embeddedworld2022
Exhibition&Conference
... it's a smarter world

QtRVSim – RISC-V Simulator for Computer Architectures Classes

Pavel Pisa, Jakub Dupak, Karel Koci
Department of Control Engineering, FEL
Czech Technical University in Prague
Prague, Czech Republic
pisa@cmp.felk.cvut.cz, dev@jakubdupak.com,
cynerd@email.com

Michal Stepanovsky
Department of Computer Science, FIT
Czech Technical University in Prague
Prague, Czech Republic
michal.stepanovsky@fit.cvut.cz

***Abstract*—This paper presents a RISC-V based computer system simulator, QtRVSim (https://github.com/cvut/qtrvsim), designed for teaching and learning computer systems principles. The simulator allows students to run assembly programs and observe the instruction execution on single-cycle and pipelined microarchitectures. The simulator graphically displays the major components in the datapath, including the register file, the arithmetic-logic unit, memory caches, peripherals, and the control unit with control signals. QtRVSim is free and open-source software available on GitHub and as a WebAssembly application online. For additional materials and related computer architecture projects, see https://comparch.edu.cvut.cz/.**

Keywords—Computer science education; RISC-V simulator; instruction cycle; single-cycle microarchitecture; pipelined microarchitecture; memory subsystem; cache memory; peripherals; assembly language

I. INTRODUCTION

Computer architecture and organization courses play a vital role in understanding modern computer systems. Students should understand fundamental concepts of the design and organization of the central processing unit, memory subsystem, input/output subsystem, and their integration into a computer system to be able to use the computer resources effectively. Many students can easily understand programming languages like C, C++, Java, and C#. However, our experience shows that they encounter difficulties understanding the instruction cycle implementation and its consequences for the low-level optimization of their programs. Additionally, many students could not bridge the gap between low-level hardware and a high-level programming language. Students often understood how to add two integer numbers in hardware, but they could hardly imagine how to use this knowledge to implement actual instructions. To solve this problem, we have tried using several different simulators over the last decade that are capable of visualizing simple microarchitectures. Students positively responded to the incorporation of graphical presentation and animation in the simulators, saying it significantly increased their understanding of all concepts. We found that the simulator must meet certain criteria to be beneficial to students:

- **Instruction set architecture (ISA).** There are obvious choices such as x86 and x64. However, these ISAs are complicated and unsuitable for students without prior knowledge of computer architectures. We have a positive teaching experience with MIPS32 because of its simplicity. However, MIPS32 is proprietary and becoming outdated. The next option would be ARM ISA though it is also proprietary. RISC-V ISA is open, simple, and currently receiving much attention.
- **Graphical User Interface (GUI).** The simulator should provide a GUI for student interaction.
- **Computer system components visualization.** The simulator should go beyond the components defined in ISA (register file, main memory, program counter) and show at least a control unit and control signals, arithmetic-logic unit, cache memories, and ideally peripherals.
- **Instruction cycle visualization.** The simulator should provide information about the instruction fetched from memory and its execution in each simulated microarchitecture.
- **User-friendliness.** The simulator should be user-friendly with a minimal learning curve.
- **Portability.** The simulator should run on major platforms, such as Linux, Microsoft Windows, and macOS. A web application is also a significant benefit.

