SCOoOTER: A RISC-V Processor Framework and Tool Flow for an Architecture-to-Layout Design Course

Markus Scheck
Yannick Lavan
Christoph Spang
Andreas Koch
{scheck,lavan,spang,koch}@esa.tu-darmstadt.de
Technische Universität Darmstadt
Darmstadt, Hesse, Germany

Abstract

This paper introduces SCOoOTER, an open-source, highly-configurable RISC-V processor designed for courses on advanced processor architecture and design, reaching from architecture down to chip implementation. SCOoOTER offers students the opportunity to explore and extend processor functionality through a modular architecture and a wide range of configuration options, incorporating advanced architectural concepts such as dynamic instruction scheduling, speculative execution, branch prediction, and multi-threading, providing a rich environment for hands-on learning. The processor is integrated with several educational tools, such as automated testing suites, debugging tools, built-in performance evaluation, and FPGA and ASIC design generation, enabling students to efficiently explore and validate different design decisions down to the physical layer. Except for FPGA synthesis and (optional) functional verification, SCOoOTER exclusively relies on freely available, open-source tools to ensure accessibility and avoid cumbersome NDAs, high per-seat EDA licensing costs, and the associated organizational overhead. Our evaluation shows that SCOoOTER achieves competitive performance to the *CVA6* application-class processor, while also being more accessible to novice users due to its education-focused documentation and extensive integration and evaluation tools. Furthermore, SCOoOTER is aligned with widely-used textbooks on processor design, making it an ideal platform for students to learn and apply modern processor architecture concepts. Additionally, we highlight the potential for students to enhance SCOoOTER by implementing additional functionalities, which serves as a hands-on introduction to modern processor design and optimization techniques. Overall, SCOoOTER serves as a powerful educational platform that enables students to engage deeply with processor architecture at all abstraction levels and gain experience relevant to both research and industry practices in processor design.

WCAE '25, June 21, 2025, Tokyo, Japan

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

CCS Concepts

• **Hardware → Integrated circuits**; • **Applied computing → Education**; • **Computer systems organization → Architectures**.

Keywords

computer architecture, CS education, RISC-V, speculation, out-of-order

ACM Reference Format:
Markus Scheck, Yannick Lavan, Christoph Spang, and Andreas Koch. 2025. SCOoOTER: A RISC-V Processor Framework and Tool Flow for an Architecture-to-Layout Design Course. In *Proceedings of Workshop on Computer Architecture Education (WCAE '25).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Many current computer science courses on processor design focus on simple, in-order architectures, system-on-chip design, and/or utilize processors not intended for educational purposes (cf. Section 3), resulting in a high barrier of entry for exploration and modification. Additionally, advanced concepts such as speculation, reordering, superscalarity, and multi-threading are rarely addressed beyond basic theoretical explanations. As a result, students may be familiar with these concepts in theory but often lack the practical knowledge needed to implement them or assess the trade-offs involved (e.g., area requirements or power consumption). Our goal is to bridge this gap and provide resources that enable students to effectively engage even with these advanced topics.

The primary outcome of our efforts is the *Speculative Configurable Out-of-Order Teaching-Enhanced RISC-V Processor* (SCOoOTER). SCOoOTER is a highly configurable, legible, and extensible reference implementation. Legibility is achieved using Bluespec SystemVerilog (BSV), a high-level hardware description language (cf. Section 4.4). Extensibility is supported through generic interfaces, a modular design approach, and the open, extensible RISC-V instruction set architecture. To facilitate integration with existing educational materials, the design is closely aligned with Hennessy and Patterson's widely used computer architecture textbook [13].

SCOoOTER's wide configurability allows for exploration of hardware trade-offs and performance benefits associated with different design choices. Since such trade-offs can vary significantly across platforms, SCOoOTER particularly targets ASIC synthesis. While FPGAs allow for rapid prototyping and are easier to deploy, ASICs provide broader design flexibility and higher performance, making

them more suitable for in-depth exploration of hardware cost. We specifically target an open 130nm ASIC process, where actual fabrication is easily accessible via ChipCreate [5] or similar low-cost multi-project wafer services, enabling students to receive actual physical chips of their designs.

Beyond the processor's features, usability and debuggability are central to ensuring a productive learning experience. SCOoOTER includes pre-configured simulation targets, support for multiple test suites and custom binaries, a graphical pipeline viewer, an extensive tracing infrastructure, and automated implementation for both ASIC and FPGA designs. Except for FPGA synthesis and optional random-stimuli functional verification, SCOoOTER relies exclusively on freely available open-source tools, facilitating easy distribution of all dependencies, e.g., in the form of pre-configured virtual machines.
Key features of SCOoOTER include:

- Support for the RV32I[M][A] instruction set
- Broad configurability (e.g., buffer sizes, execution unit mix, ISA extensions, branch predictors)
- Automated ASIC and FPGA design generation
- Comprehensive debugging, simulation, and testing infrastructure
- Free and open-source availability with detailed documentation

The remainder of this paper is structured as follows: Section 2 outlines the student requirements and intended use cases for SCOoOTER. Section 3 discusses related courses and similar processors, highlighting SCOoOTER's key benefits. Section 4 presents the architecture of SCOoOTER, explains our choice of HDL, and describes the implementation of multithreading, multiprocessor systems, and the provided peripherals. Section 5 details the educational tools and features available in SCOoOTER. Section 6 evaluates SCOoOTER in terms of power, performance, area, and legibility. Section 7 demonstrates an example usage scenario of SCOoOTER. Section 8 summarizes our findings, and Section 9 outlines potential future enhancements.

