Formal Verification of RISC-V Processor Chisel Designs

Shidong Shen[1,2], Yicheng Liu[1,2], Lijun Zhang[1,2], Fu Song[1,2], and Zhilin Wu[1,2(✉)]

[1] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{shensd,liuyc,zhanglj,songfu,wuzl}@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China

Abstract. Chisel is an open-source high-level hardware construction language embedded in Scala to facilitate 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 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, and extend. This considerably lowers the advantage of RISC-V for designing highly customizable processors. In this work, we present 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, thus can produce customized RISC-V processor variants. We then devise a novel queue-based 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. We implement our approach in an open-source tool and demonstrate its efficacy on two representative open-source RISC-V processor designs in Chisel (i.e., riscv-mini and NutShell). The experiment results confirm the efficacy of our approach, capable of discovering 7 real-world unknown non-conformance bugs and all the 10 manually injected bugs. It is also three-orders-of-magnitude more efficient than the state-of-the-art symbolic-execution based approach.

Keywords: Chisel · RISC-V · formal verification · reference model · synchronization · ISA conformance · model-checking

T. Bourke et al. (Eds.): SETTA 2024, LNCS 15469, pp. 142–160, 2025.
https://doi.org/10.1007/978-981-96-0602-3_8

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 specifications of RISC-V ISA in two volumes. The first volume [30] specifies the unprivileged ISA, including the base integer instructions (RV32I/64I) and optional unprivileged extensions, such as Multiplication and Division extension (M), and Compressed instructions (C). The second volume [31] 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) [2], 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 write complex, modular circuit generators that can produce circuit designs in synthesizable Verilog/SystemVerilog. This generation methodology enables the creation 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 [26], BOOM [4], riscv-mini [15], NutShell [23], and XiangShan [35]), and is playing an indispensable role in the promotion of the agile hardware development methodology [17,37].

Motivation. Modern processors are becoming more and more complex, consequently, ensuring their correctness in the pre-silicon stage becomes a central issue in the processor development. In this work, we focus on formal verification for the (functional) 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 [39] and the commercial tools FormalISA [1] and OneSpin [24]) first compile Chisel designs into low-level implementations in Verilog/SystemVerilog, then check their conformance to the RISC-V ISA specifications in the form 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, and extension of the reference models or assertions challenging and error-prone, hindering the flexibility of RISC-V ISA for 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 the first *end-to-end* formal verification approach for the (functional) 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. Sect. 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. Our model not only strictly subsumes the functionalities of, but also is an-order-of-magnitude more succinct than the Verilog/SystemVerilog reference model in riscv-formal.
2. We propose an end-to-end formal verification approach for RISC-V processor designs in Chisel by devising a novel queue-based 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. Sect. 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, that can be solved by off-the-shelf model-checkers.
3. We implement our approach in a tool using the open-source SMT-based model-checker Pono [20]. We validate the effectiveness and efficiency on riscv-mini [15] and NutShell [23], two representative open-source RISC-V processor designs in Chisel (cf. Sect. 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 10 manually injected bugs (5 bugs per DUT). Compared with the state-of-the-art open-source tool riscv-formal [39][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 [6,7], which is only able to discover 1 out of 7 bugs. Moreover, on this bug, our approach is significantly faster than the symbolic-execution based tool (0.52 s vs. 58 min).

Related Work. Formal verification has been widely studied and adopted for checking functional correctness of software and hardware designs [9]. 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

[1] In other words, it is the first work for the formal verification of RISC-V processor designs in Chisel that do not go through low-level Verilog/SystemVerilog or SVA.

[2] We do not compare with the two aforementioned commercial tools FormalISA [1] and OneSpin [24]), as they are not publicly available.

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. Sect. 5).

