Simulation and Formal: The Best of Both Domains for Instruction Set Verification of RISC-V Based Processors

Simulation and Formal: The Best of Both Domains for Instruction Set Verification of RISC-V Based Processors

Ckristian Duran*, Hanssel Morales*, Camilo Rojas*, Annachiara Ruospo†, Ernesto Sanchez† and Elkim Roa*
* Integrated Systems Research Group - OnChip, Universidad Industrial de Santander - Colombia
†Politecnico di Torino - Italy. e-mail: ckristian.duran@correo.uis.edu.co

***Abstract*—The instruction set architecture (ISA) specifies a contract between hardware and software; it covers all possible operations that have to be performed by a processor, regardless of the implemented architecture. Verifying the instruction execution against a golden execution model following the ISA is becoming a common practice to verify processors. Despite many potential applications, existing verification frameworks require an extensive test set to cover most of the processor states. In this paper, we suggest a verification scheme combining two different domains, simulation- and formal-verification, establishing a methodology for exclusive error detection. The first approach drives automatic program generation using genetic algorithms to maximize coverage of the test and the contrast against an instruction set simulator. The second is a formal verification approach, where an interface carries specific processor states according to the ISA specification. By combining these two, we present a reliable way to perform more accurate instruction verification by increasing processor state coverage and formal assertions to detect different kinds of errors. Compared to extensive torture test sets, this approach reaches a more significant number of internal states by taking advantage of the exercised abstractions. Among remarkable results to highlight, the proposed approach detected a RISC-V ISA specification gap revealing ambiguity from two different verification perspectives.**

I. Introduction

The instruction set architecture (ISA) is the main specification of any processor implementation, and it contains detailed information about the instruction execution, the data registers, and the memory interactions through an external bus. Processors are commonly described by using a hardware description language in a register transfer level (RTL). RTL implementations must be able to execute any instruction specified in the ISA. To verify the processor against the ISA, designers and certifiers run a set of tests to verify the processor by using hardware simulation. Processor architectures must be verified to give solidness over the instruction execution before and after the circuit is synthesized in a technology node.

Running an extensive software test set inside a hardware description simulation is a common approach to perform ISA verification in processors. This test set is compared to an execution model, which checks the execution results for each instruction specified in the ISA. However, guaranteeing an accurate execution verification model to cover all processor states is challenging. Moreover, the processor model might not be accurate enough to perform comparisons against the ISA specification. A RISC-V case-of-study conduits comparisons

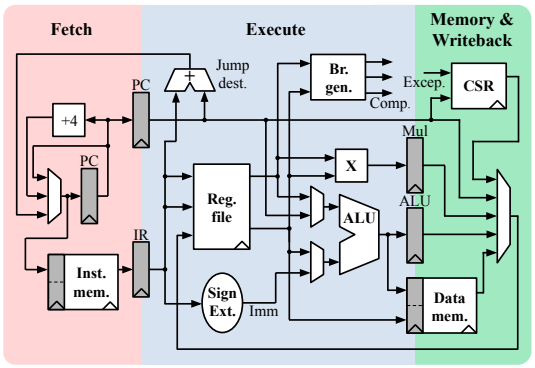

Fig. 1. Microarchitecture of the processor under verification (PUV). PUV comprise 3-stage single issue in order pipeline.

to simulation models implementing an evolutionary framework with a coverage-based optimization function [1]. In order to verify the instruction simulator, coverage-guided fuzzing may be performed [2]. Authors in [3], apply a different approach by using evolutionary algorithms to compare the execution of two different processors.

Although a coverage-based optimization aims to envelop all the processor states, formal verification can explore all possible states, according to specifications, using assumptions and checking errors using assertions [4]. For Arm processors, a specification language named ”architecture specification language” (ASL) allows any Arm processor to be scaled in the RTL environment [5]. Arm verifies its processors in any architectural implementation by extending the ASL language to include formal properties. A RISC-V formal verification framework allows the implementation of verification intellectual property (IP) specified in formal verification assertions [4]. The test framework may be suited to several architectural implementations through a *RISC-V formal* interface by adding internal processor states.

