order core, which is the foundation of modern high performance processors.

## 3 COURSE DESCRIPTION

The class is designed as an advanced undergraduate (and introductory graduate-level) course in computer architecture. Students are expected to have an understanding of undergraduate computer organization (topics such as pipelining, caches, and virtual memory), software design (simple data structures and algorithms) and digital logic (logic gates, combinational and sequential logic, and ideally experience writing Verilog or another HDL). The end-goal is to provide students with a detailed understanding of how a single-core CPU, including the core pipeline and cache-memory hierarchy, is designed and interacts with software (see Figure 2), and to demonstrate the numerous trade-offs in design and implementation.

In this section we describe the lecture, lab, and individual project components in detail from the perspective of a computer architecture educator. The final project – detailed design of a processor using SystemVerilog, performed in teams of four to five students – is described in detail in Section 4.

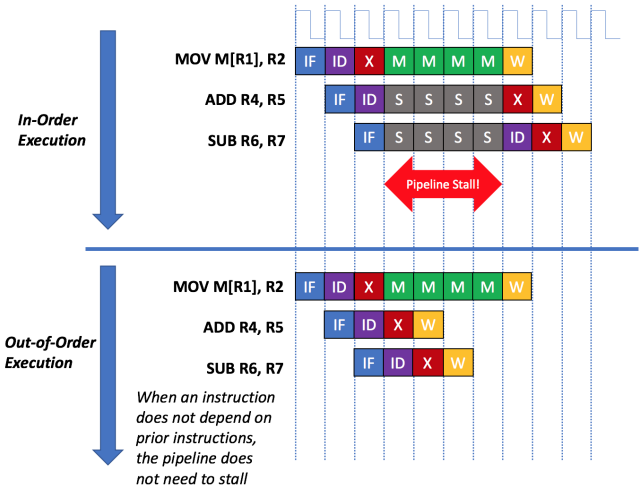


**Figure 1: In-order (top) vs out-of-order execution (bottom). An out-of-order pipeline schedules instructions based on the availability of execution units rather than program order. Instructions can finish execution in any order and need not stall unless waiting for a result from a prior instruction.**

### 3.1 Lecture and Lab

When considering the lecture schedule each term, we have one simple goal: provide students with enough knowledge to successfully complete the final project. This means teaching a number of topics in the first 6-7 weeks to give students sufficient background to write a final project proposal document. The first half lecture schedule consists of the following topics:

- Review of pipelined processors and hazards, including control, structural, and data hazards. Includes a high-level discussion of data forwarding and speculation-and-squash.
- Introduction to Tomasulo's algorithm [19, 20], as originally implemented without a reorder buffer. Possibly including scoreboarding at the instructor's discretion.
- Local and global branch predictor theory and example designs [24], as well as more advanced designs like G-share and TAGE [11, 18, 23].
- Intel P6 [17] and MIPS R10K [22] processors. These designs differ from the earlier Tomasulo designs by the introduction of a reorder buffer, allowing precise interrupts.

The second half lecture schedules includes memory and caches [9], power usage, multiprocessors, and finally special topics in computer architecture such as multithreading, case studies of new or unusual ISAs (e.g. IA64), static optimization, and prefetching [12].

There are seven lab sessions, each containing around 45 minutes of instruction. Six labs have assignments, due a week later. The labs teach practical skills such as SystemVerilog, the industry-standard Synopsys toolchain, shell scripting, and version control. The topics of labs are enumerated in Table 1. Each lab is designed to be completable in about an hour or two. Each student interacts personally with course instructor to get their lab checked upon completion, either in-class or later during office hours. We find this allows students to begin to interact one-on-one with the course staff and get some quick feedback about their coding and design style.

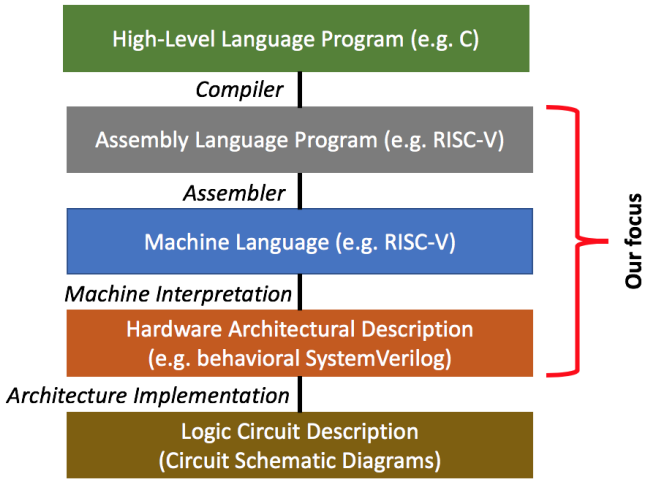


**Figure 2: A hierarchy of computing stack from software to hardware. We focus on the interaction of low-level software (assembly and machine code) with the top level of hardware.**

