RISCALAR
A Cycle-Approximate, Parametrisable RISC-V Microarchitecture Explorer & Simulator

Josiah Mendes
National University of Singapore / Imperial College London
{e097832}@u.nus.edu, {jam419}@ic.ac.uk

***Abstract*—Riscalar is a highly parameterisable, extensible, and modular computer architecture simulation tool designed around the RISC-V ISA. The main feature of Riscalar that differentiates it from other RISC-V educational simulators is its ability to explore a large design space with configurable pipeline widths, execution orders, functional units, cache sizes and branch predictors. It is capable of simulating user designed programs through standard cross-compilation tools, and can be used to explore the relationship between processor microarchitecture, compiler optimisations and program performance. This paper details the design and implementation of Riscalar using Rust, highlighting the abstractions made to implement a cycle-approximate simulator.**

***Index Terms*—computer microarchitecture modelling, computer architecture educational tool, performance modelling, RISC-V**

I. Introduction

Developing an understanding of computer microarchitecture is an essential part of every Computer Engineering undergraduate program. Most courses will explain the theory behind how different parts of the processor facilitate code execution, increase performance and combat bottlenecks.

To further this understanding, a practical aspect is usually included in the course. This may be the implementation of a synthesisable RISC processor using a hardware description language, or to build a simple functional software simulator written in a high level language such as C++ or Java. While both of these approaches help to facilitate students' understanding of computer microarchitecture and also to build their programming skills, they do not place enough of an emphasis on performance and the constraints that computer architects face in the real world. It is easy for the focus to be purely placed on functional correctness, resulting in unrealistic designs of single-cycle architectures with long critical paths and a very low clock frequency.

Another approach to verifying theoretical knowledge is by observing the behaviour of computer architecture through the use of simulators. This involves students configuring systems, acquiring statistics and interpreting the obtained data to identify areas of improvement for the microarchitecture. Such simulators can fall into two categories, those that model an existing system or those that provide the user with a set of choices to design a new microarchitecture at the expense of reduced accuracy. An efficient simulation tool coupled with comprehensive data collection mechanisms allows students to explore the performance impact of different system configurations while also being able to observe how there may be an impact on power or area. When such a simulator can be coupled with visual descriptions of the process each unit within the CPU is carrying out at a given time, this becomes a powerful tool for students to engage with the constraints faced when designing a processor. Such a simulator can also be helpful in teaching students how to optimise their code for a particular microarchitecture which is particularly useful in embedded programming.

While the idea of using simulation to teach Computer Engineering is not new, as efforts have being carried out for over three decades, there is a gap in the field where modern simulation tools targeting popular instruction set architectures today such as RISC-V are unable to match the same level of simulation customisation that certain older "outdated" tools provide. To fill this gap, this paper presents a micro-architecture simulator named Riscalar that is highly configurable and parametrisable designed specifically for the RISC-V ISA.

The report is structured as follows: Section II covers the microarchitecture simulation landscape today, with a particular focus placed on education and simulation tools targeting RISC-V and how Riscalar is classified as a simulator compared to other similar tools; Section III introduces different aspects of the RISC-V ISA that need to be considered when implementing a functional simulator, including architectural registers, instruction decoding and execution, privilege levels and exceptions; Section IV shows how the cycle approximate simulator is implemented, covering instruction execution together with the configurable parameters that affect how many cycles an instruction takes to retire and the components outside of the main pipeline that affect performance.

II. Literature Review

A. Simulators for Education

The advantages of using simulation as a tool for teaching computer architecture has been discussed by many authors since the early 1990s, such as [1], [3], [2]. More recently, [12] and [15] confirm that these findings still remain true, and can be applied to new learning modes today such as distance-learning, flipped classroom teaching modes, blended

remote learning and MOOCs (Massive Open Online Courses). Limiting such courses to textbooks and lectures can be ineffective, and simulators provide a way for students to understand concepts through their own thinking abilities helping them to master the content [13] [14].

