# Large-Scale RISC-V Processor Verification Using Automated Design Inspection and a Generic Simulation Method*

**Gabriel Gomes**[1], **Julio Avelar**[1], **Gabriel Oliveira**[1], **Enzo Bertoloti**[1], **Rodolfo Azevedo**[1]

[1]Instituto de Computação - Universidade Estadual de Campinas - Campinas - SP - Brasil
{g212736, j241163, g247700, e248361}@dac.unicamp.br, rodolfo@ic.unicamp.br

***Abstract.*** *In recent years, RISC-V ISA has gained popularity and research on RISC-V processor verification has increased. However, most studies focus on a small number of cores and rely on implementation-dependent characteristics. To address this limitation, we propose a generic verification framework that monitors high-level processor states through the fetch interface and the register file write port. The execution traces are compared against a reference model. Our framework leverages a Large Language Model to collect the necessary files and signals of new processors, allowing manual code reductions of 85%. We evaluated the framework by testing 21 LLMs in the inspection task and simulating a custom benchmark on 21 processors, identifying 16 bugs across 8 different cores.*

## 1. Introduction

Before the 2000s, processor design was predominantly controlled by a handful of companies and their Instruction Set Architectures (ISAs). This scenario significantly influenced processor development, including the verification flow. Because access to ISAs was restricted, verification methodologies became specialized to the few implementations that were under development. For example, the development of the Power9 processor by IBM [Schubert et al. 2018] required verification tools focused on the unique components and interconnections of the processor.

The creation and popularization of RISC-V has allowed the release of a wide range of processors designed according to the same ISA. The common architecture has facilitated the creation of reusable verification tools, but ensuring compatibility across numerous microarchitectures remains a significant challenge. In response to this context, verification efforts of RISC-V have increased [Yosys 2025, Synopsys 2025], but most tools are still specialized for their target processors [Kabylkas et al. 2021, Joannou et al. 2024, Herdt et al. 2020]. The abundance of distinct implementations also makes it difficult to select an appropriate processor from the many available options.

Another challenge for large-scale verification is reducing the time to set up the verification environment, which can be achieved through automation. Large Language Models (LLMs) have shown promising results in the analysis of source code and the generation of Hardware Description Language (HDL) code [Thakur et al. 2023], which can assist with repetitive tasks such as module instantiations.

To address these challenges, we propose an agile and general solution based on two key components. The first component leverages an LLM to automatically identify

*This work was supported by the Coordination for the Improvement of Higher Education Personnel (CAPES) – Finance Code 001.

the relevant source files for the simulation and extract the necessary signals to the verification infrastructure, thereby reducing the amount of manually written code. The second component uses simulation trace comparison, in which the PC flow and the register file commits of the processor under test and a reference model are compared for each program in a custom benchmark. This method is highly reusable, as it requires no modifications to the processor code.

## 2. Related Work

### 2.1. Reference Model Comparison

Reference model comparison is a verification technique in which the processor under test is compared to a reference –or golden –model that is typically a high-level language description of the ISA. To apply the method, both implementations run the same test program and generate execution traces containing information regarding each instruction. If the traces differ, the cause must be investigated as it may indicate a bug.

Many studies leverage reference model comparison for RISC-V processor verification, but most approaches target a single core or a small set of cores. They usually require HDL files modifications that are tightly coupled to microarchitectural details, which makes it difficult to reuse or generate the code, and ultimately hinders automation.

Dromajo is an Instruction Set Simulator (ISS) introduced in [Kabylkas et al. 2021]. It is used as the golden model in a co-simulation setup to verify three RISC-V processors, relying on Direct Programming Interface (DPI) calls to synchronize the implementations. The authors describe the instruction completion detection mechanism used for one of the cores: the processor code is modified to monitor the reorder buffer and to invoke the ISS at each commit.

An additional five processors were verified in [Joannou et al. 2024]. The proposed simulation setup compares the RTL implementation against the golden model using an extended version of the RISC-V Formal Interface (RVFI) [Yosys 2025], known as RVFI-Direct Instruction Injection (RVFI-DII). DII is a technique in which instructions are injected directly into the processor's fetch interface, enabling the simulation to execute specific instructions rather than the entire program, which makes debugging more efficient. This technique requires highly processor-specific code. For instance, Flute, an in-order scalar core, used an instruction ID attached to the program counter which was propagated through each stage of the pipeline.

Table 1 summarizes different RISC-V processor verification works, highlighting the base technique used and the number of cores verified. It can be seen that it is a common practice to verify only one or a few processors. An exception is the Fabscalar [Choudhary et al. 2011] tool, that verifies 12 cores. However, these cores are merely variations automatically generated by the same tool and share similar, well-defined structures.

