ISAUDIT: FINDING BUGS IN RISC-V CORES BY CHECKING AGAINST THE ISA SPECIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Hardware flaws are permanent: a CPU bug cannot be patched once fabricated, so it can only be fixed before tape-out. Whether a core behaves as specified by its instruction set architecture (ISA) is thus settled by pre-silicon verification. Yet existing techniques check proxies for the ISA specification rather than the specification itself, either a reference implementation that can itself be wrong or hand-written properties that cover only a subset of the specification, so a core can pass every check and still violate the specification. While Large Language Models (LLMs) offer automation potential, letting one interpret the specification and judge conformance directly faces two challenges: *interpretation fidelity*, where a prose clause may be misinterpreted, and *verdict trust*, where even a correct interpretation yields a verdict without mechanically checkable evidence. To bridge this gap, we introduce ISAUDIT, a specification-grounded auditing framework that makes the LLM's interpretation of the specification executable and checkable. It first maps each normative statement to the RTL and expresses it through two heterogeneous representations, repaired from their disagreements until adjudication settles a formal property traceable to its source clause. It then checks the settled property with a model checker, backing each violation with a concrete counterexample for triage. Evaluation on five open-source RISC-V cores shows that ISAUDIT uncovers 33 previously unknown bugs, outperforming baselines by 200–3,200%.

1 INTRODUCTION

Hardware defects are unusually costly to correct after fabrication: unlike software vulnerabilities, they cannot generally be removed by deploying a patch. An instruction set architecture (ISA) specifies the behavior that software may rely on, and a processor must implement this contract before it is fabricated. Checking this contract requires engineers to translate normative statements—rules that say what a processor must or must not do—from hundreds of pages of prose, tables, and exceptions into *formal properties*: executable conditions over the signals and events of a register-transfer-level (RTL) hardware implementation. This translation, rather than merely running a verification tool, is the expensive and error-prone part of pre-silicon verification.

Existing processor-validation techniques instead avoid this translation by checking proxies for the specification. Differential fuzzers execute programs on both a processor core and a reference implementation, and report behavioral mismatches (Solt et al., 2024). Formal suites such as RISCV-FORMAL check a manually selected set of properties (YosysHQ, 2026) (Figure 1a). General-purpose coding agents can read both the specification and the implementation, but their reports lack an executable chain from a source clause to a mechanically reproducible result (Anthropic, 2026; OpenAI, 2026). These approaches leave complementary blind spots: a reference implementation can itself implement the specification incorrectly, while hand-written properties cover only the behaviors that engineers anticipated. Consequently, a clean result establishes only agreement with the chosen proxy; it does not establish conformance to the complete ISA specification. What is missing is a source-traceable path from each normative statement to an executable formal property, together with evidence for interpreting each model-checking result.

Large Language Models (LLMs) offer a way to automate this translation, but a property formalized along a single path remains only one unverified interpretation of the specification. The property may compile and even pass every check while silently omitting a condition, and reviewing it through the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

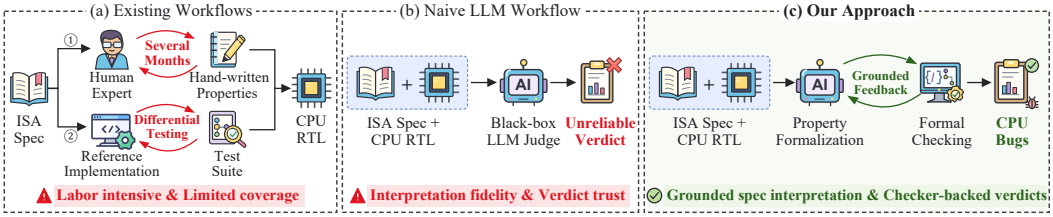

Figure 1: Comparison with existing efforts. (a) Existing workflows either involve an expert handwriting properties, or differential testing against a reference implementation. (b) A naive LLM workflow reads the specification and the RTL and produces a verdict, but it may misinterpret the specification and its verdict is untrustworthy. (c) ISAUDIT formalizes the specification into properties, checks them with a model checker, and feeds grounded evidence back, so that each reported bug is backed by a grounded interpretation and mechanical checking.

same representation may reproduce the same omission. This creates two challenges. First, *interpretation fidelity* is uncertain: an LLM may omit an exception, read a permission as a requirement, or bind a requirement to the wrong signal or cycle. Second, *verdict trust* is uncertain: an LLM's PASS or FAIL (Figure 1b) is a verdict without independently checkable evidence. This risk is asymmetric: an LLM's PASS, issued without the property ever being mechanically solved, offers no evidence against a genuine violation, which can then be silently missed as a false negative. A model checker produces a mechanically reproducible result for a given property, but by itself cannot show that the property captures its source clause or that a reported failure belongs to the RTL rather than the verification setup.

These limitations suggest that neither an extracted property nor a raw model-checker result should be trusted in isolation. **Our key insight is to turn specification-to-property translation into a refutable process:** before a property is admitted for checking, the same ISA clause is expressed through two heterogeneous, executable paths. One path constructs a view grounded in the target RTL's structure and events, while the other constructs an executable view grounded in the specification. Each disagreement between the two views then drives adjudication and repair. Unlike model voting or self-consistency over natural-language answers, the two paths expose errors through representations with different inputs and failure modes.

We instantiate this idea in ISAUDIT, a specification-grounded auditing framework with two stages. *Refutation-Driven Formalization* begins by analyzing the target RTL's structure and events, extracting normative statements from the ISA specification, and mapping each statement to the relevant RTL signals and cycles. It then assigns each mapped clause to two workers: an *RTL-grounded worker* constructs an RTL view from the target RTL and its extracted structure, events, and mappings, while a *specification-grounded worker* constructs a specification view primarily from the original specification text. Adjudication and repair compare these two executable views, use each disagreement to revise the responsible interpretation, and emit each settled property as an executable SystemVerilog assertion. *Evidence-Backed Checking* first checks that each property is correctly bound and not vacuous, then checks it with a model checker. For each result, a multi-source review places the original specification text alongside the executable property, counterexample or reachability evidence, and relevant RTL path. It also checks whether the verification environment, rather than the RTL, explains an apparent violation, and if so re-solves the property under a corrected environment. Only failures attributed to the RTL are reported as bugs, and property errors are fed back to formalization for iterative refinement. In summary, we make the following contributions:

- We propose to make both an LLM's interpretation of the ISA specification and its verdict on the RTL checkable, using executable formal properties as the bridge between the two.
- We design *Refutation-Driven Formalization*, which refutes an RTL view against a specification view of the same clause to improve the fidelity of each property, and *Evidence-Backed Checking*, which checks each property with a model checker to back each verdict with reviewable evidence.
- We implement ISAUDIT and evaluate it on five open-source RISC-V cores against four baselines. It finds 33 previously unknown bugs, 3–33$\times$ as many as the baselines, at 95.6% precision.