[10] presented a variety of educational simulators for teaching computer architecture and organisation available in 2009, classifying educational simulators into two groups, full-system creation and system simulation. The first group primarily centered around tools that were explicitly designed for doing general purpose digital design or professional simulation. Riscalar falls closer to the second group with configurable parameters that need to be set before program runtime, along with tools such as SimpleScalar [6].

SimpleScalar is a very popular tool in both the author's own undergraduate experience and in this field with its attached papers being cited over 2000 times. It is a suite of command-line hardware simulators that emphasizes performance and flexibility over simulation detail. The `sim-outorder` simulator in SimpleScalar has a completely configurable pipeline, execution model, branch predictor and cache hierarchy, making it especially valuable. It serves as one of the main inspirations for Riscalar and many other works such as [9]'s neural network based performance prediction model for teaching trade-offs based on SimpleScalar performance; [4] which implemented a power evaluation methodology extension for SimpleScalar; and [5] which provided a SimpleScalar graphical interface. [7] provides a more comprehensive overview of extensions made to SimpleScalar to improve it as a teaching tool. While SimpleScalar can be considered to be a seminal work in the educational simulator tooling space, it uses a highly customised toolchain for program development and has fallen behind in ISA support.

RISC-V is an increasingly popular choice as an ISA within the computer architecture research and teaching space since its introduction in 2011 [21]. It is a load-store architecture that is designed with a base instruction set and a set of extensions that add instructions for special operations [21]. [18] presented a summary paper covering 15 openly-released RISC-V cores available in 2019. In more recent years, it also has been rising in popularity as an industry choice for commercial processor implementations, such as in Western Digital's storage processors[19], Espressif's hobbyist microcontrollers[22], and Google's Titan M2 security chips for their Pixel 6 smartphones [23].

As a result of both RISC-V's increasing popularity in industry and educational focus, many recent simulators designed for education have also used this ISA.

[24] presented a visual simulator of processor pipelines based on the RISC-V ISA with a built-in assembler, compiler support and cache simulator. While not parameterisable, it provides the option for users between 5 pre-determined microarchitectures to see how different pipeline depths and forwarding schemes affect execution. [25] also presented a tool for the investigation of slow-downs in RISC-V program execution and further investigation of the internal state of the pipeline architectural blocks through a web-based interface. The web-based interface was especially novel, considering most simulators run as standalone software [10], it reduces the barrier of entry allowing any device with a compatible web-browser to be able to use this simulator. [16] extended this approach and provided a full C compiler and assembly viewer/editor within the browser, as well as adding a slightly customisable Verilog HDL generation for a soft-core implementation.

B. General Computer Architecture Simulation

[17] presents a very detailed overview of computer architecture simulators, stating that classification can be based on several properties:

- *Detail of Simulation*
- *Scope of Target Simulation*
- *Input to the Simulator*

Under this categorisation, Riscalar is a:

1) Cycle-Approximate Simulator: As the aim of this simulator is to provide students with an understanding of how different microarchitecture decisions affect a program's runtime and performance, detail is less of a concern, with the focus being on system-level parameters, allowing other details to be abstracted away.

2) Application Level Simulator: Full system simulation reduces simulation speed due to increased simulation complexity, takes a large amount of time to develop and requires a full kernel image for application profiling thus increasing the barrier to entry [8]. Application level simulation is the alternative approach which is taken, using techniques such as system call emulation to mimic the OS functions or limiting simulators to only running machine code. [8] [6].

3) Execution Based Simulator: Execution-based stimulus provides a more realistic target in comparison to trace-driven simulation through the ability to execute mis-predicted branches. From an educational point of view, this approach allows more experimentation as arbitrary code can be compiled and run on the simulator. Additionally, it will also expose users to the cross-compilation toolchain leading to an understanding of the tools that fulfil this purpose. Considering the lack of available RISC-V hardware to obtain traces, the execution driven approach also reduces barrier of entry.

