Journal of Systems Architecture 175 (2026) 103761

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

χRVFormal: Formal verification of RISC-V processor Chisel designs☆

Shidong Shen [a,b], Shijie Chen [a,b], Yicheng Liu [a,b], Lijun Zhang [a,b], Fu Song [a,b,c], Zhilin Wu [a,b,*]

[a] Key Laboratory of System Software (Chinese Academy of Sciences), Institute of Software, Chinese Academy of Sciences, Beijing, China
[b] University of Chinese Academy of Sciences, Beijing, China
[c] Nanjing Institute of Software Technology, Nanjing, China

ARTICLE INFO

Keywords:
Chisel
RISC-V
Formal verification
ISA conformance
Model-checking

ABSTRACT

Chisel, an open-source high-level hardware construction language embedded in Scala, was proposed to facilitate the development of parameterizable, reusable circuit design generators. It is becoming increasingly popular and has been used to design many RISC-V processor variants. Formal verification has been adapted to rigorously check the (functional) correctness of RISC-V processor designs. However, the RISC-V instructions therein are specified in the low-level hardware languages Verilog/SystemVerilog, which are challenging to develop, maintain, customize, and extend. This considerably lowers the advantage of RISC-V for agilely designing highly customizable processors. In this work, we present χRVFormal, the first end-to-end approach for formally verifying the correctness of RISC-V processor designs fully at the Chisel high-level. Specifically, by utilizing the object-oriented and functional programming constructs offered by Chisel, we develop a high-level reference model of RISC-V instructions in Chisel. This reference model is a succinct, modular, and parameterized RISC-V processor design generator, supporting the RV32/64IMCBZicsr instruction sets, three privilege levels (M/S/U), and Sv39 virtual memory system, thus can be easily instantiated to produce customized reference model instances. We then devise a synchronization mechanism between the RISC-V processor Chisel design and the reference model by which the correctness verification of the RISC-V processor design is reduced to the model-checking problem and off-the-shelf model checkers can be harnessed. Our synchronization mechanism supports both in-order single issue RISC-V processors and out-of-order superscalar RISC-V processors. We implement our approach in an open-source tool and demonstrate its efficacy on three representative open-source RISC-V processor designs in Chisel: riscv-mini, NutShell, and BOOM, where the former two are in-order single issue processor designs while BOOM is an out-of-order superscalar processor design. The experimental results confirm the efficacy of our approach, capable of discovering 7 real-world unknown non-conformance bugs and all the 15 manually injected bugs, significantly more effective than two state-of-the-arts. It is also three-orders-of-magnitude more efficient than the state-of-the-art symbolic-execution based approach.

1. Introduction

Background. RISC-V is a royalty-free and open standard Instruction Set Architecture (ISA) based on the well-established principles of Reduced Instruction Set Computer (RISC). The RISC-V instruction set manual provides detailed natural language specifications of RISC-V ISA in two volumes. The first volume [1] specifies the unprivileged ISA, including the base integer instructions (RV32/64I) and optional unprivileged extensions, such as Multiplication and Division extension (M), and Compressed instructions (C). The second volume [2] specifies the privileged ISA and those beyond the unprivileged ISA, including privileged instructions and additional functionalities required for running operating systems and attaching external devices. The scalable and extendable RISC-V ISA brings a new level of flexibility in designing highly customizable processors.

Along with RISC-V, Chisel (Constructing Hardware in a Scala Embedded Language) [3], an open-source hardware description language (HDL), was proposed. Chisel features hardware construction primitives and Scala's object-oriented and functional programming constructs, allowing designers to easily and quickly write complex, modular circuit generators that can produce circuit designs in synthesizable Verilog/SystemVerilog. This generation methodology enables the creation

☆ This article is part of a Special issue entitled: 'SETTA 2024' published in Journal of Systems Architecture.

* Corresponding author at: Key Laboratory of System Software (Chinese Academy of Sciences), Institute of Software, Chinese Academy of Sciences, Beijing, China.
E-mail addresses: shensd@ios.ac.cn (S. Shen), chensj@ios.ac.cn (S. Chen), liuyc@ios.ac.cn (Y. Liu), zhanglj@ios.ac.cn (L. Zhang), songfu@ios.ac.cn (F. Song), wuzl@ios.ac.cn (Z. Wu).

https://doi.org/10.1016/j.sysarc.2026.103761
Received 7 June 2025; Received in revised form 7 February 2026; Accepted 28 February 2026
Available online 3 March 2026

of parameterizable, reusable components and libraries, such as the FIFO queue and arbiters in the Chisel Standard Library. Compared to the classic HDLs (e.g., Verilog, SystemVerilog, and VHDL), Chisel offers a higher level programming abstraction for hardware designs. Since its introduction, Chisel has been widely used to design many RISC-V processor variants (e.g. RocketChip [4], BOOM [5], riscv-mini [6], NutShell [7], and XiangShan [8]), and is playing an indispensable role in the promotion of the agile hardware development methodology that can reduce time-to-market and improve quality and productivity [9–11].

Motivation. Modern processors are becoming more and more complex, consequently, ensuring their (functional) correctness in the pre-silicon stage becomes a central issue in the processor development. In this work, we focus on formal verification for the correctness of RISC-V processor designs in Chisel, that is, to check whether the implementations of RISC-V instructions in Chisel conform to their specifications.

The state-of-the-art verification techniques for Chisel (namely, the open-source tool riscv-formal [12] and the commercial tools FormalISA [13] and OneSpin [14]) first compile Chisel designs into low-level implementations in Verilog/SystemVerilog, then check their conformance to the RISC-V ISA specifications in the form of Verilog/SystemVerilog reference models or SystemVerilog assertions (SVA) via model-checking. Though available, such low-level representation of RISC-V ISA specifications makes the development, maintenance, diagnosis, customization, and extension of the reference models or assertions challenging and error-prone, hindering the flexibility of RISC-V ISA for agilely designing highly customizable processors. Thus, a succinct, modular, and parameterized reference model, as well as an accompanied formal verification tool, is highly-required.

Contribution. In this work, we propose χRVFormal, the first *end-to-end* formal verification approach for the correctness of RISC-V processor designs fully at the Chisel high-level.[1]

Specifically, we make the following major contributions.

1. We develop a high-level, succinct, modular, and parameterized reference model for RISC-V ISA specifications in Chisel as a single-clock-cycle RISC-V processor generator, by utilizing the object-oriented and functional programming constructs offered by Chisel (cf. Section 3). The reference model implements all the common unprivileged instructions and various features of privileged instructions, including control and status registers (CSR), exception handling, and virtual memories. To the best of our knowledge, it is the most comprehensive reference model among all the open-source RISC-V reference models that are accompanied with formal verification tools. For instance, our model not only strictly subsumes the instructions of, but also is an-order-of-magnitude more succinct than the Verilog/SystemVerilog reference model in riscv-formal. Furthermore, our reference model can be easily configured to produce various customized models.
2. We propose an end-to-end formal verification approach, χRVFormal, for RISC-V processor designs in Chisel by devising a synchronization mechanism between the RISC-V processor DUT (design under test) and our reference model, to deal with the non-fixed delays resulted from the memory accesses as well as the additional challenges posed by the virtual memory in the DUT (cf. Section 4). With our synchronization mechanism, the correctness verification of DUT w.r.t. the RISC-V ISA specifications is reduced to the model-checking problem of the synchronized Chisel DUT with the reference model, that can be solved by off-the-shelf model checkers. Furthermore, χRVFormal also supports the traditional verification flow at the Verilog/SystemVerilog low-level by compiling the synchronized Chisel DUT design with its reference model into Verilog/SystemVerilog and then harnessing off-the-shelf Verilog/SystemVerilog formal verification tools such as SymbiYosys.
3. We implement our approach in a tool using the open-source SMT-based model checkers Pono [15] and SMTBMC [16]. We validate the effectiveness of our approach on riscv-mini [6], NutShell [7] and BOOM [5], three representative open-source RISC-V processor designs in Chisel (cf. Section 5). The results confirm the effectiveness of our approach, discovered 7 real-world unknown ISA non-conformance bugs (2 in riscv-mini and 5 in NutShell) and all the 15 manually injected bugs (5 bugs per DUT). Compared with the state-of-the-art open-source tool riscv-formal [12],[2] while our approach is comparable in terms of verification efficiency, riscv-formal *only* discovered 2 of 7 real-world bugs. We also compare with a recent symbolic-execution based tool [17,18], which is only able to discover 1 out of 7 bugs using significantly more time than our tool χRVFormal (58 min vs. 0.52 s).

This article is an extended version of our preliminarily work presented at SETTA 2024 [19], with the following substantial new material.

1. A new SymbiYosys-based verification flow for generating transition systems was added to support the latest version of Chisel (cf. Section 2).
2. A new verification strategy, called *ActionCheck*, was proposed to verify more complicated processor designs such as out-of-order superscalar RISC-V processors at the cost of verification accuracy (cf. Section 4.2).
3. The reference model has been extended to support the Bit-Manipulation instructions (including Zba, Zbb, Zbc, Zbs, Zbkb, Zbkc, and Zbkx) and out-of-order superscalar RISC-V processors. We also restructured the reference model to separate the computation and register state maintenance in order to support the ActionCheck verification strategy (cf. Section 3).
4. The formal verification tool χRVFormal has been extended accordingly and additional experiments were conducted to compare two verification strategies and existing verification tools, using riscv-mini, NutShell and BOOM, an out-of-order superscalar processor that did not appear in the SETTA 2024 paper. We also include attempts at a software-model-checking based approach to assess its feasibility for verifying RISC-V processors. (cf. Section 5).

Outline. The rest of the article is organized as follows. Section 2 gives an overview of our end-to-end verification approach for RISC-V processor designs in Chisel. Section 3 describes the design of our reference model. Section 4 describes two verification strategies and the synchronization mechanism between the RISC-V processor DUT and the reference model. Section 5 reports experimental results. Section 6 discusses related work. We conclude this work in Section 7.

The source code of our RISC-V ISA reference model and formal verification tool, namely χRVFormal, are available at:
https://github.com/iscas-tis/ChiRVFormal.

2. Approach overview

In this section, we give an overview of our approach (see Fig. 1), comprising the following three key components.

[1] Here "fully at the Chisel high-level" means that the reference model, signal synchronization, assertion generation are completely within Chisel ecosystem, irrelevant to how the intermediate verification step of the synchronized model is achieved, using either Chisel high-level or low-level verification engine with an automated reduction.

[2] We do not compare with the two aforementioned commercial tools FormalISA [13] and OneSpin [14], as they are not publicly available.