2 Background and Motivation

2.1 ISA Specifications and RTL Implementations

For RISC-V, the ISA specification is a prose document that defines the instructions, the control and status registers (CSRs), and traps (RISC-V International, 2026b). Its *normative statements*, or clauses, are scattered across prose and tables (Armstrong et al., 2019) and often depend on conditions or exceptions stated elsewhere: the clause that `jal` writes the address of the following instruction to register *rd*, for example, is stated in one paragraph, while its exception for a misaligned target appears in a later one.

A processor core implements this specification at the *register-transfer level* (RTL), which specifies cycle by cycle how its registers are updated, so an event such as an instruction retiring is visible only as particular signals at a particular cycle. The specification, however, describes behavior in terms of the *architectural state*, the registers, CSRs, and memory visible to a program. It leaves the microarchitecture open, so cores realize the same architectural effect with different signals and timing (Hur et al., 2021). Correctness is therefore judged at the architectural level: as in prior work, a deviation of a core's architectural behavior from the specification is regarded as a *bug* (Thomas et al., 2025; Bölcskei et al., 2025).

2.2 Formal Property Checking on RTL

A clause is checked mechanically by stating it as a *formal property*, an assertion over RTL signals evaluated at every clock cycle (Kande et al., 2024) whose condition selects the relevant cycles and whose expression states the requirement (YosysHQ, 2026). For the `jal` clause, the condition selects a retiring `jal` that did not trap and whose destination is not `x0`, and the expression compares the value written to *rd* with the instruction address plus four. Writing it also requires *binding* it to the target core: locating the signals that carry the referenced state and sampling them at the right cycle. A property is *faithful* to its clause when it expresses the same requirement, including the clause's conditions and exceptions.

A *model checker* then checks the property against the RTL: it encodes the design and the assertion as a formula and asks a SAT or SMT solver for an input sequence that violates the assertion (Biere et al., 1999). If none is found, the property *passes*. If one exists, the solver returns a *counterexample* trace. A pass is *vacuous* if the condition never holds, which a separate cover check rules out.

The core is checked inside a *verification environment*, a wrapper that models its memory and constrains its inputs by assumptions, so an under-constrained environment can yield *spurious counterexamples* that reflect the setup rather than a bug. The environment also exposes the core's *RISC-V Formal Interface* (RVFI) (YosysHQ, 2026), signals that report each retired instruction in the cycle it retires, while state beyond RVFI must be observed on core-specific internal signals.

3 Related Work

Dynamic testing. Dynamic testing typically runs input programs on both the RTL under test and a reference implementation and reports mismatches in architectural state; the input programs come from conformance test suites (RISC-V International, 2026a), constrained-random generators (CHIPS Alliance, 2026), or fuzzers (Hur et al., 2021; Solt et al., 2024; Guo et al., 2025). What it finds is limited on both sides: a bug surfaces only if some input program reaches it, and a mismatch counts only if the reference implementation is itself correct and up-to-date, yet reference implementations have bugs of their own (Xu et al., 2023) and may not track each new ISA extension. ISAudit instead checks the RTL against the specification itself, with a model checker rather than randomly generated inputs that happen to trigger it.

Formal verification. Formal verification instead checks the RTL against hand-written properties, from suites such as riscv-formal (YosysHQ, 2026) to executable ISA models (Reid et al., 2016); a suite covers the behaviors its authors selected, and a model can itself contain errors (Reid, 2017). Recent work therefore generates SystemVerilog assertions from natural-language design specifications with LLMs (Kande et al., 2024; Yan et al., 2025; Bai et al., 2025; Imperial & Zheng, 2026),

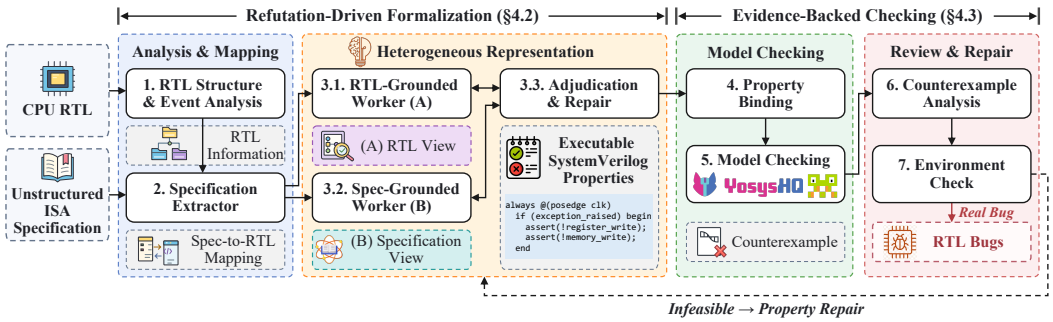

Figure 2: ISAUDIT Overview. *Refutation-Driven Formalization* (§4.2) maps ISA clauses to the target RTL (steps 1-2), has an RTL-grounded and a specification-grounded worker express each clause (steps 3.1-3.2), and adjudicates their disagreements into SystemVerilog properties (step 3.3). *Evidence-Backed Checking* (§4.3) binds each property to the verification environment (step 4), runs model checking (step 5), and reviews every counterexample against the specification, property, and environment before reporting a bug or feeding a repair back to formalization (steps 6-7).

judged against a golden RTL, injected mutants, or an LLM's alignment score, so a provable but unfaithful assertion passes (Mashnoor et al., 2026; Wu et al., 2026). ISAUDIT also derives properties from prose, but from the ISA specification, and treats the LLM's interpretation of each clause as a claim to be checked rather than a property to be trusted.

LLM-driven auditing and cross-checking. Outside hardware, LLMs audit implementations against their specifications, reporting RFC violations as judged by the LLM (Zheng et al., 2025) or turning specification clauses into formats and tests executed against the implementation (Ye et al., 2026; Ma et al., 2024). In neuro-symbolic systems the LLM translates and a symbolic engine decides (Pan et al., 2023; Liu et al., 2026), leaving the translation unverified; cross-checking LLM outputs is the usual remedy, but LLM errors are correlated: models agree 60% of the time when both err (Kim et al., 2025), and a transformation that does not disrupt this correlation yields little improvement (Dai et al., 2025). ISAUDIT does not take agreement between its two representations as evidence of correctness: they are grounded in different evidence, primarily the RTL source and the specification text, and the verdict on a property rests on the model checker's grounded result on the RTL.

4 DESIGN

4.1 OVERVIEW

ISAUDIT takes two inputs: the RISC-V ISA specification and the RTL of a target processor core. Rather than asking an LLM whether the RTL is correct, it constructs an executable formal property for each in-scope normative statement and checks both the property and its model-checking result. Every property keeps its source statement, the RTL location and observations used to evaluate it, and the evidence behind the final assessment.

Figure 2 organizes the two stages around the artifact that each block hands to the next: *Refutation-Driven Formalization* turns each clause into an adjudicated SystemVerilog Assertion (SVA), and *Evidence-Backed Checking* turns each admitted property into a verdict attributed to the RTL, the property, or the environment, or left inconclusive.