### 2.2. Verification Tool Generalization

The RISC-V Formal Framework [Yosys 2025] is an open-source tool comprising a formal specification of the RISC-V ISA and formal testbenches for supported processors. It defines a generic interface, the RVFI, which can be used not only in formal methods but also in alternative ones. In particular, some designers have adopted the RVFI as a standard trace format for verifying their cores via co-simulation [Joannou et al. 2024].

**Table 1. RISC-V Processor Verification Work with number of verified processors.**

| Work | Base Method | Processors |
|---|---|---|
| [Kabylkas et al. 2021] | Ref. Model Comp. | 3 |
| [Joannou et al. 2024] | Ref. Model Comp. | 5 |
| [Herdt et al. 2020] | Ref. Model Comp. | 1 |
| [Wang et al. 2020] | RTL Simulation | 1 |
| [Choudhary et al. 2011] | Ref. Model Comp. | 12* |
| [Orenes-Vera et al. 2021] | Formal Verification | 1 |
| [Weingarten et al. 2024] | Formal Verification | 1 |
| [Bruns et al. 2023] | Ref. Model Comp. | 1 |
| [Cui et al. 2023] | RTL Simulation | 1 |
| [Rokicki et al. 2019] | Ref. Model Comp. | 1 |
| [Jiang et al. 2024] | Ref. Model Comp. | 1 |
| [Bruns et al. 2022] | Ref. Model Comp. | 1 |
| [Xu et al. 2022] | Ref. Model Comp. | 2 |

*Automatically generated processors
Formal Verification: proves design correctness using mathematical principles
RTL-level Simulation: compares signals and low-level details

Synopsys ImperasDV [Synopsys 2025] is a commercial verification suite for RISC-V processors. It provides a variety of tools, including reference model comparison and coverage analysis. Similar to the RISC-V Formal Framework, ImperasDV defines a generic interface called the RISC-V Verification Interface (RVVI).

Even though these two tools represent significant advances toward generalization, their verification interfaces require deep changes to the processor, posing a challenge to achieving rapid, large-scale applicability.

Finally, the RISC-V Certification Steering Committee [RISC-V Foundation 2025] deserves mention. It is currently evaluating the most suitable test suite to ensure compliance with the RISC-V ISA. Although the tests are intended for large-scale use, they rely on directed testing, which is generally less effective than reference model comparison verification methods.

## 2.3. LLM Automated RTL Code Generation

LLMs have demonstrated strong capabilities in natural language processing as well as in generating high-level programming code [Nijkamp et al. 2023]. More recently, their potential for generating RTL code has also begun to be explored.

Thakur et al. [Thakur et al. 2023] aimed to improve the generation of Verilog code by LLM by fine-tuning an existing model with a new dataset composed of Verilog code collected from GitHub and textbooks. The fine-tuned model was then evaluated in a range of tasks, including: (1) describing the module's functionality, and (2) solving Verilog coding problems such as designing sequential logic circuits. The fine-tuned model outperformed its pre-trained counterpart, demonstrating that LLMs can improve their RTL generation capabilities and be effectively used in hardware design.

RTLCoder is an open source LLM introduced by Liu et al [Liu et al. 2025] that automatically generates RTL code. It is presented as a promising tool to support agile hardware development. RTLCoder can be applied to a variety of tasks, including the creation of security assertions, bug fixing, and the generation of repetitive code structures such as arithmetic units and decoders.

After presenting the current scenario, it is evident that robust models for automated RTL code generation are rapidly emerging and becoming valuable tools in computer architecture development.

## 3. System Architecture

### 3.1. Overview

Our verification methodology is presented in Figure 1. An RTL wrapper that translates the processor's interface to the standard Wishbone [OpenCores 2010] is used to connect the design under test to the Generic Simulation Testbench. In some cases, a protocol adapter is instantiated to carry out this translation. The testbench is implemented using Cocotb [The FOSSi Foundation 2025], which uses the configuration files automatically generated by the LLM. The simulation then receives an arbitrary ELF file, which is executed by the processor to generate a trace. Finally, the trace is compared with that of the reference model (Spike[1] simulator). Figure 2 summarizes the verification steps.