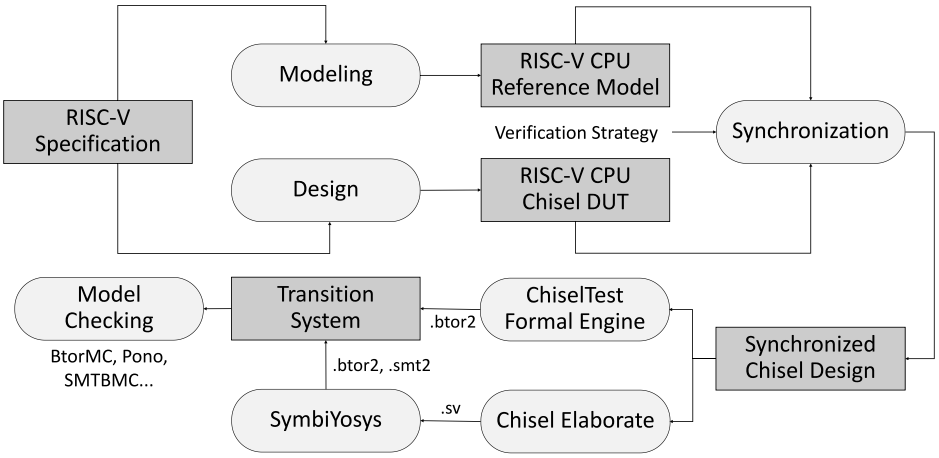

Fig. 1. Overview of our approach.

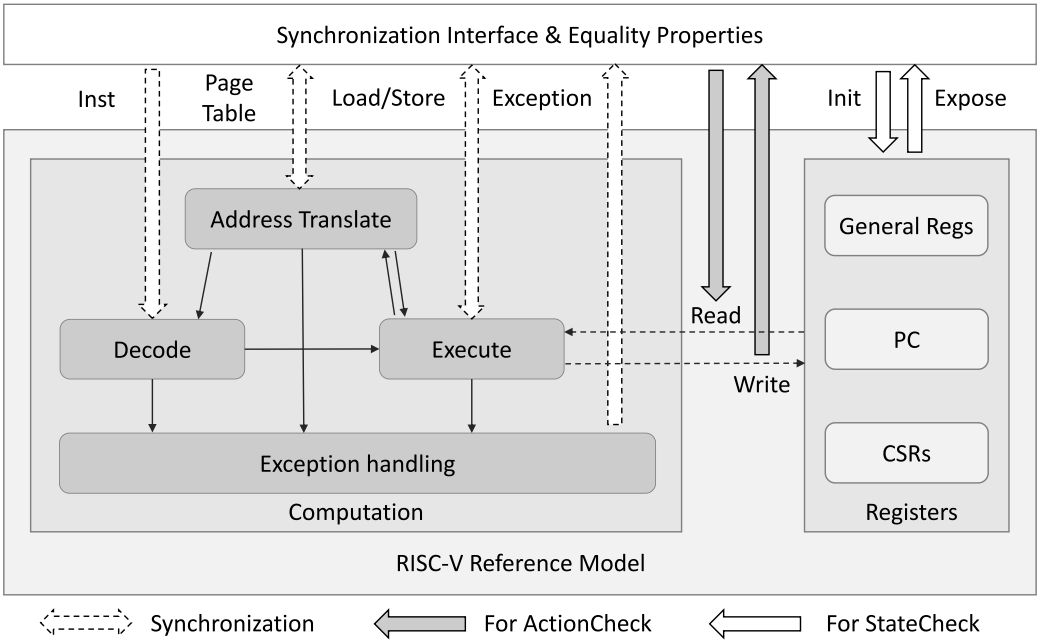

Fig. 2. The RISC-V reference model in Chisel.

Reference model. At first, following closely the official two volumes of the RISC-V ISA natural language specifications, we design a succinct and parameterized reference model for RISC-V processor design generator in Chisel with high confidence of correctness, where both the unprivileged and privileged instructions are covered. We elaborate the design details of the reference model in Section 3.

Verification strategies and synchronization. To verify whether a RISC-V CPU Chisel DUT conforms to the RISC-V ISA specifications, the DUT is synchronized with the reference model and the results of the instruction executions in the DUT and the reference model are compared for equality checking. However, such a synchronization is nontrivial since there is a significant gap of the micro-architectures between the DUT and the reference model. To fill this gap, we propose two verification strategies (StateCheck and ActionCheck), enabling verification engineers to select the appropriate approach based on the DUT's micro-architecture and identify which signals require synchronization. With the verification strategies, we propose a queue-based synchronization method, by which a synchronized Chisel design is obtained. We discuss the details of the synchronization challenges, the verification strategies and the synchronization method in Section 4.

Model-checking. We verify the synchronized Chisel design by model-checking for which the design will be transformed into a transition system. We leverage `chiseltest.formal.verify` [20], the official formal verification engine for Chisel up to version 6.0, to produce the transition systems in the format of BTOR2 from Chisel programs. Nevertheless, while `chiseltest.formal.verify` is capable of dealing with two open-source CPU designs (namely, riscv-mini and NutShell) in Chisel considered in this work, it generates invalid BTOR2 files out of BOOM [5], another CPU design considered in this work. As a result, we also incorporate an alternative approach where the synchronized Chisel design is transformed into SystemVerilog, which in turn is transformed into the transition system in the format of BTOR2 or SMT2 by using SymbiYosys [16].

Given the resulting transition system, we reduce the verification task to the model-checking problem. The verification process creates a model-checking instance for each assertion, enabling parallel verification and improved scalability for complex designs. These model-checking instances then can be passed to model checkers such as BtorMC [21] and Pono [15] for BTOR2, or SMTBMC [16] for SMT2, where various model-checking algorithms, e.g., BMC [22], k-induction [23], and PDR [24], could be harnessed.

3. The RISC-V reference model in Chisel

In this section, we elaborate the design details of our RISC-V reference model in Chisel. In particular, we illustrate how to model privileged instructions in a natural and succinct way by utilizing the object-oriented and functional programming constructs of Chisel. The architecture of our reference model is shown in Fig. 2.

Intuitively, the RISC-V reference model is a modular, and parameterized RISC-V processor design generator in Chisel that can be easily instantiated to produce customized single-clock-cycle RISC-V processor reference model instances. As a result, it involves the common steps, namely, decoding, executing, and writing back to update registers, in a typical RISC-V processor design. The decoding step identifies the op-code instruction from its binary code; the executing step performs the actual operation of the op-code instruction (e.g., computation, memory access, or control transfer); and the write-back step is responsible for updating the values of general-purpose registers (GPRs), Program Counter (PC), and Control and Status Registers (CSRs) after instruction execution. The reference model comprises the computation module and the registers module. The computation module implements the decoding, executing, and the components for modeling privileged instructions (i.e. address translation and exception handling) in a stateless manner, thus it performs only combinational operations. Instead, the register module is responsible for updating register values, thus it is stateful. This modular design allows for flexible verification strategies.

The RISC-V reference model is capable of supporting the modeling of RISC-V 32/64 Integer (I), Multiplication and Division (M), Compressed (C), Control and Status Register (Zicsr), and B (Bit-Manipulation, covering Zba, Zbb, Zbc, Zbs, Zbkb, Zbkc, Zbkx) instruction sets, as well as the three privilege levels, namely, Machine (M), Supervisor (S), and User (U), and finally the Sv39 virtual memory system.

Our RISC-V reference model is designed lazily in the sense that it covers most of the instructions occurring in the open-source RISC-V cores such as Rocket, BOOM, NutShell and XiangShan. In terms of the coverage of the RISC-V ISA, our RISC-V reference model covers around 20% of the instructions in the latest version (version 20250508) of the official RISC-V ISA manual.[3] Although at first sight, our reference model is far from complete with respect to the RISC-V ISA manual, we would like to emphasize that to the best of our knowledge, it is the state-of-the-art most comprehensive reference model among the open-source RISC-V reference models that are accompanied with formal verification tools (cf. Section 6 for a detailed comparison). More importantly, compared to these reference models, our reference model is designed and implemented in a succinct, modular, and parameterized style, allowing additional instruction sets to be added easily when required.

3.1. Succinct, modular, and parameterized design

One of the notable features of our RISC-V reference model is its succinct, modular, and parameterized design generator in Chisel, that is,

- the reference model comprises only around 4600 lines of Chisel code,
- the reference model resembles the modular structure of the RISC-V instruction set manual,
- the parameters therein can be instantiated to produce customized RISC-V reference model instances with different verification strategies, bit-widths and ISA extensions.

As shown in Table 1, the Chisel reference model contains the following six modules.

- The *Interface* module is responsible for signal synchronization with the DUT. It comprises 423 lines of Chisel code.
- The *Checker* module is responsible for generating the assertions that will be used for equivalence checking between the DUT and the reference model. It comprises 354 lines of Chisel code.
- The *CoreTop* module serves as the "main" module of the reference model. It comprises 425 lines of Chisel code.
- The *CommonDecode* module is responsible for instruction decoding and dispatching. It comprises 121 lines of Chisel code.
- The *Utils* module encapsulates the data structures used in the reference model. It comprises 642 lines of Chisel code.
- The *Instruction-Set* module implements the instructions in various RISC-V ISA extensions (e.g. *IBase, B, C, M, Zicsr, Zifencei* and privileged instructions) in a modular and parameterized style as well. It comprises 2644 lines of Chisel code. Moreover, the implementation of the privileged instructions contains 1018 lines of code and occupies the largest portion of the Instruction-Set module. Finally, the *Instruction-Set* module is designed in a modular way so that new ISA extensions can be added as sub-modules easily.

Listing 1: Example for the parameter instantiation of the reference model

```
FormalConfig: RVConfig = RVConfig(
    XLEN = 64,
    extensions =  "MCZicsrZifenceiSU",
    functions = Seq("Privileged", "TLB")
  )
```

Users can easily create an instantiation of the reference model by configuring the parameters. For instance, according to the configuration in Listings 1, a 64-bit variant of the reference model will be generated, where the *I, M, C, Zicsr, and Zifencei* ISA extensions and *M/S/U* privilege levels are supported, moreover, the TLB component (Translation Lookaside Buffer) is included.

For a particular instantiation of the parameters, when the reference model is compiled into FIRRTL (a hardware intermediate representation language for Chisel) [25], only the ISA specifications and configurations that are related to this instantiation will be compiled in the resulting FIRRTL code while the others are dropped. This design choice often reduces the sizes of the transition systems, thus alleviating the state-explosion problem of model-checking. On the other hand, if the low-level Verilog/SystemVerilog language is used to implement a RISC-V reference model, then the code size would be much larger and it is not very convenient to support different bit-widths or ISA extensions. For instance, in riscv-formal [12], around 100,000 lines of Verilog/SystemVerilog code are used to model the RISC-V ISA specifications with additional 3523 lines of Python scripts for configuring the bit-widths and ISA extensions of the reference model, although it models less instructions than our reference model.

In contrast, our Chisel-based RISC-V reference model comprises only about 4600 lines of code (cf. Table 1). Based on the publicly available development records, riscv-formal, which was implemented in Verilog/SystemVerilog, involved a larger and more experienced team and a longer development cycle (around six engineers and over 400 commits), whereas our reference model was developed by a small student team of three graduate students with about 200 commits. While the comparison of these numbers may be overly simple and biased, the comparison still indicates that the high-level, object-oriented and functional-programming styles of the Chisel language considerably reduce the efforts of the implementation and maintenance.

[3] https://riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications

Table 1
Modules of the reference model.