4.2 REFUTATION-DRIVEN FORMALIZATION

4.2.1 ANALYSIS AND MAPPING

RTL Structure and Event Analysis (step 1). ISAUDIT first builds a *probe harness* H that exposes retirement events through the RISC-V Formal Interface (RVFI) and configuration-selected observations, and records reset behavior, free environment inputs, and the signals a property may reference. A preprocessing step inserts the required observation ports into the RTL. H, not an arbitrary internal

signal, is the binding authority for all properties. ISAUDIT also builds a source-located *RTL index* I over modules, ports, state elements, and control logic, each entry retaining its file and line. The index is a retrieval artifact that limits the RTL-grounded worker and the reviewer to evidence relevant to the current property.

Specification Extractor (step 2). ISAUDIT does not translate a clause straight into an assertion. It first writes a *property record* $q = \langle s_q, a_q, m_q \rangle$: s_q quotes the clause verbatim (what the ISA requires), a_q points to the RTL files and lines that implement the behavior (where to look), and m_q states the requirement in structured form with the signals to observe and its applicability conditions (how to judge it). The record is the fixed question and the SVA encoding a revisable answer: a wrong encoding is repaired while the requirement in q stays unchanged, so a repair cannot rewrite the requirement itself.

The clause-to-hardware mapping is therefore explicit. An ISA clause speaks about architectural behavior, such as "a CSR read with a zero source operand must not raise an exception," which no single RTL signal encodes. The extractor walks the RTL index to find the instruction class and implementation region that realize the behavior, chooses the retirement or configured observer event at which it becomes visible, and stores both in a_q and m_q. A behavior the fixed harness cannot observe is kept as a record marked as needing an observer rather than converted into a property over an unrelated signal. This yields the Spec-to-RTL Mapping artifact in Figure 2.

Because a single pass over a long specification can silently omit a clause, a mechanically triggered coverage-closure pass follows. The specification is preprocessed into individually anchored normative tags, and every tag unmatched by the property table receives exactly one disposition: a new property, an existing property that covers it, a structurally unavailable observation, inapplicability to the fixed configuration, or definitional or ordering text outside the checked behaviors. A missing RTL feature does not make a mandatory requirement inapplicable but yields an existence or behavior property. Coverage is claimed over the normative tags of the selected chapters.

4.2.2 HETEROGENEOUS REPRESENTATION

Algorithm 1 gives the full control flow for constructing the adjudicated property. The central design choice is that no property is admitted on a single worker's reading of the clause: each candidate is compared with one built within a different evidence boundary. "Heterogeneous" refers to evidence boundaries, not output language: both workers emit SystemVerilog, so their candidates are directly comparable. One is grounded primarily in the implementation view and the other in the specification view, so a misreading formed within one route does not carry over to the other. For a property record q, let S denote its specification context, R_q the RTL retrieved through its anchors, E the environment contract, and H and I the shared harness and index. The two workers receive $C_R(q) = \{q, I, H, R_q, E\}$ and $C_S(q) = \{q, S, I, H, E\}$, respectively. The shared record, index, harness, and environment contract fix the signals both candidates may reference; the contexts differ in whether the worker reads the retrieved RTL R_q or the specification text S.

Alg. 1: Heterogeneous Property Construction

Require: record q; contexts $C_R(q)$, $C_S(q)$, Γ_q; budget B
Ensure: p^* with provenance, or INCONCLUSIVE

for $i \leftarrow 0$ **to** B **do**
  $(p_R, p_S) \leftarrow W_R(C_R(q)) \parallel W_S(C_S(q))$
  if σ **then** ▷ identical drafts, Eq. (1)
    CRITIQUE$(p_R, p_S,$ NONINDEPENDENT$)$
    continue
  end if
  $(z, p^*, \delta, r) \leftarrow J(\Gamma_q, p_R, p_S)$
  if $z =$ DECIDED **then**
    return $(p^*,$ PROVENANCE$(q, R_q))$
  else if $z =$ REWRITE **then**
    $C_r(q) \leftarrow$ REPAIR$(C_r(q), \delta)$
  else
    return INCONCLUSIVE(δ)
  end if
end for
return INCONCLUSIVE(BUDGETEXHAUSTED)

RTL-Grounded Worker (A, step 3.1). The RTL-grounded worker W_R uses C_R to select implementable signals, events, and timing from the fixed design.

Spec-Grounded Worker (B, step 3.2). The specification-grounded worker W_S uses C_S to derive the expected behavior from the source text without seeing raw implementation logic. Both must emit an executable candidate over signals declared by H, including an assertion, a cover for its triggering condition, and the minimum meaningful checking depth.

Adjudication and Repair (step 3.3). The workers run independently and, during repair, each sees only its own previous candidate, so neither is anchored to the other's output. Let $p_R = W_R(C_R(q))$ and $p_S = W_S(C_S(q))$, let $\mathrm{Exec}(\cdot)$ recognize the required executable property shape, let $N(\cdot)$ re-

move comments and normalize whitespace, and let $\sigma \equiv \mathrm{Exec}(p_R) \wedge \mathrm{Exec}(p_S) \wedge N(p_R) = N(p_S)$. With $\Gamma_q = (q, S, I, H, R_q)$ the adjudication context, the selection rule is

$$p^* = \begin{cases} \perp_{\text{identical}}, & \sigma, \\ J(\Gamma_q, p_R, p_S), & \neg\sigma, \end{cases} \tag{1}$$

where J is the adjudicator. As line 7 shows, J returns a verdict z, a property p^*, feedback δ, and the route $r \in \{R, S\}$ whose context $C_r(q)$ is repaired. Equality is syntactic and is not evidence of correctness: byte-identical drafts defeat heterogeneous construction and are returned for materially different re-authoring (line 4).

Figure 3 illustrates the route-specific repair (line 11) in both directions for a Zicsr clause that names `CSRRC` with `rs1 = x0`. In (a), B omits this case ($A = 1$, $B = 0$) and receives the missing checks. In the opposite direction, (b) repairs A ($A = 0$, $B = 1$). The adjudicator may select either candidate or construct a corrected version, but must cite the relevant source text and RTL location and may neither invent an undeclared observation nor add an assumption to make the property pass. The two representations reach a fixed point when adjudication accepts a complete property without requesting a further repair of either route (line 9), so that neither view is revised by the other; the settled property then proceeds to mechanical checking. Byte-identical drafts do not count as such agreement. Otherwise, the procedure ends without a property once the budget is exhausted or external evidence is required.

4.3 Evidence-Backed Checking

4.3.1 Model Checking

Property Binding (step 4). Each adjudicated property passes two admission gates, kept separate so that a successful solver run cannot conceal a failed or guessed binding. The *binding gate* pastes the property into H and has Yosys elaborate the combined RTL and property, check signal bindings, and lower the design for model checking. A compile failure permits one constrained repair of the reported defect that may not remove assertions, narrow applicability, or add an assumption. A property that needs an unavailable observation is returned as requiring an observer extension rather than bound to an undriven signal.