III. Functional Simulation

Riscalar implements two simulation modes, one functional and one cycle-approximate. The functional model provides a useful reference for checking program correctness, producing an instruction profile for an unknown program by extracting statistics, and obtaining memory access information.

This functional model hides a lot of actual implementation details with a focus on decoding instructions encoded in binary format from a compiler's output and executing those instructions and observing how they affect both architectural register state and control status registers. Its development also helped to build up a reusable model for processor main

memory, including the loading of instructions, reading and writing different sized chunks of memory etc.

The functional simulator also runs a lot faster than a latency simulator due to the reduced detail, and it is used by the latency simulator to implement fast-forward execution to a particular point in time before starting detailed simulation as done by SimpleScalar and gem5 [6][11]. Both simulators share the same ISA, having full support for the base integer instruction set with support for 64 bits of address space (RV64I), combined with the standard extension for integer multiplication and division ("M extension") [21].

A. Architecture

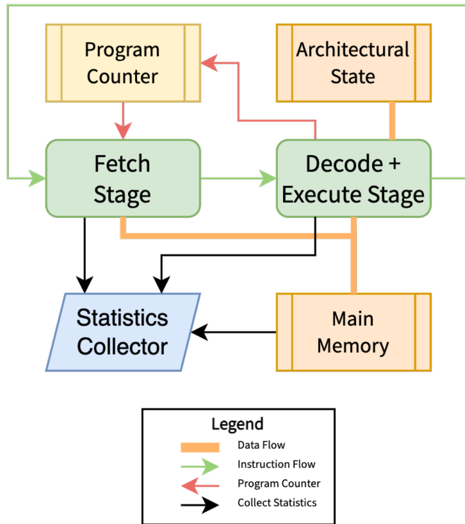

Fig. 1. Functional Simulator System Architecture

The system architecture for the functional simulator is shown in Figure 1, it is implemented with minimal components, only containing what is strictly necessary to *emulate* a RISC-V system. The two stages that are running within the simulator loop are the **Fetch** stage and the **Execute** stage which also handles instruction decoding. Each instruction is decoded and executed within the processor using the associated `execute` function, modifying either main memory through reads/writes or the architectural integer registers. If an exception occurs, the core does not trap, but exits the simulation gracefully stating the exception code and description.

B. Memory

Memory addresses in RISC-V are byte addressable, therefore the main memory is simply modelled as a vector of unsigned bytes (`u8`). The model is created as a `struct` with methods that act on immutable and mutable references to itself for reads and writes respectively. The public `read()` and `write()` functions both take an argument for address and size, with an additional parameter for value in the `write` method. Each public method then calls private methods depending on the size parameter for *bytes*, *halfwords*, *words* and *doublewords*. The memory base address is then used to translate the given address to index the vector to get or set the chosen value.

To load the binary produced by the compiler, the filename is passed as a command line argument and read into a vector of bytes using the standard `read_to_end()` function. This vector of bytes then is used to replace a portion of main memory from the base address where the core starts fetching.

C. Architectural Registers

As the number of architectural registers is fixed by the ISA, a statically allocated array of `u64` values is used in place of a vector. The zero register `x0` is hardwired to 0 and this is maintained by initialising all the values in registers to 0 on creation and ignoring all writes to `x0`. The RISC-V ISA does not define any other dedicated registers for stack pointers or return addresses, but there are standard software calling conventions in place that define which registers should be used for what as specified in Chapter 25 of [21]. One of the conventions that requires initialisation is that the stack pointer is stored in `x2`. As this register is initially 0 and the stack grows downward, the storage location for stack information is incorrect. So the known memory size and memory base address is used to initialise the value of `x2` on creation.

D. Instruction Decode and Execution

Each instruction fetched from memory is decoded and executed in a single stage for this simulator. The top level public function `execute()` separates the instruction through shifts and bitwise and operations into the different fields such as opcode, immediate, register source 1, register souce 2, register destination and then determines which private function to call based on the opcode.