Module	Sub-module	Description	LOC
Verification Infrastructure			
Interface	–	DUT signal synchronization	423
Checker	–	Assertion Generation	354
RISC-V ISA Implementation			
CoreTop	–	The "main" module	425
CommonDecode	–	Instruction decoding	121
Utils	–	Data structures	642
	IBase	Integer base	457
	B	Bit manipulation	540
	C	Compressed	314
Instruction-Set	M	Multiply/divide	120
	Zicsr	CSR operations	164
	Zifencei	Fence operations	31
	Privileged	Privileged support	1018
	Total	–	2644
Total	–	–	**4,609**

3.2. Modeling privileged instructions

The object-oriented and functional programming constructs of Chisel are utilized to model the two mechanisms related to privileged instructions, namely, exception handling and virtual–physical memory address translation.

Exception handling. At first, thanks to Chisel's high-level language features, the enable bit in the exception vector can be turned on directly by calling the designated exception triggering functions, implemented in Chisel as well. Moreover, Scala's collection functions such as `foldRight` are very effective in implementing the priority arbitration for exceptions.

Virtual–physical memory address translation. The virtual memory system plays a vital role in modern operating systems, and Translation Lookaside Buffer (TLB) is one of the critical components in a processor for fast translation between virtual and physical memory addresses. In a typical processor design, the TLB component is usually implemented as a state machine to access the memory for multiple times. Nevertheless, our RISC-V reference model in Chisel only uses combinational logic, which should be finished in one clock cycle. As a result, it is necessary for the RISC-V reference model to have more copies of the memory read/write ports. In our reference model, we implement the virtual–physical memory address translation component for Sv39, which is a three-level page table virtual memory system. As a result, three additional page-read ports and one additional page-write port are included for the address translation, besides the conventional memory access interface for the load/store instructions. (See Fig. 3 for the architecture of the address translation component.)

3.3. Validation of the reference model

Since the reference model is used as the specification in the formal verification of the ISA conformance of RISC-V CPU Chisel designs, the correctness of the reference model itself should also be validated. Nevertheless, the formal verification of the reference model with respect to the RISC-V ISA specification (in natural language) is in some sense similar to the formal verification of the formal specification of computer systems with respect to the natural-language specification, which is a well-known open problem. As a result, in this work, instead of formal verification, we utilize the following three affordable but informal means to validate the correctness of the reference model.

- We manually audit the RISC-V reference model with respect to the RISC-V ISA natural-language specification. As a design choice, the high-level features of Chisel are utilized to make the reference model closely resembling the structure of the RISC-V ISA manual. This reduces the chances of introducing mistakes in the design of the reference model and makes the manual auditing feasible.
- Since the reference model can be seen as a parameterized single-clock-cycle RISC-V processor, we also use riscv-tests [26], the official RISC-V CPU test suite, to thoroughly validate the correctness of the reference model. The riscv-tests test suite is the defacto standard benchmark for RISC-V processor verification in both academia and industry [27]. We test our reference model by using all tests that are applicable to the instruction extensions implemented in the reference model, including those test cases covering the virtual-memory address translation. In total, 114 riscv-tests cases were executed, all of which were successfully passed by the reference model, increasing our confidence on the correctness of the reference model.
- Finally, our reference model is also validated when it is applied to formally verify the various RISC-V processor DUTs. For each DUT, we carefully analyze the mismatches between the DUT and the reference model in the formal verification and confirmed that the mismatches are either the bugs in the DUT or the bugs in the synchronization of the DUT and the reference model, but not the bugs in the implementation of the reference model, which increases our confidence on the correctness of the reference model further.

4. The verification strategies and synchronization mechanism

In this section, we first outline the challenges in synchronizing the DUT with the single-cycle reference model, then introduce the verification strategies together with the synchronization mechanism to facilitate the instruction-level comparison between them. For the synchronization mechanism, we first show how to synchronize the reference model with the in-order single-issue processors, then show how to synchronize with the out-of-order superscalar processors.

4.1. Synchronization challenges

To facilitate the formal verification of the ISA conformance of the RISC-V CPU DUT with respect to the reference model, it is necessary to synchronize the DUT with the reference model. However, such a synchronization is non-trivial since there is a significant gap of the micro-architectures between the DUT and the reference model.

First, while the reference model is a (parameterized) single-clock-cycle processor, the micro-architecture of the DUT is typically different from that of the reference model, ranging from simple in-order pipelines to complex out-of-order superscalar designs with multiple issue channels. This diversity makes synchronization inherently challenging.

Another challenge arises from the fact that not all signals required for verification are preserved through the pipeline, as processor designers may not propagate all debugging and verification-required signals

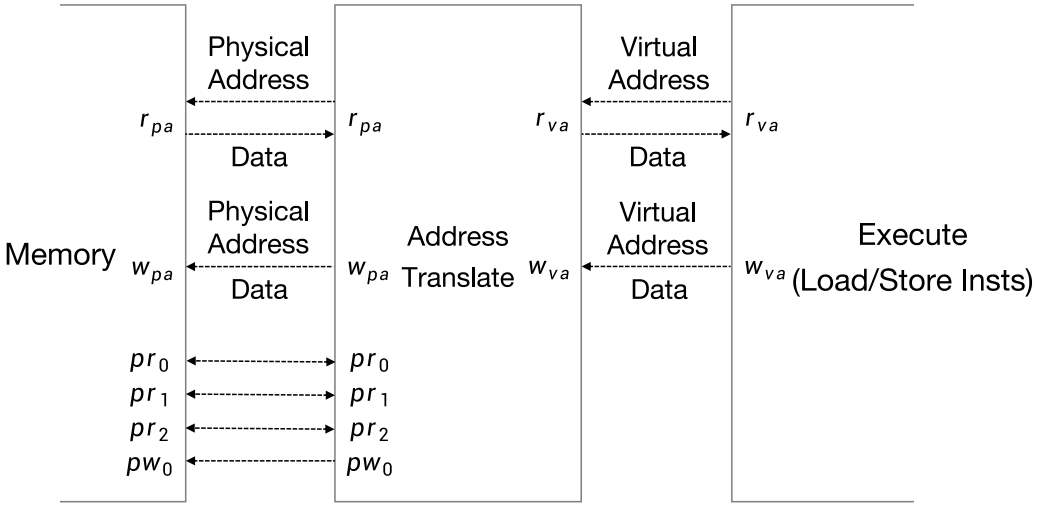

Fig. 3. Architecture of the address translation component.

to the write-back stage, e.g., instruction signals that are only used in decoding and are discarded afterwards. For verification purposes, verification-required signals must be captured and preserved in the pipeline at every stage until they reach the final write-back stage, referred to as "pipeline follower" in [28]. However, some critical signals for verification—particularly those related to load/store operations and virtual memory translations—may not be explicitly stored in pipeline registers due to implementation optimizations. Additionally, signal extraction in out-of-order superscalar processors presents further challenges due to their inherent complexity.

To address these challenges, we first present two verification strategies (StateCheck and ActionCheck) in Section 4.2, which the verification engineers can select based on the DUT's micro-architecture. Once a verification strategy is selected, the synchronization signals can be automatically identified accordingly. Then in Sections 4.3 and 4.4, the synchronizations with in-order single-issue processors and out-of-order superscalar processors are described respectively.

The synchronization mechanism is designed to bridge the gap between the complex micro-architectures (e.g. pipelines, out-of-order executions) of DUTs and the reference model. In this work, we validate empirically the correctness of the synchronization mechanism by manually inspecting the mismatches between DUTs and the reference model discovered in the formal verification and analyzing whether these mismatches are attributed to bugs in the synchronization of the DUTs and the reference model. Since it is highly challenging to formally verify the correctness of the synchronization mechanism, we leave its formal verification as the future work.

4.2. Verification strategies: StateCheck and ActionCheck

To accommodate different micro-architectures, we propose two verification strategies, namely, StateCheck and ActionCheck. They primarily differ in whether the reference model maintains internal register values itself or not, indicating whether the synchronization between the reference model and the DUT shares the same internal register values or not. We now describe each strategy in detail.

StateCheck. In this strategy, the reference model independently maintains its architectural states, including all 32 general-purpose registers (GPRs), the program counter ($\texttt{PC}_{\texttt{now}}$)[4] and control and status registers (CSRs). It updates these values cycle-by-cycle according to the instruction and relevant memory data. The resulting state is then compared against the synchronized signals from the DUT.

[4] $\texttt{PC}_{\texttt{now}}$ denotes the address of the instruction currently being executed.

Table 2
Synchronized input signals required by StateCheck and ActionCheck.

Signals	StateCheck	ActionCheck
Instruction	✓	✓
Load Data	✓	✓
PageTable Data	✓	✓
rs1 and **rs2**	✗	✓
$\texttt{PC}_{\texttt{now}}$	✗	✓
CSRs	✗	✓

ActionCheck. In contrast, ActionCheck avoids maintaining internal architectural states in the reference model. Instead, it relies on the DUT to provide the architectural inputs—such as source register values (`rs1` and `rs2`), PC ($\texttt{PC}_{\texttt{now}}$), and CSRs to simulate instruction executions. Based on these synchronized inputs and memory data, the reference model computes the expected result, i.e., the destination register (`rd`) and next program counter ($\texttt{PC}_{\texttt{next}}$),[5] and checks for consistency with the DUT's outputs.

Table 2 summarizes the input signals required by each strategy, and Table 3 lists the architectural signals checked for equivalence.

StateCheck vs. ActionCheck. Compared to StateCheck, ActionCheck requires more synchronization efforts, especially for signals like $\texttt{PC}_{\texttt{next}}$, which are often not explicitly exposed in the DUT. Despite this synchronization overhead, ActionCheck generally achieves higher verification efficiency, as it enables early detection of control-flow bugs and involves fewer verification properties. However, since it relies on DUT-provided inputs and does not track the full architectural states, the verification is coarser and thus may miss some subtle inconsistencies—similar to the limitations of riscv-formal, which failed to detect bug R:E2 (cf. Section 5.2). In contrast, StateCheck requires fewer synchronized signals and offers more comprehensive verification by maintaining the entire architectural states itself, though typically at the cost of verification efficiency.

4.3. Synchronization of in-order single-issue processors

In-order single-issue processors commit exactly one instruction per clock cycle and follow a strictly ordered pipeline, making their structure relatively simple and widely used. Both strategies ActionCheck and StateCheck can be applied to verify such designs. In the sequel, we shall

[5] $\texttt{PC}_{\texttt{next}}$ denotes the address of the next instruction to be executed, which may be determined by branch or jump instructions.

show how to synchronize the in-order single-issue RISC-V CPU designs with the reference model.

For many instructions (e.g. the integer instructions), the synchronization is not difficult since the number of clock cycles for their executions in the DUT have a *fixed* (constant) upper bound. However, when the execution of an instruction involves memory accesses, the delay can be non-fixed. In the sequel, we first show how to synchronize the load/store instructions with non-fixed delays. Then we show how to synchronize the more challenging instructions involving the virtual memory. Finally, we show how to synchronize the initial values of general-purpose registers for the StateCheck strategy.

4.3.1. Synchronization of load/store instructions with non-fixed delays

Let us use the load instructions in NutShell to illustrate the non-fixed delays.

In RISC-V ISA, there are four load instructions, namely, LD, LW, LB, and LH. In contrast, NutShell [7], a 64-bit RISC-V processor, categorizes the load instructions into partial reads and full reads. The LW, LH, and LB instructions are *partial reads* that read 32, 16, and 8 bits respectively, while the LD instruction is a *full read*, that reads 64 bits. The delay for transmitting the signal to the write-back stage in the execution of a partial-read instruction is different from that of a full-read instruction, where the delay of a partial read is one cycle shorter than a full read.