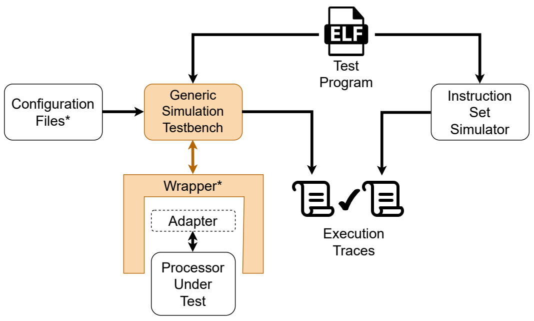


Figure 1. Diagram of the proposed tool. The wrapper* and the configuration files* were automatically created. The adapter module is optional.

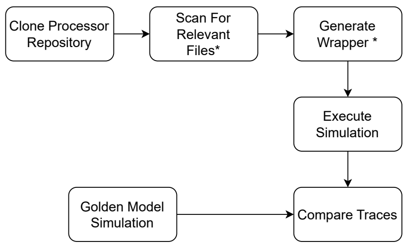


Figure 2. Proposed verification framework flow for one processor. Steps marked with * were automatically executed with the use of an LLM.

### 3.2. Automated Design Inspection and Wrapper Generation

The wrapper generation process has five steps that are either executed by scripts, by LLM, or manually.

1. Since the LLM's context window is not large enough to handle all the source code, a script preprocesses the files and creates a list containing the modules instantiated by each file.
2. The LLM receives a prompt with the created list and determines the top module along with the necessary files for a core-only system.
3. The LLM instantiates the top module in the wrapper's template.

[1]https://github.com/riscv-software-src/riscv-isa-sim

4. The processor's interface is automatically identified by the LLM to determine whether an adapter is needed. If so, the model either instantiates one of the standard adapters, which were handcrafted, or notifies the user to manually insert a custom adapter module into the template.
5. Finally, the user completes the wrapper by connecting the modules through their ports.

In summary, the wrapper consists of an HDL file containing the external interface, along with the instantiations of the processor and the optional adapter. The list of files required for a core-only system is used to generate the configuration files.

## 3.3. Simulation Setup

The generic simulation testbench was developed using the Cocotb Python verification framework [The FOSSi Foundation 2025]. In Cocotb, the design under test is instantiated as a Python object, allowing the user to control the simulation and integrate high-level models that interact with the design.

The testbench consists of an instance of the processor under test, a memory model that supplies instructions and data, a monitoring probe, and an execution trace generator. During execution, two interfaces are monitored: the instruction memory interface and the register file interface.

When an instruction-memory request occurs, the processor is executing a fetch. The requested address corresponds to the next program counter (PC) value, and the returned data correspond to the next instruction. Since the instruction memory interfaces were very similar across processors, the Wishbone interface could be used with little modifications.

The register file interface was highly consistent across processors, with all verified designs including signals equivalent to (1) write-enable, (2) write address, and (3) write data. When the write enable signal is asserted, the other two signals can be used to identify the current commit. To monitor the register file, the user must indicate the corresponding signals to the framework, which incorporates them into a small core-specific section of the testbench.

## 3.4. Challenges and Implementation Strategy

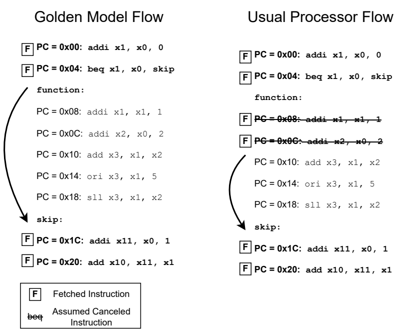
**Relationship between trace events**: Each commit of a register file must be associated with its corresponding instruction to generate a correct execution trace. Related work usually achieves this by monitoring the instruction execution along the pipeline or by checking the multi-cycle processor's state machine. Since the verification method must be processor-agnostic, it cannot rely on such microarchitectural details.

The adopted strategy to address this challenge was to monitor instruction fetches and register file commits separately, and subsequently merging them into a final execution trace. Two trace fragments are constructed as ordered lists: one containing all instruction fetches (program counter and instruction) and the other containing all register file commits (target register and written value). The lists are combined according to a general behavior: both sequences maintain the same order. Thus, the first instruction that writes to the register file (arithmetic instructions write to the register file but branches do not) is associated with the first entry in the commit list, and so on.

**PC-flow monitoring**: In most verified processor simulations, we observed flushed instructions (fetched but not fully executed). When jumps and branches are fetched, subsequent instructions at the next sequential PC addresses are speculatively fetched, similar to the behavior of a static "not-taken" branch predictor.