Each individual function is then implemented as a large `match` statement based on the passed parameters to determine which specific instruction should be executed with what values. Each function returns a `Result<(), Exception>` type so that an exception (n.b. Not a program exception, but a RISC-V exception) can be thrown by the processor if there is an issue with the decoding or execution of the instruction. This exception would not terminate the program, but gracefully be returned up the call stack and printed for the user to see.

The full list of instructions and their encoding can be found in chapter 24 of [21].

E. Exception Handling

Each operation performed by the CPU returns the Result type, with the error value being an enum that implements the 14 possible exception codes in the RISC-V ISA as specified in Table 3.6 of Volume II of the RISC-V Privileged Architectures document. The `Result<T>` type is used to handle exceptions in all parts of the core. This type is an enumeration type used for returning and propagating errors that forms part of the standard library in Rust. It represents the result of a computation that can either be successful, returning variant `Ok(T)` with a value of type `T` or an error with an associated error value of

type E with variant `Err(E)`. e.g. An unaligned read would return a `Err(Exception::LoadAddressMisaligned)` from the memory model.

On the occurrence of an exception, the RISC-V hart should *trap* (transfer control to a trap handler), but currently this is not implemented for the functional simulator. When an exception occurs, the main execution loop breaks, and the error is output on the command line.

F. Privilege Mode

At any time a RISC-V hart is running with a given privilege level that determines the running software's access to hardware during execution. RISC-V currently defines three privilege levels ordered from highest privilege to lowest - machine mode (M-mode) for inherently trusted code such as the bootloader, supervisor mode (S-mode) for the kernel, user mode (U-mode) for user processes. Implementations of a RISC-V core do not need to implement every privilege mode, only the base M-mode is required which trades off reduced isolation for lower implementation cost. The privilege levels also affect the implementation of features and the number of control and status registers (CSRs) there are in a system [20].

For the current simulation, only M-mode is implemented along with the corresponding CSRs to simplify implementation. For a pure application-based simulator, the security provided by the other privilege levels can be thought of being less necessary as the programs being run on the simulator are expected to be trusted benchmarks or code written by the user.

G. Functional Simulator Evaluation

Each instruction was individually tested to make sure that it returned the correct result, and a simple Fibonacci calculation was used as an overall integration test to make sure that the instructions worked together. As system calls are not implemented yet, the correctness of the program was checked by asserting that the function return register contained the correct value.

TABLE I
FUNCTIONAL SIMULATOR SUMMARY

Supported ISA	RV64-IM
Privilege Levels Supported	M only
Syscall Support	N
Exception Handling	N

Lines of Code	1.2K
Binary Size	505KB
MIPS[1]	57
CPI	1.00
Memory Usage[1]	664KB

[1] as tested with `fib(30)`

As this iteration of the simulator has no detail in simulation, it has a high simulation speed without any optimisation (∼14 times faster than SimpleScalar's `sim-fast` functional simulator) and a low memory footprint as shown in Table I.

IV. Latency Simulation

As most high-performance RISC-V processor implementations are not openly available for modelling and it is a significant undertaking to design every component from the ground up, the simulator is not modelled to match a specific implementation, but rather follows a general model of how each component works. Components such as the reservation station queue that are more architecturally important are chosen to be modelled in more detail, while others such as the integer multiplier are treated as black boxes. These approximations allow for greater configurability, while also allowing the simulator to run and produce results quicker than a cycle-accurate simulator. These two advantages of cycle-approximate modelling are especially important for a simulator targeted at teaching computer architecture as the actual cycle level details are less important when considering the architectural level trade-offs within microprocessor design and a faster simulation runtime allows for more rapid iteration and experimentation with different microarchitecture designs and parameters.

The simulator's architecture is based on the SimpleScalar simulation model for PISA and Alpha, with modifications to match the RISC-V ISA and simplifications are made where possible. This approach was chosen as designing an entirely new simulation model without an existing processor to use as reference would exceed the amount of time allocated for this project and go beyond the project's scope. Previous work has shown that SimpleScalar is accurate to a high enough degree for use within educational settings.