To deal with the non-fixed delays of instructions in DUTs, we propose to utilize queues. When an instruction in the DUT is executed, some necessary memory access information (such as validity, address, data and width) is recorded in a queue. When the execution of an instruction is complete (usually the write-back stage in pipelines), the information can be retrieved from the queue and compared with the signals in the reference model. Since Chisel provides an implementation of queues in its library, it is relatively easy to implement the synchronization mechanism where queues are utilized to deal with non-fixed delays.

Currently, we provide an optional configuration to use queues for synchronizing load/store memory access information in the reference model only for the StateCheck verification strategy. For the ActionCheck verification strategy, which is commonly used for verifying out-of-order superscalar processors, we do not adopt queues. This decision is made to reduce the design complexity. Indeed, in out-of-order superscalar processors, the Load/Store Unit (LSU) typically maintains its own memory access queue. As a result, the required memory access information can usually be obtained directly from the DUT based on the index of the current memory access channel instead of using queues in the reference model.

4.3.2. Synchronization of instructions involving virtual memory

The virtual memory poses additional challenges for the synchronization. As shown in Fig. 3, three page-read ports and one page-write port are used in the reference model to model the memory translations of load/store instructions. To deal with the delays resulted from the virtual memory, it is necessary to introduce five queues to store the memory access information of DUTs: one queue for each of the three page-read ports, and two queues for the read/write physical address involved in the load/store instructions (i.e., r_{pa} and w_{pa} respectively).

When a load/store instruction is executed in a DUT involving virtual memory, the corresponding memory access information is first extracted from the DUT and added into the three TLB read queues, then the reference model utilizes the information in the three TLB read queues to translate a virtual address into a physical address, which is finally compared with the value retrieved from the load/store queues that was added by the DUT to ensure that they are equal.

4.3.3. Synchronization of the initial values of general-purpose registers

When verifying in-order single-issue processors using the StateCheck verification strategy, we can further enhance its effectiveness by synchronizing the initial values of the general-purpose registers between the DUT and the reference model, in order to facilitate the earlier error detection.

Both the DUT and the reference model contain 32 general-purpose registers, denoted by $x_0, \ldots, x_{31}$ and $x'_0, \ldots, x'_{31}$, respectively. Some bugs may not be triggered directly when the values of these registers are initialized to 0. In such a situation, some load instructions should be introduced first before executing the other instructions, in order to expose the bugs. As a result, we apply the following trick so that the introduction of load instructions can be avoided: For each $1 \leq i \leq 31$, we synchronize the initial values of x_i and x'_i by introducing an additional input in_i and connecting in_i to both x_i and x'_i. Note that the initial values of the register x_0 and x'_0 are always set to 0. This trick helps to reduce the number of clock cycles to detect some bugs. Furthermore, we can even achieve the same effect by introducing only two inputs, say inputs in_1 and in_2, and synchronizing the initial values of x_i and x'_i with $i = 1, 2$. (Note that the initial values of the other registers are still 0.) This simplified trick is justified by the fact that each instruction uses at most two general-purpose registers as source registers.

4.4. Synchronization of out-of-order superscalar processors

Out-of-order superscalar processors can execute multiple instructions in a single clock cycle, typically through mechanisms such as out-of-order execution and multiple instruction issues (multi-issue), which introduces significant challenges for synchronization. In the sequel, we show how to synchronize such processors with the reference model. Specifically, we first show how to utilize the multiplexers to deal with the multiple instruction issues. Then we show how to synchronize the general-purpose registers and the program counter. Finally, we discuss the challenges we meet when trying to synchronize the instructions involving the virtual memory for out-of-order superscalar processors.

4.4.1. Multiplexer selection for multiple instruction issue

When multiple instructions are committed in the same cycle, the StateCheck verification strategy becomes difficult to apply. Since the reference model executes only one instruction per cycle, matching the state transitions caused by multiple simultaneously committed instructions in the DUT requires a sequential replay in the exact commit order. This means that the reference model must execute across multiple cycles, significantly increasing implementation complexity and moreover hindering the isolation and behavior checking of individual instructions.

By introducing ActionCheck, we provide a scalable solution to tackle this challenge. ActionCheck allows us to verify the DUT at the granularity of individual commit channels. As illustrated in Fig. 4, a multiplexer selects one of the committed instructions in each cycle using the $select_i$ signal, which serves as an input to the transition system. This can be implemented using flying wires in Chisel. We then apply ActionCheck to verify the correctness of the selected instruction. This approach makes it feasible to verify designs with out-of-order and multi-issue architectures while maintaining the precision and flexibility of cycle-level checking. A similar approach using multiplexers to select commit channels and their corresponding instructions has also been adopted in prior work, such as ISA-Formal [28].

4.4.2. Synchronization of general-purpose registers

Out-of-order execution allows instructions to be executed in a different order than they appear in the program to improve performance. To support precise exceptions, such processors rely on an in-order commit mechanism implemented via a Reorder Buffer (ROB). Together, these features require that general-purpose registers in out-of-order designs

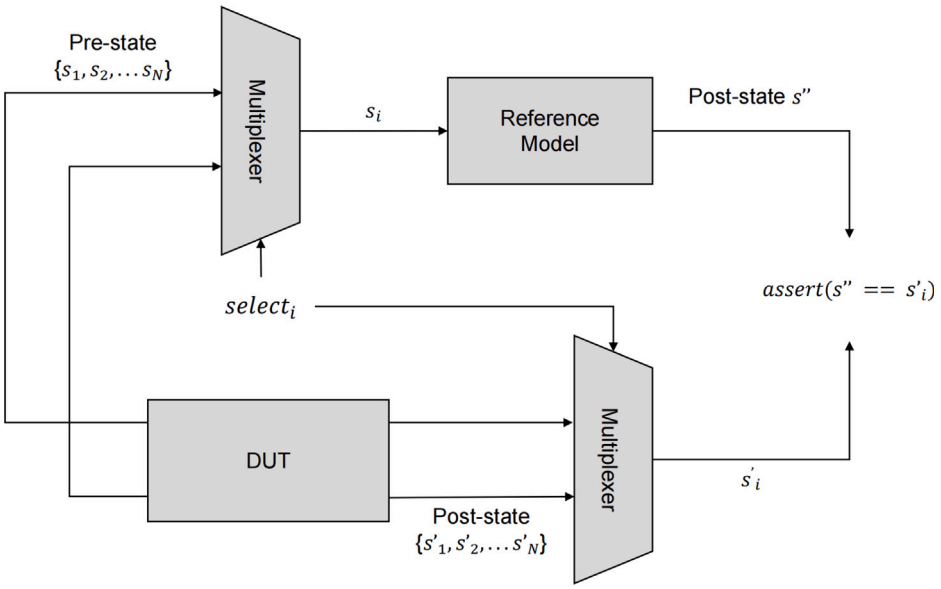

Fig. 4. Multiplexer Selection.

are mapped from a larger pool of physical registers through register renaming.

As a result, directly obtaining the values of general-purpose registers to synchronize with the reference model, as done for the in-order single-issue processors, becomes impractical. To address this issue, we maintain a shadow copy of the 32 general-purpose registers at the commit stage of the DUT, which serves as a precise ISA-level view of the general-purpose registers. Whenever an instruction is committed via the ROB, the corresponding update is reflected in the shadow registers. This approach makes it easier to track the values of the 32 general-purpose registers, even though the processor uses register renaming and executes instructions out-of-order.

We remark that special attention is required to handle data dependencies introduced by multiple instruction issues. For example, if two issue channels, `ch0` and `ch1`, execute dependent `addi` instructions in the same cycle (as shown in Listing 2), the source operand `x1` used by `ch1` should reflect the result produced by `ch0`. To mitigate this problem during synchronization, a bypassing mechanism is applied to the shadow register copy: namely, based on the issue order, each channel retrieves the most recent value of a source register as last updated by an earlier channel in the same cycle.

Listing 2: Example of register dependency in multiple instruction issue

```
ch0: addi x1,  x1, 8   # x1  = x1 + 8
ch1: addi x14, x1, 16  # x14 = x1 + 16
```

4.4.3. Synchronization of $\mathtt{PC}_{\mathtt{next}}$

For in-order single-issue processors, control transfer instructions (such as conditional branches and jumps) can be verified by comparing either the current program counter ($\mathtt{PC}_{\mathtt{now}}$) for the StateCheck strategy or the next program counter ($\mathtt{PC}_{\mathtt{next}}$) for the ActionCheck strategy between the DUT and the reference model, because these signals are readily available and sufficient for validating control flows in in-order single-issue processors.

However, the verification of out-of-order superscalar processors must use the ActionCheck strategy which has to check $\mathtt{PC}_{\mathtt{next}}$, while the extraction of $\mathtt{PC}_{\mathtt{next}}$ is challenging in out-of-order superscalar processors due to branch prediction, speculative execution, and the decomposition of control flow handling across multiple pipeline stages. Indeed, $\mathtt{PC}_{\mathtt{next}}$ is often computed over multiple cycles and dispersed across different modules, making it impractical to observe a single definitive signal at any given moment.

To address this issue, we monitor the execution of all the control-flow instructions, record their behavior, such as branch decisions, jump offsets, and partial target addresses, at write-back stage, and reconstruct the value of $\mathtt{PC}_{\mathtt{next}}$ in the commit stage using their architectural outcomes when these instructions are eventually committed. This method is effective for out-of-order superscalar processor designs, such as BOOM [5].

4.4.4. Challenges in the synchronization of instructions involving the virtual memory

The synchronization of out-of-order superscalar processors in this work does not support the instructions involving virtual memory. We justify this technical choice by analyzing the challenges we face when trying to do so.

Recall that in the queue-based synchronization mechanism for in-order processors, five queues are introduced to deal with delays resulted from the virtual memory for each memory access in the DUT. In the out-of-order superscalar processors with virtual memory, in a single clock cycle, multiple (possibly unbounded) memory accesses can be issued to the memory management unit for translating virtual addresses to physical addresses. As a result, it is challenging (if not impossible) to extend queue-based synchronization mechanism for virtual memory to out-of-order processors, since possibly unbounded number of queues have to be introduced.

The idea of using a pipeline follower to record multiple memory accesses to the memory management unit in a single cycle would have a similar problem since unbounded number of registers should be used to store these memory accesses.

Considering the aforementioned challenges, a more practical idea is to utilize an abstraction (aka. specification) of the virtual-memory management unit in the formal verification of RISC-V ISA processors and verify the virtual memory management unit with respect to the specification separately, which we decide to investigate in the future. As a result, in this work, for out-of-order superscalar processors, we restrict our attention to arithmetic and control-flow instructions that occur in most RISC-V processors.

Table 3
Signals compared for equivalence checking under StateCheck and ActionCheck.