Model Checking (step 5). The *reachability gate* checks the property's cover before its assertion: an unreachable cover means the intended scenario was never exercised under the current harness, so the result is recorded as vacuous, not PASS. For a reachable property, the model-checking depth is set from the candidate's minimum latency and the first reached cover step, and the checker returns PASS, FAIL, or TIMEOUT.

4.3.2 Review and Repair

A raw solver status is conditional on the encoded property and environment, so this block first checks whether the evidence supports both before anything is regenerated. For PASS, structural checks over the admitted SVA count assertions, covers, and assumptions and flag disabled assertion enables or constant-true conditions. A reviewer then compares the source statement, property code, reachability record, structural checks, and RTL index and labels the property sound, weak, mismatched, or undecidable. Only a reachable PASS judged sound becomes a reviewed PASS; a weak or mismatched property yields targeted repair feedback.

Counterexample Analysis (step 6). For FAIL, ISAudit records a VCD counterexample and extracts the retirement and internal-signal trace around the failing step. The reviewer must state separately the facts visible in the trace, the behavior the source statement requires, and the behavior the relevant RTL path implements, and only then classify the failure as an RTL-defect candidate, a property error (including a binding error), an environment artifact, or inconclusive; this ordering discourages it from bending the expected behavior to fit the trace. Only an evidenced property defect is returned to formalization; conclusions about the RTL are never used as repair instructions.

Environment Check (step 7). Counterexamples can also exploit unconstrained environment inputs. When the reviewer identifies a configured request/acknowledgment signal as load-bearing, ISAudit rechecks the same property on a scratch copy of the RTL with the corresponding protocol constraint,

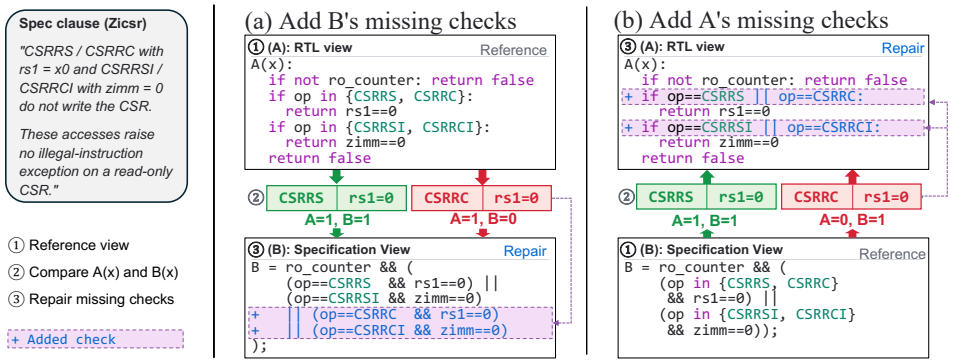

Figure 3: Bidirectional repair on a Zicsr clause (both SystemVerilog candidates shown as pseudocode). Where A and B disagree, the clause decides which view to repair.

leaving the canonical RTL untouched. A surviving counterexample that the second review again attributes to the RTL is retained as a reviewed RTL-defect candidate. A passing recheck classifies the original FAIL as an environment artifact. A vacuous, timed-out, or unsupported recheck, like free inputs outside the declared contract, leaves the result inconclusive.

Review findings close the loop: a weak or mismatched PASS and a property-error FAIL trigger a new heterogeneous-formalization attempt under the fixed requirement, and each reported bug carries its clause, property, trace, and decision provenance.

ISAUDIT Implementation. We implement ISAUDIT as a Python pipeline around Yosys (Wolf & Glaser, 2013), `yosys-smtbmc`, and Boolector (Niemetz et al., 2015). One core-parameterized driver runs indexing, formalization, checking, review, and reporting, persists the clause, SystemVerilog, SMT, log, VCD, and decision artifacts for each property, and exposes isolated direct- and dual-formalization modes for the comparison in Section 5.

5 EVALUATION

5.1 EVALUATION SETUP

We evaluate to answer three research questions: (**RQ1**) whether ISAUDIT finds more real bugs than the four baselines, (**RQ2**) how reliable its two modules, Refutation-Driven Formalization and Evidence-Backed Checking, are, and (**RQ3**) how sensitive its results are to the LLM backbone.

Setup. We evaluate five 32- and 64-bit cores—`PicoRV32`, `Kronos`, `VexRiscv`, `CVA6`, and `Rocket`—against CASCADE (Solt et al., 2024), RISCV-FORMAL (YosysHQ, 2026), CLAUDE CODE (Anthropic, 2026), and CODEX (OpenAI, 2026), with details and selection criteria in Appendix A. Our main metric is the number of previously unknown bugs, i.e., those not publicly reported in issue trackers or pull requests. Focusing on such bugs mitigates the risk of data contamination, especially for LLM-based tools: since previously unknown bugs could not have appeared in LLMs' pre-training corpora, their count reflects each tool's bug-finding ability. We also report precision by validating every finding against its ISA clause and RTL.

5.2 RQ1: BUGS FOUND IN REAL-WORLD CORES

Table 1 shows that ISAUDIT finds 43 true bugs (33 new) on all five cores with two false positives (95.6% precision), versus 1, 1, 6, and 11 new bugs for CASCADE, RISCV-FORMAL, CLAUDE CODE, and CODEX. RISCV-FORMAL's 42 mechanical failures cluster into ten root causes. One is a real defect (uninitialized `mepc` state on `Kronos`). The other nine stem from retirement-interface misalignment, configuration-dependent property assumptions, and under-constrained formal environments, and on `Rocket` alone RVFI adaptation issues produced 33 correlated failures in six root causes. Because the baseline reports property violations without any semantic or environmental review, such integration-level discrepancies surface directly as bug candidates. Root-cause attribution also changes the interpretation of CASCADE's three `Kronos` mismatches: one generated program

Table 1: Bug detection statistics across the five RISC-V designs. TP and FP count true and false positives grouped by root cause within each core and tool. Precision is TP/(TP+FP). The "New bugs" row counts previously unknown bugs among TP (the percentage in parentheses is how many more new bugs ISAUDIT finds than that tool). "–" indicates that RISCV-FORMAL was not run on `CVA6` because its existing verification flow does not provide a compatible RVFI harness and configuration for `CVA6`.

