SPECGUARD: WHAT SHOULD LLMS SEE TO GENERATE ASSERTIONS THAT CATCH RTL BUGS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly used to write test oracles from natural-language specifications. The artifact under test is often part of the model's context. When that artifact is incorrect, the context contains the defect the oracle is meant to detect, and the model may encode it in the oracle. This is a form of information leakage, and no check applied after generation can reverse it. We study this problem in hardware assertion generation, where injected bugs provide ground truth for detection. SystemVerilog Assertions (SVA) generated by frontier models detect only 5.1–11.7% of injected bugs. An analysis of suites from prior generators reveals four failure modes. Two attributes make these failures irreparable after generation. First, no logically equivalent rewrite removes a restated defect. Second, retaining only the assertions that pass on the design under test discards exactly those that detect its bug. We formalize the remedy as information-constrained generation, in which the model's context depends on the design only through structure that its defects leave unchanged. We instantiate this formulation as SpecGuard, which withholds the implementation's behavior, poses one question per enumerated case, and derives temporal form and scope deterministically. SpecGuard was evaluated on the 83-design AssertLLM2 benchmark, using a local 14.8-B model with int4 quantization. Its assertions detected up to 66% of the AssertLLM2 bugs in a single run, compared with 11.7% for the best-performing of six frontier models. Furthermore, on more complex hierarchical designs from NVDLA and Google Coral NPU, it achieves up to a 51% bug-detection rate, compared with 14% for existing state-of-the-art assertion-generation techniques.

1 INTRODUCTION

Language models are increasingly used to generate the checks against which code is evaluated, ranging from unit-test assertions to postconditions derived from natural-language descriptions of intended behavior (Endres et al., 2024). This practice introduces a subtle risk, which can be illustrated with a simple example: suppose a model is asked to write a test for a function whose loop bound is off by one. If the function's implementation appears in the prompt, the model may read the incorrect bound directly and generate an assertion that reflects it. The resulting test passes when run against the flawed implementation, yet fails once the bug is fixed. While this is the defect the test was meant to detect, has instead been encoded into the oracle itself. This is a form of information leakage: the input to the model contains information about the quantity its output is later evaluated against (Kaufman et al., 2012). Empirical evidence supports this concern. Examining 24 open-source Java projects, Konstantinou et al. (2024) find that LLM-generated test oracles frequently capture what a program actually does rather than what it is supposed to do.

The same risk arises in hardware verification, where it can be measured exactly. There the oracles are SystemVerilog Assertions (SVA), checked formally or in simulation against a design's register-transfer level (RTL) description (Kern & Greenstreet, 1999). An assertion that has absorbed a defect holds on the design that carries it. We therefore measure the risk by mutation testing: we insert a known bug into a correct design, and an assertion detects it only if it holds on the correct design and fails on the mutated one (DeMillo et al., 1978). Existing LLM generators let the design under test into generation in different ways. AssertAIN places the RTL in the prompt, ChIRAAG revises its assertions from simulation logs of the RTL, and AssertLLM reads the specification together with the signals' widths (Tarek et al., 2026; Mali et al., 2024; Yan et al., 2025). Their assertions are fluent

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.

and well formed. Yet on the AssertLLM2 benchmark, assertions from frontier models detect only 5.1–11.7% of injected bugs (Wu et al., 2026a), and published evaluations find that neither the choice of model nor its scale moves this rate appreciably (Section 3.1). We therefore argue that the limit lies in what the model is shown and what it is asked.

To identify this limit, we examine the test suites produced by prior generators and trace their missed bugs to four underlying failure modes, each rooted in a flaw in what the model is shown or asked to do: **implementation capture** (restating the RTL rather than the specification), **missing structure** (the absence of register or wire information), **wrong scope** (visibility limited to top-module signals), and **incomplete enumeration** (a single question standing in for many cases) (Wu et al., 2025; Mashnoor et al., 2026; Wang et al., 2025). Two properties render these failures irreparable once generation has occurred. The first, **equivalence invariance**, holds because no downstream rewrite can remove a defect that has simply been restated rather than corrected. The second, **self-filtering**, arises when only assertions that pass on the design under test are retained, a process that discards precisely those assertions capable of detecting the design's bug. Section 3 measures the prevalence of each failure mode, Appendix B examines them across our measured runs, and Section 4.1 provides a formal proof of the two properties.

We therefore constrain the model's input rather than repair its output. We formalize **information-constrained generation**, in which the context depends on the design only through structure that its defects leave unchanged (Section 4.1). Structure is which signals a design declares and when each may change, whereas behavior is what the RTL computes. SpecGuard instantiates this formulation for SVA. A parser extracts the structure into an inventory, and only the inventory enters a prompt. Code derives each question, together with the timing and scope of each assertion, from the inventory. The model supplies only the relation, which it takes from the specification. Our main contributions are as follows:

- We trace missed bugs to four failure modes in what the model is shown or asked. Two attributes make these failures irreparable after generation explaining why stronger models and refinement loops barely raise detection, and why compile and proof rates overstate quality.
- We cast oracle generation as a leakage problem and formalize information-constrained generation. This gives the model the artifact's structure but not its behavior. We prove this constraint must hold at the input and reduces the model's task to the relation, since timing and scope follow from structure.
- We introduce SpecGuard, which instantiates the formulation for SVA and asks one question per case the structure enumerates. With a local 14.8B model quantized to int4, its generated assertions detect upto 66% of AssertLLM2 bugs against 11.7% for the strongest of six frontier models (Wu et al., 2026a). Furthermore, SpecGuard detected upto 51% of bugs in hierarchical designs from NVDLA and Google Coral NPU.

2 Background and Related Work

Assertions and their evaluation. An SVA assertion is checked at every clock edge, in simulation by a tool such as Verilator (Snyder, 2004) or formally by a tool such as SymbiYosys with the Z3 solver (YosysHQ, 2024; De Moura & Bjørner, 2008). Bounded model checking (BMC) restricts a formal proof to a fixed number of cycles. An assertion fixes four things: the signals it names, the relation among them, its temporal form, and the module it is bound into. The temporal form places the relation in time: `A |-> B` requires `B` in the cycle where `A` holds, and `A |=> B` one cycle later. Mutation testing measures whether a suite catches bugs. A mutant applies one small change to the design, and an assertion detects it when it holds on the correct design and fails on the mutant. In RTL, a mutant rewrites one operator, constant, signal, bit range, condition, assignment or clock edge. AssertLLM2 applies this method to 83 open-source designs with 20 mutants each (Wu et al., 2026a).

Oracles and what a model is shown. Judging an oracle requires an independent definition of correct behavior (Weyuker, 1982), which a prose specification does not provide (Barr et al., 2014). Formal verification against the RTL is therefore only a pseudo-oracle, since it certifies the RTL wherever the two disagree. A survey of language-model oracles recommends surfacing such disagreements (Mughal & Bilal, 2026). Retrieval (Lewis et al., 2020), prompt compression (Jiang et al., 2023) and tool-use loops (Yao et al., 2022) control how much context a model receives. Accuracy degrades

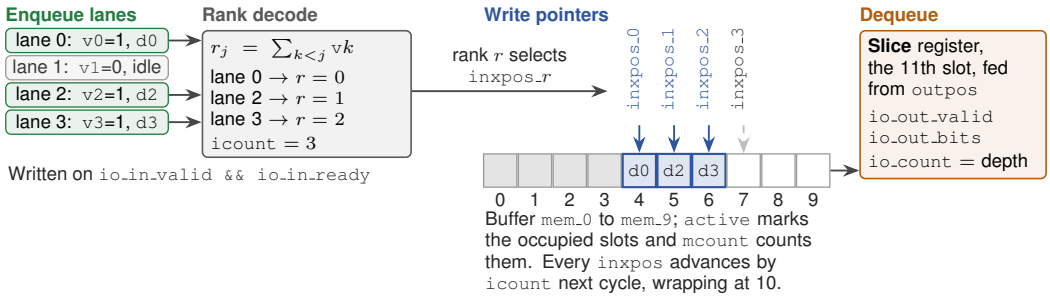

Figure 1: FifoX, the running example, during one enqueue cycle with lanes 0, 2 and 3 valid.

when irrelevant material surrounds the relevant (Liu et al., 2024), and self-refinement overstates its own correctness (Huang et al., 2025; Tang, 2026). None of these techniques decides what a model must not be shown, which in learning terms is whatever leaks the target (Kaufman et al., 2012).