Types of signals	StateCheck	ActionCheck
Instruction	PC_{now}	PC_{next}
General Regs	All general regs (x0 - x31)	`rd`
CSRs	All implemented CSR regs	
Exception	Valid, No, PC_{now}, Inst	
Memory	Valid, Address, WriteData, Width	
PageTable	Valid, Address, WriteData, Width	

4.5. Equality checking in synchronized Chisel designs

To verify whether the implementation of each instruction in the DUT conforms to its RISC-V ISA specification, it is necessary to check whether the results after the execution of the instruction in the DUT and in the reference model are equal. Table 3 shows the set of signals of the DUT that are involved in the equality checking. Usually, the signals in the DUT should be preprocessed before the equality checking. The preprocessing involves extracting related signals (e.g., destination register address and PC values) and synchronizing their timing to align with instruction commit points. After that, the DUT and the reference model are synchronized during which assertions are added to check equality of considered signals between them. The number of instantiated equivalence assertions is tool-parameter dependent, with up to 80 assertions automatically generated. Finally, the synchronized Chisel design together with assertions is used to generate model-checking instances.

5. Evaluation

In this section, we first evaluate the effectiveness and efficiency of our approach and then compare the performance of our approach against other two open-source state-of-the-art tools for formal verification of RISC-V processor designs, namely, riscv-formal [12] and the symbolic-execution based tool in [17,18]. Both tools have been used in formally verifying several RISC-V processor designs and found real-world bugs [18,29]. (We do not compare with the commercial tools FormalISA [13] and OneSpin [14], as they are not publicly available.)

5.1. Benchmarks and experimental setup

RISC-V processor Chisel designs. We use three representative open-source RISC-V processor Chisel designs as evaluation subjects: riscv-mini [6], NutShell [7] and BOOM [5].

- **riscv-mini** is a simple *single-issue in-order* RISC-V processor with a 3-stage pipeline, comprising 3499 lines of Chisel code in total. The architecture of riscv-mini is shown in Fig. 5. It supports RV32I and the machine-level ISA. It also contains simple instruction caches (ICache) and data caches (DCache).
- **NutShell** is a *single-issue in-order* RISC-V processor with a 9-stage pipeline, comprising 8859 lines of Chisel code in total. The architecture of NutShell is shown in Fig. 6. It supports various instruction extensions such as I, M, A, C, Zicsr, and Zifencei. Moreover, it includes three privilege levels, namely, M, S, and U. It also supports the Sv39 virtual memory system, and implements TLB to translate virtual addresses to physical addresses. As a result, it is capable of running Linux. NutShell is configurable, e.g., it can be configured as a 32-bit processor (with a 5-stage pipeline).
- **BOOM** (Berkeley Out-of-Order Machine) is a *multi-issue out-of-order* RISC-V processor with a 7-stage pipeline, comprising 13,250 lines of Chisel code. The architecture of BOOM is shown in Fig. 7. Apart from RV64GC and three privilege levels, it also supports the Sv39 virtual memory system and implements TLB. BOOM is based on the RocketChip RISC-V processor [4] while enhancing out-of-order execution capabilities. The micro-architecture of BOOM is parameterizable with multiple configurations. Based on the MediumBoom (a dual-issue configuration), we minimized the size of L1 cache (nSets = 2, nWays = 1) and disabled the floating-point unit to maintain full functionality while keeping the design tractable for automated formal verification.

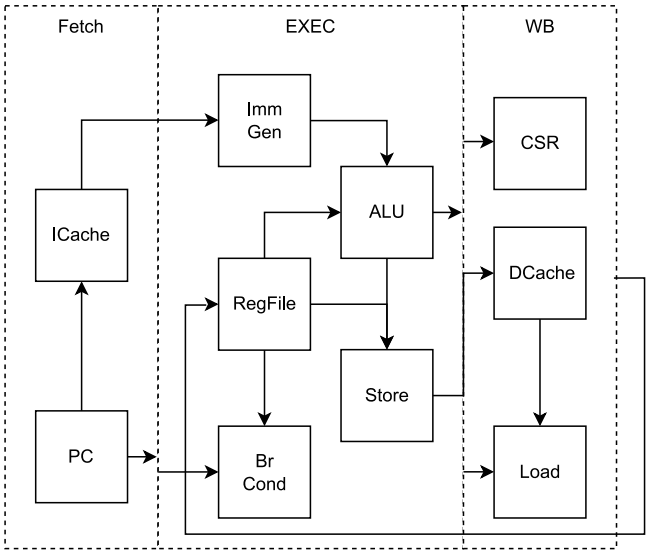

Fig. 5. The architecture of riscv-mini.

Table 4
Architecture details of RISC-V processor designs.

	riscv-mini	NutShell	BOOM
ISA	RV32I	RV64IMACZicsr Zifencei	RV64IMACZicsr Zifencei,Zihpm
Architecture	single-issue in-order	single-issue in-order	dual-issue out-of-order
Pipeline	3 stages	9 stages	7 stages
Cache	I/D Cache	✗	I/D Cache
TLB	✗	✓	✓
Privilege	M	M, S, U	M, S, U
Branch Prediction	✗	✓	✗

Details of these three RISC-V processor designs used for evaluation are summarized in Table 4.

Model-checking instances. To reduce the conformance checking of RISC-V processor designs w.r.t. RISC-V ISA specifications to model-checking instances that can be solved by off-the-shelf model checkers, we allow users to specify the set of interested RISC-V instructions and use the Chisel assume statements to restrict the op-codes of RISC-V instructions in both the DUT and the reference model to the interested ones. After that, the DUT and the reference model are synchronized in which assertions are added to check equality of considered signals between them. The synchronized Chisel design together with assertions is used to generate model-checking instances, where a model-checking instance is generated and stored in the BTOR2 [21] or SMT2 [30] format for each assertion property. The model-checking instances of riscv-mini and NutShell are generated directly from the synchronized Chisel designs by leveraging ChiselTest's `chiseltest.formal.verify` function [20] in the BTOR2 format, where each model-checking instance of riscv-mini contains 3395 state bits and 378 input bits, and each model-checking instance of NutShell contains 71,581 state bits and 7312 input bits. In contrast, the model-checking instances of BOOM are generated by first translating the synchronized Chisel design into SystemVerilog and then translating SystemVerilog design

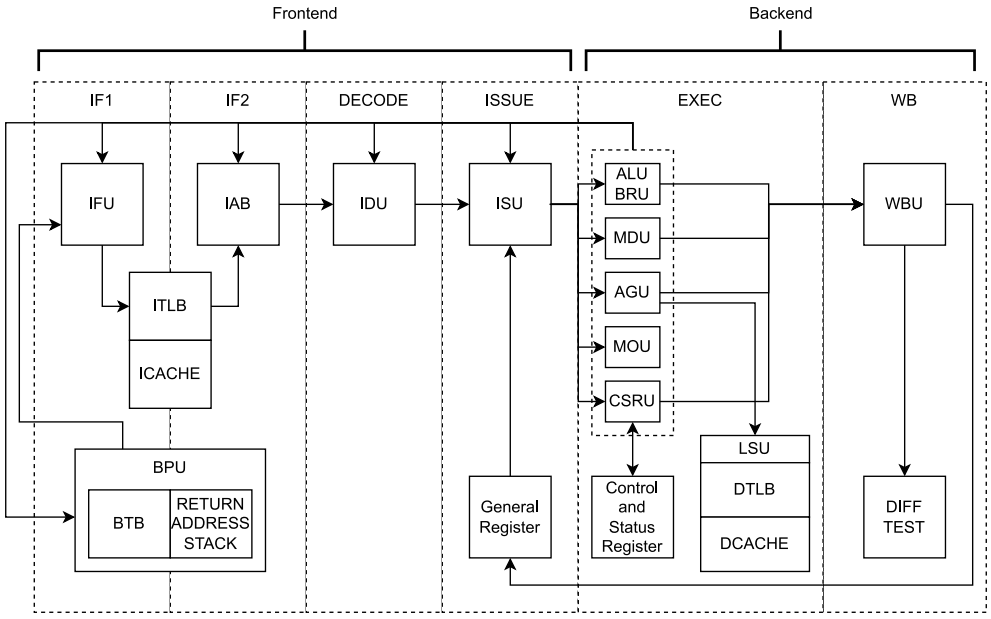

Fig. 6. The architecture of NutShell.

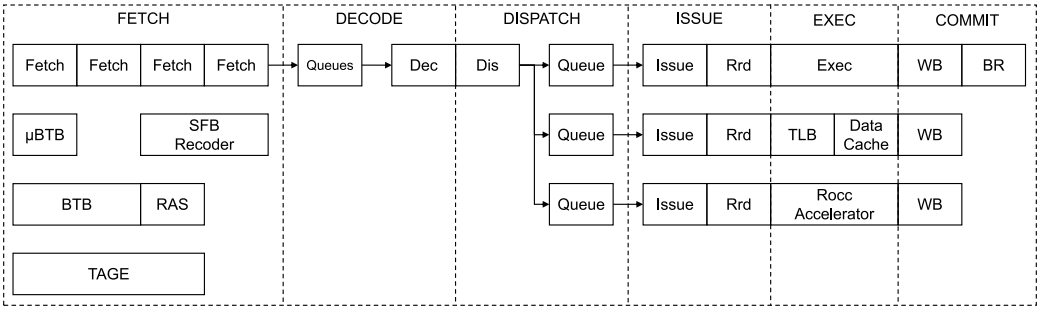

Fig. 7. The architecture of BOOM.

into model-checking instances via SymbiYosys [16] in the SMT2 format, where each model-checking of BOOM contains 45,168 state bits and 19,475 input bits. We note that both ChiselTest and SymbiYosys failed to generate BTOR2-format model-checking instances for BOOM, where ChiselTest was unable to convert the BoomTile module into a transition system due to BOOM's highly parameterized architecture, and SymbiYosys generated invalid BTOR2-format model-checking instances that contained bit vectors with zero length, violating the BTOR2 specification, rather than being caused by memory exhaustion or excessively long conversion times. Meanwhile, SMT2-format model-checking instances for both riscv-mini and NutShell can be generated via SymbiYosys which are indeed used for a fair scalability comparison with riscv-formal.

Experimental setup. In this work, we primarily use the model checker Pono [15] for verifying riscv-mini and NutShell, as the BTOR2-format model-checking instances generated by ChiselTest are compatible with Pono. For BOOM whose model-checking instances are in the SMT2 format, we have to use the model checker SMTBMC [16], because Pono only accepts BTOR2-format model-checking instances. We use Pono for riscv-mini and NutShell because it is more recent than SMTBMC. When comparing the scalability with riscv-formal, to be fair, the model checker SMTBMC is used for SMT2-format model-checking instances of riscv-mini and NutShell. We utilize the bounded model-checking (BMC) engine of both Pono and SMTBMC to solve the model-checking instances,[6] with Boolector as the backend solver, where the maximal step bound is set to 40 with a timeout of 24 h (see Table 5). All the model-checking experiments were run on a machine with Ubuntu 20.04LTS, 64 GiB RAM, 24 Intel(R) Core™ i9-13980HX CPU @ 5.60 GHz processors.

Table 5
Experimental configuration.

DUT	Format	Model Checker	Backend Solver
Instruction-set conformance checking			
riscv-mini	BTOR2	Pono	Boolector
NutShell	BTOR2	Pono	Boolector
BOOM	SMT2	SMTBMC	Boolector
Scalability evaluation (fair comparison with riscv-formal)			
riscv-mini	SMT2	SMTBMC	Boolector
NutShell	SMT2	SMTBMC	Boolector
BOOM	SMT2	SMTBMC	Boolector
Algorithm: BMC	Max bound: 40	Timeout: 24 h	

[6] We also tried some other model-checking algorithms (e.g., PDR and k-induction) to verify the model-checking instances of riscv-mini and NutShell. It turns out that the model-checking instances are too large to be verified in a reasonable amount of time (24 h).