| Lab | Content |
|---|---|
| 1 | Getting Started with SystemVerilog HDL |
| 2 | Synopsys Build System and Tools |
| 3 | SystemVerilog Style Guide |
| 4 | Version Control and Intro to RISC-V ISA |
| 5 | Linux Shell Scripting in Bash |
| 6 | SystemVerilog Features |
| 7 | Final Project Memory System |

**Table 1: Topics for each lab lecture**

## 3.2 Individual Projects

Students review basic digital logic and computer architecture by completing three individual projects. These projects develop students' skill at writing combinational and sequential logic, as well as troubleshooting and modifying an existing code base.

*3.2.1 Combinational Logic (Priority Selector).* This project requires students to write priority selectors using combinational logic and gives an introduction to hierarchical module design by using two-bit priority selectors to build larger priority selectors. This project introduces students to common Verilog constructs such as assign statements and logic blocks.

*3.2.2 Finite State Machine (Integer Square Root).* We give students a sample multiplier with a four-stage pipeline design which computes sums of partial products across multiple clock cycles. First, students are asked to reconfigure the multiplier's pipeline stages, synthesize the design, and benchmark performance. This ensures students know how to use Design Compiler and perform post-synthesis simulation and benchmarking, as well as understanding the tradeoff between critical path and adding additional pipeline stages.

Then students design a hardware module to compute the integer square root of a 64-bit value (effectively performing binary search in hardware). This gives students experience designing and implementing a finite state machine in Verilog.

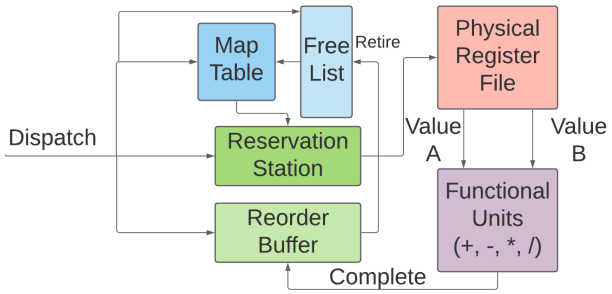


**Figure 3: Example implementation of a MIPS R10K pipeline. Architected registers are renamed to a physical register. Instructions are issued from the reservation station to functional units, and results are reported to the reorder buffer.**

| Easier Features | More Difficult Features |
|---|---|
| Advanced Branch Predictor | Early Branch Recovery |
| Associative Cache | Multithreading (SMT) |
| Prefetcher | Superscalar (2-N ways) |
| Return Address Stack | Writeback Cache |

**Table 2: Examples of commonly implemented advanced features. Some features such as a superscalar design must be integrated into the entire pipeline, while smaller features such as a stride prefetcher can be decoupled from the pipeline and often take only a day or two to complete.**

*3.2.3 In-Order Processor.* The last individual project asks students to fix pipeline hazards in a five-stage RV32IM processor. RV32IM is the integer and multiplication extension of RISC-V, supporting bitwise operations, conditional and unconditional branches, loads/stores, and integer add and multiply (no division or floating point). We provide students with a synthesizable SystemVerilog implementation of a standard five-stage pipeline. A top-level module integrates all pipeline stage modules with a simple memory interface and testbench. The memory interface is a simple read/write bus design which connects to a testbench function (we do not use synthesized memory).

The pipeline as given has no control for hazards. Arbitrary RV32IM programs (of which we provide a test library) can run correctly by inserting four noop instructions between each instruction. Students are instructed to fix three hazards:

- Data hazard created by register read occurring in the decode stage. Students must implement detect-and-forward logic.
- Control hazard created by branches. Students are to speculate "not taken" to avoid stalling the pipeline. If the branch resolves to "taken", squash the affected pipeline stages before the instruction commits.
- Structural hazard created by the (simulated) memory bus. As this bus can only accept a single request at a time, contention is created between the memory pipeline stage and the fetch stage. Students must stall the fetch stage for one cycle.

We encourage students to think carefully before adding additional logic, especially additional registers. Additional registers means additional allowable states for the CPU, and this creates proportionally more bugs to be found during testing.