In this paper, we present a verification scheme that combines two functional platforms. We implemented a μGP-based RISC-V program generator, which uses a genetic algorithm to find the best program that covers most of the processor states. For every individual generated from the genetic algorithm, we perform a comparison against a simulation program as a golden model. Such simulation can be performed by any RISC-V simulator, but for this paper, we chose the *Spike*

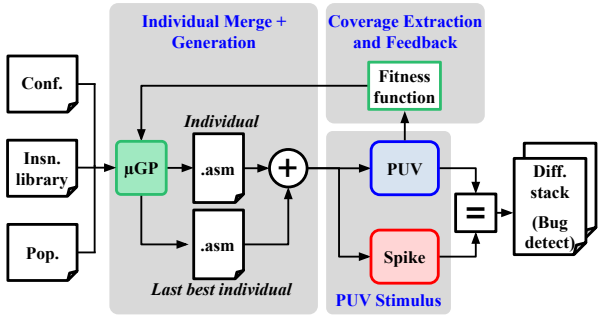

Fig. 2. Test generation and simulation-based verification methodology for the PUV a RV32IM based processor.

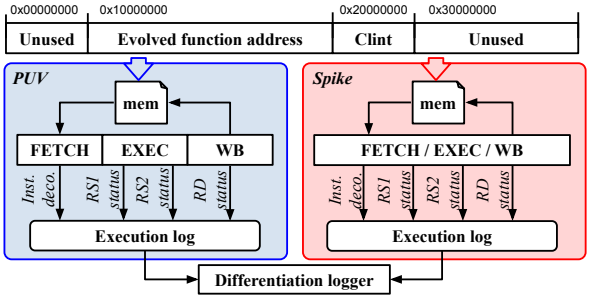

Fig. 3. Parallel execution on the PUV and *Spike* using the same memory model. The execution is logged from the states of each model.

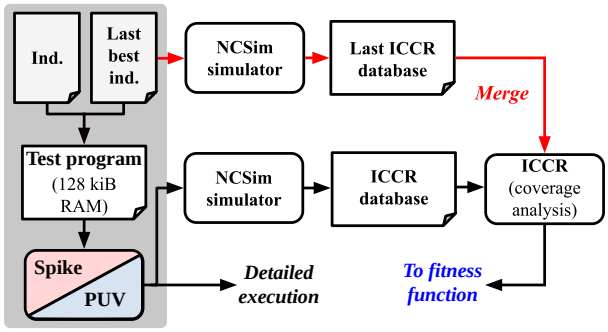

Fig. 4. Simplified block diagram of the code coverage extraction processes using Cadences tools.

simulator following the suggestion by the RISC-V foundation [6]. In addition to the program generator, a second approach based on formal verification is also utilized. We attached a set of verification IP with formal properties named *RISC-V formal* [4]. This IP compares internal states of the processor to a formal specification derived from the RISC-V ISA. We show the detection of induced errors conducted in both simulation and formal approaches. The framework was capable of detecting a consistent misunderstanding in the ISA for asynchronous processor interruptions.

II. Core Verification

A. Using Spike and µGP

We established the simulation-based verification framework on *µGP* to generate effective test programs [3]. *µGP* stands on a test program generation algorithm, handling an evolutionary core. We compare results from test programs run in both the processor and a RISC-V instruction simulator. The RISC-V foundation provides *Spike* [6], the golden RISC-V ISA simulator within a functional model for instruction execution. *Spike* is compliant with the RV32I or RV64I base ISA, supporting M, A, F, D, and Q extensions among extra features.

We designed an in-house 32-bit RISC-V ISA based processor, which is the processor under verification (PUV) in this framework. The PUV is an ultra-low power consumption processor intended for low-performance tasks, comprises a three-stage pipeline single-issue in-order, and is compatible with I, M, and C RISC-V extensions. Fig. 1 describes the microarchitecture of the PUV.