The following sections detail the architecture of the latency simulator, the execution stages and their respective parameters, and the other customisable components within the simulator.

A. Architecture

For latency simulation, the functional simulator was extended to add the various structures that model the individual components contained within a modern microprocessor needed for both in-order and out-of-order execution with dynamic instruction issue and execution in a **cycle-approxiate** manner. It inherits some of the same capabilities of the functional simulator such as exception handling, privilege levels, and memory management, and focuses on extending the detail of instruction execution simulation.

It uses a 5-stage fixed pipeline that is controlled by calling multiple functions in each simulated cycle, moving instructions and data through the different pipeline stages to fetch, dispatch, issue, write-back and commit instructions.

The additional components within the latency simulator are shown in Figure 2. Each rounded green rectangle within the diagram represents a processor function that moves instructions through a certain pipeline stage. The blue hexagonal shapes are data structures used to hold information moving between stages during execution. These data structures may not exist in an actual RTL implementation of a processor, but are required for the simulator to track the simulated state. The yellow rectangles represent the different structures that are required to

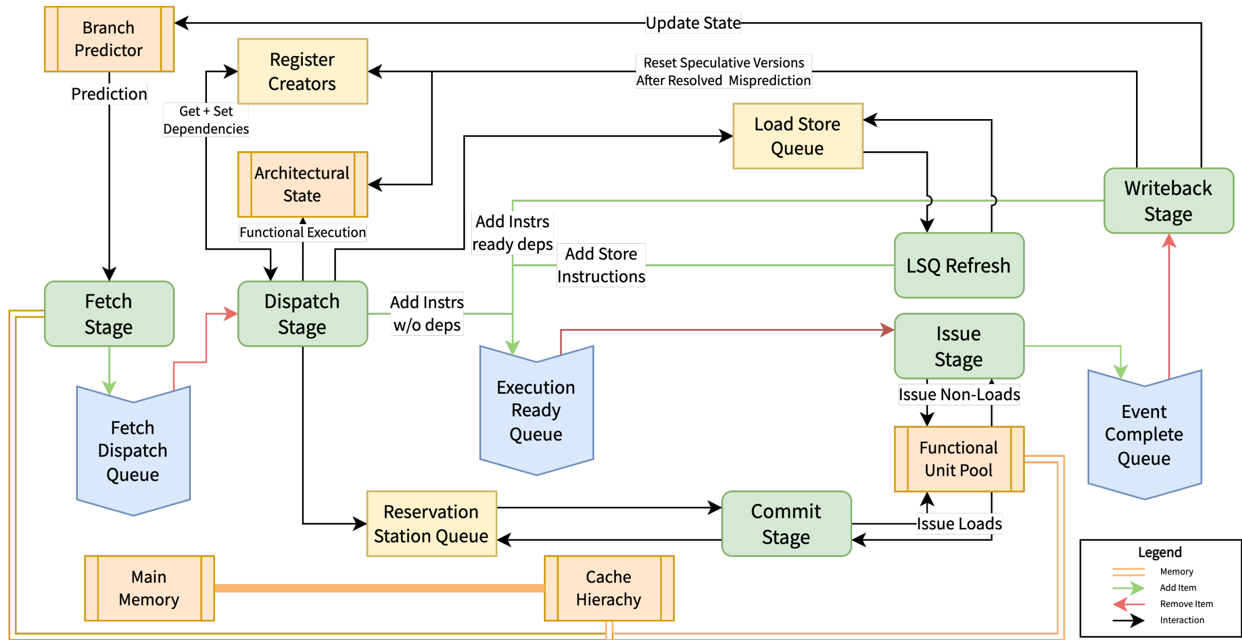

Fig. 2. Cycle Approximate Latency Simulator Architecture

model a processor that can dynamically schedule instructions on the fly with dependency analysis. The orange rectangles represent the components that are not directly related to the processor execution model, but would typically exist in an implementation. They include but are not limited to the branch predictor, caches and a functional execution pool.