Table 6
Bugs found in riscv-mini and NutShell.

Bug	Step	Time	Assume Insts	Insts	Hex	VM
R:E1	4	0.52 s	RV32I	BLTU x29, x1, −2	0xFE1EEFE3	✗
R:E2	5	2.58 s	RV32I	SW x1, 38(x1)	0 × 0210A323	✗
				SB x31, −1(x5)	0xFFF28FA3	
N:E1	10	2.56 s	RV32/64I	LD x4, −377(x1)	0xE870B203	✗
N:E2	29	47 m 27.66 s	Load/Store	SD x1, 1976(x19)	0 × 7A19BC23	✓
N:E3	19	11 m 25.44 s	Zicsr	SRET	0 × 10200073	✗
			MRET, SRET	SRET	0 × 10200073	
			ECALL, EBREAK			
N:E4	29	97 m 31.34 s	Load/Store	SD x0,48(x22)	0 × 020B3823	✓
N:E5	29	67 m 11.29 s	Load/Store	LD x0,384(x2)	0 × 18013003	✓

5.2. Case studies on riscv-mini, NutShell and BOOM

We apply our approach χRVFormal to check ISA conformance of riscv-mini, NutShell and BOOM w.r.t. the official RISC-V ISA specifications, and find *2 RISC-V ISA non-conformance bugs* in riscv-mini and *5 RISC-V ISA non-conformance bugs* in NutShell. The 2 bugs found in riscv-mini have been reported to the designers as GitHub issues.[7] The 5 bugs found in NutShell have been confirmed by the designers.

Table 6 reports the statistics of the experiments related to the 7 bugs,[8] where columns **Step** and **Time** show the number of steps and the execution time respectively when the bug was found, column **Assume** shows the instructions that are enforced by the assume statements, and column **Insts** reports the instruction represented by the counterexample when the bug was found, and column **VM** shows whether the virtual memory is enabled or not. The label R/N in the name of a bug represents that it is a bug of riscv-mini/NutShell.

We would like to remark that when the virtual memory is enabled in NutShell, we set the value of the first 20 bits of the `satp` register[9] to be `h80000` and the value of the `mstatus` register[10] to be `h000E0800`. Among the 5 bugs found in NutShell, the bugs N:E4 and N:E5 are correlated. When the bug N:E4 was found, the number of steps in the BMC algorithm is 29 and the execution time is 97m31.34s. Then we adapted the assume statement to enforce that `PPN[1:0]` is 0,[11] thus bypassing the bug N:E4. Finally, we reran the model checker Pono and discovered the bug N:E5 after 67m11.29s.

In the sequel, we describe more details about the 2 bugs found in riscv-mini and 5 bugs found in NutShell.

R:E1. Instruction-address-misaligned exceptions are missed for branch instructions: riscv-mini enforces the instruction address alignment only for (unconditional) jump instructions (e.g., `JAL`), but not for (conditional) branch instructions (e.g., `BLTU`). As a result, when the addresses are misaligned in branch instructions, no exceptions will be triggered.

R:E2. Store address-misaligned exception does not flush the subsequent store instructions correctly: When the address in a store instruction is misaligned and an exception is triggered, riscv-mini does not flush the subsequent store instructions correctly. For instance, if the instruction `SW x1, 38(x1)` is followed by the instruction `SB x31, -1(x5)`, then although the execution of the first store instruction triggers an address-misaligned exception, the second store instruction, which should be flushed and not be executed, will still be executed partially in riscv-mini and a write request will be issued to the memory. As we shall see in Section 5.3, riscv-formal fails to detect this bug, indicating that even on unprivileged instructions, riscv-formal can miss some bugs.

N:E1. Non-writable high bits of the mtval register: When an exception occurs, the high bits of the `mtval` register (more precisely, `mtval[63:39]`) in NutShell are not writable, which violates the RISC-V ISA specification.

N:E2. Invalid PPN [2] bits in Sv39 physical addresses: In RISC-V ISA specification, the physical addresses in the Sv39 virtual memory system should be 56 bits, while in NutShell, Sv39 physical addresses have only 32 bits.

N:E3. Missing privilege checking in the execution of xRET instructions: According to the RISC-V ISA specification, an `xRET` instruction (where `x` = M, S) can only be executed in a privilege mode that is the same as or higher than `x`. When the initial value of the `mstatus` register is set to be `h00001800`, and two consecutive `SRET` instructions are executed, after the execution of the first `SRET` instruction, the privilege mode is changed to U, as a result, the execution of the second `SRET` instruction should raise an exception. Nevertheless, NutShell fails to do so.

N:E4. Missing super-page checking in virtual address translation: According to Step 6 of the virtual address translation process in the RISC-V ISA specification (see Section 4.3.2 of [2]), some lower `PPN` bits of the Sv39 page table entries (PTE) should be 0 during the page-table translation. For instance, during the first translation (where level = 2), `PPN[level-1:0]` (i.e. `PPN[1]` and `PPN[0]`) should be kept as 0. A violation of this restriction would raise a page-fault exception. Nevertheless, NutShell does not handle such unaligned super-page exceptions.

N:E5. Missing some corner cases on X, W, R, V bits of PTE: To determine whether a leaf PTE has been found, the processor should do some checks according to Steps 3 and 4 in the RISC-V ISA specification (Page 82, Volume 2). NutShell implements these checks by using nested conditional statements. After converting the counterexample reported by the Pono model checker to the truth table (see Table 7), we can see that in the third case, according to the RISC-V specification, a page-fault exception should be raised, while NutShell fails to do so. We should emphasize that a single corner case among 16 combinations of X, W, R, V bits is hard to be discovered by simulation or testing, which shows that formal verification is capable of discovering some deep bugs that might be elusive for informal verification methods (e.g. simulation or testing).

5.3. Comparison with the other verification approaches

As shown above, our approach is able to discover real-world unknown bugs in the two open-source processor designs riscv-mini and NutShell. In the sequel, we compare the performance of our approach with the other two verification approaches of RISC-V processor designs, riscv-formal [12] and the symbolic-execution based approach in [17, 18], using their publicly available tools.

[7] Please refer to the issues 71 and 72 at https://github.com/ucb-bar/riscv-mini/issues.

[8] The bugs N:E1 and N:E2 reveal that the values of some signals' bit-widths in NutShell do not conform to the RISC-V ISA specification, although they can be seen as the pragmatic choices made by designers.

[9] The `satp` register is a read/write register that controls the supervisor-mode address translation and protection. It only exists when the supervisor mode is enabled.

[10] The `mstatus` register is a read/write register that keeps track of and controls the core's current operating state.

[11] PPN denotes the physical page number.

Table 7
Truth table of X, W, R, V bits of Bug E5's PTE Flag.

X	W	R	V	Spec	NutShell
0/1	0/1	0/1	0	Page Fault	Page Fault
0	1	0	1	Page Fault	Page Fault
1	1	0	1	Page Fault	None
1	0	0	1	Found	Found
0/1	0/1	1	1	Found	Found
0	0	0	1	Not Found	Not Found

5.3.1. Comparison with riscv-formal

We first compare the bug-detection capability of our approach with riscv-formal, then compare their scalability.

Comparison of the bug-detection capability. We compare the detection capability of our approach with riscv-formal by checking whether riscv-formal can discover the 7 bugs found by our approach (see Table 6). It turns out that it can only discover 2 out of 7 bugs.

- Among the 2 bugs found in riscv-mini by our approach, i.e. the bugs R:E1 and R:E2, riscv-formal fails to find R:E2, although it can find R:E1. It is because riscv-formal checks the ISA conformance *only* when an instruction is committed, but the flushed instruction in R:E2 is not committed, thus fails to detect. In contrast, our approach checks the conformance when memory read/write requests occur, thus does not miss the bug R:E2.
- Among the 5 bugs found in NutShell by our approach, i.e. N:E1–N:E5, riscv-formal cannot discover the four bugs (N:E2–N:E5) related to virtual memories or privileged instructions, namely, it can only find the bug N:E1.

Furthermore, the RVFI interface on which riscv-formal relies on is much heavier for the developers, compared to the interface in our approach, because it requires much more information from the DUT than ours.

Comparison of the scalability. We compare the scalability of our approach with riscv-formal on riscv-mini, NutShell and BOOM, by checking the conformance of the RV32I instruction set on riscv-mini, and the conformance of RV32/64I instruction sets on NutShell and BOOM. To be fair, the scalability evaluation is conducted under the same toolchain, namely, we use SMT2-format model-checking instances and the model checker SMTBMC with Boolector [21] as the backend SMT solver. To facilitate the comparison, we manually inject into each of the three processor designs the following five bugs for RISC-V Integer instructions.

#1. **Unsigned comparison for SLTI.** The SLTI (Set Less Than Immediate) instruction conducts a signed comparison between a register and an immediate value. We inject the fault by neglecting the sign and apply an unsigned comparison.
#2. **SUB with the highest bit set to 0.** We modify the implementation of the SUB (SUBtraction) instruction by setting the highest bit of the result to 0.
#3. **BNE to BEQ.** We modify the implementation of the BNE (Branch Not Equal) instruction to the BEQ (Branch Equal) instruction.
#4. **Incorrect signed comparison in BLTU.** The BLTU instruction compares two registers as unsigned integers. We manually introduced a bug by modifying it to use signed comparison, causing incorrect branching for large unsigned values.
#5. **ADDI with the lowest bit set to 0.** We modify the implementation of the ADDI instruction by setting the lowest bit of the result to 0.

We would like to remark that although all three processors, i.e. riscv-mini, Nutshell, and BOOM are injected with the same bugs and only a few instructions are covered by these injected bugs, the execution of these instructions in each of the three processors will go through the whole pipeline in this processor (since the executions of all the instructions in a processor share the same micro-architecture of this processor), thus the complexity of the micro-architecture or the size of the state space, indeed infects the performance of the formal verification.

Moreover, to be fair, all processors are verified using the same instruction-set constraints (e.g., RV32I for riscv-mini and RV32/64I for NutShell and BOOM), that is, we add the "assume" statements so that the processors only commit the instructions satisfying these constraints.

Then we run our approach and riscv-formal to detect these bugs and compare their performance. Recall that when attempting to find a non-conformance bug for a DUT, we reduce the problem into the problem of checking multiple assertions against the DUT and create multiple model-checking instances, one instance for each assertion. In contrast, riscv-formal generates one model-checking instance per RV32/64I instruction, and verifies the correctness of that instruction via its corresponding assertion. After generating all model-checking instances, both approaches run multiple SymbiYosys [16] processes on the 24 processors in parallel. Both our approach and riscv-formal will stop when any of their processes finds a bug.

The experimental results are reported in Table 8. Note that we record the time when the first bug is reported.

From the results, we can see that on the smallest design, i.e., riscv-mini, our approach performs worse than riscv-formal, primarily because our approach generates more model-checking instances than riscv-formal, thus takes longer time. In contrast, on NutShell, our approach is comparable with to riscv-formal. On the largest design, i.e., BOOM, our approach significantly outperforms riscv-formal, because the cost of model-checking dominates the overall verification time. When compared the verification strategy StateCheck with ActionCheck, we can observe that ActionCheck in general outperforms StateCheck.