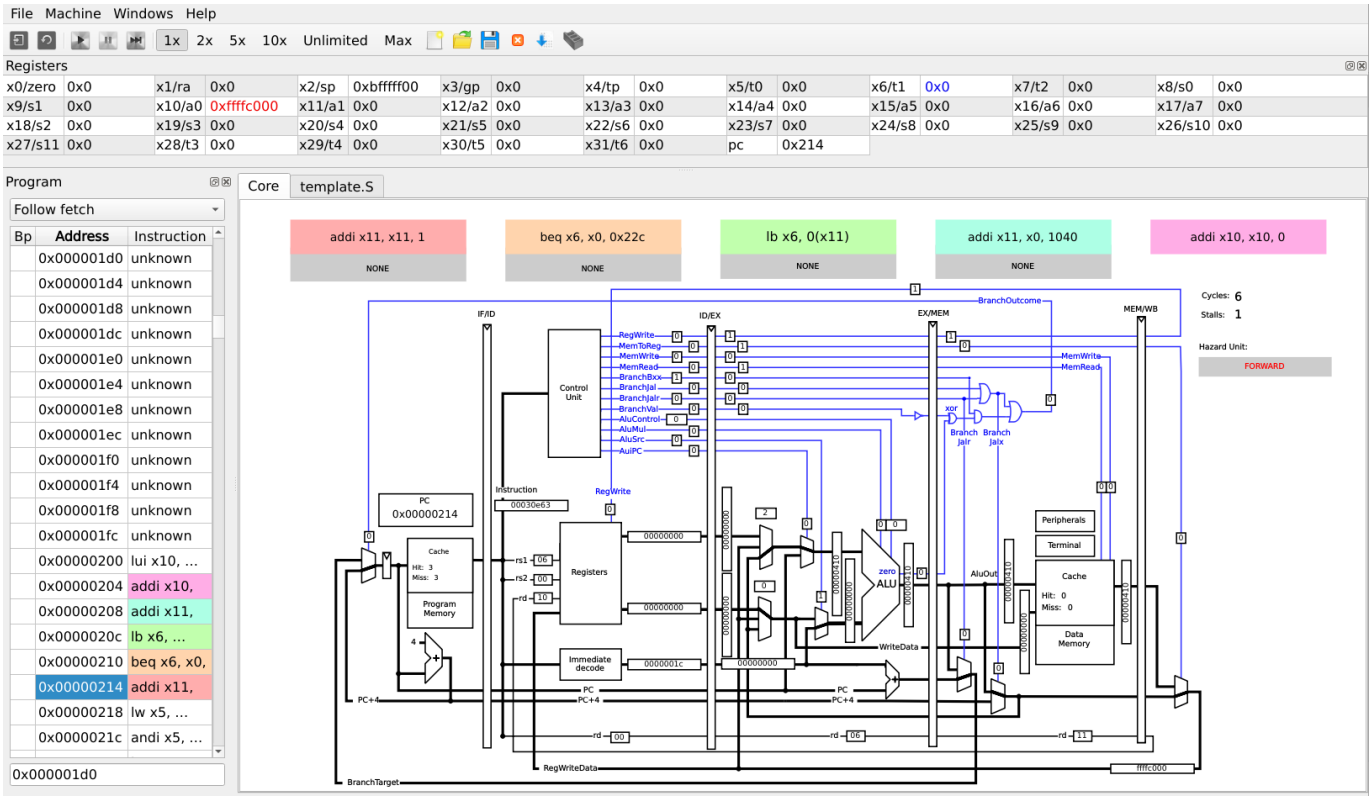

Fig. 1. QtRVSim simulator, showing a five-stage pipelined microarchitecture

II. Features of QtRVSim

Fig. 1 illustrates the graphical design of the simulator. It is composed of multiple panels/windows, each displaying a specific part of the simulator, e.g., code editor, register file, microarchitecture datapath, and peripherals. The user can change the position and the size of individual panels or close them. In this section, we briefly describe the main features of the QtRVSim simulator.

A. Integrated Assembler

QtRVSim includes a built-in assembler and a basic program editor so students can easily write, debug, and assemble programs without third-party programs. QtRVSim recognizes a small subset of GNU assembler directives and some custom directives. For example, the assembler supports directives to show particular tabs/windows to ensure students get the same starting point while working with provided source codes. Unknown directives are ignored to allow for code that both an integrated and a standalone assembler can compile.

B. Datapath and Instruction Cycle Visualization

The simulator shows the main components in the datapath, including their inputs and outputs. For instance, students can explore ALU inputs and outputs for each instruction, where these inputs are coming from, and where they are leading. Students can execute instructions step-by-step, run an animation of the execution at the rate of 1, 2, 5, or 10 instructions per second, or execute the program at maximal speed. The simulator provides the relevant information about the instruction processing so that the student easily understands how these instructions are implemented in hardware.

C. Various processor microarchitectures

The QtRVSim supports multiple microarchitecture variants. Specifically, the following microarchitectures are currently predefined:

- M1: Single-cycle microarchitecture without pipeline and cache memories
- M2: Single-cycle microarchitecture without pipeline, but with cache memories
- M3: 5-stage pipelined microarchitecture without hazard unit (no hazard resolving) and without cache memories
- M4: 5-stage pipelined microarchitecture with hazard unit and with cache memories

Microarchitecture M1 can be used as the introductory micro-architecture. Many students often understand the base CPU building blocks; however, they have difficulty realizing how to interconnect them to support a given ISA. Microarchitecture M2 is a slightly extended microarchitecture that includes cache memories, thus, giving valuable insights when learning memory hierarchy concepts. Microarchitecture M3 introduces a 5-stage integer pipeline without hazard resolving, thus allowing students to investigate the behavior of such processors in the presence of inter-instruction dependencies. Finally, microarchitecture M4 concludes the basic concepts of pipelined instruction processing.