Assertion generators. Prior generators differ mainly in what they condition on. The first group reads the RTL. GoldMine mines invariants from simulation traces (Vasudevan et al., 2010), AssertionForge fuses a specification graph with RTL facts (Bai et al., 2025), STELLAR prompts once per RTL execution path (Rajabi et al., 2025), and AssertCoder, DeepAssert, AssertGen and AssertMiner reach internal signals through the RTL or its hierarchy (Tian et al., 2025; Wang et al., 2025; Lyu et al., 2025; 2026). These generators risk implementation capture. The second group reads only the specification. AssertLLM prompts once per signal (Yan et al., 2025), and Spec2Assertion yields mostly width and connectivity checks (Wu et al., 2025), the symptom of missing structure (Mashnoor et al., 2026). The third group selects on the design under test. ChIRAAG refines until no assertion fails in simulation (Mali et al., 2024), SANGAM keeps what formal verification proves (Gupta et al., 2025), and related loops judge or prove outputs on the RTL (Tarek et al., 2026; Krishnamurthy et al., 2026). These criteria are self-filtering, and their gains in proven assertions plateau (Nie et al., 2026). Learning-based work trains specialized models on RTL-grounded data or against a property-equivalence verifier (Wu et al., 2026b; Zou et al., 2026), and is scored by equivalence to reference assertions rather than by detection. Benchmarks check assertions formally (Pulavarthi et al., 2025), on hierarchies (Nie et al., 2026), and by mutation (Wu et al., 2026a).

3 Case Study: Where Generated Oracles Fail

As a case study, we use FifoX, a buffer in Google's Coral neural processing unit (NPU) (Google Coral Team, 2026). We inject 48 bugs into it, and Appendix B reports every run in full. FifoX stores ten entries, `mem_0` to `mem_9` (Figure 1). Each cycle, it accepts data on four input **lanes**. A valid lane is written through a write pointer. The pointer's **rank** is the number of valid lanes below that lane. In the cycle shown, lane 3 uses the pointer of rank 2.

3.1 Model or Information?

Prior studies find that model choice matters little. On AssertLLM2, six frontier models detect 5.1–11.7% of injected bugs, and the highest compile rate does not belong to the best detector (Wu et al., 2026a). Model scale yields no consistent gains either: a fine-tuned 14B model outperforms GPT-5 with one attempt per task, but loses with sixteen (Wu et al., 2026b).

Neither study explains why a stronger model detects so few additional bugs. Therefore we run three existing generators on a local Qwen3-14B (int4) and on GPT-4o, score each suite against the 48 FifoX bugs (Table 1) (Yan et al., 2025; Tarek et al., 2026; Mali et al., 2024). The stronger model improves every surface measure: it writes more assertions, halves the width checks, and raises coverage for two of three systems. Detection, however, barely moves; five of six suites detect none of the 48 bugs, and the sixth detects only 4. Three of those four change one expression, checked only by AssertLLM on GPT-4o via `(mcount <= 10 - icount) |-> io_in_ready`.

A stronger model, however, cannot answer a question it is never asked. A bug changes what a register becomes on the next cycle, when it may change, or what it holds after reset. Detecting it requires a

Table 1: Prior generators on FifoX with a local Qwen3-14B and GPT-4o. Coverage: driven signals in some assertion's cone of influence.

System	Model	Assertions	Width-only (%)	Single instant (%)	Coverage (%)	Detected
AssertLLM	Qwen3-14B	2	100	–	28.6	0 of 48
	GPT-4o	31	39	37	71.4	4 of 48
AssertAIN	Qwen3-14B	4	0	0	14.3	0 of 48
	GPT-4o	8	0	0	57.1	0 of 48
ChIRAAG	Qwen3-14B	17	0	0	42.9	0 of 48
	GPT-4o	8	0	0	42.9	0 of 48

claim that relates a signal across cycles under a stated condition, and such a claim covers only the case its condition names. Prior generators ask one question per property the specification states, yet a property can cover many cases. On FifoX, one rule decides which pointer each lane writes through, and it covers ten cases as shown in Figure 1. One question about that rule yields one assertion and leaves nine cases unchecked. It does not depend on the model used.

Table 2: FifoX detection by question granularity, with the model fixed.

Question asked	Assertions	Hold (%)	Detected (%)
One per property the specification states	77	43 (55.8)	5 (10.4)
One per lane	50	43 (86.0)	17 (35.4)
One per lane and pointer pair	90	90 (100.0)	41 (85.4)
Both, over three repetitions	622	410 (65.9)	136 of 144 (94.4)

We test this by fixing the model and varying question granularity. Every arm uses the local Qwen3-14B, reads the same specification and signal list, and is scored against the same 48 bugs. An assertion **holds** when it passes on correct RTL, and **detects** a bug when it **holds** and fails on the RTL carrying that bug. Table 2 reports one repetition per arm. One assertion per property, as prior generators ask, detects 5 of the 48 bugs. One question per lane detects 17. One question per reachable lane and pointer pair, the 10 cases of Figure 1, detects 41. It shows, what the model is asked matters more than which model answers. We omit this case from the evaluation results, as it arose during debugging and involved human intervention.

3.2 Four Failure Modes

To find what the suites miss, we start from the surviving bugs. Each is traced to the signal its mutation changes, and to what the generator did about that signal: it (i) asked nothing, (ii) received no answer, (iii) refused the answer it received, or (iv) shipped a claim too weak to fail. Bugs that share an

Table 3: Bugs detected under two conditions differing only in what each failure mode or attribute concerns. Equivalence invariance is a logical identity, not a measurement.

Failure	First		Second	
	Condition	Detected	Condition	Detected
Implementation capture	pointer value read from the buggy RTL	0 of 2	pointer value implied by the specification	2 of 2
Missing structure	register claimed in its input cycle	10 of 20	register claimed one cycle later	14 of 20
Wrong scope	bug below the top module	28 of 94	bug in the top file	54 of 125
Incomplete enumeration	one question per lane	2 of 14	one question per lane and pointer pair	11 to 12 of 14
Self-filtering	suite kept by validation on the buggy design	0 of 2	every property written	2 of 2
Equivalence invariance	the copy as written	0 of 2	the copy rewritten equivalently	0 of 2

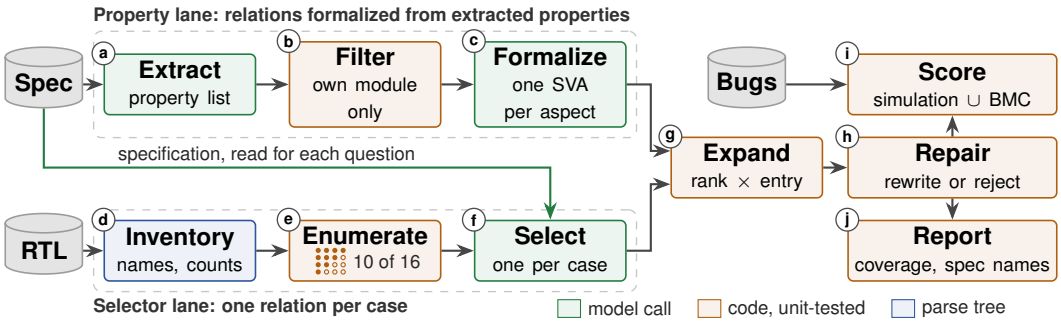

Figure 2: Overview of SpecGuard.

outcome form one mode. Subsequently, we measure each mode by comparing two conditions that differ only in that respect, scored by the same harness against the same bugs (Table 3).