2 Intended Use Cases and Prerequisites

SCOoOTER is designed for two primary use cases: (1) enabling students to experiment with various design choices by modifying the configuration file and evaluating the impact of different settings, and (2) allowing students to inspect and extend the processor implementation, such as by adding custom branch predictors or functional units for new instructions. This process is facilitated by a clear modular structure, comprehensive documentation, and standardized, generic interfaces. In both cases, SCOoOTER aims to move students beyond theoretical, pen-and-paper exercises and toward an intuitive, hands-on understanding of the impact of architectural decisions and added functionalities.

Before using SCOoOTER, students should acquire a foundational understanding of the underlying concepts, e.g., through a course that directly incorporates SCOoOTER, via prerequisite coursework, or through recommended literature. For background reading, we suggest [13, Ch. 3, Ch. 5, App. C] to introduce pipelining, instruction- and thread-level parallelism, speculative execution, and superscalar architectures. Additionally, Bluespec's resources [6], [18], [2] are recommended for learning Bluespec SystemVerilog (see Section 4.4), while also covering basic pipeline architectures and epoch-based misprediction handling [2], and introducing the RISC-V ISA [18].

3 Related Work

Courses using different RISC-V implementations already exist. We have identified three categories of such courses and explain how SCOoOTER will fill a gap in the existing offerings.

(1) The first category uses *available complex processors*. Harris et al. [11] introduce a course in which students complete multiple labs, built around the *SweRV EH1* processor. The first part of the course focuses on processor peripherals and the implementation of RISC-V programs, while the second part introduces the SweRV EH1 micro-architecture. Although the course excels at teaching system-on-chip architecture, students may struggle to fully understand or modify SweRV EH1, as it is highly optimized and implemented in low-level SystemVerilog.

(2) The second category uses *embedded-class pipelined processors*. Ahmadi-Pour et al. [1] introduce the *microrv32* processor, implemented in Verilog, and target both FPGA and simulation in their course.

Lowe et al. [15] present the in-order *DINO* processor, implemented in Chisel and available in single-cycle and pipelined versions. Their course consists of four labs in which students: (1) create an ALU control unit, (2) extend it into a single-cycle processor, (3) build a pipelined processor, and (4) add branch prediction. For each lab, students are provided with templates derived from the *DINO* processor.

Syafalni et al. [30] present a framework for RISC-V-based learning on a Xilinx PYNQ FPGA.

While these simple architectures offer a low barrier of entry and are well-suited for introductory-level teaching, they do not reflect the complexity of modern high-performance architectures that would be the subject of an advanced class. Our work aims to provide follow-up educational resources that address this gap.

(3) The final category has students *implement an entire processor from scratch*. Jamieson et al. [14] share their experiences having students design and implement a RISC-V processor and compose a system-on-chip similar to a microcontroller, targeting FPGA deployment. They focus on tooling and introduce the architecture of a microcontroller.

McGrew et al. [16] present an overview of tools for processor design and implementation, and discuss pedagogical decisions such as choosing between simulators, FPGAs, or thought experiments, and selecting an appropriate ISA.

Zekany et al. [34] describe a course in which student groups design an *out-of-order* RISC-V processor. The course begins with the construction of simple combinational circuits, followed by state machines, and an in-order processor. Students then reuse components to implement an out-of-order processor. Bonus points are awarded for advanced features such as superscalar execution, prefetching, exceptions, or branch prediction. Because full processor design can be overwhelming for beginners, the course divides the project into smaller tasks and preliminary exercises.

We acknowledge that implementing an entire processor is both insightful and engaging, however, it is time-consuming and may not fit the constraints of all university curricula (e.g., tight credit point limits on individual courses or the entire study programme).

We see SCOoOTER as a natural extension of the already excellent resources provided by the prior works.

Additionally, many universities offer excellent courses on FPGA and ASIC design, which allow students to build efficient hardware and understand the mapping of HDL code to physical implementations. The goal of SCOoOTER, however, is somewhat different: we aim to help students both grasp advanced processor design concepts specifically and, in turn, to evaluate the trade-offs among different architectural choices at the physical level. We believe these two types of courses can complement and enhance each other.

After considering existing courses, we now compare SCOoOTER itself to currently available application-class RISC-V processors. The *CVA5* [20] and *CVA6* [33] processors provide limited reordering by allowing out-of-order writeback.[1]

However, in contrast to SCOoOTER, they still *issue* instructions in order and offer only limited configurability, such as limited functional unit mix configurations and limited branch predictor types and options. Both are implemented in low-level SystemVerilog, which results in verbose source code and reduced accessibility. On the other hand, both processors are fully Linux-capable, which is future work for SCOoOTER.

VexRiscv [29] and *NaxRiscv* [28], both written in SpinalHDL, a hardware construction language embedded in Scala, feature extensive plugin systems. *VexRiscv* is a configurable-depth pipeline processor, whereas *NaxRiscv* employs a dynamic pipeline with strong reordering capabilities. Their plugin systems enable wide configurability, but the deep abstraction stack and Scala being a complex base language risk obscuring the resulting hardware and introducing a steep learning curve, depending on the nature of the desired configuration changes. Like *CVA5* and *CVA6*, both processors can be configured to be fully Linux-capable.