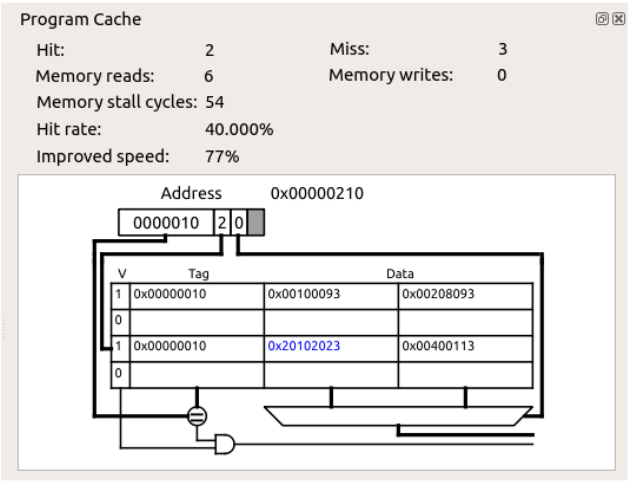

Fig. 2. Instruction cache window

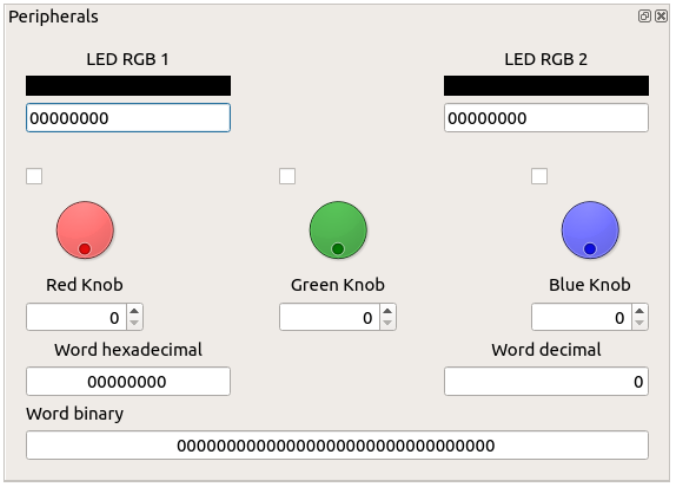

Fig. 3. Dedicated peripheral with two RGB LEDs and three knobs

As shown in Fig. 1, each instruction is colored depending on the pipeline stage where it is currently processed.

D. *Instruction and Data Cache Simulation*

The simulator allows students to study the computer system without and with memory caches (i.e., instruction cache and data cache) to learn the fundamental concepts of the memory hierarchy. In addition, they can study the cache behavior for various cache organizations (block size, cache size, associativity) and under various writeback and replacement policies (write-through no-write-allocate, write-through write-allocate, writeback, least recently used (LRU), least frequently used (LFU), and random replacement). Fig. 2 illustrates the behavior of an instruction cache for given parameters and programs. The instruction cache window shows the total number of hits and misses, the number of reads and writes, memory stall cycles, hit rate, and improved speed.

E. *Peripherals*

The QtRVSim simulator also offers several simple peripherals mapped to the memory address space, allowing students to learn about the memory-mapped I/O (MMIO) concepts. The first peripheral is a simple serial port (UART) connected to the terminal window. Thus, students can experiment with two basic character devices – a keyboard as the input device and a character display as the output device. The next implemented peripheral is a 480x320 LCD with a simple 16-bit per pixel (RGB565) frame buffer mapped into the memory address space. Moreover, as shown in Fig. 3, the QtRVSim supports more advanced I/O devices — three independent knobs (red, green, and blue) with buttons and two RGB LEDs.

F. *Operating System Emulation*

QtRVSim supports an operating system (OS) emulation feature, allowing students to enable system services, stop on known/unknown system calls, stop on interrupt entry, and stop and step over exceptions. Currently, the emulator supports a few Linux kernel system calls (*exit, read, write, close, openat, brk, ftruncate, readv, writev*). Although a very small subset of system calls is implemented, it is sufficient to demonstrate and explain the principles of interaction between user programs and the operating system.

III. BUILT-IN EXAMPLES

Currently, QtRVSim provides three built-in examples:

A. *Simple-lw-sw-ia.s*

This example is used to illustrate how to access the main memory. A user-defined constant is stored at the user-defined address in the main memory. The program uses LW instruction to load the value from the main memory into a specified register. Then, by using the SW instruction, the register content is written back to the memory at the adjacent position. This example can be used as a starting point to learn the QtRVSim interface and learn cache memory principles.

B. *Template-os.s*

This example shows how the user program interacts with the operating system. Specifically, it uses the system call WRITE to print out the text string “Hello world!” on the terminal. Then terminates by calling the system call EXIT. The received output is illustrated in Fig. 4.

C. *Template.s*

Finally, to illustrate polled I/O (or software-driven I/O), the last example accesses the peripheral directly from the user program. Like the previous example, it prints out the predefined text string “Hello world!” on the terminal.

Fig. 4. Output message from the build-in example *“Template-os.s”* printed out on the Terminal window

IV. Implementation

When designing the internal structure of the simulator, we paid close attention to extensibility and maintainability. Here we provide some examples of our design. Further details can be found in theses of the involved students [1-3].