Four modes account for the misses, and each is a gap in the model's input. **Implementation capture** is the leakage of Section 1: the assertion states the relation the RTL implements rather than the one the specification requires (Appendix B.3). **Missing structure** lacks a grounding fact, so a register claim is placed in the cycle of its inputs rather than the cycle after (Appendix B.4). **Wrong scope** gives the model a context narrower than the defect: the assertion is bound to the top module, whereas the bug never reaches an output (Appendix B.5). **Incomplete enumeration** is an underspecified query, in which one answer stands for every case a rule covers (Appendix B.6).

Two attributes keep these failures in place after generation. Under **equivalence invariance**, no logically equivalent rewrite removes a captured relation. Under **self-filtering**, keeping only the assertions that pass on the design under test discards exactly those that detect its bug. Both of them are detailed in Section 4.1, and proven in Appendix C.1.

None of the four failures is visible to the checks a generator is usually judged by. In one AssertLLM2 repetition, 193 of the 280 assertions that hold on correct RTL detect no bug, yet a syntax or proof rate counts each as a success. The failures must therefore be prevented while an assertion is generated.

4 SpecGuard Framework

SpecGuard (Figure 2) takes a natural-language specification and the RTL of the design under verification (DUV), but the model never sees the RTL. SpecGuard parses it with the slang SystemVerilog parser (Popoloski, 2024) into an **inventory** ⓓ. A property lane (ⓐ–ⓒ) and a selector lane (ⓔ–ⓕ) obtain relations from the specification, and code expands, repairs and scores the resulting assertions (ⓖ–ⓙ). Section 4.1 states the problem SpecGuard solves, and Algorithm 1 gives the procedure. We use the circled letters throughout the paper to denote blocks from Figure 2.

4.1 Problem Formulation

The formulation asks three things of the artifact under test: a specification, a structure that can be read without evaluating behavior, and defects that leave that structure unchanged. A program's signatures and types would serve as the structure for unit-test oracles (Konstantinou et al., 2024). We state it for RTL, where all three hold and detection has ground truth. We write a design as $D = (\sigma, \beta)$. Its **structure** σ is what the inventory records, the facts a parse tree states without evaluating an expression. It contains the signal set V and four facts about each $v \in V$: its width, its **module scope** $\mu(v)$, its **drive kind** $\kappa(v) \in \{\text{REG}, \text{WIRE}\}$, and whether it may **hold**. The module scope is the module that declares v, and v may hold when it is written under a condition other than reset. Indexed signals sharing a stem, such as `mem_0` to `mem_9`, form a **signal family**, and the clock and the reset's polarity complete σ. Its **behavior** β is the set of assignments that compute each signal. The correct design is $D^\star = (\sigma, \beta^\star)$, a mutant is $D_m = (\sigma_m, \beta_m)$ for $m \in M$, and the generator receives the specification S and a design under verification $D_u \in \{D^\star\} \cup \{D_m\}_{m \in M}$ without knowing which.

An assertion $a = (N_a, r_a, \tau_a, b_a)$ names signals $N_a \subseteq V$, states a relation r_a among them, places it in time by its temporal form τ_a, and is bound into a module b_a (Section 2). Writing $D \models a$ when a holds on every trace the checker explores, an assertion detects a mutant, and a suite A scores, as:

$$\det(a, m) = [D^\star \models a] \wedge [D_m \not\models a], \qquad \mathrm{Det}(A) = \tfrac{1}{|M|}\,\big|\{m \in M : \exists a \in A,\ \det(a, m)\}\big|. \quad (1)$$

A generator maps a context c and a question q to an answer, $G_\theta(c, q)$. **Information-constrained generation** chooses a context builder ϕ, a question set Q and a repair R to solve

$$\max_{\phi,\,Q,\,R}\ \mathbb{E}\Big[\mathrm{Det}\Big(R_\sigma\big(\{G_\theta(\phi(S,\sigma), q) : q \in Q(\sigma)\}\big)\Big)\Big] \quad \text{s.t. } \phi, Q, R \text{ read } D_u \text{ only through } \sigma, \quad (2)$$

and none of them evaluates an assertion on D_u. Each of the four failure modes of Section 3.2 break this program. Implementation capture lets $\beta(D_u)$ into the context, missing structure withholds κ from it, wrong scope ignores μ, and incomplete enumeration leaves Q to the model. Assumption 1, Lemmas 1 and 2, and Propositions 1 and 2 justify each constraint. These are proven in Appendix C.1.

Assumption 1 (Structure-preserving mutants)**.** *Each scored mutant changes one statement of the RTL and leaves every declaration intact, retaining the correct design structure, $\sigma_m = \sigma$.*

The seven mutation operators of Appendix D.4 rewrite an expression, a condition, an assignment or a clock edge inside a statement. A condition mutant can nonetheless remove the gate that lets a register hold, and 81% of the scored mutants satisfy the assumption when σ is parsed from each mutant and compared with $D^\star$.

Lemma 1 (Structure carries no bug)**.** *Under Assumption 1, the context $\phi(S, \sigma(D_u))$ is the same whether D_u is the correct design or a mutant. The distribution of the generated suite therefore does not depend on which bug, if any, D_u carries.*

A context carrying $\beta(D_u)$ differs from the correct design's exact bug location, leaking the quantity the suite is scored against (Kaufman et al., 2012). The leak cannot be removed after generation.

Proposition 1 (Equivalence invariance)**.** *If $D_u = D_m$ and a restates β_m, so that $D_m \models a$, then $\det(a, m)$ is false, and so is $\det(a', m)$ for every a' logically equivalent to a.*

A repair preserves meaning, therefore, it cannot undo a capture. The constraint of Equation 2 thereby sits on ϕ rather than on R.

Proposition 2 (Self-filtering)**.** *Let $F_u(A) = \{a \in A : D_u \models a\}$ be the assertions that hold on D_u. If that design is the mutant D_m, no assertion in $F_u(A)$ detects m.*

An acceptance rule evaluated on the design under test thus removes exactly the detectors of its bug.

Lemma 2 (Structural determinacy)**.** *Given the relation r_a, the binding and the temporal form are functions of σ. The binding b_a is a module that declares every $v \in N_a$. A claim about the value of a register v is placed one cycle after its operands and stated over their `$past` values, admitting $v =$ `$past`(v) as well when v may hold. A claim about a wire is placed in the cycle of its operands, and the reset guard follows the reset's polarity.*

Consequently R_σ needs no model call, and the model's task reduces to the relation, $r_a \sim p_\theta(\cdot \mid S, \phi(\sigma), q)$. The questions follow from σ as well. A rule over a signal family has a case set $C(\sigma)$ that code computes, and one question per case guarantees that every case is asked, whereas one question per family leaves C to the model, which covered at most 5 of FifoX's 10 lane and pointer pairs over six repetitions (Section 3.2).

4.2 Context construction

Context construction implements ϕ (Algorithm 1, lines 2–6). A restatement of the RTL cannot be repaired once written (Proposition 1). Therefore, the rule holds on the prompt: the model receives the names it may use and when each signal may change (Lemma 2), never how a signal is computed. SpecGuard therefore replaces the RTL with an **interface card** built from the inventory, which names the RTL's constants but never their values. A guard (line 15) halts the run whenever a prompt contains an RTL construct, such as an `always` block, beyond a one-line quotation in the specification. A scope filter ⓑ withholds from the top module any property about another module and routes it to the module that declares its signals. The card's remaining rules, and the handling of supplementary documents are detailed in Appendix C and Appendix D.11, respectively.