B. Execution Stages

1) Instruction Fetch: Each time the fetch stage function is called, it repeatedly obtains instructions based on the predicted program counter given by the branch predictor and places the instructions within the **Fetch Dispatch Queue**. This continues until the Fetch-Dispatch Queue is full or the fetch pipeline width is reached or there is an I-cache miss. Both the fetch pipeline width (`fetch_width`) and queue size (`fetch_queue_size`) can be set by the user. An I-cache miss results in the fetch stage stalling.

2) Instruction Dispatch: This stage obtains instructions from the Fetch Dispatch Queue in program order, functionally executes the instruction so that simulation can focus on latency modelling, creates a Reservation Station and places the instructions in either the Load Store Queue (LSQ) or the Reservation Station Queue (RSQ). The LSQ and RSQ holds all the instructions that are currently in-flight within the processor, together with each instruction's output and input dependencies. Parameters such as `issue_inorder`, `rsq_size`, `lsq_size` determine how many instructions can be dispatched in a single cycle. If an instruction has no outstanding unresolved dependencies, it can also be sent directly to the **Execution Ready Queue** which holds references to all reservation stations in-flight with satisfied input dependencies.

3) Instruction Issue: This stage takes instructions from the Execution Ready Queue and attempts to get functional units to execute each instruction. If a functional unit is available, it is set to busy and a event is set to complete in the future and inserted into the **Event Queue** for the writeback function to handle. The Event Queue is a priority queue that contains references to in-flight Reservation Stations and when they complete using a functional unit, ordered by completion time. Otherwise, the instruction is reinserted into the **Execution Ready Queue** to retry instruction issue on the next cycle. The functional execution pool configuration (# of functional units, unit issue latency etc.) and the `issue_width` control how many instructions can be processed.

4) Instruction Write Back: The write-back stage takes the pre-scheduled events from the functional units recorded on the Event Queue that happen in the current cycle or previous cycles. For each event, if it indicates a mis-predicted branch being resolved, the entries within the LSQ and RSQ after the instruction are squashed, incurring the branch misprediction penalty. Additionally, for each event that has output dependencies, the register creator table is updated to remove the output dependency, and each dependent instruction that was dispatched after the current instruction is informed that one of their input dependencies has been satisfied. If all inputs are ready, they are sent to the Execution Ready Queue.

5) Instruction Commit: In the final stage, the RSQ is iterated over from oldest to newest with the LSQ and each completed entry is committed in program execution order and removed. Stores are assigned to a memory port if there is one available for use. As many entries are committed until `commit_width` limit is hit, or an incomplete instruction is found. The factors that affect this stage include the `commit_width`, the functional unit configuration.

C. Components

1) Branch Prediction: Three types of branch predictors are implemented for simulation: perfect prediction, always taken and always not taken. Perfect prediction is implemented by correcting the prediction address after obtaining the correct branch address in the functional execution of the branch instruction during the dispatch stage. The two static predictors are implemented as proofs of concept that additional more complex branch predictors can be implemented, including dynamic branch predictors.

2) Functional Unit Pool: The functional unit pool allows flexible processor configurations to target a specific instruction mix. For example, a program that has a high percentage of division instructions can add more integer division units to reduce stalls caused by a lack of available resources to execute the division and increase IPC. The pool is modelled as a table of functional units. Each functional unit is treated as a black box in terms of implementation, where only the issue and operation latency of each unit can be customised to either model a fully-pipelined or partially pipelined or un-pipelined unit implementation.

3) Register Creator Table: The register creator table keeps track of which architectural registers are created by which instruction. Each entry within the create table is either null, indicating that there is no in-flight instruction that writes to the register exists, or contains a reference to an in-flight reservation station. When an instruction that uses a non-null create table entry is dispatched, a reference to the dependent output instruction is added to the creator, so that on write-back the dependent can be updated.