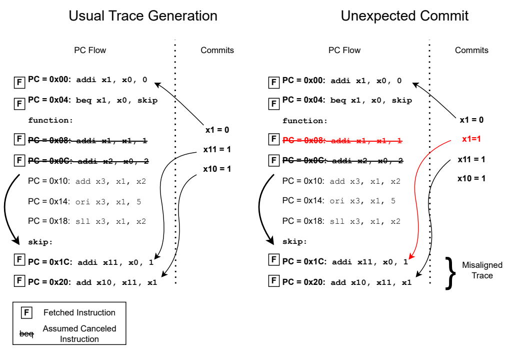
The chosen approach to address this challenge involves filtering the control-flow fragment of the processor's execution trace. If the core fetches an instruction that does not follow the reference flow, it is discarded from the trace under the assumption that it was a speculative fetch. After all assumed speculative (canceled) instructions are removed, the filtered processor flow must match the reference flow. Figure 3 illustrates an example of canceled instructions being removed from the final control flow.

Nevertheless, it is still necessary to verify whether the assumed canceled instructions were indeed canceled. This challenge is addressed during the merging of the fetch and register file commit traces, as explained next.



Golden Model Flow
F PC = 0x00: addi x1, x0, 0
F PC = 0x04: beq x1, x0, skip
function:
PC = 0x08: addi x1, x1, 1
PC = 0x0C: addi x2, x0, 2
PC = 0x10: add x3, x1, x2
PC = 0x14: ori x3, x1, 5
PC = 0x18: sll x3, x1, x2
skip:
F PC = 0x1C: addi x11, x0, 1
F PC = 0x20: add x10, x11, x1
Usual Processor Flow
F PC = 0x00: addi x1, x0, 0
F PC = 0x04: beq x1, x0, skip
function:
F ~~PC = 0x08: addi x1, x1, 1~~
F ~~PC = 0x0C: addi x2, x0, 2~~
PC = 0x10: add x3, x1, x2
PC = 0x14: ori x3, x1, 5
PC = 0x18: sll x3, x1, x2
skip:
F PC = 0x1C: addi x11, x0, 1
F PC = 0x20: add x10, x11, x1
F Fetched Instruction
~~beq~~ Assumed Canceled Instruction

**Figure 3. Control flow monitoring. Some instructions are assumed to be speculative fetches and are discarded from the execution trace.**



Usual Trace Generation
PC Flow
Commits
F PC = 0x00: addi x1, x0, 0
F PC = 0x04: beq x1, x0, skip
function:
F ~~PC = 0x08: addi x1, x1, 1~~
F ~~PC = 0x0C: addi x2, x0, 2~~
PC = 0x10: add x3, x1, x2
PC = 0x14: ori x3, x1, 5
PC = 0x18: sll x3, x1, x2
skip:
F PC = 0x1C: addi x11, x0, 1
F PC = 0x20: add x10, x11, x1
x1 = 0
x11 = 1
x10 = 1
F Fetched Instruction
~~beq~~ Assumed Canceled Instruction
Unexpected Commit
PC Flow
Commits
F PC = 0x00: addi x1, x0, 0
F PC = 0x04: beq x1, x0, skip
function:
F ~~PC = 0x08: addi x1, x1, 1~~
F ~~PC = 0x0C: addi x2, x0, 2~~
PC = 0x10: add x3, x1, x2
PC = 0x14: ori x3, x1, 5
PC = 0x18: sll x3, x1, x2
skip:
F PC = 0x1C: addi x11, x0, 1
F PC = 0x20: add x10, x11, x1
x1 = 0
x1=1
x11 = 1
x10 = 1
Misaligned Trace

**Figure 4. Merging of fetches and commits fragments. If an unexpected commit (in red), it is associated with the wrong instruction and creates a failing trace entry that can be detected.**

In Figure 4, two examples of the trace generation process are shown. On the left, the normal case is illustrated, where the commits are correctly associated with their respective instructions: `x1 = 0` is associated with the `addi` at `PC=0x00`, `x11 = 1` with the `addi` at `PC=0x1C`, and so on. On the right, an unexpected commit `x1 = 1` is shown, made by the `addi x1, x1, 1` instruction (in red). Since the trace generator assumed that this instruction was canceled, the unexpected commit is incorrectly associated with the next valid instruction, causing all subsequent associations to be incorrect as well, resulting in a misaligned trace. This behavior will trigger a mismatch in the trace comparison, which can be used to detect the bug.