Fig. 2 summarizes the methodology for simulation-based verification used in this work. *µGP* automatically generates a set of programs called individuals by using a constrained genetic algorithm. According to the feedback (the fitness function computed by the collecting coverage metrics), the best programs are improved, and *µGP* aims to increase the covered code while reaching more states on the PUV hardware. *µGP* is formally described in [7]. In the current experiments, *µGP* uses the evolutionary standard setup provided by the tool.

The framework loads the programs directly into the simulated memory. The PUV and *Spike* execute the individuals simultaneously, and the execution states are compared to find discrepancies in the ISA interpretation. Figure 3 shows the state comparison using an execution log. The PUV simulation logs the execution by each clock cycle. This log is compared on valid states with the execution in the *Spike* simulator. We compare the execution state in the program counter and the register states from all the pipeline stages.

We extracted coverage metrics from the execution of the test program in the PUV using commercial tools. In this case, we employed the NCSim simulator and the incisive comprehensive coverage (ICCR), the latter a code coverage analysis tool, as described in Fig. 4. Metrics coverage data is related to the percentage of executable statements during the simulation, the percentage of the Boolean expression evaluated, among others. Finally, the extracted coverage is used to compute the fitness value that is fed back to *µGP*, guiding the test program generation until one of the stop conditions triggers.

B. Using RISC-V formal

RISC-V formal is a non-invasive processor-independent formal verification description of RISC-V based processors [4]. It consists of a processor-independent formal description of the RISC-V ISA and the specification for the *RISC-V formal* interface (RVFI) among additional features. This interface transmits the execution status of the current instruction when the calculation is ready for all the internal registers. The current version of the interface allows us to verify against general-purpose registers, the program counter (PC), fetched

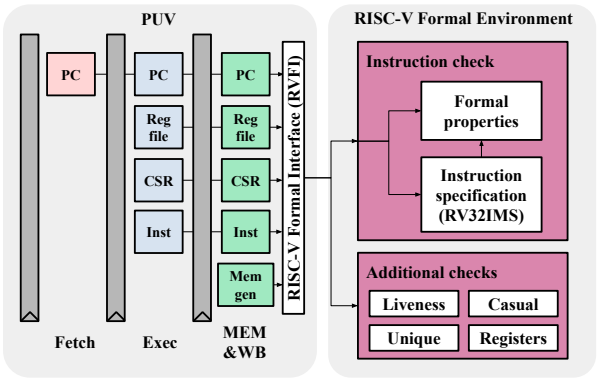

Fig. 5. Simplified block diagram of an interconnection using a *RISC-V formal* interface.

instructions, a generalized memory interface, control and status registers (CSR), and some internal processor flags.

Fig. 5 exhibits a simplified diagram of the interconnection using the *RISC-V formal* interface (RFVI) to verify a processor formally. The RVFI must carry the final state of execution of any instruction to a formal environment. In the processor, we took all the necessary signals from any stage of the processor architecture to the write-back stage (MEM & WB). In this way, the formal properties inside the *RISC-V formal* environment compare the instructions specifications against the internal finish-states of the processor.

The processor formal verification scheme performs tests over the CSRs, but there are no tests available for the behavior of a RISC-V processor based on external interruptions. The RISC-V specification has three kinds of interruptions: external, software and timer. Depending on the processor mode, different flags should be triggered, and the processor should jump to the exception vector in the CSR. We observed that there is no precise specification about the interruption flag sequence to change the internal processor flags and the PC. Moreover, the RISC-V privileged ISA specification does not specify the execution status of the processor when an external interruption occurs.

III. Interruption Specification Absence

The RISC-V interruption specification does not give details of how an interruption must be performed [8]. This description absence is related to the sequence in which a processor performs an interruption. That absence opens up different ways to implement an interruption depending on the core designer. Fig. 6 presents a waveform that compares the interruption procedure in the PUV and *Spike*. Both execute the main program (in blue) until the interruption is triggered. Once the interruption triggers the PUV, its control unit immediately stores the actual PC into the CSR and flushes the execution and write-back stages corresponding to the flushed PC. After storing the PC, the processor sets the PC to the interruption handler code (in red). Later, the last instruction of the trap-handler program reaches the write-back stage, and the PC