## 4 PROCESSOR DESIGN PROJECT

The final project occurs during the last eight weeks of class. Students form groups of four or five to build a correct implementation of a high-performance out-of-order processor which supports RV32IM. Performance is measured by time for each test case to execute, as a function of the clock period and average instructions executed per cycle (IPC). The project is complex enough that having a group of five is helpful even with increased communication overhead.

### 4.1 Project Design Principles

The project has two components: the baseline and advanced features. The baseline consists of an OoO pipeline following either Intel P6- or MIPS R10K-style microarchitecture, along with instruction and data caches, a branch predictor, and a load-store unit. Figure 3 shows an example of a MIPS R10K style OoO pipeline with dispatch/issue logic and reorder buffer (ROB).

*4.1.1 Starter Code.* Students can freely reuse code from the in-order processor project, of which the most useful modules are the instruction decoder and the arithmetic logic unit (ALU). We include a simple instruction and data cache (32 lines, direct-mapped). To facilitate students' designs, we provide several reference utility modules such as priority selectors, the former of which can be used throughout the project wherever arbitration is needed. The in-order processor also serves as a ground-truth simulator, generating a correct memory map after the program execution and a log of all instructions executed and register writes.

*4.1.2 Ensuring Out-of-Order Execution.* We require students to use the partial-products pipelined multiplier from the individual project to perform any multiplication instructions. This ensures that at least one instruction will take multiple cycles and have a simple explorable design space of clock period vs. latency. Even a simple benchmark consisting of a single multiply instruction followed by a number of add instructions in a repetitive loop will then be executed out-of-order.

Students make their own determination of how to schedule available instructions to execution units in their pipeline (commonly referred to as the "issue" stage). Different designs are desirable; for example scheduling high-latency instructions early, and identifying older instructions to avoid starvation. In practice these designs rely on an optimized priority selector and can be quite difficult to do in a single, short clock cycle. We instead recommend students schedule a random available instruction for execution and optimize the issue logic later if desired.

### 4.2 Design Process

Before the project officially begins, instructors hold meetings with each group to discuss which advanced features they propose to implement. Table 2 shows a list of advanced features that groups can add to their design. During the course of the project, we have two mandatory and one optional milestones to measure students' progress and ensure they remain on-track.

*4.2.1 Writing Modules.* All groups begin by completing one fully-debugged module at the first milestone. Students are encouraged to code as a whole group for this phase of the project to agree on coding style and practices. We recommend each group construct a high level architecture and decide on the interfaces used throughout the design. The first module serves as a reference point for the interfaces between different units. We test and grade the first module and testbench by introducing small bugs and seeing whether the testbench will catch them (e.g., flipping an "OR" to "AND", or subtracting instead of adding, etc). This emphasizes the importance of unit testing and incentivizes students to identify bugs early.

We teach in lab how to write parameterizable modules and require that the first module be parameterizable as parameterization allows for easier changes and better analysis of the design later on. Before writing the first module, the instructors explain how to build basic hardware data structures such as first-in-first-out (FIFO) queues, stacks, and bit-maps. Afterwards, they provide more high level design advice and serve more as consultant rather than instructors, since we want to give the students as much freedom as possible in their design choices. After the first module, we recommend groups have at least two people working together so that if someone is absent later on, there is at least one other person who can debug the module.

*4.2.2 Pipeline Integration.* By the second milestone meeting, we expect groups to have a functional OoO pipeline core capable of fetching and executing instructions (but no data memory operation yet). We recommend students begin integrating modules a week before the milestone and finish advanced features unrelated to memory. Integration is a difficult and time-consuming process. Poor coding style often leads to ambiguous behavior, which may cause bugs. As SystemVerilog simulators tend to be tolerant of ambiguous code, these bugs may be difficult to find post-simulation. To mitigate this, we are meticulous about teaching correct coding style from the beginning of the semester. We suggest all groups reserve one week for the final integration with the memory system.

*4.2.3 Debug and Synthesis.* We recommend students unit-test, ensure adequate test coverage, and synthesize each of the individual modules as pipeline integration is easier when they do more unit-testing earlier. Students have access to a test suite of more than 35 test cases and can optionally write their own in C or assembly. Students can also contribute a new test case back to the class test suite. Once a group has a working pipeline, we recommend they log the history of register writes tagged with the corresponding instructions, as this is the only way to verify correctness without being able to modify memory. Also, groups can identify exactly which instruction is incorrectly executed if it performs a register write. This log also serves as a list of committed instructions, which can be compared to the log generated by the in-order core to see if control-flow instructions such as branches and jumps have taken the correct direction. After the second milestone, when groups have a fully functioning pipeline that can modify memory, groups check the memory map their processor produces after program execution with the provided in-order processor.