**Memory write monitoring**: We decided not to include memory write operations in the execution trace. This exclusion is due to the difficulty of correctly monitoring byte and halfword store operations across all processors using the memory interface signals. Most processors feature a write strobe or byte-enable signal that indicates which bytes are being written to memory, signals that are typical of cache interface. However, some processors (refer to Table 2) instead implement stores using read-modify-write (RMW), common of memory interfaces. So far, our infrastructure does not detect those distinct patterns, but we found that errors in these instructions are later detected by a subsequent load from the same address. The test benchmarks include programs that account for this behavior, with load instructions immediately following write operations.

## 4. Evaluation

### 4.1. Automated Design Inspection and Wrapper Generation

We executed an experiment to identify the most suitable LLM for automation tasks described in section 3.2. Nine different model families were included: Qwen [1], Gemma [2], StarCoder [3], DeepSeek [4], Phi [5] , Codestral [6], Llama [7], Llava [8], and Granite [9]. Each family comprises subsets of different sizes and other variants (e.g. general, coder), totaling 21 models. The 21 processor projects listed in Table 2 were used as input for the LLMs, as well as ten other open-source cores available on GitHub and GitLab.

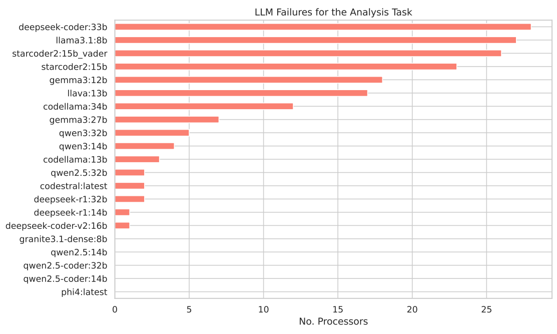


**Figure 5. Number of processors for which the LLMs failed in the source files and top module identification task.**

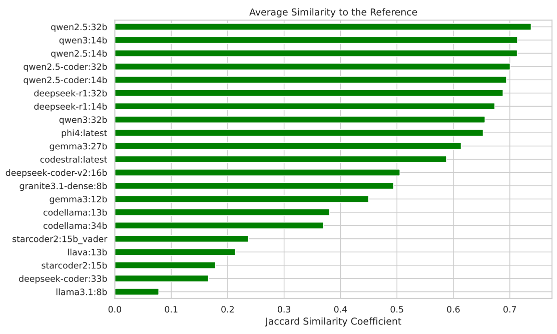


**Figure 6. Similarity between LLM and manual solutions for the source files and top module identification task.**

Figure 5 presents the experimental results for the LLMs tested. If an LLM fails to complete any of the tasks described in section 3.2, the entire flow for the corresponding processor is considered a failure. The behaviors that lead to a failure were (1) the LLM providing a script to solve the problem instead of the actual solution or (2) the LLM generating the output with incorrect formatting, making it unparseable. The high failure rate of some models may be explained by the fact that hardware description languages (HDLs) were underrepresented in their training data. For example, in the development of one of the worst-performing LLMs, StarCoder2 [Lozhkov et al. 2024], no HDLs are mentioned.

[1]https://github.com/QwenLM/Qwen
[2]https://ai.google.dev/gemma
[3]https://github.com/bigcode-project/starcoder
[4]https://github.com/deepseek-ai/DeepSeek-Coder
[5]https://azure.microsoft.com/en-us/products/phi
[6]https://mistral.ai/news/introducing-codestral/
[7]https://arxiv.org/abs/2302.13971
[8]https://arxiv.org/abs/2304.08485
[9]https://arxiv.org/abs/2304.08485

We then evaluated the quality of the generated results by comparing the LLM outputs with manually created solutions for step 2 in section 3.2, which involves the creation of configuration files. These configuration files consist on makefiles with a list of the necessary source files for the simulation, the top module, and some configuration flags for the simulator. To perform the comparison, we populated Python sets with the names of the source files along with the top module name. The Jaccard coefficient was then calculated using the LLM and manual sets, resulting in the similarity metric.

Figure 6 shows that approximately half of the LLM achieved an average similarity greater than 60%. It can be seen that the model family had a significant influence on the results. Also, that the size of the model (measured in billions of parameters, such as *32b* and *14b*), on the other hand, showed less influence on the similarity. The presented automation significantly improved productivity, as the qwen2.5 model was used to generate 70% correct configuration files and reduce manual effort.