Design	CASCADE		RISCV-FORMAL		CLAUDE CODE		CODEX		ISAUDIT (ours)	
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
`PicoRV32`	1	0	0	3	1	1	2	2	6	1
`Kronos`	0	3	1	0	8	0	5	1	11	0
`VexRiscv`	0	0	0	0	0	4	6	0	19	1
`CVA6`	0	0	–	–	3	2	3	0	5	0
`Rocket`	1	0	0	6	0	0	4	1	2	0
Total	2	3	1	9	12	7	20	4	43	2
New bugs	**1 (+3,200%)**		**1 (+3,200%)**		**6 (+450%)**		**11 (+200%)**		**33**	
Precision	**40.0%**		**10.0%**		**63.2%**		**83.3%**		**95.6%**	

never executes the target CSR operation, while the other two mismatches survive after the alleged triggering instruction is neutralized. We therefore count them as attribution false positives, not detections of the target bug families. CASCADE targets execution-visible mismatches against a reference model, so defects that manifest only as missing or incorrect traps, such as CSR-existence checks, lie outside its reach. We include it as a representative differential fuzzer whose original evaluation covers all five cores. On `Rocket`, ISAUDIT confirms two bugs whereas CODEX finds four: the remaining two require longer temporal sequences or more specific architectural environments than the generated properties capture, so the current mechanical checking configuration does not expose them. This reflects the coverage of the generated properties, not a triage error, and it does not imply that the RTL is correct.

5.3 RQ2: COMPONENT ANALYSIS

5.3.1 QUALITY OF THE FORMALIZATION

Bug counts alone do not show whether the formalized properties are faithful to their clauses, because a property can pass every check while omitting a condition. We therefore compare Refutation-Driven Formalization with *Direct Formalization*, which gives one worker the union of the specification and RTL contexts and formalizes each clause along a single path without cross-checking, with the harness, mechanical gates, solver, and review unchanged. The comparison covers the integer computational instructions of the unprivileged specification: every core implements them, their deterministic results separate formalization errors from environment effects, and their `x0`, XLEN, shift, overflow, and sign-extension rules retain hardware-specific corner cases. For each core's configuration, we manually derive the applicable clauses into reference properties. Precision is the fraction of formalized properties that match a reference property, and recall is the fraction of reference properties for which ISAUDIT formalizes one.

Table 2: *Precision* (%) and *Recall* (%) of the properties formalized by *Direct Formalization* and *Refutation-Driven Formalization* across the five RISC-V designs.

	Direct Formalization						Refutation-Driven Formalization (Ours)					
	Kronos	PicoRV32	VexRiscv	Rocket	CVA6	**Overall**	Kronos	PicoRV32	VexRiscv	Rocket	CVA6	**Overall**
Precision	77.8%	73.7%	76.5%	62.2%	100.0%	**75.5%**	100.0%	89.5%	93.8%	100.0%	100.0%	**97.1%**
Recall	77.8%	77.8%	76.5%	80.0%	77.8%	**77.9%**	100.0%	94.4%	100.0%	93.3%	94.4%	**96.5%**

Table 2 shows that Refutation-Driven Formalization raises precision from 75.5% to 97.1% and recall from 77.9% to 96.5% across the five cores. The errors of direct formalization are semantic rather than syntactic: properties omit the immutable `x0` condition, weaken an XLEN-dependent overflow rule, or apply an RV32 interpretation to `Rocket`'s RV64 behavior. The three properties still incorrect under the dual path are isolated boundary omissions, two on `PicoRV32` missing one equivalent operand or encoding case and one on `VexRiscv` sampling an auxiliary retirement indication one cycle from the architectural retirement event. None recurs across cores or appears in both views.

The three missed reference properties are conservative abstentions: one requires a stable mapping for a multi-cycle architectural event and two require observations absent from the fixed harness, so ISAUDIT records the observation boundary rather than emitting a superficially complete property.

5.3.2 RELIABILITY OF THE TRIAGE

Within Evidence-Backed Checking, the model checker's result for a property is mechanical and reproducible. The fallible step is the review that attributes a FAIL to the RTL, the property, or the environment, because an LLM makes this attribution. We therefore evaluate the reliability of this triage: we group failing properties by independent root cause, manually label each cause as a bug or not against the ISA and execution evidence, and withhold these labels from Evidence-Backed Checking. Figure 4 compares the triage outcome with these labels: precision is the fraction of retained reports that are bugs, and recall the fraction of bugs that the triage retains. This recall covers only decisions after a mechanical FAIL.

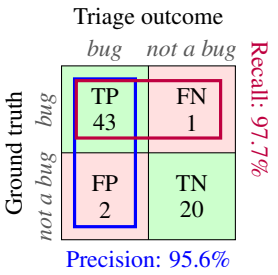

Figure 4: Confusion matrix for counterexample triage in Evidence-Backed Checking.

Figure 4 shows 43 true positives, 2 false positives, 20 true negatives, and 1 false negative, which give 95.6% precision and 97.7% recall. Both false positives stem from local mismatches between the property and the implementation's interface semantics: on `PicoRV32` the property took memory addresses as byte addresses whereas the observation interface reports word-aligned bus addresses, and on `VexRiscv` it assumed PC continuity across consecutive retirements although the RVFI observer does not record synchronously excepting instructions. In both cases the solver faithfully falsified the stated property, yet the counterexample violates no ISA rule. The 20 true negatives are property or environment failures correctly rejected by the review. The single false negative arose from conservative adjudication: the evidence at first review did not separate a property-expression issue from the RTL execution path, so the review returned the counterexample for property repair, and re-checking the RTL path along it confirmed a real bug. In sum, the review misjudges three of the 66 root causes, all at the boundary between property and implementation semantics. None reflects a systematic failure across designs.

5.4 RQ3: SENSITIVITY TO THE LLM BACKBONE

We assess backbone sensitivity on `VexRiscv` and `Kronos`, the two cores on which ISAUDIT finds the most bugs in RQ1, so that a difference between backbones is more likely to be visible in the bug count. We replace ISAUDIT's default backbones (see Appendix A) with `DeepSeek-V4-Pro` (an open-weight model) and `gpt-5.6-sol` in turn; each replacement applies to every LLM invocation in ISAUDIT, with all other settings unchanged. On `VexRiscv`, the (TP, FP) counts are (19, 1) with the default backbones, (11, 0) with `DeepSeek-V4-Pro`, and (17, 0) with `gpt-5.6-sol`; on `Kronos` they are (11, 0), (6, 1), and (10, 1). Across the two cores, `gpt-5.6-sol` retains 27 of the 30 true positives found with the default backbone (90.0%) at 96.4% precision, and `DeepSeek-V4-Pro` retains 17 (56.7%) at 94.4% precision. The number of confirmed bugs thus varies with backbone capability, while precision aggregated over the two cores stays at or above 94% in every configuration, because every reported bug must rest on a model-checker counterexample that survives review.

6 CONCLUSION

We presented ISAUDIT, which audits real RISC-V cores by constructing executable formal properties directly from the ISA specification. Refutation-Driven Formalization derives each property through two workers with different evidence boundaries and evidence-based adjudication, and Evidence-Backed Checking turns mechanical results into attributed verdicts through multi-source review. Heterogeneous construction raises formalization precision from 75.5% to 97.1%, and the triage retains 43 confirmed defects while rejecting 20 non-defects. On five open-source cores, ISAUDIT finds 43 confirmed bugs, 33 of them previously unknown, more than each of four baselines. Specification-derived, traceable properties adjudicated by mechanical evidence are a practical route to auditing real processor cores.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods and as baselines: as a component of the proposed method, a large language model (LLM) formalizes the ISA specification into properties, binds them to the RTL, and triages the results of model checking; as an object of study, two general-purpose coding agents are the baselines. The models and configurations are described in Appendix A. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, or interpret results. The remaining tasks with required disclosure (generate synthetic data sets, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, assist with translation, clean and reformat dataset, and support qualitative and thematic data analysis) are not applicable to this work.