Logical synthesis translates the HDL into a netlist of standard cells, which can be either macro blocks such as adders or simple logic gates. This is a critical step in hardware design since most designs are written by behavior and not their hardware construction, but ultimately must be capable of being built into actual hardware. A typical problem that students face is using uninitialized values

that are not reset or using values without checking validity, resulting in garbage values propagating down the pipeline. In behavioral simulation, this can be hidden, since there are usually some values passed along and do not affect other logic. Post synthesis however, the standard cell registers generate a value of X instead of a 0 or 1, which means unknown. Xs can propagate down to other logic to produce unknown values, effectively disrupting the processor's state.

The other aspect of synthesis is performance. Groups are benchmarked by their total run time of the test programs, which is the product of the number of cycles executed and clock period. Synthesis includes static timing of the design and reports critical paths, area, and other potential problems such as circular logic. To reduce the latency of the critical path, groups can add more pipelining in the logical chain, reduce the size of data structures or sometimes completely give up certain functionalities. They learn the important trade-off between cycles per instruction (CPI) vs. clock speed, a problem that every logic designer faces.

## 4.3 Why RISC-V?

Before RISC-V the final project used the Alpha ISA. The obvious advantages of migrating to RISC-V are relevance and adoption by industry; for example current commercial products running on RISC-V instructions exist. Furthermore, the open-source aspect makes RISC-V available to everyone in the world without having to worry about licensing and export control. For our course, aside from implementing the necessary infrastructure and writing new supporting materials, we encountered no downsides to switching to RISC-V. Students have a chance to apply more knowledge from class in the real world. However the important advantages of using RISC-V in the classroom are more subtle, as we detail below.

*4.3.1 Better test programs.* RISC-V supports the full GNU toolchain including a compiler and binary utilities such as objdump, which disassembles executables. Previously, we had only assembly-based test programs that were hand-designed mostly to test for corner cases. While these are useful to find bugs, they are poor metrics for measuring performance because they are short and do not reflect real workloads. We have a few assembly test programs that represent common algorithms such as quicksort, but these are still limited in number of instructions. We found that groups became discouraged from implementing certain features if the benefits cannot be reflected within a short execution time, such as advanced branch predictors requiring significant time to warm up.

With GCC, we can develop large test programs with reduced effort and workloads are more representative of real applications. We still create assembly programs for edge cases, but now we also have much larger C programs that serve as better indicators of performance. This also makes it easier for students to write their own tests and more likely to contribute a test case back to the class. Another advantage is the ability to change optimization level during compilations. By optimizing the program at different levels, different versions of the same program can be generated for every C program. To maximize performance, we use higher optimization levels such as O2 or O3 for the grading test suite. Before the revision, we had 20 public test cases. Now there are over 35, of which the C programs can be compiled into several versions with different optimization levels to catch bugs.

*4.3.2 Simplicity.* RISC-V is a clean-sheet design suitable for educational use, and is unencumbered by the need for legacy support. On the other hand, Alpha has not been actively developed for more than 15 years. The students must support the RV32IM instruction set except the system calls (emulation of divide instructions is optional), totaling about 40 instructions. While a subset of any commercial ISA may achieve similar results, it is difficult to provide compiler support for only the subset. Since RISC-V is highly modularized, that support is built-in. Fewer than 10 pages of the ISA documentation supports all the instructions for the class.

*4.3.3 Understanding of the computing stack.* Although not the main purpose of the class, in the end students can see that the majority of the C library functions can be represented by permutations of about 40 instructions. It is valuable for students to able to draw the connection between software and hardware. Seeing how data structures map to memory and commonly used algorithms execute on hardware they have built themselves is a unique experience.

## 4.4 Student Reports and Project Grading

After submitting their project, students write a final report detailing their design and advanced features. We are particularly interested in performance evaluations of advanced features and as instructors we often learn one or two new things each semester. In some cases, we ask students to demonstrate an advanced feature to prove it works properly. Then, each group does a 10-minute talk on their design to the rest of the class, again showcasing any unique advanced features or findings.

We have an autograder harness that runs all benchmark programs and checks correctness via final memory state. The autograder also extracts the cycle count and clock period for each project, and computes a normalized performance score based on relative performance for each group. The most painful process is synthesizing each group's project to ensure it works properly after synthesis (we don't run every single program post-synthesis as this would take too much time during final exam time). We review code by hand to check for cheating, though happily cheating appears to be quite rare.

