RISCV-PySim: A Modular and Flexible Python-Based RISC-V Simulator

Carlos Rojas Morales[1,2]*, Víctor Asanza[1], Julian Pavon[1,2], Ivan Vargas Valdivieso[1,2], Erick Brandon Cureño Contreras [3], and Adrian Cristal[1]

[1]Barcelona Supercomputing Center (BSC)
[2]Universitat Politècnica de Catalunya (UPC)
[3]Centro de Investigación en Computación, Instituto Politécnico Nacional (CIC-IPN)

Abstract

This work presents RISCV-PySim, a Python-based simulator for RISC-V processors. RISCV-PySim is designed to incorporate and evaluate custom instructions, facilitating the development of Domain-Specific Hardware Accelerators (DSHA). As a case study, we implemented a General Matrix Multiply (GEMM) benchmark using scalar, vector and systolic array approaches. To estimate the speed-up between these three implementations, we implemented a simplified model of a in-order Central Processing Unit (CPU). Then, we compared the efforts to do the same using the State-of-the-Art (SOTA) gem5 simulator.

RISCV-PySim is a particularly useful tool to verify the correctness and semantics of new instructions, a task that would require significant effort in the SOTA simulators. Additionally, the modular architecture of RISCV-PySim enables seamless integration with various performance models, allowing adapting to specific needs.

Introduction

The State-of-the-Art (SOTA) Domain-Specific Hardware Acceleratorss (DSHAs) have demonstrated improvements in performance and energy efficiency over general-purpose Central Processing Units (CPUs) [1]. Furthermore, the RISC-V Instruction Set Architecture (ISA) is designed to take advantage of this by allowing the integration of custom instructions, unlike traditional ISAs. To develop new ISA extensions and custom instructions, it is crucial to have tools to evaluate these new instructions before starting the Hardware (HW) implementations. There are several SOTA simulators that are capable of modeling and evaluating these new instructions [2]. However, SOTA simulators have become increasingly complex, to the point where evaluating new instructions has become a time-consuming and cumbersome process.

This work introduces RISCV-PySim, a modular and flexible Python-based simulator that simplifies the integration of custom instructions into the RISC-V ISA. It follows a top-down approach, starting with application requirements and acceleration strategies to facilitate the development and validation of DSHAs. Instead of executing the compiled binary outputs, RISCV-PySim runs Assembly (ASM) code generated by the standard RISC-V compiler, allowing for seamless integration of custom instructions into the standard C code.

*Corresponding author: carlos.rojas@bsc.es

Methodologies

The simulator is based on an Finite State Machine (FSM) with two decoupled modular components: The *Functional Engine*, which executes the program and generates traces, and the *Performance Engine*, which evaluates the execution cycles (Fig. 1).

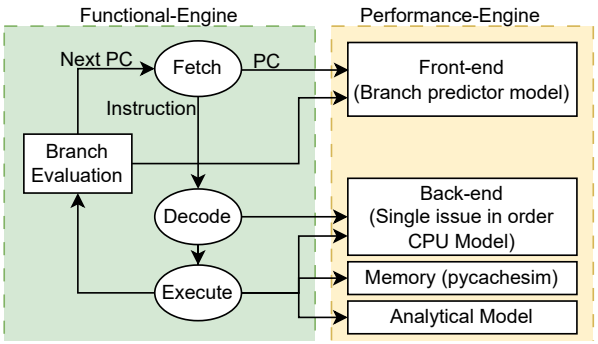

Figure 1: *Proposed RISCV-PySim architecture*

Functional Engine: This component executes instructions atomically, allowing execution traces to be sensed to an independent performance model implementation. Its main components are: **Fetch:** Retrieves the instruction from Level 1 Instruction Cache (L1-I) and controls the program execution flow through the Program Counter (PC). **Decode:** Decodes the instruction, obtaining the operands and the specific computation function that must be executed. **Branch Evaluation (BE):** Executes the evaluation of conditional and unconditional branches to determine the next PC. **Execute Stage:** Executes the computa-

tion function of each instruction, invoking statically generated execution functions from the Decode stage.