Formal verification of CHERI-RISC-V processor designs was studied using SVA [12], where CHERI-RISC-V[3] 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 [10], 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 [6,7]. They compile SpinalHDL designs into Verilog which in turn are compiled into C++. The RISC-V processor design in C++ is synchronized with a RISC-V Instruction Set Simulator 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. Sect. 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 [32]. 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., [11,13,21]. There is also a trend of adapting effective software testing techniques (e.g. concolic testing and fuzzing) to the informal verification of hardware designs [8,14,19,36,38]. Nevertheless, they are orthogonal to this work.

Outline. 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 presents a queue-based synchronization mechanism between the RISC-V processor DUT and the reference model. Section 5 reports experimental results. We conclude this paper in Sect. 6.

The source code of our RISC-V ISA reference model and formal verification tool is available at: https://github.com/iscas-tis/riscv-spec-core.

2 Overview

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

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

Reference Model. At first, following closely the official two volumes of the RISC-V ISA manual, 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.

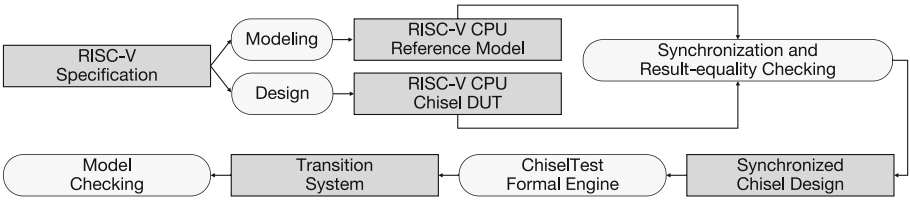


Fig. 1. Overview of our approach

Synchronization. To verify whether a RISC-V CPU Chisel design conforms to the RISC-V ISA specifications, the RISC-V CPU 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. After synchronization, a synchronized Chisel design is obtained.

Model-Checking. The synchronized Chisel design is transformed by the formal engine in ChiselTest [16] to a transition system, which is then tackled by the model-checkers e.g. Pono [20], where various model-checking algorithms e.g. BMC [3], k-induction [28], and PDR [5], could be harnessed.

3 The RISC-V Reference Model in Chisel

In this section, we elaborate the design 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 RISC-V reference model is shown in Fig. 2. Intuitively, the RISC-V reference model can be seen as a modular, and parameterized RISC-V processor design generator in Chisel that can produce customized single-clock-cycle RISC-V processor variants.

As a result, it involves the common steps of decoding, executing, and register-status updating (a.k.a. write-back) in a typical RISC-V processor design. The step of register-status updating is responsible for updating the values of general registers and Control and Status Registers (CSR) after the execution of an instruction is finished. The reference model also involves several components for modeling privileged instructions, including exception handling and the address translation unit that translates the virtual addresses into physical addresses. Equipped with these components, the RISC-V reference model is capable of supporting the modeling of RISC-V 32/64 bit-widths Integer (I), Multiplication and

Division (M), Compressed (C), and Control and Status Register (Zicsr) instruction sets, as well as the three privilege levels, namely, Machine (M), Supervisor (S), and User (U), and finally the Sv39 virtual memory system.

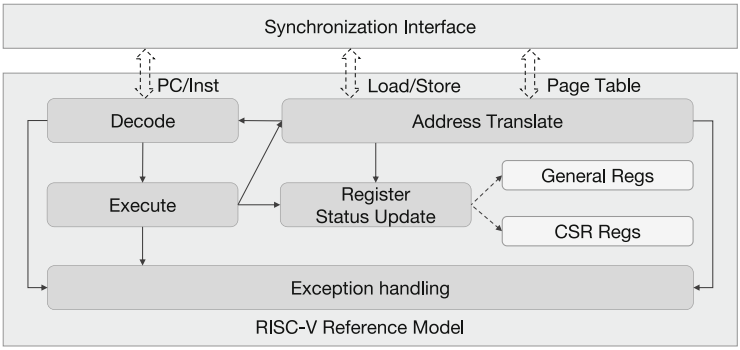


Fig. 2. The RISC-V reference model in Chisel

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 3,000 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 processor designs with different bit-widths, and ISA extensions.