# 5 ADVANCED FEATURES

Student-designed advanced features are an important component of the final project, allowing students to experiment with different techniques to increase performance. To earn credit for advanced features, a group must demonstrate correct implementation of a feature and show the performance impact on certain test programs or explain why performance was not impacted. In this section we describe advanced features commonly implemented by project groups, the relative level of difficulty, and the impact on performance.

## 5.1 Specific Advanced Features

*5.1.1 Advanced Branch Predictor.* Students are expected to, at a minimum, provide a branch target buffer and simple (e.g. bimodal) branch predictor. In a typical semester one or more groups will implement a global, Gshare [11], TAGE [18], or other advanced branch

predictor. Since any branch predictor is ultimately a speculative structure, demonstrating an advanced branch predictor follows the algorithm correctly is a more challenging verification problem than ensuring correctness of execution. Also, advanced branch predictor designs such as TAGE [18] will not provide a performance improvement on our standard test programs because none of the programs provide sufficient instructions to warm-up the predictor. We ask students to provide (if possible) an example program that (even if contrived) performs better with their branch predictor than with trivial branch prediction.

*5.1.2 Superscalar (2-N ways).* Most groups implement a superscalar design by widening each pipeline stage to support more than a single instruction. We constrain students to loading or storing a single 64-bit block from main memory each cycle, so there is an obvious benefit to supporting at least 2-way superscalar, with diminishing returns for 3-way or higher (depending on cache implementation). We encourage students to design each module to be superscalar from initial design, as rewriting modules to be superscalar late in the project is more challenging. Generally students will see a substantial speedup on a number of test programs, making this one of the best advanced features for the effort invested.

*5.1.3 Cache and Load/Store Queue Improvements.* Students are provided a simple 32-line direct-mapped data and instruction cache. Many improvements are possible and conceptually simple, though the sum of these optimizations often leads the data cache to be the most complex module in their design. Adding writeback provides a substantial benefit by reducing the amount of traffic on the memory bus. A set-associative design reduces the number of conflict misses. Implementing a 2-way or 4-way cache is fairly straightforward; a fully-associative cache is challenging.

*5.1.4 Early Branch Recovery.* A speculative branch must be resolved prior to retirement (commit) from the reorder buffer (ROB). A branch can be resolved earlier and the ROB partially cleared to correct for the new branch target. This can be challenging to implement while constrained to a clock period, as an algorithm is necessary to check multiple instructions (and keeping track of which have been resolved) every cycle. This provides a reasonable amount of performance improvement for the effort required.

*5.1.5 Prefetcher.* Prefetching is one of the most common features for students to implement and is conceptually quite simple: upon fetching from memory, fetch more than the current instruction. With a 32-bit ISA and 64-bit memory layout, storing the second instruction in each block is clearly advantageous. Simple stride prefetchers which fetch the next block or two tend to provide good performance improvement for the effort invested, while more-complex prefetching designs are harder to get working properly and debug. Each group cannot exceed 32 lines of cache, so the potential for prefetching without evicting valid instructions is limited.

*5.1.6 Exceptions.* While we do not support asynchronous exceptions (interrupts) yet, we do let students implement exceptions triggered by system calls or certain instructions that need to be emulated, such as divide. To achieve this, students have to support system calls such as MCALL and MRET and integrate the Control Status Registers (CSRs) into their core. For correctness and ordering, students must ensure the system call is the only instruction in the pipeline, i.e. the ROB must be empty and no further instructions can be dispatched until the system call is retired. We provide students with a software exception handler and techniques to emulate divide and remainder instructions. We typically give groups a flat percentage boost, since this feature does not improve performance.

*5.1.7 Multithreading (SMT).* The principal challenge of SMT is to implement the load-link and store-conditional instructions in the core pipeline and cache hierarchy. This requires students to understand mutexes and shared memory, but often students know of these topics from other courses or experience. Once students have hardware support for the LL/SC opcodes, they can design an experiment where two threads will increment a counter. Due to the complexity of this feature, we typically schedule a demo session where the group will convince us they have implemented the feature correctly. We typically award student groups who implement this successfully with a 30% boost on performance.

*5.1.8 Multicore.* Groups implementing a multicore design are encouraged to design a simple core quickly so they have a few weeks to work on cache coherence. To get full points for this feature, a group must demonstrate implementation of at least two cores communicating with an MSI protocol and execution of interleaving threads. For groups implementing this feature we will typically host an additional lecture or two to teach MSI/MESI protocol a few weeks early. As with multithreading, we don't grade this feature directly but instead ask groups to demonstrate their design.