Algorithm 1: Information-constrained oracle generation. Every model call is G_θ, and letters ⓐ–ⓙ are the stages of Figure 2; Sections 4.2–4.3 and Appendix D.7 instantiate each step for SVA.

Input: Specification S, artifact under test D_u with top unit t, model G_θ
Output: Oracle suite A

1 **Function** `BuildContext` (S, D_u) **:**
2 $\quad \sigma \leftarrow \textsc{ExtractStructure}(D_u)$; /* ⓓ names, scopes, update timing; no value evaluated */
3 $\quad c_\mu \leftarrow \textsc{StructureView}(\sigma, \mu)$ for each unit μ ; /* constants by name, never by value */
4 $\quad P \leftarrow G_\theta(c_t, \textsc{Requirements}(S))$; /* ⓐ what the specification states */
5 $\quad P_t \leftarrow \{p \in P :$ every element p names lies in $t\}$; /* ⓑ scope filter */
6 $\quad P_\mu \leftarrow$ each other $p \in P$, routed to the one unit μ declaring its elements, else dropped;
7 $\quad$ **return** $\sigma, \{c_\mu\}, \{P_\mu\}$;
8 **Function** `Decompose` $(S, \sigma, \{c_\mu\}, \{P_\mu\})$ **:**
9 $\quad Q \leftarrow \{(p, x, \mu) : p \in P_\mu,\ x \in G_\theta(c_\mu, \textsc{Aspects}(p))\}$; /* ⓒ one question per aspect */
10 $\quad Q \leftarrow Q \cup \{(\rho, k) : \rho \in \textsc{Rules}(\sigma),\ k \in C_\rho(\sigma)\}$; /* ⓔ one per case, cases by arithmetic */
11 $\quad Q \leftarrow Q \cup \{(\textsc{update}, v) : v \in \textsc{StateElements}(\sigma)\}$;
12 $\quad A \leftarrow \emptyset$;
13 $\quad$ **foreach** $q \in Q$ **do**
14 $\quad\quad \pi \leftarrow \textsc{Prompt}(S, c_{\mu(q)}, q)$;
15 $\quad\quad$ **if** π *outside its quoted* S *reveals behavior of* D_u **then halt**;
16 $\quad\quad y \leftarrow G_\theta(\pi)$; /* ⓕ one relation */
17 $\quad\quad$ **if** $y \neq \textsc{Skip}$ **then**
18 $\quad\quad\quad A \leftarrow A \cup \textsc{Expand}(y, \sigma)$; /* ⓖ one oracle per member of a family */
19 $\quad\quad$ **else**
20 $\quad\quad\quad$ record q as silence;
21 $\quad$ **return** A;
22 **Function** `Repair` (A, σ) **:**
23 $\quad$ **foreach** $a \in A$, *with* b_a *the unit whose view its question used* **do**
24 $\quad\quad$ **if** a *names an element outside* b_a, *or cannot be parsed* **then** reject a;
25 $\quad\quad$ suspend a during reset, by the reset's polarity;
26 $\quad\quad$ **if** a *claims a state element* v *in the step of its operands* **then**
27 $\quad\quad\quad$ move the claim one step later, over the operands' previous values, admitting v unchanged if v may hold;
28 $\quad$ **return** A ; /* ⓗ no model call; no rule evaluates a on D_u */
29 $A \leftarrow$ `Repair(Decompose(`S, `BuildContext(`S, D_u`)), `σ`)`;
30 report the verdict of each $a \in A$ on $D^\star$ and each D_m ; /* ⓘ never used to select */

4.3 Question Decomposition

Decomposition implements $Q(\sigma)$ (lines 9–18). The selector lane asks one **question** per case a rule covers, and one per member of each family the inventory lists, such as each register that may hold and each one-bit flag, and it repeats these questions for every submodule. Deterministic program computes the cases as shown in ⓔ from Figure 2 . On FifoX, the rank of lane j cannot exceed j, so four lanes yield 10 reachable lane and pointer pairs rather than 16. An unanswered question is recorded as silence rather than turned into an assertion, and expansion ⓖ turns each answer into one assertion per entry of the family. Table 20 lists every question, and Appendix C works through FifoX.

Subsequently, Repair (lines 24–26) sets each answer's timing, reset guard and module binding from the inventory without a model call, as Appendix D.7 details. Each assertion's verdict on each bug is reported but never used to select an assertion to avoid self-filtering (Proposition 2).

5 Evaluation

5.1 Experimental setup

Experiments run on an AMD Ryzen 9 9950X with an NVIDIA RTX 5090. SpecGuard runs a local Qwen3-14B under int4 quantization, served by Ollama, and the baselines run GPT4o. Every arm is

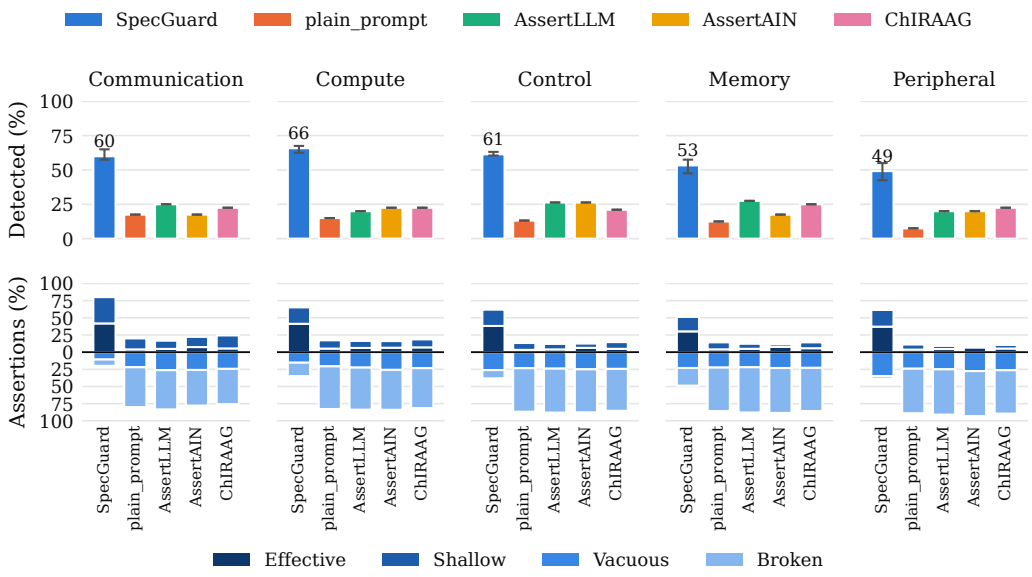

Figure 3: Bugs detected and suite composition per design family of AssertLLM2. The error bar is the range over three repetitions.

run three times, and we report the mean with the range over repetitions. We evaluate on AssertLLM2, whose 83 designs each ship 20 single-bug mutants (Wu et al., 2026a), and on five hierarchical designs: Coral's FifoX (Google Coral Team, 2026) and four NVDLA blocks (NVIDIA Corporation, 2026). Further details are provided in Appendix A.

5.2 Metrics

Detection is the fraction of bugs for which at least one assertion holds on correct RTL and fails on the buggy RTL. We also classify every shipped assertion. It is **broken** when it fails on correct RTL, **vacuous** when it holds but its antecedent never fires, **shallow** when it fires but constrains no value, and **effective** when it fires and fixes a signal's value. A suite can be sound and still catch nothing, so we report both measures (Endres et al., 2024; Schuler & Zeller, 2011).

5.3 Detection results

In Figure 3, we compare SpecGuard with the benchmark's own specification-only prompt and three prior generators (Yan et al., 2025; Tarek et al., 2026; Mali et al., 2024). Every tool reads the same specification and is scored by the same harness against the same mutants. SpecGuard detects upto 66% of AssertLLM2 mutants against 27% for the strongest baseline, and leads on all of the five families of designs. The composition of generated assertion explains the gap. At least SpecGuard's effective assertions range from 13% to 24% and against 0–7% for the baselines. Detailed results and discussions on why the number is at this range are shown in Appendix D.1.