BOOM [35] is a powerful out-of-order processor written in Chisel, which is also embedded in Scala, featuring an advanced memory subsystem and OS support. Like *VexRiscv* and *NaxRiscv*, the deep abstraction stack and complexity of the chosen HDL risk obscuring the generated hardware and increasing the learning curve. *TOOOBA* [3], which is implemented in BSV, is a superscalar out-of-order processor but provides limited configurability, e.g., employing a fixed execution unit mix.

All of the above processors are tightly coupled with a cache hierarchy, whereas SCOoOTER directly accesses memory, since our focus lies on the processor micro-architecture itself, with the *uncore* (such as the cache subsystem) being the topic of future work. While the processors discussed above are already optimized and widely adopted, we believe SCOoOTER offers superior educational value through its modularity, configurability, English-language documentation, and supporting tooling.

4 Architecture and Implementation

The architecture of SCOoOTER aligns with Hennessy and Patterson's popular textbook on computer architecture [13]. Figure 1 shows the overall architecture. Each component follows a generic interface and modular design to enable replacement. SCOoOTER consists of three main components, which we briefly describe in the following subsections, focusing on key design choices. For a detailed architectural explanation, we refer to the SCOoOTER documentation. After introducing the processor's architecture, we discuss our choice of HDL and our support for the implementation of multithreading and multiprocessor systems.

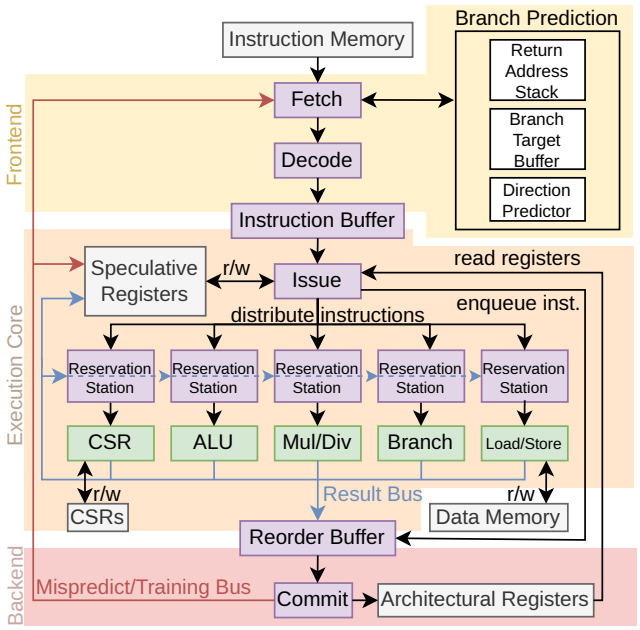

Figure 1: Overall architecture of SCOoOTER. The frontend fetches instructions and calculates the next PC. The execution core implements Tomasulo's algorithm. The backend updates the architectural state. Pipeline stages and buffers holding instructions are shown in violet, functional units in green, and elements holding program state in gray.

4.1 Frontend

The frontend is responsible for fetching and decoding instructions, supplying them to the execution core, and determining the next PC. The *fetch stage* retrieves instruction words from memory and advances the PC. The *decode stage* expands instructions from their compact in-memory form to a wider internal format and places them into an *instruction buffer* between the frontend and the execution core. A secondary component is the *branch prediction unit*, which assists the fetch stage in selecting the next PC. The branch predictor speculatively determines the instruction to follow a branch, avoiding pipeline flush penalties in deep, reordering pipelines (cf. Figure 5) when the prediction is correct.

Among the many available branch prediction schemes, we implement a *return address stack (RAS)* for predicting return targets and a *branch target buffer (BTB)* for predicting conditional branch and call targets, as both are widely used [13, p. 497]. Their sizes are freely configurable, and both can be disabled. The RAS optionally supports misprediction recovery, which mitigates corruption caused by wrong-path instructions that alter the stack before being detected. To enable this, each instruction may carry metadata

[1] As definitions vary, we use the term *writeback* for result generation and buffering, and *commit* for actually writing to the architectural register file.

such as the stack head pointer or entry, allowing recovery during misprediction [26]. This improves prediction accuracy at the cost of increased buffer sizes.

Branch direction (taken or not taken) is most efficiently predicted by tracking the historical outcome of branching instructions [13, p. 458ff.]. The *smiths* predictor uses two-bit saturating counters to add hysteresis while tracking previous results of the same instruction. *Gshare* adds a *branch history register (BHR)* to correlate outcomes with the history of previous dissimilar branches. *Gskewed* introduces redundancy using multiple predictor tables and index functions to reduce interference when two instructions map to the same entry in the predictor. We select those predictors as they pedagogically build upon one another, supporting progressive complexity (cf. Figure 2). Additionally, they illustrate core concepts in branch prediction like basic indexing, history-based correlation, and interference mitigation. The type of branch direction predictor, number of entries, and BHR size are all configurable. If no branch predictor is enabled, SCOoOTER predicts branches as not taken, a behavior the RISC-V C compiler optimizes for by favoring the fall-through paths [32, Sec. 2.5.2].