4) Speculative Registers, Memory and Register Creator Table: The latency simulator can simulate the effect of a mis-prediction on performance, but it can also check if a prediction is correct. Therefore, to simplify speculative tracking, a set of registers, memory and register creator table can be added so that executing a mis-predicted branch does not wrongly affect the architectural state. For registers and the register create table, a copy is maintained with a valid bitmap to determine if the architectural registers should be read from or from the speculative registers. For memory, a hashmap is used to keep track of changes to memory that happen on a mis-predicted path, with adjusted read and write functions to match.

D. Evaluation

The latency simulator also passed every individual instruction test. Sweeping parameters also showed the expected performance improvements and degradation across the different available statistics. Table II shows that as expected, when compared to the functional simulator, simulation speed decreases by 3.5x, and the memory usage jumps up by a order of 3. However, the simulator remains faster than SimpleScalar's 'sim-outorder' which with a similar configuration executes at 0.2 MIPS. As this simulator has not been optimised, it is possible that spending further time on optimisation would allow the simulation speed to increase further, and have a reduced memory footprint.

TABLE II
LATENCY SIMULATOR SUMMARY

Supported ISA	RV64-IM
Privilege Levels Supported	M only
Syscall Support	N
Exception Handling	N

Lines of Code	3K
Binary Size	1.5MB
MIPS[1]	$\sim$ 16
CPI	Config Dependent
Memory Usage[1]	600MB

[1] as tested with `fib(32)` on a 4-wide OoO pipeline

V. Future Work

While developing Riscalar and undertaking the literature review, three main areas of improvement were noted for future development.

1) User Experience: While a graphical user interface is available for Riscalar, built using a Flutter front-end, the experience remains crude and unpolished. The importance of graphical user interfaces for simulation tools was noted previously, which is why the CLI libraries have been built with GUI support in mind, but further time is needed to develop a first-class user experience.

2) Simulation Capabilities: Currently, the simulator only supports the RV64IM instruction set. Adding other RISC-V extensions such as the floating point extension and being able to configure which extensions are supported by the simulator would increase the relevance of the tool. Adding generalised estimates for power consumption and area would also make the tool more useful as these are other constraints that are emphasised when designing microprocessors. Implementing system call emulation for Riscalar would also reduce the compiler restrictions needed now such as `nostdlib`.

3) Optimisation + Multiplatform: As mentioned in the previous section, the latency simulator has not been analysed for optimisation and has a significant memory footprint. With optimisation, it may be possible to create a version of the latency simulator that runs within the web browser using WebAssembly, allowing more people to access and use it.

VI. Conclusion

This work has presented Riscalar, a highly parameterisable computer architecture simulator for the RISC-V ISA. Through the description of its design and implementation, it can be observed that it may not necessarily be the most true-to-life simulation tool, but its targeting of the RISC-V ISA and the intuitiveness that it provides around system-level computer architecture design make it a valuable resource for teaching. The flexible nature allowing thousands of system parameter combinations is sure to help students understand the trade-offs that computer architects face in industry. It is hoped that through further development and extension, Riscalar can improve to become a better, more engaging tool for computer engineering undergraduates and the wider education and scientific community alike.

References

[1] A. Dunworth and V. Upatising, “Umac: A simulated microprogrammable teaching aid,” *ACM SIGCSE Bulletin*, vol. 21, no. 3, pp. 39–43, 1989.

[2] D. Foley, “Microcode simulation in the computer architecture course,” *ACM SIGCSE Bulletin*, vol. 24, no. 3, pp. 57–59, 1992.

[3] S. D. Bergmann, “Simulating and compiling a hypothetical microprogrammed architecture with projects for computer architecture and compiler design,” *ACM SIGCSE Bulletin*, vol. 25, no. 2, pp. 38–42, 1993.

[4] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations,” *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 83–94, 2000.

[5] C. T. Weaver, K. C. Barr, E. D. Marsman, D. Ernst, and T. M. Austin, “Performance analysis using pipeline visualization.,” in *ISPASS*, 2001, pp. 18–21.