5.4 Detection on larger designs

AssertLLM2 places most logic in the top module, therefore, wrong scope barely shows there. We therefore test five hierarchical designs of up to 435 modules at depth seven. None ships bugs, therefore we inject single-point mutants (Appendix D.4). Most logic lies below the top, and screening keeps 74.5% of FifoX's mutants against 3.5–8.9% on the NVDLA blocks.

In Figure 4, we report the results against the baselines' configurations (Yan et al., 2025; Tarek et al., 2026; Mali et al., 2024). SpecGuard detects upto 52% of the bugs on the NVDLA blocks, and the

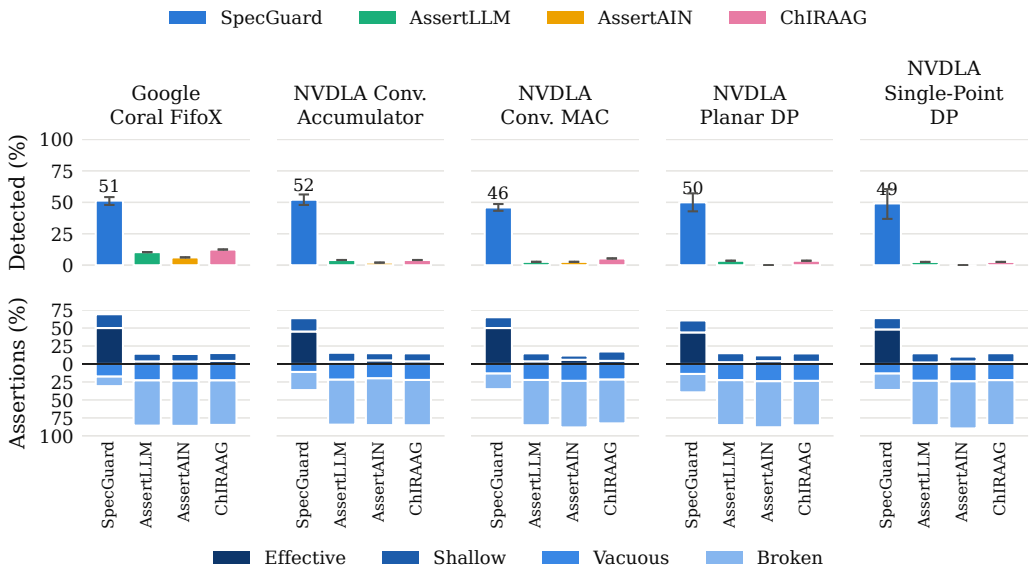

Figure 4: Bugs detected, above, and what each suite is made of, below, on Google Coral's FifoX and NVDLA's four blocks.

rate falls with hierarchy depth rather than design size. Module routing binds a property into the submodule that declares its signals, where 33–48% of assertions bind below the top module.

5.5 Discussion

Upon in-depth analysis of the generated assertions and the bugs that go undetected. We found two major reasons behind this.First, the specification is incomplete. The AssertLLM2 specifications are generated documents that stop above cycle level. For about 8 of the 41 bugs, no document names the signal or states the relation. The UART's bit-rate event, for example, fires when a counter reaches 15, but neither the specification nor the manual gives that threshold. No generator that reads the specification alone can catch such a bug. This is further supported by the results for the more complex designs. Since, SpecGuard reached the same boundary, the number of *effective* and *shallow* assertions remains almost similar. Second, SpecGuard rejects some claims on purpose. A claim may name one bit of a vector, such as `state[5]`, only when the specification names that bit, so that no index is guessed. A bug that changes which bit is read therefore escapes in 8 of 9 cases. The only noticable difference between the assertion generated for AssertLLM2 benchmark and complex designs is that, in case of the complex designs, SpecGuard requires more repairs to generate compilable assertions. This is not a severe issue since, the language model captures the relation correctly, and the rest of the structure can be fixed deterministically. This process is elaborated in Appendix D.7.

6 Conclusion

We present SpecGuard, an SVA generation framework that governs the information a model receives rather than the model itself. SpecGuard withholds implementation behavior, since exposing it biases generation toward the implementation rather than the specification. Instead, SpecGuard supplies structure as a parse tree, derives one narrow question per reachable case, and repairs the model's output deterministically. On AssertLLM2, SpecGuard's assertions detect up to 66% of bugs per design class using a 14.8B model at int4, versus 5.1–11.7% for frontier models on the same mutants. SpecGuard also detects bugs in deeply hierarchical designs invisible at the top-level port, such as NVDLA and Google Coral, by binding each assertion into the module declaring its signals, reaching up to 51% detection there. These results show assertion quality follows from the information given, not model scale, making SpecGuard a practical route to verification without a frontier model.

AI USE STATEMENT

A large-language-model coding assistant was used during this project for three tasks. First, drafting and iterating on RTL, testbench, and Python tooling code. The authors reviewed and executed that code, and verified it against an independent fp64 golden model, before any result in this paper was reported. Second, drafting figure-generation scripts and diagrams, which the authors checked for numerical and visual accuracy against the underlying data. Third, drafting and editing portions of this manuscript, which the authors reviewed, fact-checked against the measurements, and revised. Generative AI tools were not used to produce the reported experimental results. RTL simulation, synthesis, and benchmark numbers were produced by the stated toolchain. Citations were not generated or selected without verification. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This is a study on Language model's capability of formulating oracle for hardware verification and developing an effective method for systemverilog assertion generation. It does not involve human subjects, personal data, or a dataset release. We utilize the publicly available AssertLLM2 Wu et al. (2026a), Googel Coral Google Coral Team (2026), and NVDLA codes for evaluation NVIDIA Corporation (2026).

REPRODUCIBILITY STATEMENT

We are uploading a zip file with a readme file inside detailing how to run SpecGuard.

REFERENCES

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)*, pp. 85–92. IEEE, 2025.

Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 1(FSE):1889–1912, 2024.

Google Coral Team. Coral npu: Energy-efficient ml accelerator. `https://github.com/google-coral/coralnpu`, 2026.

Adarsh Gupta, Bhabesh Mali, and Chandan Karfa. Sangam: Systemverilog assertion generation via monte carlo tree self-refine. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 180–187. IEEE, 2025.

Chenghua Huang, Zhizhen Fan, Lu Wang, Fangkai Yang, Pu Zhao, Zeqi Lin, Qingwei Lin, Dongmei Zhang, Saravan Rajmohan, and Qi Zhang. Self-evolved reward learning for llms. In *International Conference on Learning Representations*, volume 2025, pp. 15026–15048, 2025.

Kai Huang, Peng Zhu, Rongjie Yan, and Xiaolang Yan. Functional testbench qualification by mutation analysis. *VLSI Design*, 2015(1):256474, 2015.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 13358–13376, 2023.

Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):1–21, 2012.

Christoph Kern and Mark R Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. Do llms generate test oracles that capture the actual or the expected program behaviour? *arXiv preprint arXiv:2410.21136*, 2024.

Ramesh Krishnamurthy, Danial Chitnis, and Themis Prodromakis. Closing the loop on llm-generated rtl assertions with quality-aware formal verification. *arXiv preprint arXiv:2606.21451*, 2026.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

Hongqin Lyu, Yonghao Wang, Yunlin Du, Mingyu Shi, Zhiteng Chao, Wenxing Li, Tiancheng Wang, and Huawei Li. Assertgen: Enhancement of llm-aided assertion generation through cross-layer signal bridging. In *2025 IEEE 34th Asian Test Symposium (ATS)*, pp. 25–30. IEEE, 2025.