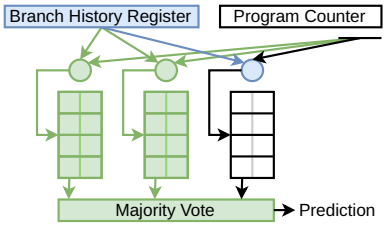

Figure 2: Branch predictors supported by SCOoOTER. Smiths is shown in black and consists of a table of 2-bit saturating counters indexed by the LSBs of the PC. Gshare adds a branch history register and combines it with the PC before accessing the table (shown in blue). Gskewed adds two counter tables with differing combination functions for the table index and a majority vote (shown in green).

4.2 Execution Core

The execution core computes instruction results. It uses the widely adopted Tomasulo algorithm for register renaming to resolve data hazards and enable instruction reordering for higher throughput [13, Sec. 3.5].

Instructions first enter the *issue stage*, which reserves space in a *reorder buffer (ROB)* to preserve program order for commit.

Each instruction is tagged with its ROB slot for identification. A *speculative register file* tracks pending writes and speculative values. The issue stage updates this file with pending write information and reads operands, tagging instructions accordingly. Issued instructions proceed to a *reservation station (RS)* associated with their *execution unit*. These buffers latch operands as they become available and queue ready instructions for execution. Execution units compute results and broadcast them via a result bus, which is received by the ROB, speculative register file, and applicable reservation stations.

Since speculative instructions must not alter architectural state until confirmed correct, state-modifying units (e.g., CSR and load/store) *defer* execution until the instruction reaches the ROB head. Because instruction readiness is based on operand availability, later instructions may execute before earlier ones. SCOoOTER supports configurable ROB and RS sizes, execution unit mix, and the number of instructions issued per cycle. The fetch and commit stages scale accordingly to match increased issue throughput with increased issues per cycle.

4.3 Backend

The backend includes the *commit stage* and the *architectural register file*. The commit stage retrieves instructions from the ROB and checks branch prediction outcomes. It informs the fetch stage of branch results for updating predictors and initiates redirection on misprediction. On misprediction, it also flushes the speculative register file, restoring state from the architectural register file. Correct-path instructions are committed to the architectural register file.

4.4 HDL Selection

Since RTL-style HDLs such as Verilog or VHDL, and even modern *hardware construction languages* (HCLs) such as Chisel and SpinalHDL, often require verbose and difficult-to-follow descriptions that will likely slow the students' progress, we opt for a higher-level HDL. We choose *Bluespec SystemVerilog (BSV)* [17] for the following reasons: (1) BSV is open-source and does not require licenses, (2) BSV comes with a fast simulator exceeding RTL simulation throughputs, (3) BSV does not require manual handling of resets, clock signals or stalls, (4) BSV features strict type safety to detect common mistakes, (5) BSV provides a comprehensive standard library, reducing the code's verbosity, and (6) BSV includes *Guarded Atomic Actions*, a powerful concurrency execution model that is far more expressive than the parametrized structural descriptions targeted by modern HCLs such as Chisel or SpinalHDL. We consider BSV to strike a good balance between C/C++/SystemC-based high-level synthesis tools — which often obscure the resulting hardware — and the verbosity of (System)Verilog.

4.5 Multithreading and Multiprocessor Systems

Hardware multithreading is implemented by modifying the *fetch stage* to alternate between the individual hardware threads' program counters. Instructions are tagged with a thread ID, and the backend maintains separate architectural register files for each hardware thread (commonly called a *hart* in the RISC-V community). CSRs and speculative registers are replicated as well.

As shown in Figure 3, coarser-grained parallelism in the form of simple *multiprocessor* systems is supported for SCOoOTER through a shared arbitration unit that serializes memory access across processors. Requests are queued and processed in order. Instruction arbitration (fetch only) requires read logic (red blocks in Figure 3), while data arbitration (load/store) supports both read and write operations. *Atomic operations* are supported to enable inter-thread or inter-processor communication. The arbitration unit manages the link register for RISC-V *load-reserved/store-conditional* instructions and ensures that atomic *load-modify-store* instructions execute

in isolation by stalling further memory operations until they are complete.

The number of hardware threads and processor cores is configurable. The RISC-V *A* extension is automatically enabled when configuring SCOoOTER for multiple threads or processors. Note that the current SCOoOTER version has been designed with a focus on teaching the use and implementation of atomic operations, which often "fall by the wayside" in many architecture courses. However, as processor count increases, memory arbitration becomes a bottleneck in the current implementation. To address this, we plan to introduce a more powerful uncore with a modular, multi-level cache system in future versions.

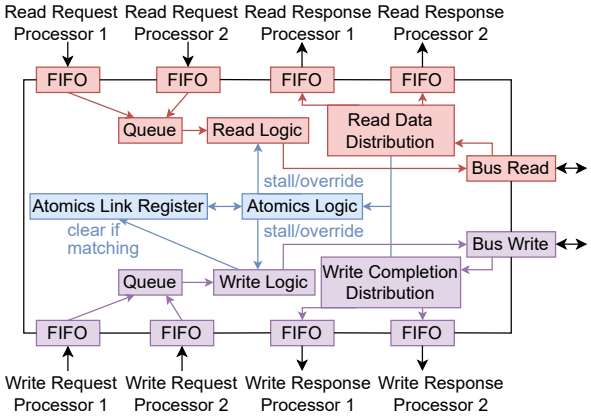

Figure 3: Memory arbitration for multiprocessor systems. Requests are serialized via queues. Red blocks show the read logic, violet blocks the write logic, and blue blocks the atomic logic. Atomic read-modify-write accesses are executed reading, modifying and storing the value while stalling all reads and writes.