[6] T. Austin, E. Larson, and D. Ernst, “Simplescalar: An infrastructure for computer system modeling,” *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[7] D. Burger, T. M. Austin, and S. W. Keckler, “Recent extensions to the simplescalar tool suite,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 4–7, 2004.

[8] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder, “Automatic logging of operating system effects to guide application-level architecture simulation,” in *Proceedings of the joint international conference on Measurement and modeling of computer systems*, 2006, pp. 216–227.

[9] A. Beg and W. Ibrahim, “An online tool for teaching design trade-offs in computer architecture,” in *Proc. International Conference on Engineering Education*, Citeseer, 2007, pp. 3–7.

[10] B. Nikolic, Z. Radivojevic, J. Djordjevic, and V. Milutinovic, “A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization,” *IEEE Transactions on Education*, vol. 52, no. 4, pp. 449–458, 2009.

[11] N. Binkert, B. Beckmann, G. Black, *et al.*, “The gem5 simulator,” *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[12] F. Sanchez, D. Megas, and J. Prieto Blazquez, “Simr: A simulator for learning computer architecture,” *International Journal of Engineering Education*, vol. 27, no. 2, p. 238, 2011.

[13] R. Hasan and S. Mahmood, “Survey and evaluation of simulators suitable for teaching for computer architecture and organization supporting undergraduate students at sir syed university of engineering & technology,” in *Proceedings of 2012 UKACC International Conference on Control*, IEEE, 2012, pp. 1043–1045.

[14] P. Jamieson, “Does badge-based learning buck the grading curve? an educational experiment in computer architecture,” in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, The Steering Committee of The World Congress in Computer Science, Computer ..., 2014, p. 1.

[15] P. Prasad, A. Alsadoon, A. Beg, and A. Chan, “Using simulators for teaching computer organization and architecture,” *Computer Applications in Engineering Education*, vol. 24, no. 2, pp. 215–224, 2016.

[16] R. Agrawal, S. Bandara, A. Ehret, M. Isakov, M. Mark, and M. A. Kinsy, “The brisc-v platform: A practical teaching approach for computer architecture,” in *Proceedings of the Workshop on Computer Architecture Education*, 2019, pp. 1–8.

[17] A. Akram and L. Sawalha, “A survey of computer architecture simulation techniques and tools,” *IEEE Access*, vol. 7, pp. 78 120–78 145, 2019.

[18] R. Höller, D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer, and M. Linauer, “Open-source risc-v processor ip cores for fpgas—overview and evaluation,” in *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, IEEE, 2019, pp. 1–6.

[19] A. Shilov, *Western digital rolls-out two new swerv risc-v cores for microcontrollers*, Dec. 2019. [Online]. Available: https://www.anandtech.com/show/15231/western-digital-rollsout-two-new-swerv-riscv-cores.

[20] A. Waterman, K. Asanovic, J. Hauser, S. Inc., and C. D. E. D. U. of Calfornia Berkeley, “The RISC-V Instruction Set Manual Volume ii: Unprivileged ISA Version 20191213,” Tech. Rep., Dec. 2019.

[21] A. Waterman, K. Asanovic, S. Inc., and C. D. E. D. U. of Calfornia Berkeley, “The RISC-V Instruction Set Manual Volume i: Unprivileged ISA Version 20191213,” Tech. Rep., Dec. 2019.

[22] E. Systems, *Introducing esp32-c3*, Nov. 2020. [Online]. Available: https://www.espressif.com/en/news/ESP32_C3.

[23] D. Kleidermacher, J. Seed, B. Barbello, and S. Somogyi, *Pixel 6: Setting a new standard for mobile security*, Oct. 2021. [Online]. Available: https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html.

[24] M. B. Petersen, “Ripes: A visual computer architecture simulator,” in *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, IEEE, 2021, pp. 1–8.

[25] G. Mariotti and R. Giorgi, “Webrisc-v: A 32/64-bit risc-v pipeline simulation tool,” *SoftwareX*, vol. 18, p. 101 105, 2022.