## 5.2 Student Choices

Students, of course, prefer advanced features with the most performance gain for the least amount of work. Nearly all groups (except one or two each semester) implement superscalar and at least a few cache improvements. Groups tend to use the R10K microarchitecture more often than P6, although occasionally an ambitious group will do something quite different, such as a Palacharla-inspired design [14]. Many groups build a more-advanced branch predictor (i.e. more complicated than a simple bimodal predictor), but groups that implement early branch recovery often stick with a simpler predictor. Multicore, multithreading, and exception handling remain niche choices as each is much harder to implement than the more-common features and cannot be easily decoupled should it prove difficult to get working. These features also have a relatively high opportunity cost, as a group tends to be kept quite busy and without sufficient time left over to implement simple features.

# 6 LOGISTICS AND SOFTWARE SUPPORT

*6.0.1 Schedule and Staffing.* The OoO design project is the last eight weeks of the class, spanning more than half the semester. By the midterm, the students will have seen all lectures teaching the OoO pipeline concepts. There are several milestones for the final project, and at each the instructors meet with every group individually. We find milestone meetings are enormously helpful for keeping students on-track and helping them resolve problems early. Further details on schedule, grading and staffing are omitted here for space but available in our SIGCSE paper [25].

## 6.1 CAD Tools and Cost

*6.1.1 SystemVerilog Simulation and Synthesis.* Synopsys VCS (Verilog Compiler Simulator) is used to perform simulations for behavior and structural SystemVerilog. While VCS is proprietary, educational licenses can be obtained via Synopsys' academic programs [3]. Our college's IT department installs and maintains VCS on the campus network, and we provide a makefile for students to easily adapt for their own projects.

Should there be a concern over obtaining licenses, there are free to use, open-source alternatives for SystemVerilog simulation such as Verilator and Cocotb, which support most of the SystemVerilog functionalities while only missing a few features important to industry but not used in this class such as Universal Verification Methodology (UVM) libraries. A third option is to obtain simulation tools from FPGA manufacturers such as Altera and Xilinx, who also have academic programs.

*6.1.2 Synthesis.* We use Synopsys Design Compiler for SystemVerilog synthesis to build structural netlists (also described in SystemVerilog) of logic gates from behavioral SystemVerilog. This tool can also be obtained via Synopsys' educational programs. As with VCS, Design Compiler is available on our campus network. Design Compiler performs better with additional memory, and we find students have the best experience synthesizing their projects on a standalone workstation in one of our computing labs instead of via the shared campus login servers. We provide a TCL script to set all the necessary parameters, which has the additional benefit of leveling the playing field on performance between groups.

Open-source alternatives exists such as Yosys, and academic commercial alternatives are sold by both Altera and Xilinx. For the standard cell library we use Synopsys' included general technology libraries. Alternative open source technology libraries are also available. Since this is not a physical design class, the technology library does not need to be advanced. Design Compiler reports power, area, and critical path numbers for the design, but we are concerned only with critical path.

*6.1.3 RISC-V GNU Toolchain.* The GNU suite of toolchains are used for compilation and linking. For the most recent semester we used GCC 9.2 with flags to support the correct subset of RISC-V instructions. We developed a custom linker script to support placing text and data in the specific memory regions that complies with the memory model provided to students as part of the final project. Additionally, we need the RISC-V version of elf2hex to convert the ELF (executable) produced after linking into a human readable hex file, which is then read by the testbench. All of these tools are open-source and standard for software development. RISC-V is an officially supported target when building GCC. The GNU toolchain also includes the binary utilities like objdump, which is used to disassemble the ELF and provide the assembly.

# 7 STATISTICS AND DISCUSSION

In the interest of space, we have omitted some details in this section. Full details are available in our SIGCSE paper [25].