4.6 Uncore Peripheral Subsystem

In addition to the processor, SCOoOTER includes peripheral components instantiated by IDMemAdapter, our uncore subsystem (see Figure 4). We provide the standard RISC-V *CLINT* and *PLIC* interrupt controllers. Additional peripherals can be connected via a minimal BSV PutGet interface. The interface can be adapted to AXI4 Full or Lite to connect additional peripheral device IP. For testbench environments, IDMemAdapter includes the *RVController* for testbench communication from the *TaPaSCo-RISC-V* [12] software framework, which forms the base of our simulation environment. New peripherals can be added by instantiating memory connections and assigning them address spaces. The interconnect automatically updates to accommodate the new configuration.

5 Educational Features and Tooling

This section introduces some of SCOoOTER's education-focused tools and features.

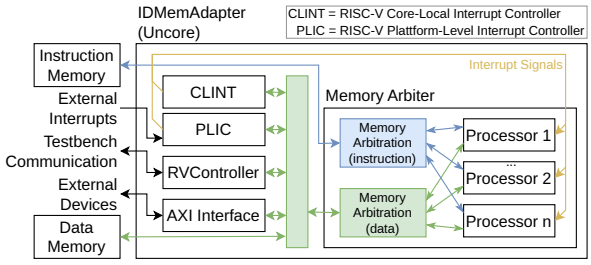

Figure 4: Uncore subsystem with provided peripheral devices. Instruction fetching paths are shown in blue. Load/store paths are shown in green. Devices are connected via a configurable interconnect.

5.1 Automated Testing and Verification

Since we encourage students to modify, explore, and extend the architecture, we must also enable them to test whether the processor still operates correctly. To this end, we provide coarse test suites for quickly testing for significant mistakes, as well as long-running, fine-grained testing for more complete verification.

Coarse testing is possible by executing the *riscv-tests* [24] (for basic instructions), *riscv-arch-tests* [23] (for exceptions), and custom parallel reduction tests (for atomic instructions). Higher test coverage is enabled by running the *Embench-IoT* [10] benchmarks and by executing randomly generated tests and comparing the execution traces to an *instruction set simulator (ISS)*. Test generation and comparison to an ISS is supported by *Core-V-Verif* [19]. Note that Core-V-Verif requires a commercial UVM-capable simulator, since, to the best of our knowledge, no open alternative is available yet. Each suite can be executed as a make target and requires no additional setup. We also provide a *GitLab CI* file and an associated *Docker* container to run these tests automatically, such that students can identify mistakes in their commits quickly, and educators are able to get a quick overview of the functionality of all students' results.

5.2 Debug Tooling

During the implementation of their modifications, students may frequently encounter issues, such as failing tests, that need to be debugged. While logic waveform dumping allows for detailed exploration of the simulated design, it is neither fast nor user-friendly. Therefore, we have implemented additional tools for quick and efficient debugging. SCOoOTER generates log files in the *Kanata* format. Using the Konata [25] tool, students can visually track the pipeline progress and identify stuck instructions or hazards. Figure 5 shows a sample Konata diagram. Additionally, SCOoOTER has an extensive logging-based debugging infrastructure. Debug outputs can be enabled per component in a global configuration file. This allows students to trace updates of specific units and explore their issues. Lastly, SCOoOTER implements the industry-standard RVFI [31] interface as a tracing interface for committed instructions. This interface can be used by additional hardware units monitoring the processor's execution trace or during waveform debugging to understand which instructions are executed.

Figure 5: Konata tracing of pipeline execution for SCOoOTER (65X configuration, cf. section 6) running the XOR RISC-V ISA test. Different colors correspond to different pipeline stages during instruction execution. Gray instructions are flushed following mispredictions, highlighting the benefit of good branch prediction to increase throughput.

5.3 Performance Evaluation Tooling

After creating and validating their modifications, students will typically need to gather performance data. SCOoOTER simplifies this by integrating *Embench-IoT* [10], a widely used benchmark suite. Our simulation framework reports elapsed cycles per test and prints the number of correctly / incorrectly predicted branches. Faster evaluations for longer workloads are supported using FPGA hardware. Both mapping to the FPGA as well as interacting with the hardware are automated using the *TaPaSCo* [9] hardware/software framework. Since the software simulation environment is compatible with the *TaPaSCo-RISC-V* [12] FPGA hardware implementation, no changes to the test binaries are necessary.

5.4 Automated Design Generation

In addition to cycle-based performance measurements, the maximum frequency and required hardware area need to be considered to gauge whether a micro-architectural design decision is actually beneficial. Using FPGA and ASIC tools can be challenging without significant user experience. Hence, we *automate* the design generation process as much as possible. FPGA designs are automatically created using TaPaSCo-RISC-V [12]. Note that, since TaPaSCo targets AMD/Xilinx FPGAs, the proprietary Vivado tool suite must be installed when using this feature. ASIC designs are generated using the open *OpenLane* [22] flow. By default, we use Efabless-provided `EF_SRAM` [8] memory macros for instruction and data memory. Integration of other memory macros, such as those created by OpenRAM [21] and DFFRAM [7], is possible via our included SRAM abstraction layer `BlueSRAM`. The ASIC design is generated by our flow as a turnkey macro, which can be placed into the *Caravel* [4] SoC wrapper, which provides I/O pads, clocking, reset, and on-chip debugging infrastructure.