Performance Engine: This component configures architectural parameters to estimate the performance of a given use case. It includes: **Front-end:** Implements a Branch Predictor (BP) model. **Back-end:** Simulates a *single-issue* (*in-order*) CPU. **Memory:** Uses the `pycachesim` [3] tool to estimate the cycles required for memory accesses. **Analytical Model:** We used a simplified modeling approach [4] to estimate performance considering latencies, branch mispredictions, and accesses to Level 1 Data Cache (L1-D). However, the modularity of the simulator allows the interchange of a variety of performance models according to the desired accuracy and requirements.

Use case

To evaluate RISCV-PySim, we used a General Matrix Multiply (GEMM) benchmark using scalar, vector and custom instructions. Custom instructions manage a DSHA based on a systolic array architecture to multiply 8x8 matrices with 8-byte elements, stored in *row-major* format. We introduce three custom instructions to compute with the systolic array *write_ buffer*, *read_ buffer* and *multiply*. To maintain program order, the systolic array instructions are executed in the commit stage, and the CPU pipeline is stopped until the execution has finished. The algorithm 1 describes the operation of the systolic array, where the *write_ buffer* function transfers data from *memory* to the systolic array. The multiplication operation is performed by the *multiply* function, which processes data from *bufferA* and *bufferB*, storing the result in *bufferC*. Finally, the *read_ buffer* function transfers the results from the systolic array back to the main *memory*.

Algorithm 1: Systolic Array 8x8

```
for i ← A.N; i+ = 8 do
    for k ← B.M; k+ = 8 do
        bufferA ← block(A_{i→i+7:k→k+7})
        for m ← B.N; m+ = 8 do
            bufferB ← block(B_{k→k+7:m→m+7})
            bufferC = multiply()
            block(C_{i→i+7:m→m+7}) += bufferC)
        end
    end
end
```

We implemented a simple analytical performance model based on the 64-bit Sargantana processor, which supports the RV64G ISA, RVV 0.7.1, and a 7-stage pipeline optimized for frequencies above 1 GHz [5]. The processor features an in-order pipeline that supports scalar, vector, and custom instructions. For the vector data-path we used a maximum vector-length of 2 elements. The model includes a non-blocking L1-D, optimizing load operations by handling multiple cache misses. The memory subsystem consists of three levels: **L1:** 3-cycle latency for hits (32KB). **L2:** 30-cycle latency for hits (512KB). **main memory:** fixed latency model of 250 cycles.

Discussion

The results show that in the used GEMM benchmark, the acceleration between the scalar implementation and the vector implementation was 1.68X. Similarly, the speed-up achieved between the scalar implementation and the systolic array implementation with a custom instruction was 9.63X. In this regard, using a very simple performance model, we observe the potential speed-up between the three implementations, scalar, vector, and systolic array.

Integrating the architecture and functional model of custom instructions and the systolic array accelerator required only 67 lines of code, demonstrating the feasibility of implementing ISA extensions in a very efficient manner compared to doing the same implementation in the gem5 simulator that required the modification of twelve different files and approximately 300 lines of code [2]. RISCV-PySim is a particularly useful tool to evaluate and verify the semantics of new custom instructions, which would otherwise require significant effort to implement in many SOTA simulators. Furthermore, its modular architecture allows for seamless integration with a variety of performance models according to specific needs.

References

[1] Yuze Chi et al. "Democratizing domain-specific computing". In: *Communications of the ACM* 66.1 (2022), pp. 74–85.

[2] Ayaz Akram and Lina Sawalha. "A survey of computer architecture simulation techniques and tools". In: *Ieee Access* 7 (2019), pp. 78120–78145.

[3] RRZE-HPC. *pycachesim: Python Cache Hierarchy Simulator*. `https://github.com/RRZE-HPC/pycachesim`. AGPL-3.0 License. 2024.

[4] Atanu Barai et al. "PPT-SASMM: Scalable analytical shared memory model: Predicting the performance of multicore caches from a single-threaded execution trace". In: *Proceedings of the International Symposium on Memory Systems*. 2020, pp. 341–351.

[5] Víctor Soria-Pardos et al. "Sargantana: A 1 GHz+ in-order RISC-V processor with SIMD vector extensions in 22nm FD-SOI". In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE. 2022, pp. 254–261.