Additionally, we used generative AI tools to edit the manuscript to improve readability. We have reviewed all AI-assisted work. The authors inspected every edit for accuracy and originality. All figures, tables, and citations are authored by us and are not AI-generated. Every finding reported in this paper was validated by the authors against the ISA specification and the RTL before inclusion. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects or personal data. All subject cores are open-source projects under permissive licenses, and the ISA specification is publicly available.

There may be a concern that malicious parties could also use ISAUDIT to find exploitable flaws in processor designs. However, ISAUDIT targets pre-silicon RTL, and developers can use it to find and fix discrepancies between a core and its ISA specification before tape-out, after which a hardware flaw can no longer be patched.

All previously unknown bugs found by ISAUDIT have been responsibly disclosed to the developers of the respective projects. For each bug, we provided a detailed report with the violated specification clause and a concrete counterexample trace to facilitate reproduction. Throughout this process, we have followed the responsible disclosure process.

REFERENCES

Anthropic. Claude Code: An agentic coding tool. `https://claude.com/product/claude-code`, 2026.

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019.

Yunsheng Bai, Ghaith Bany Hamad, Syed Suhaib, and Haoxing Ren. AssertionForge: Enhancing formal verification assertion generation with structured representation of specifications and RTL. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, 2025.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999.

Matej Bölcskei, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Encarsia: Evaluating CPU fuzzers via automatic bug injection. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

CHIPS Alliance. riscv-dv: Random instruction generator for RISC-V processor verification. `https://github.com/chipsalliance/riscv-dv`, 2026.

Yihan Dai, Sijie Liang, Haotian Xu, Peichu Xie, and Sergey Mechtaev. Reducing hallucinations in LLM-generated code via semantic triangulation. arXiv preprint arXiv:2511.12288, 2025.

Zihui Guo, Miaomiao Yuan, Yanqi Yang, Liwei Chen, Gang Shi, and Dan Meng. DiveFuzz: Enhancing CPU fuzzing via diverse instruction construction. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*, 2025.

Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DifuzzRTL: Differential fuzz testing to find CPU bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

Jaime Rafael Imperial and Hao Zheng. SpecAlign: A semantic alignment framework for SystemVerilog assertion generation. arXiv preprint arXiv:2605.25181, 2026.

Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. (Security) assertions by large language models. *IEEE Transactions on Information Forensics and Security*, 19, 2024.

Elliot Myunghoon Kim, Avi Garg, Kenny Peng, and Nikhil Garg. Correlated errors in large language models. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

Yanfang Liu, Mingjun Wang, Peng Xu, Rongliang Fu, Bei Yu, and Tsung-Yi Ho. AnalogVerifier: A neuro-symbolic framework for analog circuit verification. In *Forty-third International Conference on Machine Learning (ICML)*, 2026.

Xiaoyue Ma, Lannan Luo, and Qiang Zeng. From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter IoT devices. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

Nowfel Mashnoor, Hadi Kamali, and Kimia Azar. From language to logic: Bridging LLMs & formal representations for RTL assertion generation. In *2026 IEEE 44th VLSI Test Symposium (VTS)*, 2026.

Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0: System description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1), 2015.

OpenAI. Codex CLI: A lightweight coding agent that runs in your terminal. `https://github.com/openai/codex`, 2026.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023.

Alastair Reid. Who guards the guards? Formal validation of the Arm v8-M architecture specification. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 2017.

Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of ARM processors with ISA-Formal. In *Computer Aided Verification (CAV)*, 2016.

RISC-V International. riscv-arch-test: RISC-V architectural certification tests. `https://github.com/riscv/riscv-arch-test`, 2026a.

RISC-V International. The RISC-V instruction set manual: Volume I, unprivileged architecture, and volume II, privileged architecture. `https://github.com/riscv/riscv-isa-manual`, 2026b. Source revision 59c2aec (2026-09-03).

Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: CPU fuzzing via intricate program generation. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

Fabian Thomas, Eric García Arribas, Lorenz Hetterich, Daniel Weber, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. RISCover: Automatic discovery of user-exploitable architectural security vulnerabilities in closed-source RISC-V CPUs. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*, 2025.

Clifford Wolf and Johann Glaser. Yosys: A free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

Yuchao Wu, Wenji Fang, Jing Wang, Wenkai Li, Ziyan Guo, and Zhiyao Xie. AssertLLM2: A comprehensive LLM benchmark for assertion generation from design specifications. arXiv preprint arXiv:2605.27472, 2026.

Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. AssertLLM: Generating hardware verification assertions from design specifications via Multi-LLMs. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASPDAC '25)*, 2025.

Hengdi Ye, Bing Shui, Jielun Wu, Yufan Zhou, Baowen Xu, and Qingkai Shi. Generating precise format specification for network protocols through adversarial LLM interactions. In *35th USENIX Security Symposium (USENIX Security 26)*, 2026.

YosysHQ. riscv-formal: RISC-V formal verification framework. `https://github.com/YosysHQ/riscv-formal`, 2026.

Mingwei Zheng, Chengpeng Wang, Xuwei Liu, Jinyao Guo, Shiwei Feng, and Xiangyu Zhang. RFCAudit: AI agent for auditing protocol implementations against RFC specifications. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025.

A Evaluation Details

Table 3: Overview of the evaluated designs at the upstream commits used in our experiments (Version), detailing the ISA (optional extensions of VexRiscv in brackets), pipeline depth, hardware description language (HDL), lines of code of the package each project designates as the core (LoC; cloc, blanks and comments excluded; testbenches, SoC wrappers, deprecated packages and vendored sub-modules excluded).

Design	ISA	Pipeline	HDL	LoC	Version
`PicoRV32`	RV32IMC	multi-cycle	Verilog	2,664	`a473fc8`
`Kronos`	RV32I_Zicsr_Zifencei	3-stage	SystemVerilog	1,591	`13678d4`
`VexRiscv`	RV32I[M][A][F[D]][C]	5-stage	SpinalHDL	11,758	`c4b2a55`
`CVA6`	RV64IMAFDC	6-stage	SystemVerilog	32,016	`6348e9e`
`Rocket`	RV64GC	5-stage	Chisel	11,449	`55bcad0`

Subjects. We evaluate ISAudit on five real-world open-source RISC-V cores drawn from the evaluation suites of comparison tools (Solt et al., 2024; YosysHQ, 2026). They also span different complexities and ISA extensions: `PicoRV32`, a size-optimized, unpipelined 32-bit core; `Kronos`, a three-stage in-order 32-bit core; `VexRiscv`, a five-stage in-order 32-bit core with caches and an FPU, generated from SpinalHDL; `CVA6`, a six-stage in-order 64-bit core with an MMU that boots Linux; and `Rocket`, the five-stage in-order 64-bit core of the Chisel-generated Rocket Chip. They range from 1.6k to 32k lines of core RTL in four hardware description languages, and from a microcontroller core with no pipeline to Linux-capable application cores, as detailed in Table 3.