On BOOM, the large discrepancy in execution time of riscv-formal for different bugs is mainly due to its parallel execution approach, which checks RV32/64I instructions in the alphabetical order. For example, issues such as the *#5.* ADDI bug are quickly detected, while others, like the *#2.* SUB bug, take significantly longer. Additionally, riscv-formal checks 49 tasks per channel, leading to a total of 98 tasks for BOOM's two channels that must be processed in parallel. In contrast, our approach only requires 16 tasks, independent of the number of channels, thus the task execution time is not influenced by the alphabetical order of the instructions.

5.3.2. Comparison with the symbolic-execution based approach

We compare the performance of our approach with the symbolic-execution based approach in [17,18]. For readability, we recall the workflow of the symbolic-execution based approach in the sequel.

- At first, the RISC-V processor design in the SpinalHDL (an open-source high-level hardware description language similar to Chisel) is translated into Verilog using SBT (a build tool for Scala projects).
- The Verilog design is translated into C++ using the tool Verilator.
- A Voter module is utilized to synchronize the C++ design with a C++ RISC-V ISS (Instruction Set Simulator), where the RISC-V Formal Interface (RVFI) [31] is added to the RISC-V processor to facilitate its connection to the Voter module.
- The synchronized C++ design is compiled into LLVM bytecode by Clang.

Table 8
Scalability comparison (execution time in seconds) between our approach and riscv-formal.

Processor	Bug	χRVFormal (StateCheck)		χRVFormal (ActionCheck)		riscv-formal	
		Seconds	Step	Seconds	Step	Seconds	Step
riscv-mini	#1	19.86	4	19.45	4	**3.43**	4
	#2	28.57	4	14.04	4	**1.81**	4
	#3	31.27	5	10.58	4	**3.41**	4
	#4	23.25	5	10.95	4	**3.50**	4
	#5	20.65	4	13.22	4	**3.36**	4
	Avg.	24.72	4.4	13.65	4	**3.10**	4
NutShell	#1	**76.21**	10	98.72	10	96.40	10
	#2	**78.87**	10	150.26	10	97.04	10
	#3	151.71	14	**66.29**	10	100.00	10
	#4	171.00	14	**80.98**	10	95.94	10
	#5	**76.90**	10	96.79	10	95.05	10
	Avg.	110.94	11.6	98.61	10	**96.89**	10
BOOM	#1	N.A.	N.A.	**2178.03**	20	45 124.31	20
	#2	N.A.	N.A.	**2251.92**	20	55 725.25	20
	#3	N.A.	N.A.	**1717.55**	20	8139.99	20
	#4	N.A.	N.A.	**1785.99**	20	3895.47	20
	#5	N.A.	N.A.	1831.89	20	**1390.65**	20
	Avg.	N.A.	N.A.	**1953.08**	20	22 855.13	20

- Finally, the dynamic symbolic-execution engine KLEE is harnessed for verification.

In [18], a non-pipelined RISC-V processor called MicroRV32 was verified. MicroRV32 supports RV32-IMC, but lacks Cache and TLB modules. It consists of 3193 lines of Verilog code and 7497 lines of C++ code (after translating Verilog to C++).

At first, we attempt to use the symbolic-execution based approach to detect the RISC-V specification non-conformance bugs in riscv-mini and NutShell that were found by our approach (see Table 6).

We translate the NutShell and riscv-mini Chisel designs into SystemVerilog, add the RVFI, and establish the connections to the Voter module, so that KLEE can be utilized eventually to perform symbolic-execution. We set the time bound to 86,400 s (or 24 h) for detecting these bugs. It turns out that *the symbolic-execution based approach discovers only one bug in Table 6, namely R:E1, within the time 58 m 39 s (Recall that R:E1 can be discovered by our approach in 0.52 s)*. Note that the C++ RISC-V ISS does support privileged instructions and the poor performance of the symbolic-execution based approach is due to its poor scalability. Compared with MicroRV32, riscv-mini has 2349 lines of Verilog code and 8155 lines of C++ code, while NutShell has 18,039 lines of Verilog code and 19,513 lines of C++ code, after translating these Chisel designs into Verilog, then to C++ with Verilator [32]. Note that although the code sizes of riscv-mini and MicroRV32 are comparable, riscv-mini contains a *3-stage pipeline*, while MicroRV32 is a *non-pipelined* processor, therefore, these bugs are hard to detect in riscv-mini by the symbolic-execution based approach.

With the idea that the manually injected bugs might be easier to detect, we also manually inject into riscv-mini and NutShell the 5 bugs for RV32I instructions in Table 8. We also set the time bound to 86,400 s (or 24 h). It turns out that *the symbolic-execution based approach can discover only two bugs in riscv-mini, namely the bugs #3 and #4, within the time 85m59s and 1356m10s, respectively (Recall that the two bugs can be discovered by our approach in 10.58s and 10.95s, respectively)*. It fails to discover the other 3 bugs in riscv-mini and all the 5 bugs in NutShell. These experimental results demonstrate that our approach is significantly more effective and efficient than the symbolic-execution based approach in the bug detection.

5.3.3. Attempts of software-model-checking based approach

There is also another potential approach of formal verification of RISC-V CPU Chisel designs: translating the DUT into Verilog, then to C/C++, synchronizing the C/C++ design with the RISC-V reference model in C/C++, and finally utilizing software model checkers for C/C++ to fulfill the verification. Several reference models based on C/C++ or SystemC have been designed for RISC-V ISA, including Spike [33], QEMU [34], and riscv-vp [35].

Compared with this software-model-checking based workflow, the primary advantage of our approach is that it is naturally enabled by Chisel. Since the DUT itself is implemented in Chisel, expressing both the DUT and the reference model within the same hardware framework avoids cross-language translation and supports a more faithful verification flow.

Moreover, it turns out this software-model-checking based approach is *not* mature enough for a comparison. We tried two tools that translate Verilog to C/C++, namely, V2C [36] and Verilator [32].

- When we use V2C to translate the Verilog design (generated from the Chisel design) of NutShell into C, the translation fails and only an incomplete Verilog design is produced. Thus, V2C is not yet ready for translating real-world RISC-V Verilog processors such as NutShell.
- When we use Verilator to translate the Verilog design of NutShell into C++, the translation succeeds and produces a complete C++ design. Nevertheless, the generated C++ design contains some new features in C++11, which are not yet supported by the state-of-the-art software model checkers for C/C++, i.e., CBMC [37], ESBMC [38], SMACK [39], and CPAchecker [40].

Even if these engineering challenges were resolved, the translation itself incurs additional burden on the formal verification, since a single hardware clock cycle of the DUT should be simulated by a sequence of multiple software program statements and the number of execution steps in the BMC of the resulting software programs is considerably larger than the number of clock cycles in the BMC of DUT. The experiment results in Section 5.3.2 echoes this observation: When using Verilator to translate the DUT into a C++ design and applying the symbolic-execution engine KLEE to check the conformance of the DUT with the reference model riscv-vp, a bug is discovered in 58m39s, whereas our approach detects the same bug in only 0.52s. This huge

Table 9
Comparison of RISC-V reference Models.

Model	Spike [33]	riscv-sail [42]	riscv-formal [12]	riscv-vp [35]	χRVFormal
Language	C/C++	Sail C/C++	Verilog/SystemVerilog Python	C/C++	Chisel
#SLOC	48,426	Sail: 18,343 C/C++: 28,640	Verilog/SystemVerilog: 104,911 Python: 3523	30,175	4609
Inst. set	All	All	RV32/64IB	RV32/64I MAFDC	RV32/64I MCBZicsr
Privilege	✓	✓	✗	✓	✓
Vitrual Memory	✓	✓	✗	✓	✓
Simulation	✓	✓	✗	✓	✓
Formal Verification	✗	✗	✓ (Model-checking)	✓ (Dynamic symbolic-execution)	✓ (Model-checking)

performance gap indicates that the software-based formal verification approaches for hardware verification is much less scalable than the formal-verification approaches directly in the hardware level (such as our approach).

6. Related work

Formal verification has been widely studied and adopted for checking functional correctness of software and hardware designs [41]. Hereafter, we only discuss formal verification studies on RISC-V and Chisel.

As aforementioned, the open-source tool riscv-formal and the commercial tools FormalISA and OneSpin verify Chisel designs by first compiling them into low-level Verilog/SystemVerilog and then checking their correctness using Verilog/SystemVerilog reference models or SVA. Instead, we develop a high-level, modular, and parameterized reference model, that is a RISC-V processor design generator in Chisel. It is more comprehensive yet more succinct than the Verilog/SystemVerilog reference model in riscv-formal, thus more convenient for maintenance, diagnosis, and extension of the reference model. We also propose the first end-to-end formal verification approach for RISC-V processor Chisel designs, which is more effective than riscv-formal with comparable verification efficiency (cf. Section 5). Table 9 summarizes a detailed comparison of different RISC-V reference models with respect to the programming language of the model and configuration script, the number of source code lines, featured instruction sets and privilege mode, accompanied simulation and formal verification tools. It is easy to observe that our reference model is the most succinct one. Although Spike and riscv-sail supports all the instruction sets, they cannot be used for formal verification. While riscv-vp supports some instruction sets that we do not support, it only supports symbolic-execution based verification which is less effective and less efficient than ours as demonstrated in Section 5.3.2.

Specifically, the additional instruction sets supported by riscv-vp include the atomic extension (A) and the single- and double-precision floating-point extensions (F and D). All these instructions can be functionally executed in riscv-vp either concretely or symbolically. However, dynamic symbolic execution differs from model checking and it would be computational expensive to verify DUT with these instructions using model-checking. Indeed, atomic behavior can only be observed in a multi-core verification context and floating-point operations (F and D) require a large number of clock cycles to complete, all exceeding the practical capacity of bounded model checking.

Formal verification of CHERI-RISC-V processor designs was studied using SVA [43], where CHERI-RISC-V[12] is an extension of RISC-V ISA with an alternative model to offer memory safety. To reduce the manual effort of ISA modeling, a trace notation was proposed and used to model the RISC-V ISA [44], based on which a complete set of SVA properties are generated for detecting functional bugs in RISC-V processor designs.

KLEE, a dynamic symbolic-execution engine, has been utilized to analyze RISC-V processor designs in SpinalHDL [17,18]. They compile SpinalHDL designs into Verilog which in turn are compiled into C++. The RISC-V processor design in C++ is synchronized with riscv-vp [35] in C++ as the reference model, on which dynamic symbolic-execution is applied to find non-conformance bugs. Dynamic symbolic-execution often suffers from the path-explosion problem when the DUT has many conditional branches, thus limited in efficiency for a high coverage analysis. The experimental results show that our approach is significantly more effective and efficient than this approach (cf. Section 5).

Recently, BDD-based formal verification was leveraged to verify the correctness of *non-pipelined* RISC-V processor designs with polynomial time and space complexity [45]. While efficient, their approach and closed-source tool cannot be applied to verify *pipelined* RISC-V processor designs such as riscv-mini and NutShell, due to the lack of support for clocks.