5.5 Resources for Students

Processor features and tooling capabilities are useless if students are unable to discover them. Hence, we provide extensive documentation. Building on the prerequisites discussed in Section 2, the documentation introduces all of SCOoOTER's features, configuration options, and the implementation and simulation environments. It also explains the architecture and the structure of the code base, which itself is thoroughly commented.

6 Evaluation

We evaluate the performance and usability of SCOoOTER under different configurations, comparing it to *CVA6*. We chose *CVA6* because it can be configured to have feature parity with SCOoOTER and has a similar pipeline design, differing solely in maintaining instruction order while reading operands. Specifically, we utilize the *CV32A60X* and *CV32A65X* configurations of *CVA6*, while disabling all features not supported by SCOoOTER (i.e., the cache, memory management, and extensions outside of `RV32IMA_zicsr`). The main difference between those configurations is the number of ROB entries, where *CV32A65X* has eight slots while *CV32A60X* only has four. The equivalent SCOoOTER configurations are referred to as *60X* and *65X*. We evaluate the maximum frequency, required area, wall-clock performance, and legibility as reported by students. Note that SCOoOTER is capable of scaling to significantly larger configurations. The largest configuration successfully implemented on Caravel features dual-issue execution, 32 ROB entries, and eight-slot reservation stations.

6.1 Area and Frequency

Since we exclusively use open-source tooling and an open ASIC process, we can fully report technology-specific results. We target the *SkyWater 130A* process [27] using the *OpenLane* [22] implementation suite with the default configuration provided with SCOoOTER. The generated designs contain only the processor cores with no memory, caches, or peripheral components, ensuring a fair comparison. The results are shown in Table 1.

Configuration	Area (um²)	fmax (MHz)
CVA6 (CV32A60X)	887526	56
SCOoOTER (60X)	1322280	83
CVA6 (CV32A65X)	946298	53
SCOoOTER (65X)	1432540	77

Table 1: ASIC data on SKY130 for CVA6 and SCOoOTER

SCOoOTER requires about 1.5 times more area than *CVA6*. This is likely due to SCOoOTER's larger execution core, which includes added reservation stations and speculative registers. On the other hand, SCOoOTER achieves a higher clock frequency, likely due to optimizations for the *SkyWater 130A* process. Both processors show similar scaling of area with different configurations.

Although the achieved maximum clock frequencies are not exceptional, this is primarily due to the legacy 130nm process and the comparatively lower quality of results produced by open-source EDA tools relative to mature commercial alternatives. Despite these limitations, we consider the *SkyWater 130A* process as optimal for classroom use, as it is widely supported by open-source tools and provides adequate frequency and area scaling to demonstrate the effect of different design choices.

We advocate for the use of open alternatives in educational settings because, depending on class size, it may be infeasible for all students to sign NDAs or have access to commercial EDA tool licenses. Furthermore, *OpenLane* is more accessible for novice users than the very complex industrial-strength tools, due to its simple unified configuration file and easy-to-use launch commands.

6.2 Performance

Figure 6 presents the wall-clock time for SCOoOTER and *CVA6* executing the *Embench-IoT* benchmarks in simulation [10]. For CVA6, we include results both with and without its cache to assess the cache's performance impact, assuming a similar clock frequency in both cases. SCOoOTER consistently outperforms the cacheless CVA6 configurations. As expected, 65X is faster than 60X, while *CV32A65X* and *CV32A60X* show the opposite trend. This inversion likely stems from CVA6's more efficient load/store implementation, which reduces stalls and diminishes the performance gains from increased instruction-level parallelism. Consequently, the higher clock rate of CV32A60X predominantly impacts the results. Overall, while SCOoOTER already has competitive performance, the planned enhancements to its memory subsystem, such as a more advanced load/store unit or added caches, will further improve efficiency.

6.3 Accessibility by Students

SCOoOTER has been successfully used in three student theses, with students reporting only few hurdles when they worked with the code. The thesis topics were:

- Building a cache for SCOoOTER in BSV
- Extending the cache with TileLink coherency
- Adding NoMMU Linux support to SCOoOTER

While students had no issues modifying and extending the processor, the theses yielded just proof-of-concept implementations that require further work before being suitable for wider classroom use. For example, the cache currently only supports FPGA targets and operates at a low frequency, while the Linux port needs further automation and currently lacks a full root filesystem.

7 Sample Learning Scenario

We demonstrate SCOoOTERs utility by describing a typical evaluation task to be performed by students: comparing the efficiency of the four provided branch direction predictors. We select this example because branch prediction is crucial for the performance of deeply-pipelined reordering processors, helping to avoid the high cost of pipeline flushes (Figure 5 shows an example of the pipeline flushing penalty). Furthermore, we believe it serves as a good introductory exercise. Varying the branch predictor type can be achieved by changing just a single line of SCOoOTER's configuration. Instead of crafting their own simulation environment and integrating it with a benchmark, students can use SCOoOTER's built-in simulation environment with *Embench-IoT* as workload. Launching the entire test suite is simplified to a single `make` command. The simulation will then provide students with the number of correctly and incorrectly predicted branches.