Baselines. Cascade (Solt et al., 2024) performs differential testing against a reference implementation, whereas riscv-formal (YosysHQ, 2026) checks hand-written properties. The two general-purpose coding-agent baselines, Claude Code (Anthropic, 2026) and Codex (OpenAI, 2026), receive the same RTL and ISA specification as ISAudit and are asked to audit the core. Claude Code uses `Claude-Opus-5.5` with reasoning effort set to `high` and Codex uses `gpt-5.6-sol` with reasoning effort set to `high`.

Metrics and validation. The end-to-end comparison reports true and false positives grouped by root cause, precision, and the number of previously unknown bugs. We validate each finding by inspecting the governing ISA clause, the implicated RTL, and the available counterexample or execution evidence. For Cascade, we additionally remove the alleged triggering instruction and rerun the program: a mismatch that persists is not attributed to that target bug. Unresolved candidates are excluded from TP, FP, and precision.

ISAudit and baseline configurations. ISAudit uses `Claude-Sonnet-5` for the specification-grounded worker, the adjudicator, and the reviewer, and the stronger `Claude-Opus-5.5` for the RTL-grounded worker, since grounding each property in the fixed design is the most challenging step; all roles run at temperature 0 with at most two evidence-driven repair rounds. *Direct Formalization* (§5.3.1) uses a single worker on `Claude-Opus-5.5` at the same temperature and repair-round budget; it receives the union of the two workers' contexts, $C_R(q) \cup C_S(q) = \{q, I, H, R_q, S, E\}$, formed by concatenation without further reduction, and formalizes the property along one path without cross-checking or adjudication, after which the candidate passes through the same harness, mechanical gates, solver, and review as the dual-path output. Properties are compiled with Yosys 0.68 and checked with SymbiYosys 0.68, smtbmc, and Boolector (Niemetz et al., 2015). Model checking uses a depth of 25–120 cycles, a timeout of 3,600 seconds per property, and a memory limit of 32 GB. For riscv-formal, depth and timeout follow each core's shipped configuration: 20–40 cycles and 180 s on `Kronos`, 24–40 cycles and 300 s on `PicoRV32`, 266–280 cycles and 900 s on `VexRiscv`, and 25–120 cycles and 900 s on `Rocket`, with the same 32 GB memory limit. Each Cascade campaign runs 10 rounds of 24 core-hours per core (240 core-hours in total) with 32 workers. Both agents receive the same inputs as ISAudit, i.e., the RTL and the ISA specification, and are asked to find violations.

Run statistics. Table 4 summarizes the main sweep of ISAUDIT per core: properties submitted to checking, solver outcomes, confirmed bugs, and solver time. The coverage-closure pass produced 96 deduplicated normative tags and 248 applicable clauses in total.

Table 4: Per-core run statistics of ISAUDIT: properties checked, solver outcomes, confirmed RTL bugs, and solver time.

Design	Prop.	PASS	FAIL	Vacuous	Timeout	Inconcl.	Bugs	Solver (h)
PicoRV32	68	53	15	0	0	0	6	2.37
Kronos	81	23	56	2	0	0	11	13.69
VexRiscv	49	7	40	1	0	1	19	2.28
CVA6	389	245	94	8	22	20	5	22.36
Rocket	326	225	62	7	16	16	2	19.60
Total	913	553	267	18	38	37	43	60.30

Detailed information on baseline findings. The two CASCADE true positives are the PicoRV32 reserved-field handling error in FENCE and the Rocket incorrect inclusion of ECALL in minstret. Its three Kronos reports cannot be attributed to the target bug families: one generated program never executes the target CSR operation, and in the other two cases the RTL/reference mismatch survives after the alleged triggering instruction is neutralized. We therefore count these reports as attribution false positives. RISCV-FORMAL produced 42 mechanical failures, which reduce to ten distinct candidates after root-cause clustering. Only one corresponds to an RTL defect (six Kronos branch checks share one root cause, uninitialized mepc); the remaining nine arise from retirement-interface misalignment, configuration-dependent property assumptions, and insufficiently constrained formal environments. On PicoRV32, the DIV/REM checks are affected by signed-expression and formal arithmetic semantics, and the liveness check permits indefinitely stalled external responses, yielding traces that cannot occur in a real environment; on Rocket, 33 failures reduce to six classes caused mainly by RVFI adaptation, where instruction, PC, register, and memory fields are not aligned to the same retirement event and internal pipeline signals are taken as architectural events. Because the baseline reports mechanical property violations without subsequent semantic and environmental review, these integration-level discrepancies appear directly as bug candidates. For CLAUDE CODE, 12 of 19 adjudicated root causes are true positives and seven are false positives.

B PREVIOUSLY UNKNOWN BUGS DISCOVERED BY ISAUDIT

Table 5: Detailed information on the 33 previously unknown bugs identified by ISAUDIT. Each entry records the violated specification clause, RTL location, root cause, and hardware failure class.

ID	Design	Clause	RTL location	Root cause	Failure class
PicoRV32 (4 findings)					
PIC-RTL-001	PicoRV32	Privileged ISA, minstret.	picorv32.v, count_instr update.	The counter is incremented at decode/dispatch instead of architectural retirement.	Counter/ retirement
PIC-RTL-002	PicoRV32	Unprivileged ISA, FENCE.	picorv32.v, FENCE decode and register writeback.	The reserved rd field can enter the normal writeback path.	Instruction semantics
PIC-RTL-005	PicoRV32	M extension, MUL/DIV/REM.	picorv32.v, picorv32_pcpi_mul and PCPI response arbitration.	A stale multiplier ready/result response is not reset-gated and can win arbitration over a later M-extension instruction.	Execution-unit arbitration
PIC-RTL-006	PicoRV32	Unprivileged ISA, misaligned stores.	picorv32.v, CATCH_MISALIGN, memory address, and write-strobe generation.	With trapping disabled, the effective address is aligned down and an SW still emits a full-word strobe instead of being split or rejected.	Memory alignment
Kronos (8 findings)					
KRO-RTL-001	Kronos	Zicsr, CSRRS[I]/CSRRC[I] no-write condition.	kronos_csr.sv, CSR write-enable decode.	Zero source masks are still classified as CSR writes, disturbing mcycle/minstret.	CSR semantics

Continued on the next page

Table 5 (continued)