For a particular instantiation of the parameters, when the reference model is compiled into FIRRTL (a hardware intermediate representation language for Chisel) [18], only the ISA specifications that are related to this instantiation will be compiled in the generated 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 [39], around 30,000 lines of Verilog/SystemVerilog code are used to model the RISC-V ISA specifications, moreover, additional 2,551 lines of Python scripts are used to configure the bit-widths and ISA extensions of the reference models.

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).

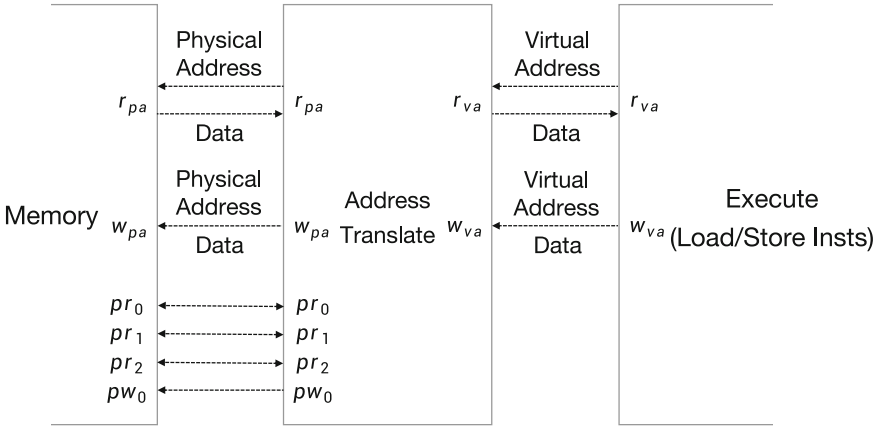


Fig. 3. Architecture of the address translation component

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 guaranteed. Nevertheless, the formal verification of the correctness of the reference model is similar to the formal verification of the temporal logic specifications in model-checking, which is in some sense a dead-end. Therefore, we resort to the following two informal means to enhance our confidence on the correctness of the reference model.

- As a design choice, we utilize the high-level features of Chisel to make the reference model closely resembling the organization of the RISC-V ISA manual in a succinct and modular way. This definitely reduces the chances of introducing human-mistakes in the design of the reference model and eases the manual inspection of its correctness.
- Since the reference model can be seen as a parameterized single-clock-cycle RISC-V processor, we also use riscv-tests [25], the official RISC-V CPU test suite, to thoroughly validate the correctness of the reference model.

4 Synchronization

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 between the micro-architectures of the DUT and the reference model. For instance, while the reference model is a (parameterized) single-clock-cycle processor, the DUTs normally involve pipelines. To bridge this gap, in the synchronization, we take the signals in the last stage of the pipeline (usually the write-back stage) and check whether they are equal to those in the reference model.

To this end, for many instructions (e.g. the integer computational instructions), the synchronization is not difficult since the number of clock cycles for their executions in DUTs have a *fixed* (constant) upper bound. However, the delays of load/store instructions can be *non-fixed*, since they involve memory accesses. In the sequel, we show how to deal with the non-fixed delays of the load/store instructions in DUTs during synchronization. Moreover, we also discuss how to tackle the additional challenges when virtual memory are used in DUTs so that the executions of the load/store instructions involve the translation of virtual addresses into physical addresses.

4.1 Synchronization of Instructions with Non-fixed Delays

When the execution of an instruction involves memory accesses, the delay can be non-fixed. For instance, some instructions may be implemented using state machines so that the delays in their executions may be greater than those of the other instructions.

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 [23], 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.

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.