Hongqin Lyu, Yonghao Wang, Jiaxin Zhou, Zhiteng Chao, Tiancheng Wang, and Huawei Li. Assertminer: Module-level spec generation and assertion mining using static analysis guided llms. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 432–438. IEEE, 2026.

Bhabesh Mali, Karthik Maddala, Vatsal Gupta, Sweeya Reddy, Chandan Karfa, and Ramesh Karri. Chiraag: Chatgpt informed rapid and automated assertion generation. In *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 680–683. IEEE, 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)*, pp. 1–5. IEEE, 2026.

Mughal and Bilal. LLM-based test oracles: Source-of-authority taxonomy, a systematic literature review. *arXiv preprint arXiv:2607.05031*, 2026.

Maohua Nie, Jiang Zhu, Jingqun Zhang, Zhichen Zeng, Jiayi Wang, Sibo Zhang, Jialin Wang, and C-J Richard Shi. Hiersva: A data synthesis pipeline, dataset, and benchmark for llm-driven hierarchical hardware formal verification. *arXiv preprint arXiv:2606.13706*, 2026.

NVIDIA Corporation. NVDLA open source hardware, 2026. URL `https://github.com/nvdla/hw`. Accessed: March 18, 2026.

A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

Michael Popoloski. slang: A multi-threaded SystemVerilog compiler frontend. `https://github.com/MikePopoloski/slang`, 2024.

Vaishnavi Pulavarthi, Deeksha Nandal, Soham Dan, and Debjit Pal. Assertionbench: A benchmark to evaluate large-language models for assertion generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 8058–8065, 2025.

Saeid Rajabi, Chengmo Yang, and Satwik Patnaik. Stellar: Structure-guided llm assertion retrieval and generation for formal verification. *arXiv preprint arXiv:2601.19903*, 2025.

David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 90–99. IEEE, 2011.

Wilson Snyder. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, 2004.

Hanzhang Tang. Compact constraint encoding for llm code generation: An empirical study of token economics and constraint compliance. *arXiv preprint arXiv:2604.07192*, 2026.

Shams Tarek, Dipayan Saha, Khan Thamid Hasan, Sujan Kumar Saha, Mark Tehranipoor, and Farimah Farahmandi. Assertain: Automated security assertion generation using large language models. *arXiv preprint arXiv:2604.01583*, 2026.

Enyuan Tian, Yiwei Ci, Qiusong Yang, Yufeng Li, and Zhichao Lyu. Assertcoder: Llm-based assertion generation via multimodal specification extraction. *arXiv preprint arXiv:2507.10338*, 2025.

Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 626–629. IEEE, 2010.

Yonghao Wang, Jiaxin Zhou, Hongqin Lyu, Zhiteng Chao, Tiancheng Wang, and Huawei Li. Deepassert: An llm-aided verification framework with fine-grained assertion generation for modules with extracted module specifications. *arXiv preprint arXiv:2509.14668*, 2025.

Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

Fenghua Wu, Evan Pan, Rahul Kande, Michael Quinn, Aakash Tyagi, David Kebo Houngninou, Jeyavijayan Rajendran, and Jiang Hu. Spec2assertion: Automatic pre-rtl assertion generation using large language models with progressive regularization. *arXiv preprint arXiv:2505.07995*, 2025.

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*, 2026a.

Yutong Wu, Chenrui Cao, Pengwei Jin, Di Huang, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, and Xing Hu. Qimeng-codev-sva: Training specialized llms for hardware assertion generation via rtl-grounded bidirectional data synthesis. *arXiv preprint arXiv:2603.14239*, 2026b.

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*, pp. 614–621, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

YosysHQ. Symbiyosys: A front-end driver for yosys-based formal verification. `https://github.com/YosysHQ/sby`, 2024.

Qingyun Zou, Yingze Li, Tianen Liu, Bingsheng He, and Weng-Fai Wong. Reward-weighted on-policy distillation with an open property-equivalence verifier for nl-to-sva generation. *arXiv preprint arXiv:2605.13501*, 2026.

A Implementation Details

Table 4: Experimental setup: identical inference stack and tokenizer for every method, fixed run matrix, and the implementation details for each baseline reproduction. (a) describe the model configuration. (c) lists how each baseline was reproduced; in every case the published flow is preserved and only the LLM backend is replaced with the same Qwen3-14B endpoint used by SpecGuard, isolating the methodological delta from the model delta.

(a) Inference configuration

Parameter	Value
Model	Qwen3-14B (GGUF Q4_K_M)
Serving backend	Ollama `qwen3:14b-32k`
API endpoint	OpenAI-compatible `/v1/chat/`
Tokenizer	Qwen3 (shared, all methods)
Context window	32,768 tokens
Temperature	0.1
Max new tokens	8,192
Top-p	1.0
Reasoning scratch	`<think>` disabled
Lint gate	pyslang v10 (Popoloski, 2024)

(b) Baseline reproductions

Method	Reproduction details
AssertLLM (Yan et al., 2025)	Three-stage flow preserved verbatim: (i) spec analyser, (ii) signal mapper, (iii) SVA generator with one prompt per signal in the design's signal map. No lint-feedback loop. Originally GPT-4-mini-Turbo; backend swapped to Qwen3-14B.
AssertAIN (Tarek et al., 2026)	NL-Analyzer pre-extracts CWE-tagged threat properties; LLM-as-judge filters and refines candidate assertions per CWE class. Signal validator preserved. Originally GPT-5; backend swapped to Qwen3-14B.
ChIRAAG (Mali et al., 2024)	Single-prompt strategy: full formatted spec concatenated into one LLM call. Iterative simulator-driven refinement disabled (no RTL simulator in our setup). Originally GPT-4; backend swapped to Qwen3-14B.
SpecGuard	Full pipeline: AST extraction with cluster-and-compact (`ast_use_clustering: true`, batch size 40), RTL facts card, RAG over spec docs, lint-feedback loop (≤ 10 refinement steps), optional security pass with CWE-tagged prompts.

B Case study: why generated suites miss bugs

This appendix reports in full the case study of Section 3, which measures the four failures of Section 1 on saved runs of SpecGuard with a single model, a local Qwen3-14B under int4 quantization. Table 5 lists the runs. An assertion *holds* when it passes on correct RTL, and it *detects* a bug when it holds and fails on the RTL carrying that bug. **Soundness** is the share of a suite's assertions that hold.

B.1 Model choice and scale

Table 6 reproduces the six-model comparison of AssertLLM2, whose models generate assertions from the specification alone (Wu et al., 2026a). Averaged over three runs, the bug kill ratio ranges from 5.08% for LLaMA-4-Maverick to 11.69% for Claude-Sonnet-4.5, a 2.3-fold spread. Merging each model's three runs raises the range to 8.04–18.51%. Every model compiles at least 77.93% of its assertions, whereas none detects more than 11.69% of the bugs on average.

Table 5: Runs examined in the case study, all generated with a local Qwen3-14B under int4 quantization.

Run	Configuration	Bugs	Rep.	Scoring
FifoX, per-pair suite	property and selector lanes, one question per lane and pointer pair	48	3	simulation
FifoX, per-lane suites	selector lane, one question per lane, with and without a separate gate question	12	$3 + 3$	simulation
AssertLLM2 pilot	7 designs, signal card without drive kinds	20	1	simulation
AssertLLM2 repetition 11	11 designs, drive kinds, retiming and module routing	20	1	simulation and BMC
AssertLLM2 repetition 21	7 designs, as repetition 11 with later generation fixes	20	1	simulation and BMC
Capture probe	FifoX, two hand-written properties per bug	2	–	simulation

Table 6: Six models on AssertLLM2, generating from the specification alone (Wu et al., 2026a).