While cycle-by-cycle performance plays a significant role, maximum frequency and area requirements must also be considered when evaluating an implementation. SCOoOTER addresses this by providing built-in synthesis targets. Instead of setting up a complex ASIC implementation flow from scratch, students can simply invoke a Python script, just needing to specify the target frequency and placement density. *OpenLane* is then called to automatically synthesize the design and will notify the student if the desired timing closure cannot be achieved. Alternatively, students can use another script to evaluate the design in hardware on an FPGA. In that case, students can load the bitstream onto an FPGA board for faster benchmarking. Since our flow relies on TaPaSCo for the actual FPGA implementation, a wide range of AMD/Xilinx boards, ranging from cheap Zynq-class prototyping boards to large UltraScale+ and Versal PCIe accelerators, are supported.

As a next exercise, students could enhance SCOoOTER's branch prediction capabilities by implementing additional branch direction predictors. This is straightforward, thanks to SCOoOTER's generic interfaces and modular architecture. The branch predictor interface provides all relevant information about branch outcomes from the commit stage, including the program counter (PC), predicted and actual direction, actual target, optional history information, thread ID, and instruction type. For the actual prediction, the instruction PC, thread ID, and branch type are passed to the predictor module, while the fetch stage expects a prediction and optional history information from the predictor, which will be returned along with the training data. We believe that this simple-yet-flexible interface allows students to implement most direction predictors easily. To add a new predictor, students simply need to implement a module with the correct interface, and add it to the list of available direction predictor implementations.

8 Conclusion

This paper presented SCOoOTER, a highly configurable RISC-V processor designed for use in mid-to-advanced level computer architecture classes. We demonstrated how SCOoOTER enables students to explore, modify, and extend its architecture, using our provided tools for testing, debugging, performance evaluation, and design generation down to layout. Our evaluation showed that SCOoOTER offers competitive performance even when compared to established processors like *CVA6*, while remaining accessible to novice users through its user-friendly interfaces, documentation, and integration with open-source tools. Through this accessibility SCOoOTER allows students to apply their theoretical knowledge in a hands-on manner. By utilizing open-source technologies such as *OpenLane*, we make full-stack processor design and evaluation more accessible in educational settings, helping to prepare students for advanced research and industry practice.

9 Future Work

One key area for future improvement is the load/store unit and memory architecture. We plan to implement more advanced memory systems, such as cache hierarchies and enhanced memory access controls, to boost performance, particularly for larger processor counts. This will also enhance SCOoOTER's educational value on the topic of memory subsystems. Additionally, we aim to extend SCOoOTER to support operating systems, such as Linux, by improving its memory management. This expansion will broaden the scope of SCOoOTER further, enabling more comprehensive system-level design exercises. Furthermore, SCOoOTER's scope could be expanded by adding implementations for simpler, non-reordering pipelines. This would allow SCOoOTER's support tools to be used for an even wider range of processor designs, from small multi-cycle or pipelined architectures to modern out-of-order superscalar cores. Lastly, we are in the process of developing a full course on advanced processor design and implementation, with SCOoOTER being the

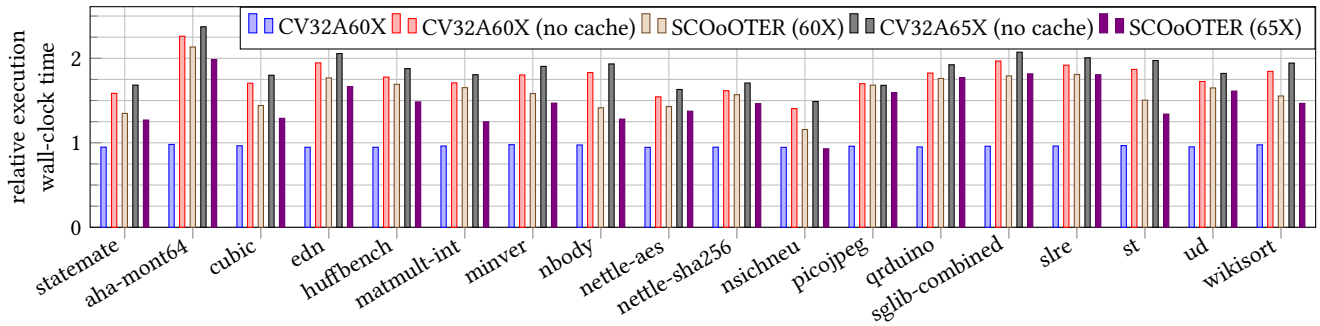

Figure 6: Relative benchmark execution wall-clock time compared to CV32A65X (with cache) (lower is better)

entry point to more complex micro-architectures. This course will offer students a hands-on introduction to processor features such as speculative execution, branch prediction, instruction reordering, and the associated *design choices*; followed by considering the *effects* of these choices when the processor is actually implemented as an ASIC.