4.2 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 Equality Checking of the Execution Results

To verify that the implementation of an instruction in the DUT conforms to the RISC-V ISA specification of the instruction, it is necessary to check that the results after the execution of the instruction in the DUT and in the reference model are equal. Table 1 shows the set of signals of the DUT that are involved in the equality checking of the execution results in the formal verification. Usually, the signals in the DUT should be preprocessed before the equality checking.

Table 1. Set of signals to be checked for equality

Types of signals	Fields to be checked
Instruction	PC
General Regs	All general regs (x0 - x31)
CSR Regs	All implemented CSR regs
Exception	Valid, No, PC, Inst
Memory	Valid, Address, Data, Width
PageTable	Valid, Address, Data, Width

5 Evaluation

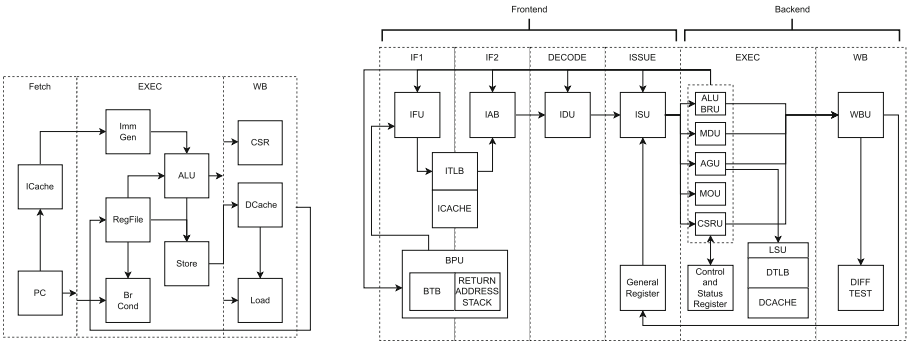
In this section, we evaluate the effectiveness and efficiency of our approach on two representative open-source RISC-V processor Chisel designs: riscv-mini and NutShell. Moreover, we compare the performance of our approach against the other two state-of-the-art approaches for formal verification of RISC-V processor designs, that is, riscv-formal and the symbolic-execution based approach in [6,7]. Both tools have been used in formally verifying several RISC-V processor designs and found real-world bugs [6,34].

5.1 Case Studies on Riscv-Mini and NutShell

We apply the approach proposed in this work to the formal verification of the ISA conformance of riscv-mini [15] and NutShell [23] w.r.t. the RISC-V ISA specifications.

- **riscv-mini** is a simple RISC-V processor with a 3-stage pipeline. It supports RV32I and the machine-level ISA. It also contains simple instruction caches (ICache) and data caches (DCache). It includes 3,499 lines of Chisel code in total. (See Fig. 4a for its architecture.)
- **NutShell** is a single-issue in-order RISC-V processor with a 9-stage pipeline. It supports the following instruction extensions: 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. It includes 8,859 lines of Chisel code in total. (See Fig. 4b for its architecture.) NutShell is configurable. For instance, it can be configured as a 32-bit processor (with a 5-stage pipeline).

RISC-V ISA Specification Conformance Checking. We use the assume statements of Chisel to restrict the op-codes of RISC-V instructions during the executions of the processors and the reference model. Then the formal verification of DUT's conformance to the RISC-V ISA specification is reduced to an



(a) riscv-mini (b) NutShell

Fig. 4. Architecture of the two processors

instance of the model-checking problem, which is then solved by the Pono model-checker [20], a well-known efficient SMT-based open-source model-checker that can be easily integrated into the ChiselTest formal engine. The model-checking instances are in the BTOR2 format [22]. Each model-checking instance resulting from riscv-mini contains 3,395 total state bits and 378 total input bits, and each model-checking instance resulting from NutShell contains 71,581 total state bits and 7,312 total input bits. We utilize the bounded model-checking (BMC) engine of Pono to solve the model-checking problem[4], where the max step bound is set to 40. The model-checking experiments were run under Ubuntu 20.04LTS with 64 GiB RAM and 24 Intel(R) Core™ i9-13980HX CPU @ 5.60 GHz processors.