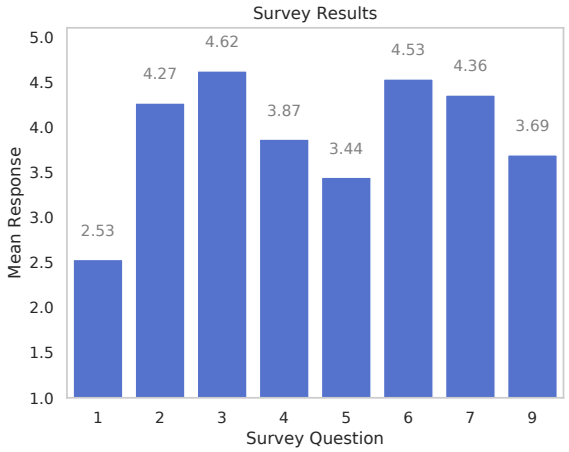


**Figure 4: Responses for survey questions in Table 3 except Q8 (for which 56% of students indicated they used the compiler) and Q10 (for which the mean response was 2.63/3).**

## 7.1 Student Survey

In Fall 2019, we surveyed students after the project to obtain their opinion of the project's level of difficulty, the value of the compiler in writing their own test cases, and how it advanced their own skills and understanding of computer architecture (Table 3). A total of 42 students responded to the survey and their responses are shown in Figure 4. Notably, students believed the project improved their SystemVerilog skills and their understanding of computer architecture concepts including the hardware/software interface. Students believed the pipeline integration and synthesis phases were more difficult than the module design phase, likely due to the number of bugs exposed during these phases. We also found only about half the class used the compiler, but those who did found it useful to write their own test cases for debugging.

## 7.2 Discussion

In a typical recent semester, about 5% of students fail or withdraw, but we try to intervene early when students do poorly on the individual projects, miss meetings with their group, or turn in low quality milestone code. Students who fail the class generally fail because they perform very poorly on both the individual projects and the group project. On the other hand, a group almost never performs poorly enough on the project to fail the class, because all groups end up with at least a semi-functional OoO processor. We find EE students often need help learning how to write disciplined code and use version control. For CS students, we help them transition from high level languages to HDL. None of this is to say our course is perfect, nor that we have anticipated every possible situation. We are constantly making improvements based on student and instructor feedback.

As instructors we have debated the value of certain advanced features and whether or not students should spend time on them. We generally do not allow students to work on hardware features orthogonal to the goals of the class, such as advanced cache coherence (e.g. MOESI) support. Simultaneous multithreading is contentious as well (pun intended) as it is tricky to implement properly.

| # | Question | Response Range |
|---|---|---|
| 1 | Before this class, how comfortable were you at reading and writing Verilog? | 1 (very uncomfortable) - 5 (very comfortable) |
| 2 | How comfortable are you at reading and writing Verilog now? | 1 (very uncomfortable) - 5 (very comfortable) |
| 3 | This project improved my understanding of computer architecture concepts (e.g. out-of-order pipelines, memory systems, performance measurements, etc). | 1 (strongly disagree) - 5 (strongly agree) |
| 4 | I have a better understanding of how software interacts with hardware. | 1 (strongly disagree) - 5 (strongly agree) |
| 5 | Rate the difficulty of the module design and unit testing phase of the project. | 1 (not difficult) - 5 (very difficult) |
| 6 | Rate the difficulty of the pipeline integration and debugging phase of the project. | 1 (not difficult) - 5 (very difficult) |
| 7 | Rate the difficulty of the pipeline synthesis and testing/debugging phase. | 1 (not difficult) - 5 (very difficult) |
| 8 | Did you use the RISC-V compiler to compile your own test cases? | 0 (no) or 1 (yes) |
| 9 | How valuable was the compiler in allowing you to debug your pipeline with test cases you wrote in C? | 1 (not at all valuable) - 5 (very valuable) |
| 10 | This class made me more interested / less interested / did not change my interest in computer architecture design and/or research. | 1 (less interested), 2 (did not change), or 3 (more interested) |

**Table 3: Survey Questions**

GPUs, FPGAs, and application-specific hardware designs are becoming increasingly popular as Moore's law slows. Beyond teaching skills such as HDL coding and CPU microarchitecture, we see a potential for technical courses to teach domain-specific hardware/software codesign. For example, our school offers a GPU programming course, and an FPGA course is in development.

## 8 CONCLUSION

We have described our experience teaching a design course in which students build an OoO processor supporting a subset of the RISC-V ISA. Changing ISAs was a worthwhile investment due to the advantages RISC-V offers compared to Alpha despite the upfront work. We hope this description is useful to others, and we encourage you to make use of or modify our materials if you find them useful!

## Acknowledgement

We are grateful to Austin Rovinski for beta testing the RISC-V project, and Siying Feng and Xueyang Liu for teaching. We thank the instructors who contributed to the course: Mark Brehob, Tom Wenisch, Jon Beaumont, and Will Cunningham. Finally we thank our students without whom this project would not be possible!