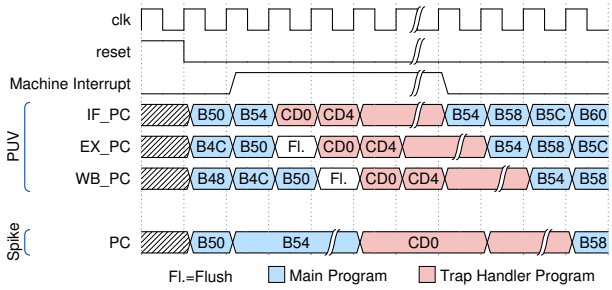

Fig. 6. PUV and *Spike* waveforms describing the internal state interaction on the interruption trigger.

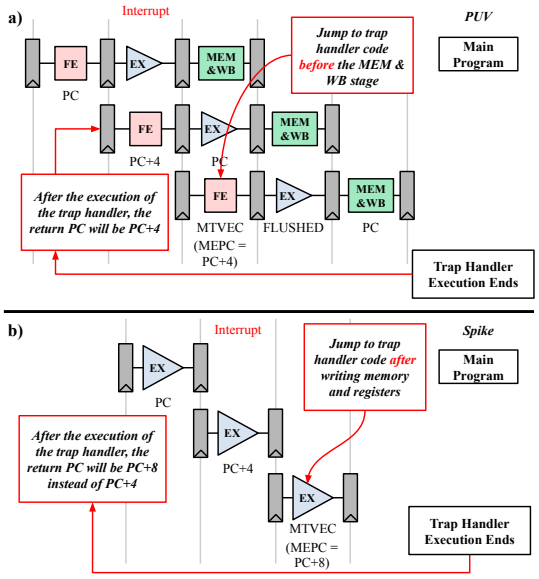

Fig. 7. Different interpretations of the RISC-V ISA interruption procedure.

returns to the stored value in the CSR, as depicted in Fig. 7(a).

At the lower edge of Fig. 6, the *Spike* approach of the interruption sequence is presented. *Spike* waits until the previous instruction (in blue) to the interruption is finished when the interruption occurs and stores the next PC into the CSR. Then, the PC jumps to the interruption handler program (in red). After the trap handler program execution ends, the PC returns to the stored value in the CSR, as shown in Fig. 7(b).

In some instances, the *Spike* approach in a real processor implementation may cause a delay in trap handler program execution. For example, in multi-cycle operations such as multiplication, the trap handler program waits to be executed until the multiplication ends. This strategy is more efficient in the sense that it does not need to flush the pipe. However, it could affect critical applications requiring a fast response to an interruption. The PUV approach reduces the execution latency of an interruption trap handler program. Furthermore, it avoids inconsistencies with instructions that may change the internal status of the memory or internal registers (such as CSRs and system bus requests).

TABLE I
ERROR DETECTION IN THE VERIFICATION MODELS.

Inserted error	μGP-Spike (450 Indiv.)	RISC-V formal (RV32IM)
ALU misbehaviour	256 Detected	Detected
Wrong comparison	1 Detected	Detected
Data hazard	1 Detected	Not detected
Interruption execution	306 Detected	Not detected

TABLE II
IMPLEMENTATION COSTS BETWEEN μGP AND RISC-V FORMAL

	μGP-Spike	RISC-V formal
Processor output	Output simulation log.	RVFI RTL port.
Testbench	RAM capability to insert external programs.	Provided. Connect RVFI.
Scripts	Adaptation needed. Build assembly and format to simulated memory.	Provided. Enable or disable flags for verification.
Exec. Time Intel-i7 9750	∼90m for 450ind. 1-core	∼45m for RV32IM 8-cores

IV. THE BEST OF BOTH DOMAINS

We combine the process of verification of the implemented RISC-V processor using the Spike and *μGP* simulation comparison, and the description of the RISC-V specification using formal verification with Yosys [9]. We compared the internal states of the processor with the verification environment. Each one of the verification covers most of the cases according to the processor architecture. To reach the most cases possible, *μGP* maximizes the coverage metrics, and Yosys evaluates the assumptions and assertions in each simulation step using boolean satisfiability (SAT).