10 Availability

SCOoOTER, along with its documentation, is available on GitHub: https://github.com/esa-tu-darmstadt/SCOoOTER

Acknowledgments

This work has been supported by the German Federal Ministry of Education and Research in the project Scale4Edge (grant: 16ME0139) as well as the Hessian Ministry of Higher Education, Research, Science and the Arts (grant: 01IS23067) within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Sallar Ahmadi-Pour, Vladimir Herdt, and Rolf Drechsler. 2021. MircoRV32: an open source RISC-V cross-level platform for education and research. In *Proceedings of the Workshop on Design Automation for CPS and IoT*. 30–35.
[2] Arvind, Rishiyur S. Nikhil, Joel S. Emer, and Murali Vijayaraghavan. 2011. *Computer architecture: A Constructive Approach*. Elsevier.
[3] Bluespec. 2024. TOOOBA. https://github.com/bluespec/Toooba. Accessed: 2025-03-26.
[4] ChipFoundry.io. 2025. Caravel. https://github.com/chipfoundry/caravel. Accessed: 2025-03-28.
[5] ChipFoundry.io. 2025. ChipCreate. https://chipfoundry.io/. Accessed: 2025-05-07.
[6] Kathy R. Czeck and Rishiyur S. Nikhil. 2010. *BSV by Example*. Createspace Independent Publishing Platform. http://csg.csail.mit.edu/6.S078/6_S078_2012_www/resources/bsv_by_example.pdf
[7] DFFRAM Contributors. 2025. DFFRAM. https://github.com/AUCOHL/DFFRAM. Accessed: 2025-03-28.
[8] Efabless. 2025. EF_SRAM. https://github.com/efabless/EF_SRAM_1024x32. Accessed: 2025-03-28.
[9] Embedded Systems and Applications Group TU Darmstadt. 2025. TaPaSCo. https://github.com/esa-tu-darmstadt/tapasco. Accessed: 2025-03-26.
[10] Embench contributors. 2023. Embench™: Open Benchmarks for Embedded Platforms. https://github.com/embench/embench-iot. Accessed: 2025-05-15.
[11] Sarah L Harris, Daniel Chaver, Luis Piñuel, JI Gomez-Perez, M Hamza Liaqat, Zubair L Kakakhel, Olof Kindgren, and Robert Owen. 2021. RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 145–150.
[12] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. 2019. A catalog and in-hardware evaluation of open-source drop-in compatible RISC-V softcore processors. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 1–8.
[13] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach* (6th ed.). Elsevier.
[14] Peter Jamieson, Huan Le, Nathan Martin, Tyler McGrew, Yicheng Qian, Eric Schonauer, Alan Ehret, and Michel A Kinsy. 2022. Computer Engineering Education Experiences with RISC-V Architectures—From Computer Architecture to Microcontrollers. *Journal of Low Power Electronics and Applications* 12, 3 (2022), 45.
[15] 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.
[16] Tyler McGrew, Eric Schonauer, and Peter Jamieson. 2019. Framework and tools for undergraduates designing RISC-V processors on an FPGA in computer architecture education. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 778–781.
[17] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 69–70.
[18] Rishiyur S. Nikhil. 2024. *Learn RISC-V CPU Implementation and BSV*. Bluespec Inc. https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design/blob/main/Book_BLang_RISCV.pdf Accessed: 2025-03-28.
[19] OpenHW Group. 2025. Core-V-Verif. https://github.com/openhwgroup/core-v-verif. Accessed: 2025-04-07.
[20] OpenHW Group. 2025. CVA5. https://github.com/openhwgroup/cva5. Accessed: 2025-04-22.
[21] OpenRAM Contributors. 2025. OpenRAM. https://openram.org. Accessed: 2025-03-28.
[22] OpenROAD contributors. 2025. OpenLane. https://github.com/The-OpenROAD-Project/OpenLane. Accessed: 2025-04-07.
[23] RISC-V Software contributors. 2025. RISC-V Architecture Test SIG. https://github.com/riscv-non-isa/riscv-arch-test. Accessed: 2025-05-15.
[24] RISC-V Software contributors. 2025. riscv-tests. https://github.com/riscv-software-src/riscv-tests. Accessed: 2025-04-07.
[25] Ryota Shioya. 2023. Konata. https://github.com/shioyadan/Konata. Accessed: 2025-03-26.
[26] Kevin Skadron, Pritpal S Ahuja, Margaret Martonosi, and Douglas W Clark. 1998. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 259–271.
[27] SkyWater PDK Authors. 2020. SkyWater SKY130 PDK. https://skywater-pdk.readthedocs.io/en/main. Accessed: 2025-03-28.
[28] SpinalHDL. 2025. NaxRiscv. https://github.com/SpinalHDL/NaxRiscv. Accessed: 2025-03-26.
[29] SpinalHDL. 2025. VexRiscv. https://github.com/SpinalHDL/VexRiscv. Accessed: 2025-03-26.
[30] Infall Syafalni, Yahwista Salomo, Chyndi Oktavia Devi, Muhammad Ali Novandhika, Nana Sutisna, Rahmat Mulyawan, and Trio Adiono. 2022. RISC-V Learning Framework using PYNQ FPGA. In *2022 8th International Conference on Wireless and Telematics (ICWT)*. IEEE, 1–8.
[31] SymbioticEDA. 2021. RISC-V Formal Verification Framework. https://github.com/SymbioticEDA/riscv-formal. Accessed: 2025-04-07.
[32] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Architecture*. Vol. I: Unprivileged ISA. SiFive Inc. and University of California, Berkeley.
[33] Florian Zaruba and Luca Benini. 2019. *The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology*. https://doi.org/10.1109/TVLSI.2019.2926114
[34] Stephen A Zekany, Jielun Tan, and James A Connolly. 2021. Teaching Out-of-Order Processor Design with the RISC-V ISA. In *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*. IEEE, 1–8.
[35] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).