To evaluate the reduction in manual effort by steps 3 and 4 in section 3.2, we measured the proportion of structural elements generated automatically by our framework. We counted syntactic elements extracted from the Verilog Abstract Syntax Tree (AST) using the PyVerilog parser [Takamaeda-Yamazaki 2015]. Each module instantiation, port declaration, signal declaration, and port connection was treated as an individual element. Our automatically generated code provides module instantiations and port declarations, while the user is responsible for completing the signal connections. By comparing the number of elements generated by the framework with the total number of elements in the final design, we obtained the automation coverage. After applying the methodology for 19 of the 21 processors in Table 2 (RPU and Leaf are written in VHDL and could not be analyzed with PyVerilog), an average of 85% coverage was obtained.

One last metric must be analyzed in this section, which is the total effort to setup the testbench. The lines of code required to apply the verification method, including configuration files and source files, were compared. The CVA6 core was verified in [Kabylkas et al. 2021] and required 280 lines of code. The Ibex core required more than 450 lines of code when verified using the RVFI-DII proposed in [Joannou et al. 2024]. The solution proposed in this work required 270 lines of code to verify Hazard3, a similar-sized core.

Even though the line counts have around the same magnitude, the code reusability differs significantly. Both cited studies depend on modifying structures that are unique to the cores: Ibex had its pipeline stages modified in order to implement the RVFI-DII interface and CVA6 had its reorder buffer modified. Our solution, on the other hand, deals only with interfaces that are typically similar or standardized. Another important aspect is that our testbench is potentially easier to generate using LLMs. It only requires interface detection and module instantiations, while related work depends on more advanced code structures and more context such as interactions between pipeline stages.

### 4.2. Generic Simulation Method

The applicability of the simulation method was evaluated by applying it to a diverse set of processors. Table 2 lists all the simulated cores together with details of their interfaces and the number of bugs identified after running a set of benchmark tests.

The *Structure* and *Byte/Half Writes* columns of Table 2 describe the processor

**Table 2. List of simulated processors and their relevant characteristics**

| Processor | Structure | Extensions | Byte/Half Writes | Required Adapter | Bugs Found |
|---|---|---|---|---|---|
| RVX[1] | Multi-cycle | RV32I | Write Strobe | Yes | 0 |
| Hazard3[2] | 3-stage pipeline | RV32IMAC_Zicsr+ | Write Strobe | Yes | 0 |
| Riscado-V[3] | Multi-cycle | RV32I | Read modify write | No | 1 |
| Tinyriscv[4] | 3-stage pipeline | RV32IM | Read Modify Write | No | 0 |
| Grande Risco 5[5] | 5-stage pipeline | RV32IMBC_Zicsr | Read Modify Write | No | 1 |
| Riskow[6] | Multi-cycle | RV32I | Read Modify Write | No | 0 |
| Risco 5[7] | Multi-cycle | RV32IM | Read Modify Write | No | 2 |
| Kronos[8] | 3-stage pipeline | RV32I_Zicsr_Zifencei | Write Strobe | Yes | 0 |
| PicoRV32[9] | Multi-cycle | RV32IMC/E | Write Strobe | No | 0 |
| AUK-V-Aethia[10] | 5-stage pipeline | RV32I | Write Strobe | No | 5 |
| Fedar F1[11] | 5-stage pipeline | RV64IM | Not Implemented | No | 2 |
| Hornet[12] | 5-stage pipeline | RV32IM | Write Strobe | No | 1 |
| Nerv[13] | Multi-cycle | RV32I | Write Strobe | No | 0 |
| Baby Risco 5[14] | Multi-cycle | RV32E | Read Modify Write | No | 3 |
| Zero-Riscy[15] | 2-cycle pipeline | RV32IMC | Write Strobe | No | 0 |
| mriscv[16] | Multi-cycle | RV32I | Write Strobe | No | 0 |
| SprintRV[17] | 5-stage pipeline | RV32IM_Zicsr | Write Strobe | No | 0 |
| Leaf[18] | 2-stage pipeline | RV32I | Write Strobe | No | 0 |
| RPU[19] | Multi-cycle | RV32IM_Zicsr | Write Strobe | Yes | 1 |
| RS5[20] | 4-stage pipeline | RV32IMACV+ | Write Strobe | Yes | 0 |
| SuperScalar-CPU[21] | 5-stage 3-issue OoO | RV32IMC | Write Strobe | Yes | 0 |