We introduced some flaws inside the datapath of the PUV to verify the correct operation of the frameworks. Table I presents the detection over the *μGP* and *RISC-V formal* of different inserted errors inside the processor. The ALU misbehavior was tested by changing one bit to the final result, affecting the multiplexer order, and adding a segmentation stage. This misbehavior is detected in both frameworks– more than half of the *μGP* tests. The faulty comparison introduces a misinterpretation of the signed comparison by using the 30th bit as the sign instead of the 31st bit. RISC-V torture does not detect this flaw, and the detection probability on the simulation-based framework is minimal, making formal verification suitable to find these kinds of errors. The data hazard removes detection when writing a register to be used in the next instruction. Software tortures are more suitable for data hazard detection. Finally, it is relevant to highlight that the interruption execution issue described in section III was detected by the proposed simulation-based approach.

Table II provides an implementation cost for *μGP* and *RISC-V formal*. *μGP* needs a simulation environment for the processor to be implemented, which uses a log to display the current status of the processor. *RISC-V formal* has a wrapper environment where the RVFI is required. *μGP* requires more software binding due to the need to push programs directly in RAM, and only a per-cycle hardware logging. *RISC-V formal* needs more hardware binding to connect and transform all signals to the RVFI. An additional setup is needed for the *μGP* in contrast with the *RISC-V formal*. The execution time is configurable for *μGP*, lasting 90 minutes for 450 individuals. The runtime for the 51 tests for the formal approach always run for 45 minutes using multi-threads.

V. CONCLUSIONS

We presented a verification framework combining two different verification approaches. The first one excites PUV and a golden RISC-V ISA simulator with a set of automatically generated tortures. Using a genetic algorithm called *μGP*, *μGP* drives the generation of testing programs following an maximization process over the coverage metrics, then a comparison is performed using the internal states of the processor with a execution model. The latter, *RISC-V formal*, defines a set of formal properties that follow the RISC-V ISA specification, through using open source formal verification tools, the behavior for each instruction is verified on the PUV against the set of formal properties.

If the verification implementation is accurate according to the specification, these two approaches must thoroughly verify the instruction set of RISC-V based processors. Unfortunately, most of the verification errors may be caused by insufficient coverage of all the cases. By using Spike (the golden model) chose to compare in the framework, exposed in Fig. 2 detects at most one out of 450 executed tests for comparator flaws and data hazards. The formal properties specified in the RTL help to detect certain behavioral flaws according to the ISA specification. Thanks to the combination of the verification schemes, we detected an absence in the processor interruption state change, which the RISC-V ISA does not specify.

REFERENCES

[1] P. D. Schiavone *et al.*, "An Open-Source Verification Framework for Open-Source Cores: A RISC-V Case Study," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2018, pp. 43–48.

[2] V. Herdt *et al.*, "Verifying Instruction Set Simulators using Coverage-guided Fuzzing*," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 360–365.

[3] F. Corno, E. Sanchez, and G. Squillero, "Evolving Assembly Programs: How Games Help Microprocessor Validation," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 695–706, Dec 2005.

[4] Symbiotic EDA, "RISC-V Formal Verification Framework," https://github.com/SymbioticEDA/riscv-formal, 2019.

[5] A. Reid *et al.*, "End-to-End Verification of Arm Processors with Isa-Formal," in *Proceedings of the 2016 International Conference on Computer Aided Verification (CAV'16)*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9780, no. 9780. Springer Verlag, July 2016, pp. 42–58.

[6] RISC-V Foundation, "Spike RISC-V ISA Simulator," https://github.com/riscv/riscv-isa-sim, 2019.

[7] E. Sanchez, M. Schillaci, and G. Squillero, *Evolutionary Optimization: the μGP toolkit*. Springer Science & Business Media, 2011.

[8] A. Waterman *et al.*, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 20190608-Priv-MSU-Ratified," EECS Department, University of California, Berkeley, Tech. Rep., Jun 2019.

[9] C. Wolf, "Yosys Open SYnthesis Suite," http://www.clifford.at/yosys/.