We 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[5]. The 5 bugs found in NutShell have been confirmed by the designers.

Table 2 includes some statistics of the experiments related to the 7 bugs[6], 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, column **Hex** gives the hexadecimal encoding of the instruction, 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.

[4] We also tried some other model-checking algorithms (e.g., PDR). It turns out that the transition systems generated from riscv-mini and NutShell are too large for them to finish in a reasonable amount of time (24 h).

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

[6] 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.

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[7] to be `h80000` and the value of the `mstatus` register[8] 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 97 m31.34 s. Then we adapted the assume statement to enforce that `PPN[1:0]` is zero[9], thus bypassing the bug N:E4. Finally, we reran the model-checker Pono and discovered the bug N:E5 after 67 m11.29 s.

Table 2. 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) SB x31, -1(x5)	0x0210A323 0xFFF28FA3	✗
N:E1	10	2.56 s	RV32/64I	LD x4, -377(x1)	0xE870B203	✗
N:E2	29	47 m27.66 s	Load/Store	SD x1, 1976(x19)	0x7A19BC23	✓
N:E3	19	11 m25.44 s	Zicsr MRET, SRET ECALL, EBREAK	SRET SRET	0x10200073 0x10200073	✗
N:E4	29	97 m31.34 s	Load/Store	SD x0,48(x22)	0x020B3823	✓
N:E5	29	67 m11.29 s	Load/Store	LD x0,384(x2)	0x18013003	✓

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.

[7] 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.

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

[9] PPN denotes the physical page number.

As we shall see in Sect. 5.2, 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 Sect. 4.3.2 of [31]), some lower `PPN` bits of the Sv39 page table entries (PTE) should be zeros 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 zeros. 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, we should do some checks according to Steps 3 and 4 in the RISC-V ISA specification (Page 82, Volume 2).

Table 3. 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

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 3), 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.

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).

5.2 Performance Comparison with the Other Approaches

As reported in Sect. 5, 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 against the other two approaches of formal verification of RISC-V processor designs, that is, riscv-formal and the symbolic-execution based approach in [6,7].

5.2.1 Comparison with Riscv-Formal

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

Comparison of the Bug-Detection Capability. We compare the detection capability of our approach and riscv-formal by checking whether riscv-formal can discover the 7 bugs found by our approach (see Table 2). 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. R:E1–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–E5, riscv-formal cannot discover the four bugs related to virtual memories or privileged instructions, that is, N:E2–E5, 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. Besides the bug-detection capability, we would like to compare the scalability of our approach and riscv-formal on riscv-mini and NutShell. Since riscv-mini supports only the RV32I instructions and riscv-formal does not support the privileged instructions, we choose to ignore the privileged instructions and focus on the RV32I instructions. For technical convenience, we choose the 32-bit version of NutShell for the comparison. As a result of this choice, in the comparison, we can use the same configuration in our reference model for both riscv-mini and NutShell. Similarly, we can use the same configuration in riscv-formal for both riscv-mini and NutShell.

To facilitate the comparison, we manually inject into the two processor designs the following five bugs for RV32I 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 a fault in riscv-mini 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 in riscv-mini 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 in riscv-mini 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 in riscv-mini by setting the lowest bit of the result to 0.

Then we run our approach and riscv-formal to detect these bugs and compare their performance. 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. Then we create multiple model-checking problem instances for these assertions, one instance for each assertion. Since the Pono model-checker is used as the backend in our approach, we run multiple Pono processes on the 24 Intel processors in parallel to solve these model-checking problem instances. On the other hand, in riscv-formal, we run multiple SymbiYosys [33] processes on the 24 processors in parallel, since SymbiYosys is used therein to solve the model-checking problem. Both our approach and riscv-formal will stop when any of their processes finds a bug.