ID	Design	Clause	RTL location	Root cause	Failure class
KRO-RTL-002	Kronos	Privileged ISA, mip and WFI.	kronos_csr.sv, mip readback; kronos_EX.sv, WFI wake logic.	Pending interrupt bits are masked by global/local enables before being exposed or used to resume WFI.	Interrupt/ WFI
KRO-RTL-004	Kronos	Privileged ISA, WFI and trap-return PC.	kronos_EX.sv, WFI interrupt trap and mepc update.	Trap entry saves the WFI instruction PC rather than the following PC, so MRET re-executes WFI.	Trap control
KRO-RTL-005	Kronos	Privileged ISA, minstret.	kronos_EX.sv, retirement pulse for ECALL/EBREAK.	The retirement counter is driven for instructions that instead raise synchronous exceptions.	Counter/ retirement
KRO-RTL-006	Kronos	Privileged ISA, reset, mepc, and mcause.	kronos_csr.sv, mepc/mcause registers and readback.	The registers lack defined reset state, and mepc lacks the required low-bit masking.	CSR reset state
KRO-RTL-007	Kronos	Privileged ISA, global interrupt enable.	kronos_EX.sv, registered core_interrupt trap decision.	The interrupt decision uses a two-cycle-old request without rechecking the current MIE or mip.	Interrupt control
KRO-RTL-009	Kronos	Privileged ISA, 64-bit counters on RV32.	kronos_counter64.sv, low-half write and carry update.	A low-half CSR write discards a simultaneously pending carry into the high half.	Counter atomicity
KRO-RTL-010	Kronos	RV32I source-register operand semantics.	kronos_EX.sv, instruction and operand pipeline registers.	Instruction and operand registers advance under different conditions, allowing an instruction to consume its predecessor's operands.	Pipeline datapath
VexRiscv (18 findings)					
VEX-RTL-002	VexRiscv	Privileged ISA, mstatus.MPP after MRET.	VexRiscv_csr.v, CsrPlugin MRET update.	MRET writes the U-mode encoding into MPP even in an M-only configuration.	Privilege state
VEX-RTL-003	VexRiscv	Privileged ISA, mepc alignment.	VexRiscv_csr.v, CsrPlugin mepc read/write and MRET target.	mepc[0] is stored and read without the required masking.	Trap-state CSR
VEX-RTL-004	VexRiscv	Privileged ISA, mtval on ECALL.	VexRiscv_csr.v, CsrPlugin exception metadata.	The generic illegal-instruction metadata path writes the ECALL encoding into mtval instead of zero.	Exception metadata
VEX-RTL-005	VexRiscv	Privileged ISA, mconfigptr.	VexRiscv_csr.v, CsrPlugin CSR decode table.	The mandatory CSR address has no implementation and falls into illegal access.	Missing CSR
VEX-RTL-006	VexRiscv	Privileged ISA, reset and mcause.	VexRiscv_csr.v, CsrPlugin mcause register.	mcause is left without a defined reset value.	CSR reset state
VEX-RTL-007	VexRiscv	Privileged ISA, mtval on interrupts.	VexRiscv_csr.v, CsrPlugin interrupt-trap update.	Interrupt entry does not overwrite mtval, so stale exception metadata remains visible.	Exception metadata
VEX-RTL-008	VexRiscv	Privileged ISA, mcounteren.	VexRiscv_csr.v, CsrPlugin CSR decode table.	A configuration with U-mode omits the required machine counter-enable CSR.	Missing CSR
VEX-RTL-009	VexRiscv	Privileged ISA, scounteren.	VexRiscv_csr.v, CsrPlugin supervisor CSR decode.	A configuration with S-mode omits the required supervisor counter-enable CSR.	Missing CSR
VEX-RTL-010	VexRiscv	Privileged ISA, misa.Extensions.	VexRiscv_csr.v, CsrPlugin misa constant/readback.	The generated extension bitmap sets B/G, omits I, and violates the E/I relationship for the implemented RV32I core.	ISA discovery CSR
VEX-RTL-011	VexRiscv	Privileged ISA, mstatus.MPRV.	VexRiscv_csr.v, CsrPlugin mstatus write mask.	MPRV is hardwired rather than writable when U-mode exists.	Privilege control
VEX-RTL-012	VexRiscv	Privileged ISA, mstatus.TW.	VexRiscv_csr.v, CsrPlugin mstatus write mask.	TW is hardwired even though a less-privileged mode is implemented.	Privilege control
VEX-RTL-013	VexRiscv	Privileged ISA, mstatus.TSR.	VexRiscv_csr.v, CsrPlugin mstatus write mask.	TSR is hardwired in a configuration that implements S-mode.	Privilege control
VEX-RTL-014	VexRiscv	Privileged ISA, sepc alignment.	VexRiscv_csr.v, CsrPlugin sepc read/write.	sepc[0] is stored and read without the required masking.	Trap-state CSR
VEX-RTL-015	VexRiscv	Privileged ISA, menvcfg/menvcfgh.	VexRiscv_csr.v, CsrPlugin CSR decode table.	Both environment-configuration CSRs are omitted despite the implemented lower privilege modes.	Missing CSR
VEX-RTL-016	VexRiscv	Privileged ISA, scause.ExceptionCode.	VexRiscv_csr.v, CsrPlugin scause field width.	Only four exception-code bits are implemented instead of bits [4:0].	CSR field width

Continued on the next page

Table 5 (continued)

ID	Design	Clause	RTL location	Root cause	Failure class
VEX-RTL-017	VexRiscv	Privileged ISA, `mstatus.TVM`.	`VexRiscv_csr.v`, CsrPlugin `mstatus` write mask.	`TVM` is hardwired although S-mode and writable `satp.MODE` are present.	Privilege control
VEX-RTL-018	VexRiscv	Privileged ISA, `sstatus` view of `mstatus`.	`VexRiscv_csr.v`, CsrPlugin `sstatus` read mask.	The supervisor view exposes `MPRV` at bit 17 instead of reading the WPRI position as zero.	CSR view/ WPRI
VEX-RTL-019	VexRiscv	Privileged ISA, `medeleg`/`medelegh` on RV32.	`VexRiscv_csr.v`, CsrPlugin machine-delegation CSR decode.	The XLEN=32 S-mode configuration implements `medeleg` without its required high-half CSR window.	Missing CSR
`CVA6` (2 findings)					
CVA6-RTL-001	CVA6	`norm:medelegh_sz_acc_enc_xlen32` (`machine.adoc:1307`).	`core/csr_regfile.sv`, missing `CSR_MEDELEGH`; default exception at line 1003.	The RV32 S-mode configurations omit the only CSR window for `medeleg[63:32]`.	Missing CSR
CVA6-RTL-002	CVA6	`norm:pmp_entry_count` (`machine.adoc:3341`).	`core/include/*_config_pkg.sv`, `NrPMPEntries=8`; `core/csr_regfile.sv`:381–382, 2922.	Eleven shipped configurations select eight PMP entries, while the architecture permits only 0, 16, or 64.	PMP configuration
`Rocket` (1 finding)					
ROC-RTL-001	Rocket	Privileged ISA, `mstatus.TW`.	`CSRFile.sv`, `TW` presence/write mask (`usingSupervisor` gate).	`TW` is conditioned on S-mode instead of the existence of any mode below M, so an M+U hart hardwires it to zero.	Privilege control