A. Instruction Decoding

All supported instructions are described in a declarative manner, which makes adding new instructions straightforward. A single table is used for decoding, encoding (integrated assembler), and simple syntax highlighting (integrated editor).

B. Memory Model

The memory model can support a large variety of configurations. We divide the memory subsystem vertically into two parts. At the upper level is the cache hierarchy, which can be almost arbitrarily structured – from no or single unified cache to complicated multilevel structures with separate ports for data and program memory accesses. In the lower level, various memory-mapped devices can be connected to the memory bus – theoretically, even during simulation. Peripheries and memory modules can be either emulated or mapped from the host system (e.g., RAM can be allocated using the host OS memory mapping functionality). The memory model data structures provide the functionality to support all combinations of the endianness of the simulated machine, the host machine, and the periphery.

C. Visualization of the Core

The visualization is designed as SVG files with special annotation for dynamic content (register and wire values, multiplexers, and links to other simulator parts). The simulator creates native Qt GUI components from the SVG files and connects the components to the simulated machine's API.

V. Teaching

The simulator is typically used during labs, where students construct assembly programs satisfying specific criteria. For example, students are tasked with writing and optimizing the bubble sort program to minimize the number of no-ops, given that the hazard unit is unavailable. In addition to GUI, the simulator is shipped with a headless CLI, which can be used for automated evaluation.

Students are expected to understand the purpose of individual components as visualized by the simulator. A typical exam task is to analyze the execution of a specific program under special conditions – such that a selected wire or signal is "stuck" on a specific value.

The simulator can save the core visualization as a vector PDF at any time of execution. We use this functionality to create comprehensive teaching materials with the visualization students are already familiar with.

VI. Future Work

We plan to extend the simulator to cover more advanced CPU features, including branch prediction, multilevel cache memories, simple virtual memory management support with TLB visualization, and multicore instruction processing with visualized MESI protocol. We will continue to extend examples, documentation, and teaching materials (see resources).

The project is mainly extended by students in final projects, theses, and following work. We kindly invite students from other universities to participate.

VII. Conclusion

This paper has introduced the QtRVSim, the RISC-V ISA-based simulator to support computer architecture courses. It supports single-cycle and pipelined microarchitectures, graphically visualizes the datapath, allows students to analyze both data and instruction cache memories, and emulates communication with peripherals (directly or via OS system calls). QtRVSim is also available as a WebAssembly application online, so no installation is required. QtRVSim is an open-source project; thus, the open-source community is welcome to develop this simulator further.

Acknowledgement

We want to thank Czech Technical University and The Ministry of Education, Youth and Sports of the Czech Republic for supporting our work from "*SP2021+ The Strategic Plan of the Ministry for Higher Education for the period from 2021*". We would like to also acknowledge the help of volunteers from the open-source community, especially Tomas Chvatal (SUSE Linux s.r.o.) and David Heidelberg, for their contributions and help with project packaging.

Resources

- Project source code and releases https://github.com/cvut/qtrvsim
- Computer architectures projects at CTU https://comparch.edu.cvut.cz/
- Undergraduate Computer Architectures course https://cw.fel.cvut.cz/wiki/courses/b35apo/en/start
- Graduate Advanced Computer Architectures course https://cw.fel.cvut.cz/b211/courses/b4m35pap/start

References

[1] J. Dupak, "Graphical RISC-V architecture simulator - memory model and project management," CTU Prague, 2021

[2] M. Hollmann, "Graphical RISC-V architecture simulator - instructions decode and execution and os emulation," CTU Prague, 2021

[3] K. Koci, "Graphical CPU Simulator with Cache Visualization," CTU Prague, 2018

[4] M. B. Petersen, "Ripes: A Visual Computer Architecture Simulator," 2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE), 2021, pp. 1-8, doi: 10.1109/WCAE53984.2021.9707149.

[5] Giorgi, Roberto and Mariotti, Gianfranco. "WebRISC-V: a Web-Based Education-Oriented RISC-V Pipeline Simulation Environment ". ACM Workshop on Computer Architecture Education (WCAE-19), 2019, pp. 1-6, doi: 10.1145/3338698.3338894.

[6] B. Nova, J. C. Ferreira and A. Araújo, "Tool to support computer architecture teaching and learning," 2013 1st International Conference of the Portuguese Society for Engineering Education (CISPEE), 2013, pp. 1-8, doi: 10.1109/CISPEE.2013.6701965.

[7] J. Djordjevic, B. Nikolic and A. Milenkovic, "Flexible web-based educational system for teaching computer architecture and organization," in IEEE Transactions on Education, vol. 48, no. 2, pp. 264-273, May 2005, doi: 10.1109/TE.2004.842918.

[8] RISC-V. (n.d.). Retrieved May 4, 2022, from https://riscv.org/