## REFERENCES

[1] 2020-08-27. *Arm M4 Processor.* https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4

[2] 2020-08-27. *Global Microprocessor Market Will Reach USD 8,894 Million By 2025.* https://www.globenewswire.com/news-release/2019/05/10/1821796/0/en/Global-Microprocessor-Market-Will-Reach-USD-8-894-Million-By-2025-Zion-Market-Research.html

[3] 2020-08-27. *Synopsys Academic Programs.* https://www.synopsys.com/community/academic-programs.html

[4] 2020-08-27. *Wikipedia: DEC Alpha.* https://en.wikipedia.org/wiki/DEC_Alpha

[5] 2020-12-01. *GitHub Repository: OpenCompArchCourse.* https://github.com/jieltan/OpenCompArchCourse

[6] 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 *Proceedings of the Workshop on Computer Architecture Education*. 1–8.

[7] Lieven Eeckhout. 2017. Is Moore's Law Slowing Down? What's Next? *IEEE Micro* 4 (2017), 4–5.

[8] Roberto Giorgi and Gianfranco Mariotti. 2019. WebRISC-V: a Web-Based Education-Oriented RISC-V Pipeline Simulation Environment. In *Proceedings of the Workshop on Computer Architecture Education*. 1–6.

[9] Norman P. Jouppi. 1990. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *SIGARCH Comput. Archit. News* 18, 2SI (May 1990), 364–373. https://doi.org/10.1145/325096.325162

[10] Jason Lowe-Power and Christopher Nitta. 2019. The Davis In-Order (DINO) CPU: A Teaching-focused RISC-V CPU Design. In *Proceedings of the Workshop on Computer Architecture Education*. 1–8.

[11] Scott Mcfarling. 1993. *Combining Branch Predictors.* Technical Report.

[12] K. J. Nesbit and J. E. Smith. 2004. Data Cache Prefetching Using a Global History Buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. 96–96. https://doi.org/10.1109/HPCA.2004.10030

[13] Subbarao Palacharla, Norman P Jouppi, and James E Smith. 1996. *Quantifying the complexity of superscalar processors.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[14] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. 1997. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 206–218. https://doi.org/10.1145/264107.264201

[15] Alberto Pirati, Jan van Schoot, Kars Troost, Rob van Ballegoij, Peter Krabbendam, Judon Stoeldraijer, Erik Loopstra, Jos Benschop, Jo Finders, Hans Meiling, et al. 2017. The future of EUV lithography: enabling Moore's Law in the next decade. In *Extreme Ultraviolet (EUV) Lithography VIII*, Vol. 10143. International Society for Optics and Photonics, 101430G.

[16] Parthasarathy Ranganathan. 2017. End of Moore's Law: Or, a Computer Architect's Mid-life Crisis?. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 1–1.

[17] J. Schutz and R. Wallace. 1998. A 450 MHz IA32 P6 family microprocessor. In *1998 IEEE International Solid-State Circuits Conference. Digest of Technical Papers, ISSCC. First Edition (Cat. No.98CH36156)*. 236–237. https://doi.org/10.1109/ISSCC.1998.672450

[18] André Seznec. 2011. A New Case for the TAGE Branch Predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 117–127. https://doi.org/10.1145/2155620.2155635

[19] Dezso Sima. 2000. The design space of register renaming techniques. *IEEE micro* 20, 5 (2000), 70–83.

[20] James E Smith and Andrew R Pleszkun. 1985. Implementation of precise interrupts in pipelined processors. *ACM SIGARCH Computer Architecture News* 13, 3 (1985), 36–44.

[21] Andrew Waterman and RISC-V Foundation Krste Asanovic. 2019. The RISC-V Instruction Set Manual, Volume I: User-Level ISA",.

[22] K. C. Yeager. 1996. The Mips R10000 superscalar microprocessor. *IEEE Micro* 16, 2 (April 1996), 28–41. https://doi.org/10.1109/40.491460

[23] Tse-Yu Yeh and Yale N. Patt. 1991. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture* (Albuquerque, New Mexico, Puerto Rico) *(MICRO 24)*. Association for Computing Machinery, New York, NY, USA, 51–61. https://doi.org/10.1145/123465.123475

[24] Tse-Yu Yeh and Yale N Patt. 1993. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th annual international symposium on computer architecture*. 257–266.

[25] Stephen A Zekany, Jielun Tan, James A Connelly, and Ronald G Dreslinski. 2021. RISC-V Reward: Building Out-of-Order Processors in a Computer Architecture Design Course with an Open-Source ISA. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 1096–1102.