Note that we record the time when the first bug is reported. The experiment results are reported in Table 4. From the results, we can see that the scalability of our approach is better than that of riscv-formal (except the bug #2).

Table 4. Scalability comparison of our approach and riscv-formal

Bug	riscv-mini				NutShell			
	Our approach		riscv-formal		Our approach		riscv-formal	
	Time[s]	Step	Time[s]	Step	Time[s]	Step	Time[s]	Step
# 1	2.28	4	3.43	4	10.32	6	29.14	6
# 2	6.72	4	1.81	4	14.84	6	14.73	6
# 3	0.66	4	3.41	4	5.47	6	32.44	6
# 4	0.50	4	3.50	4	6.54	6	32.77	6
# 5	1.62	4	3.36	4	10.28	6	32.75	6
Avg.	2.36	4	3.10	4	9.49	6	28.37	6

5.2.2 Comparison with Symbolic-Execution Based Approach

We compare the performance of our approach with the symbolic-execution based approach in [6,7]. 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 code is translated into C++ using the tool Verilator.
- A Voter module is utilized to synchronise the C++ code corresponding to a RISC-V processor with a C++ RISC-V ISS (Instruction Set Simulator), where the RISC-V Formal Interface (RVFI) [27] is added to the RISC-V processor to facilitate its connection to the Voter module.
- The C++ code is compiled into LLVM bytecode by Clang.
- Finally, the symbolic-execution engine KLEE is harnessed for verification.

In [6], a non-pipelined RISC-V processor called MicroRV32 was verified. MicroRV32 supports RV32-IMC, but lacks Cache and TLB modules. It consists of 3,193 lines of Verilog code and 7,497 lines of C++ code (after translating Verilog to C++).

At first, we try to use the aforementioned symbolic-execution approach to detect the RISC-V specification non-conformance bugs in riscv-mini and NutShell that were found by our approach (see Table 2).

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 2, namely R: E1, within the time 58 m39 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 2,349 lines of Verilog code and 8,155 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 [29]. 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 4. 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 85 m59 s and 1356 m10 s, respectively (Recall that the two bugs can be discovered by our approach in 0.66 s and 0.50 s, 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.

6 Conclusion and Future Work

In this work, we 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 two 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. Furthermore, our approach is also 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.

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 precision and efficiency in the verification.

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

References

1. Axiomise: FormalISA: RISC-V formal verification (2023). https://www.axiomise.com/riscv-formal-app/
2. Bachrach, J., et al.: Chisel: constructing hardware in a Scala embedded language. In: DAC, pp. 1216–1225 (2012)
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS, pp. 193–207 (1999)
4. RISC-V Boom: The Berkeley out-of-order RISC-V processor (2023). https://github.com/riscv-boom/riscv-boom
5. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI, pp. 70–87 (2011)
6. Bruns, N., Herdt, V., Drechsler, R.: Processor verification using symbolic execution: a RISC-V case-study. In: DATE, pp. 1–6 (2023). https://doi.org/10.23919/DATE56975.2023.10137202
7. Bruns, N., Herdt, V., Drechsler, R.: Symbolic execution framework for RISC-V processor verification (2023). https://github.com/agra-uni-bremen/symex_processor_verification