[1]https://github.com/rafaelcalcada/rvx
[2]https://github.com/Wren6991/Hazard3
[3]https://github.com/zxmarcos/riscado-v
[4]https://github.com/liangkangnan/tinyriscv
[5]https://github.com/JN513/Grande-Risco-5
[6]https://github.com/racerxdl/riskow
[7]https://github.com/JN513/Risco-5
[8]https://github.com/SonalPinto/kronos
[9]https://github.com/YosysHQ/picorv32
[10]https://github.com/veeYceeY/AUK-V-Aethia
[11]https://github.com/eminfedar/fedar-f1-rv64im
[12]https://github.com/yavuz650/RISC-V
[13]https://github.com/YosysHQ/nerv
[14]https://github.com/JN513/Baby-Risco-5
[15]https://github.com/tom01h/zero-riscy
[16]https://github.com/onchipuis/mriscv
[17]https://github.com/CastoHu/SprintRV
[18]https://github.com/daniel-santos-7/leaf
[19]https://github.com/Domipheus/RPU
[20]https://github.com/gaph-pucrs/RS5
[21]https://github.com/risclite/SuperScalar-RISCV-CPU

characteristics that influenced the design of the testbench. The *Required Adapter* column shows that six processors needed an adapter module: Hazard3 used a AHB-to-Wishbone module; RVX, Kronos, and RS5 required a Wishbone-to-Pipelined-Wishbone (introducing a one cycle delay of data signals); RPU and Superscalar-RISC-V-CPU required a custom adapter that translates the byte-enable signals.

The method was also applied to a superscalar core, which required a specific interface for multiple instruction fetches and careful trace comparisons, considering the possibility of multiple register file commits at the same cycle. The fetches, however, still follow the same relative order, as well as the memory stores.

These results indicate that the verification infrastructure is scalable, because (1) most processors have a direct mapping to the wrapper interface, (2) most of the adapters are standard and reusable, (3) and only two processors required a custom adapter. It should also be noted that the configuration files were easily generated, and that identifying the register file interface requires listing only a few signals, typically fewer than five.

## 4.3. Processor Verification via Trace Comparison

The purpose of the generic simulation setup is to allow the verification of multiple processors. The method involves simulating both the processor under test and a golden reference model to generate their respective execution traces, which are then compared. The infrastructure does not rely on a specific test format to execute the ELF file instructions, so it can handle from handwritten to randomly generated tests.

To evaluate the feasibility of the proposed method, a custom benchmark consisting of 40 programs was executed on each processor. The benchmark included manually written tests for each RV32I instruction, along with a few additional corner cases. These cases include programs with stores followed by loads and instruction combinations that trigger forwarding structures. The main bugs found after running the benchmark were the following:

**Load/store half/byte instructions errors**: AUK-V-Aethia, Fedar F1, Risco-5 and RPU exhibited bugs related to halfword and byte memory accesses. AUK-V-Aethia failed to sign-extend loaded halfwords and bytes. Fedar F1 incorrectly loaded the entire memory word at all times, even for halfword and byte load instructions. Risco-5 modified the entire memory word instead of the selected bytes during halfword and byte store operations. RPU would incorrectly load the entire word in the specific case of the unsigned instructions LBU and LHU.

**Wrong first instruction fetch**: The Hornet processor consistently skipped the first instruction of the program. Investigation revealed that the instruction address port was connected to the combinational logic for the next PC value rather than the current PC register. As a result, the processor began execution with the address port pointing to the second instruction of the program.

**Incorrect jump behavior**: Fedar F1, Grande-Risco-5 and Baby-Risco-5 exhibited control flow issues related to jump instructions under certain conditions. When fetching a jump, Fedar F1 requires two cycles to recognize the jump and update the control flow, resulting in two incorrect instructions being fetched and needing to be flushed. However, the processor's flush mechanism did not function correctly, causing these erroneous instructions to be committed. Grande-Risco-5 hangs when executing an infinite loop (e.g., the RISC-V assembly instruction `label:  j label`). Instead of repeatedly fetching the same instruction, the processor ceases instruction fetches entirely. Baby-Risco-5 failed to jump at the JALR instruction and proceeded with the program sequentially.

Table 2 indicates that some processors have more bugs than others. This variation can be attributed to differences in core maturity, influenced by the extent of verification and the designers' team profile. Processors with a higher number of detected bugs were often developed by individual designers and verified primarily through simulation using limited testbenches (for example, only a single test program was found in the Fedar F1 repository). Information regarding core maturity can assist designers in selecting processors, enabling them to choose more reliable cores or to anticipate additional verification effort when opting for less mature designs.

## 5. Conclusion

The popularization of the RISC-V ISA has led to the emergence of numerous processor cores, raising important questions regarding their verification and criteria for selection in specific applications. In this paper, we present a verification framework designed to support multiple processors effectively.