Informal verification (i.e., testing and simulation) for the functional correctness of RISC-V processor designs has been studies as well, e.g., [46–48]. There is also a trend of adapting effective software testing techniques (e.g. concolic testing and fuzzing) to the informal verification of hardware designs [49–53]. Nevertheless, they are orthogonal to this work.

7. Conclusion and future work

In this work, we have proposed the first end-to-end formal verification approach for the functional correctness of RISC-V processor Chisel designs, that is, whether a RISC-V processor Chisel design conforms to the RISC-V ISA specifications, where both unprivileged and privileged instructions are taken into account. In particular, we developed a succinct, modular, and parameterized RISC-V reference model in Chisel. We validated the effectiveness and efficiency of our approach on three representative open-source RISC-V processor designs. Our approach found 7 real-world unknown RISC-V specification non-conformance bugs in riscv-mini and NutShell. The experimental results show that our approach can discover more bugs than the state-of-the-art open-source formal verification approaches, i.e. riscv-formal and symbolic-execution based approach. Moreover, our approach is considerably more efficient than them. We should emphasize that our reference model is of independent interest and might be used in some other verification approaches, e.g. simulation, emulation, and fuzzing.

[12] CHERI is an abbreviation of "Capability Hardware Enhanced RISC Instructions".

For the future work, we would like to validate our approach on more open-source RISC-V Chisel designs. It is also interesting to combine the formal and informal approaches in order to achieve a nice balance between accuracy and efficiency in the verification. Moreover, our current framework does not yet fully support synchronization of complex signals (e.g., virtual memory operations) in out-of-order superscalar processors. Designing dedicated synchronization mechanisms for such architectures will also be an important direction for future work.

CRediT authorship contribution statement

Shidong Shen: Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Shijie Chen:** Methodology. **Yicheng Liu:** Validation, Software, Methodology, Conceptualization. **Lijun Zhang:** Writing – review & editing, Supervision. **Fu Song:** Writing – review & editing, Supervision, Methodology, Investigation, Formal analysis. **Zhilin Wu:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320101.

References

[1] A. Waterman, K. Asanović, The RISC-V instruction set manual volume I: Unprivileged ISA version 20191213, 2019.
[2] A. Waterman, K. Asanović, J. Hauser, The RISC-V instruction set manual volume II: Privileged architecture version 20211203, 2021.
[3] J. Bachrach, H. Vo, B.C. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, K. Asanovic, Chisel: constructing hardware in a scala embedded language, in: Proceedings of the 49th Annual Design Automation Conference, DAC, 2012, pp. 1216–1225.
[4] Rocket chip RISC-V CPU generator, 2023, URL https://github.com/chipsalliance/rocket-chip.
[5] RISC-V boom: The berkeley out-of-order RISC-V processor, 2023, URL https://github.com/riscv-boom/riscv-boom.
[6] D. Kim, Riscv-mini, a simple RISC-V 3-stage pipeline written in chisel, 2017, URL https://github.com/ucb-bar/riscv-mini.
[7] NutShell RISC-V CPU, 2019, URL https://github.com/OSCPU/NutShell.
[8] Xiangshan: An open-source high-performance RISC-V processor, 2023, URL https://github.com/OpenXiangShan/XiangShan.
[9] Y. Bao, T.E. Carlson, Agile and open-source hardware, IEEE Micro 40 (4) (2020) 6–9, http://dx.doi.org/10.1109/MM.2020.3002606.
[10] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, K. Asanovic, An agile approach to building RISC-V microprocessors, IEEE Micro 36 (2016) http://dx.doi.org/10.1109/MM.2016.11, 1–1.
[11] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen, L. Gou, Y. Jin, Q. Li, X. Li, Z. Li, et al., Towards developing high performance RISC-V processors using agile methodology, in: Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture, MICRO, 2022, pp. 1178–1199.
[12] YosysHQ, RISC-V formal verification framework, 2016, URL https://github.com/YosysHQ/riscv-formal.
[13] Axiomise, FormalISA: RISC-V formal verification, 2023, URL https://www.axiomise.com/riscv-formal-app/.
[14] OneSpin formal verification solutions - siemens EDA, 2024, https://eda.sw.siemens.com/en-US/ic/questa/onespin-formal-verification/.
[15] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, C. Barrett, Pono: a flexible and extensible SMT-based model checker, in: Proceedings of the 33rd International Conference on Computer Aided Verification, CAV, 2021, pp. 461–474.
[16] C. Wolf, et al., SymbiYosys, 2022, URL https://symbiyosys.readthedocs.io/.
[17] N. Bruns, V. Herdt, R. Drechsler, Symbolic execution framework for RISC-V processor verification, 2023, URL https://github.com/agra-uni-bremen/symex_processor_verification.
[18] N. Bruns, V. Herdt, R. Drechsler, Processor verification using symbolic execution: A RISC-V case-study, in: 2023 Design, Automation & Test in Europe Conference & Exhibition, DATE, 2023, pp. 1–6, http://dx.doi.org/10.23919/DATE56975.2023.10137202.
[19] S. Shen, Y. Liu, L. Zhang, F. Song, Z. Wu, Formal verification of RISC-V processor chisel designs, in: Proceedings of Dependable Software Engineering. Theories, Tools, and Applications: 10th International Symposium, SETTA, 2024, pp. 142–160.
[20] K. Laeufer, J. Bachrach, K. Sen, Open-source formal verification for chisel, in: Proceedings of the Fourth Workshop on Open-Source EDA Technology, WOSET, 2021.
[21] A. Niemetz, M. Preiner, C. Wolf, A. Biere, Btor2, btormc and boolector 3.0, in: Computer Aided Verification - 30th International Conference, CAV, 2018, pp. 587–595.
[22] A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: Tools and Algorithms for the Construction and Analysis of Systems, TACAS, 1999, pp. 193–207.
[23] M. Sheeran, S. Singh, G. Stålmarck, Checking safety properties using induction and a SAT-solver, in: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, FMCAD, 2000, pp. 127–144.
[24] A.R. Bradley, SAT-based model checking without unrolling, in: Verification, Model Checking, and Abstract Interpretation, VMCAI, 2011, pp. 70–87.
[25] P.S. Li, A.M. Izraelevitz, J. Bachrach, Specification for the FIRRTL Language, Tech. Rep. UCB/EECS-2016-9, EECS Department, University of California, Berkeley, 2016, URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html.
[26] Riscv-tests, 2015, URL https://github.com/riscv-software-src/riscv-tests.
[27] V. Herdt, R. Drechsler, Advanced virtual prototyping for cyber-physical systems using RISC-V: implementation, verification and challenges, Sci. China Inf. Sci. 65 (1) (2022) 110201.
[28] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, A. Zaidi, End-to-end verification of processors with ISA-formal, in: Computer Aided Verification: 28th International Conference, CAV, 2016, pp. 42–58.
[29] C. Wolf, Formal verification of RISC-V cores with riscv-formal, 2018, Available at: https://riscv.org/wp-content/uploads/2018/12/13.30-Humbenberger-Wolf-Formal-Verification-of-RISC-V-processor-implementations.pdf.
[30] C. Barrett, A. Stump, C. Tinelli, et al., The smt-lib standard: Version 2.0, SMT, in: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, vol. 13, 2010, p. 14.
[31] RISC-V formal interface (RVFI), 2020, https://github.com/SymbioticEDA/riscv-formal/blob/master/docs/rvfi.md.
[32] Verilator: Open-source SystemVerilog simulator and lint system, 2024, URL https://github.com/verilator/.
[33] Spike, a RISC-V ISA simulator, 2023, URL https://github.com/riscv-software-src/riscv-isa-sim.
[34] QEMU: A generic and open source machine emulator and virtualizer, 2023, URL https://www.qemu.org/.
[35] RISC-V based virtual prototype (VP), 2023, URL https://github.com/agra-uni-bremen/riscv-vp.
[36] V2c - A verilog to C translator tool, 2017, URL http://www.cprover.org/hardware/v2c/.
[37] D. Kroening, M. Tautschnig, CBMC–C bounded model checker: (competition contribution), in: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS, 2014, pp. 389–391.
[38] M.R. Gadelha, F.R. Monteiro, J. Morse, L.C. Cordeiro, B. Fischer, D.A. Nicole, ESBMC 5.0: an industrial-strength c model checker, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE, 2018, pp. 888–891.
[39] Z. Rakamarić, M. Emmi, SMACK: Decoupling source language details from verifier implementations, in: Computer Aided Verification: 26th International Conference, CAV, 2014, pp. 106–113.
[40] D. Beyer, M.E. Keremoglu, CPAchecker: A tool for configurable software verification, in: Computer Aided Verification: 23rd International Conference, CAV, 2011, pp. 184–190.
[41] E.M. Clarke, T.A. Henzinger, H. Veith, R. Bloem (Eds.), Handbook of Model Checking, Springer, 2018, http://dx.doi.org/10.1007/978-3-319-10575-8.
[42] Sail RISC-V model, 2019, URL https://github.com/riscv/sail-riscv.
[43] D. Gao, T. Melham, End-to-end formal verification of a RISC-V processor extended with capability pointers, in: 2021 Formal Methods in Computer Aided Design, FMCAD, 2021, pp. 24–33.
[44] K. Devarajegowda, E. Kaja, S. Prebeck, W. Ecker, ISA modeling with trace notation for context free property generation, in: 2021 58th ACM/IEEE Design Automation Conference, DAC, 2021, pp. 619–624, http://dx.doi.org/10.1109/DAC18074.2021.9586264.

[45] L. Weingarten, K. Datta, A. Kole, R. Drechsler, Complete and efficient verification for a RISC-V processor using formal verification, in: 2024 Design, Automation & Test in Europe Conference & Exhibition, DATE, 2024, pp. 1–6, http://dx.doi.org/10.23919/DATE58400.2024.10546693.

[46] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, G. Shurek, Constraint-based random stimuli generation for hardware verification, AI Mag. 28 (3) (2007) 13–13.

[47] F. Haedicke, H.M. Le, D. Große, R. Drechsler, CRAVE: An advanced constrained random verification environment for systemC, in: 2012 International Symposium on System on Chip, SoC, 2012, pp. 1–7.

[48] S. Fine, A. Ziv, Coverage directed test generation for functional verification using bayesian networks, in: Proceedings of the 40th Annual Design Automation Conference, DAC, 2003, pp. 286–291.

[49] Y. Lyu, P. Mishra, Scalable concolic testing of RTL models, IEEE Trans. Comput. 70 (7) (2021) 979–991, http://dx.doi.org/10.1109/TC.2020.2997644.

[50] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, J. Rajendran, HyPFuzz: Formal-assisted processor fuzzing, in: Proceedings of the 32nd USENIX Conference on Security Symposium, USENIX Security, 2023, pp. 1361–1378.

[51] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, C. Wang, MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation, in: Proceedings of the 32nd USENIX Conference on Security Symposium, USENIX Security, 2023, pp. 1307–1324.

[52] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, J. Rajendran, TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities, in: Proceedings of the 31st USENIX Security Symposium, USENIX Security, 2022, pp. 3219–3236.

[53] Y. Xu, Z. Yu, K. Wang, H. Wang, J. Lin, Y. Jin, L. Zhang, Z. Zhang, D. Tang, S. Wang, et al., Functional verification for agile processor development: A case for workflow integration, J. Comput. Sci. Tech. (2023).