8. Chen, C., et al.: HyPFuzz: formal-assisted processor fuzzing. In: USENIX Security, pp. 1361–1378 (2023)
9. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
10. Devarajegowda, K., Kaja, E., Prebeck, S., Ecker, W.: ISA modeling with trace notation for context free property generation. In: 2021 58th ACM/IEEE Design Automation Conference (DAC), pp. 619–624 (2021). https://doi.org/10.1109/DAC18074.2021.9586264
11. Fine, S., Ziv, A.: Coverage directed test generation for functional verification using Bayesian networks. In: DAC, pp. 286–291 (2003)
12. Gao, D., Melham, T.: End-to-end formal verification of a RISC-V processor extended with capability pointers. In: FMCAD, pp. 24–33 (2021)
13. Haedicke, F., Le, H.M., Große, D., Drechsler, R.: Crave: an advanced constrained random verification environment for systemc. In: SoC, pp. 1–7 (2012)
14. Kande, R., et al.: TheHuzz: instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In: USENIX Security, pp. 3219–3236 (2022)
15. Kim, D.: RISCV-MINI, a simple RISC-V 3-stage pipeline written in chisel (2017). https://github.com/ucb-bar/riscv-mini
16. Laeufer, K., Bachrach, J., Sen, K.: Open-source formal verification for Chisel. In: WOSET (2021). https://woset-workshop.github.io/WOSET2021.html
17. Lee, Y., et al.: An agile approach to building RISC-V microprocessors. IEEE Micro **36**, 1 (2016). https://doi.org/10.1109/MM.2016.11
18. Li, P.S., Izraelevitz, A.M., Bachrach, J.: Specification for the FIRRTL language. Technical report. UCB/EECS-2016-9, EECS Department, University of California, Berkeley (2016). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html
19. Lyu, Y., Mishra, P.: Scalable concolic testing of RTL models. IEEE Trans. Comput. **70**(7), 979–991 (2021). https://doi.org/10.1109/TC.2020.2997644
20. Mann, M., et al.: Pono: a flexible and extensible SMT-based model checker. In: CAV, pp. 461–474 (2021)
21. Naveh, Y., et al.: Constraint-based random stimuli generation for hardware verification. AI Mag. **28**(3), 13–13 (2007)
22. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, btormc and boolector 3.0. In: CAV, pp. 587–595 (2018)
23. Nutshell RISC-V CPU (2019). https://github.com/OSCPU/NutShell
24. Onespin formal verification solutions - siemens eda (2024). https://eda.sw.siemens.com/en-US/ic/questa/onespin-formal-verification/
25. RISCV-tests (2015). https://github.com/riscv-software-src/riscv-tests
26. Rocket chip RISC-V CPU generator (2023). https://github.com/chipsalliance/rocket-chip
27. RISC-V formal interface (RVFI) (2020). https://github.com/SymbioticEDA/riscv-formal/blob/master/docs/rvfi.md
28. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD, pp. 127–144 (2000)
29. Verilator: Open-Source SystemVerilog simulator and lint system (2024). https://github.com/verilator/
30. Waterman, A., Asanović, K.: The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Version 20191213 (2019)
31. Waterman, A., Asanović, K., Hauser, J.: The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 20211203 (2021)

32. Weingarten, L., Datta, K., Kole, A., Drechsler, R.: Complete and efficient verification for a RISC-V processor using formal verification. In: 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–6 (2024). https://doi.org/10.23919/DATE58400.2024.10546693
33. Wolf, C., et al.: Symbiyosys (2022). https://symbiyosys.readthedocs.io/
34. Wolf, C.: Formal Verification of RISC-V Cores with RISCV-formal (2018). https://riscv.org/wp-content/uploads/2018/12/13.30-Humbenberger-Wolf-Formal-Verification-of-RISC-V-processor-implementations.pdf
35. Xiangshan: An open-source high-performance RISC-V processor (2023). https://github.com/OpenXiangShan/XiangShan
36. Xu, J., Liu, Y., He, S., Lin, H., Zhou, Y., Wang, C.: MorFuzz: fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In: USENIX Security, pp. 1307–1324 (2023)
37. Xu, Y., et al.: Towards developing high performance RISC-V processors using agile methodology. In: MICRO, pp. 1178–1199 (2022)
38. Xu, et al.: Functional verification for agile processor development: a case for workflow integration. J. Comput. Sci. Technol. (2023)
39. YosysHQ: RISC-V Formal Verification Framework (2016). https://github.com/YosysHQ/riscv-formal