The framework includes an automated design inspection tool that leverages LLMs to analyze the project's source files, identifying those related to the processor. Based on this analysis, an RTL wrapper is generated around the top module to ensure compatibility with the testbench interface. Various bus interfaces such as AHB, Wishbone, and customized ones were connected using the Wrapper. This task was executed by 21 different LLMs and their outputs were compared to a manually created reference in terms of successfully generated outputs and output quality. The automation coverage in the final wrapper files was also evaluated, revealing an average of 85%.

Following this, the framework employs a generic simulation-based verification method that monitors the processor's instruction fetch and register file interfaces to produce an execution trace. This trace is then compared with that generated by the Spike golden model. Simulating a processor requires no modifications to its source code and minimal wrapper code. This approach successfully simulated 21 processors that exhibit diverse structures and interface behaviors, primarily consisting of embedded cores.

A benchmark comprising 40 manually written test programs was executed on each processor and their execution traces were compared. Using this method, 16 bugs were identified in eight different processors. The number of detected bugs can serve as an indicator of the maturity of a design and may assist designers in selecting appropriate processors.

## References

Bruns et al. (2022). Cross-level processor verification via endless randomized instruction stream generation with coverage-guided aging. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1123–1126. IEEE.

Bruns et al. (2023). Processor verification using symbolic execution: A risc-v case-study. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6.

Choudhary, N. K. et al. (2011). Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 11–22.

Cui et al. (2023). A hardware-software cooperative interval-replaying for fpga-based architecture evaluation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–2. IEEE.

Herdt, V., Große, D., Jentzsch, E., and Drechsler, R. (2020). Efficient cross-level testing for processor verification: A risc- v case-study. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–7.

Jiang, Z., Zheng, K., Bao, Y., and Shi, K. (2024). Efficient verification framework for risc-v instruction extensions with fpga acceleration. In *2024 2nd International Symposium of Electronics Design Automation (ISEDA)*, pages 345–350. IEEE.

Joannou, A. et al. (2024). Randomized testing of risc-v cpus using direct instruction injection. *IEEE Design & Test*, 41(1):40–49.

Kabylkas, N., Thorn, T., Srinath, S., Xekalakis, P., and Renau, J. (2021). Effective processor verification with logic fuzzer enhanced co-simulation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 667–678.

Liu, S. et al. (2025). Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(4):1448–1461.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., et al. (2024). Starcoder 2 and the stack v2: The next generation.

Nijkamp, E. et al. (2023). Codegen: An open large language model for code with multi-turn program synthesis.

OpenCores (2010). Wishbone system-on-chip (soc)interconnection architecture for portable ip cores. https://cdn.opencores.org/downloads/wbspec_b4.pdf. [Online; accessed 26-August-2025].

Orenes-Vera, M., Manocha, A., Wentzlaff, D., and Martonosi, M. (2021). Autosva: democratizing formal verification of rtl module interactions. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 535–540. IEEE.

RISC-V Foundation (2025). RISC-V Certification Steering Committee. https://riscv.atlassian.net/wiki/spaces/CSC/overview. [Online; accessed 26-August-2025].

Rokicki, S., Pala, D., Paturel, J., and Sentieys, O. (2019). What you simulate is what you synthesize: Designing a processor core from c++ specifications. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8.

Schubert, K.-D. et al. (2018). Addressing verification challenges of heterogeneous systems based on ibm power9. *IBM Journal of Research and Development*, 62(4/5):11:1–11:12.

Synopsys (2025). ImperasDV RISC-V Processor Verification Solution. https://www.synopsys.com/verification/imperasdv.html. [Online; accessed 26-August-2025].

Takamaeda-Yamazaki, S. (2015). Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In Sano, K., Soudris, D., Hübner, M., and Diniz, P. C., editors, *Applied Reconfigurable Computing*, pages 451–460, Cham. Springer International Publishing.

Thakur, S. et al. (2023). Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6.

The FOSSi Foundation (2025). Cocotb python verification framework. https://www.cocotb.org/. [Online; accessed 26-August-2025].

Wang, J., Tan, N., Zhou, Y., Li, T., and Xia, J. (2020). A uvm verification platform for risc-v soc from module to system level. In *2020 IEEE 5th International Conference on Integrated Circuits and Microsystems (ICICM)*, pages 242–246. IEEE.

Weingarten, L., Datta, K., Kole, A., and Drechsler, R. (2024). Complete and efficient verification for a risc-v processor using formal verification. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6.

Xu, Y. et al. (2022). Towards developing high performance risc-v processors using agile methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1178–1199. IEEE.

Yosys (2025). RISC-V Formal Verification Framework. https://github.com/YosysHQ/riscv-formal. [Online; accessed 26-August-2025].