Model (developer)	Syntax (%)	Proven / total	Kill, average (%)	Kill, union (%)
Gemini-2.5-Pro (Google)	88.29	13/20	9.76	18.51
Claude-Sonnet-4.5 (Anthropic)	93.04	21/38	11.69	16.31
GPT-5.2 (OpenAI)	95.00	22/30	10.19	14.87
DeepSeek-V3.2 (DeepSeek)	88.94	13/22	9.20	15.32
Qwen3-Coder-Plus (Alibaba)	77.93	8/18	6.82	12.18
LLaMA-4-Maverick (Meta)	89.23	5/13	5.08	8.04

Scale is examined separately by CodeV-SVA, which fine-tunes Qwen3-14B to generate SVA from natural language (Wu et al., 2026b). As Table 7 shows, the fine-tuned model reaches 75.8% functional correctness at pass@1 on NL2SVA-Human, where pass@k is the share of tasks for which at least one of k sampled generations is correct, against 71.8% for GPT-5 and 74.6% for DeepSeek-R1. The advantage holds at pass@1 alone: at pass@16, GPT-5 and DeepSeek-R1 reach 90.2% and 90.3%, above its 89.4%. Neither comparison holds the information given to the model fixed, which the FifoX runs of Section B.2 do.

Table 7: Functional correctness (%) on NL2SVA, from CodeV-SVA (Wu et al., 2026b).

Model	Human, pass@1	Human, pass@16	Machine, pass@1	Machine, pass@16
DeepSeek-R1-671B	74.6	90.3	81.0	93.3
GPT-5	71.8	90.2	81.8	93.2
Qwen3-14B	61.6	86.1	75.3	92.7
CodeV-SVA-14B	75.8	89.4	84.0	94.9

B.2 Same model, different questions

Table 2 of Section 3.1 holds the model, the specification, the 48 stratified FifoX bugs and the scoring fixed, and varies the questions the model is asked. Two of its rows are the lanes of the full suite's first repetition, scored separately, and the per-lane row is a single repetition. It is the best of three per-lane repetitions on 12 bugs, which detect 9, 5 and 7, and the only one scored on the 48. Every FifoX run reads the specification as the authors revised it after reading the RTL's lane-selection logic, and receives the array, the pointer family and the lane count as configuration.

B.3 Implementation capture

A model shown the RTL can copy a relation from the implementation where the specification states it. Table 8 isolates the consequence on FifoX's write address, the entry that pointer `inxpos_0` selects. Bugs 120 and 135 each change one constant of the decoder that maps a pointer value to an entry. For

each bug, two properties differ only in that constant, taken from the specification or from the buggy decoder line.

Table 8: Two properties per FifoX bug, differing only in the source of one decoder constant.

Bug	Constant from	Constant	Correct RTL	Buggy RTL	Verdict
120	specification	`4'h1`	holds	fails	detects
120	buggy RTL	`4'h0`	fails	holds	certifies the bug
135	specification	`4'h2`	holds	fails	detects
135	buggy RTL	`4'h3`	fails	holds	certifies the bug

The specification's constant holds on correct RTL and fails on the buggy design, detecting the bug. The decoder's constant fails on correct RTL and holds on the buggy design. A generator that keeps only assertions holding on the RTL under test therefore keeps the second property whenever that RTL carries the bug, and its suite certifies the bug. These properties are written by hand to isolate the mechanism, and no model produced them. No saved run measures how often a model shown the RTL restates such a constant. With the RTL withheld, every repetition of the full suite detects all 28 address-decode bugs among the 48, the class of bug 120. The per-lane questions detect 10 of them, as Table 11 reports, and the property lane of the first repetition detects none.

B.4 Missing structure

A register takes a new value at the clock edge after its inputs change, and a claim stated in the cycle of those inputs reads the register one cycle early. The AssertLLM2 pilot generated its suites from a signal card that listed no drive kinds, the facts that separate registers from wires. Table 9 replays the retiming rule of Appendix D.7 on those suites to find same-cycle claims about registers, without rewriting them, and reports each assertion's saved verdict on correct RTL.

Table 9: Same-cycle claims about registers in the AssertLLM2 pilot suites, by verdict on correct RTL.

		Same-cycle register claims				Other assertions		
Design	Assert.	All	Fail	Hold	No verdict	Fail	Hold	No verdict
versatile_counter	34	14	5	8	1	13	6	1
ethernet_smii_txrx	66	13	9	0	4	29	18	6
spi_core	64	18	2	13	3	7	28	11
uart	39	12	5	7	0	5	19	3
present_cipher	9	0	0	0	0	0	0	9
ima_adpcm_encoder	5	0	0	0	0	5	0	0
pwm	54	9	4	5	0	10	32	3
Total	271	66	25	33	8	69	103	33

Of the 271 pilot assertions, 66 state a same-cycle claim about a register, 24.4%. Among the 58 of these with a verdict, 25 fail on correct RTL, a rate of 43.1%, against 69 of 172 other assertions, or 40.1%. A same-cycle claim about a register thus fails on correct RTL about as often as any other claim. Retiming acts on both measures of one design. Moving versatile_counter's 14 same-cycle register claims one cycle later, with no model call, raised the assertions holding from 14 to 18 of 34 and the bugs detected from 10 to 14 of 20. That rescore covers one generation scored by simulation, and it did not record which bugs it gained or lost.

With drive kinds on the card, timing still produces failures. A blind diagnosis of repetition 11 examined 248 rows: the 237 that failed on correct RTL or received no verdict, and 11 pwm rows that formal verification refuted or could not build. It files 24 of them under timing, such as a wire the card called a register or a response bound shorter than the specification allows. The larger categories are 86 checks the specification does not support, 45 conditions or values attached to the wrong signal or event, and 40 wrong relations.

B.5 WRONG SCOPE

An assertion can name only the signals its module declares, and a bug inside a submodule reaches an assertion bound to the top module only through that submodule's outputs. Table 10 splits repetition 11's surviving bugs by the file each mutation changes. Of its 137 survivors, 66 lie in submodule files, and a Yosys equivalence check proves 7 of those equivalent to correct RTL, which no assertion can detect.

Table 10: Repetition 11 survivors by mutated file, with the assertions bound below the top module.

Design	Survivors	In submodule files (equivalent)	Submodule bugs detected	Assertions bound below the top
versatile_counter	4	0	–	0 of 58
ethernet_smii_txrx	15	0	–	0 of 51
spi_core	9	1	0 of 1	0 of 55
uart	8	8 (1)	10 of 18	15 of 41
xgate	16	14 (1)	3 of 17	0 of 64
present_cipher	12	10 (1)	0 of 10	0 of 55
ima_adpcm_encoder	9	0	–	0 of 61
reed_solomon	17	16 (2)	2 of 18	2 of 50
srdy-drdy-library	11	9 (1)	9 of 18	1 of 46
sd_fifo	20	4	0 of 4	1 of 16
pwm	16	4 (1)	4 of 8	0 of 20
Total	137	66 (7)	28 of 94	19 of 517

Module routing binds an assertion into the module that declares its signals. On uart, which bound 15 of its 41 assertions below the top, 10 of 18 submodule bugs were detected. On present_cipher, which bound none, none of its 10 were. Routing is not the only path, since pwm detected 4 of 8 submodule bugs with every assertion bound to the top. No run switches routing off, and the table accordingly shows an association rather than an effect. Over the pilot's first six designs, the location-based diagnosis placed 17 of 74 located misses in a submodule, and counted by file, 24 of their 102 misses lie in submodule files.

The spi_core bug M_0000 shows the limit. It changes the condition that sets `gb`, a register of the `fifo4` submodule that separates a full buffer from an empty one, from a conjunction to a disjunction. A replay of repetition 11's stimulus reproduces 1,019 of its 1,020 saved verdicts. In that replay, the `gb` registers of the two buffer instances diverge from correct RTL in 15 cycles, whereas none of the top module's five outputs ever does. Only an assertion bound inside `fifo4` can detect this bug, which survives both repetitions. On NVDLA, 114 of the 151 bugs kept for scoring lie outside the top file, and the convolution MAC is the exception, since all 37 of its kept bugs lie in the top file.

B.6 INCOMPLETE ENUMERATION

A specification states a rule once, for a generic case, and a question about the rule receives an answer for one representative case. FifoX has 10 reachable lane and pointer pairs, and Table 11 counts how many each selector suite covers. Asked once per lane, the model answered each lane with one pointer expression, and six repetitions covered 5, 5, 3, 5, 2 and 4 pairs. Asked once per pair, three repetitions covered 9, 10 and 10. The rank decode, the logic computing how many valid lanes lie below each lane, carries 14 of the 48 bugs. The per-lane suite detects 2 of them, against 12, 12 and 11 for the full per-pair suites.

Requesting two obligations in one question loses one of them. When the per-lane question asked for the write condition with the pointer, lanes whose condition kept the handshake gate yielded 100 assertions, all holding on correct RTL. Lanes that dropped the gate yielded 70, none of which held. The split per lane follows from the saved totals, which equal the kept lanes' assertion counts in each of the three repetitions. Of the 100, 20 can never fire, since their guard compares a one-bit valid signal with 2 or 3. Asking for the gate in a separate question raised the assertions holding from 100 of 170 to 103 of 110. The same change bounded each lane's rank by its index, and the two effects are not separated.

Table 11: Lane and pointer pairs covered, and FifoX bugs detected, per selector arm.

Suite	Rep.	Pairs of 10	Assert.	Sound (%)	Detected of 12	Rank decode	Address decode
Per lane, joint condition	1	5	70	40 (57.1)	6 of 12	–	–
	2	5	70	50 (71.4)	6 of 12	–	–
	3	3	30	10 (33.3)	4 of 12	–	–
Per lane, separate gate	1	5	50	43 (86.0)	9 of 12	2 of 14	10 of 28
	2	2	20	20 (100.0)	5 of 12	–	–
	3	4	40	40 (100.0)	7 of 12	–	–
Per pair, full suite	1	9	167	133 (79.6)	8 of 12	12 of 14	28 of 28
	2	10	222	146 (65.8)	8 of 12	12 of 14	28 of 28
	3	10	233	131 (56.2)	9 of 12	11 of 14	28 of 28

B.7 Sound suites that verify little

An assertion that holds on correct RTL can still detect nothing. Table 12 reports, for each design of repetition 11, how many assertions hold and how many of those detect none of the scored bugs. Of 280 assertions that hold, 193 detect no bug, and in repetition 21, 104 of 162 detect none. The reed_solomon suite holds 48 of its 50 assertions and detects 3 of 20 bugs, and the pwm suite holds all 20 and detects 4. The xgate row measures the harness rather than the suite, since the testbench held xgate in reset. Such suites hold on correct RTL, the property that syntax and formal pass rates reward, while verifying little of the intended behavior.

Table 12: Assertions holding on correct RTL, and the share of those detecting no bug.

Design	Assert.	Hold (%)	Hold, detect none (%)	Bugs detected
versatile_counter	58	26 (44.8)	9 (34.6)	16 of 20
ethernet_smii_txrx	51	17 (33.3)	13 (76.5)	5 of 20
spi_core	55	29 (52.7)	19 (65.5)	11 of 20
uart	41	20 (48.8)	9 (45.0)	12 of 20
xgate	64	49 (76.6)	48 (98.0)	3 of 19
present_cipher	55	25 (45.5)	18 (72.0)	8 of 20
ima_adpcm_encoder	61	25 (41.0)	12 (48.0)	11 of 20
reed_solomon	50	48 (96.0)	29 (60.4)	3 of 20
srdy-drdy-library	46	18 (39.1)	14 (77.8)	9 of 20
sd_fifo	16	3 (18.8)	3 (100.0)	0 of 20
pwm	20	20 (100.0)	19 (95.0)	4 of 20
Total, repetition 11	517	280 (54.2)	193 (68.9)	82 of 219
Total, repetition 21	272	162 (59.6)	104 (64.2)	56 of 139

B.8 Summary

With the model fixed, the question decided FifoX detection, which rose from 5 to 41 of 48 bugs as questions narrowed from properties to lane and pointer pairs. Structure changed detection on one design, where retiming raised it from 10 to 14 of 20, while same-cycle register claims failed on correct RTL no more often than other claims. Scope bounded what top-level assertions reach, with 66 of 137 repetition 11 survivors in submodule files. Across models, frontier detection on AssertLLM2 varied 2.3-fold, and a holding assertion detected no bug in 193 of 280 cases.

C Method details

C.1 Proofs

Proof of Lemma 1. $\sigma(D_u) = \sigma$ for every admissible D_u, and ϕ reads D_u only through σ. □

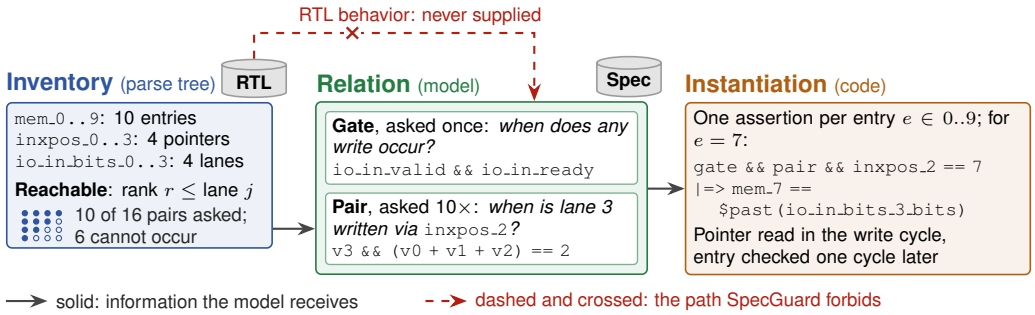

Figure 5: Division of labor on FifoX for the selector lane.

Proof of Proposition 1. By Equation 1, $\det(a, m)$ requires $D_m \not\models a$. Equivalent assertions hold on the same designs, so $D_m \models a'$ as well. □

Proof of Proposition 2. $a \in F_u(A)$ means $D_m \models a$, which Equation 1 excludes. □

Proof of Lemma 2. A register takes its value at the clock edge after its operands are sampled, and keeps it when a gated write is not taken; a wire follows its operands combinationally. Each fact is a field of σ. □

C.2 The selector lane on FifoX

Figure 5 shows how the selector lane divides the work on FifoX. Questions are paraphrased, answers are quoted from a logged run, and vk abbreviates `io_in_bits_`k`_valid`. The array `mem`, the pointer family `inxpos`, and the four lanes are given as configuration, since the inventory does not state which family indexes which. A **pair** question asks when one lane is written through the pointer of one rank, and a separate **gate** question asks once when any write occurs, which keeps the two obligations apart. The rank of lane j cannot exceed j, so L lanes yield $L(L+1)/2$ reachable pairs, 10 rather than $L^2 = 16$ for FifoX. Expansion turns each answer into one assertion per entry, as the right panel of the figure shows. A vector whose bits the design tests individually, such as a one-hot state vector, yields one **phase** per bit and one question per phase, and a protocol question asks once what the environment guarantees about the inputs, which formal verification assumes.

C.3 Rules of the interface card

Two rules follow from what the card exposed. The model uses every listed name as a signal, and a loop variable such as `i` yields `inxpos[i]`, which does not compile outside its loop, so the card lists module-level signals only. A specification moreover documents a whole hierarchy in one file, whereas a formalization prompt carries one module's signals. On FifoX, this led to properties about the Slice, the submodule at the right of Figure 1, being written with the nearest FifoX names, which motivates the scope filter of Section 4.2. Where the module a property concerns can be identified, the property is formalized against that module's card and bound into it.

D Detailed results

D.1 Per-design results

Table 13 pools the per-family results of Section 5.3 over AssertLLM2. Its frontier-model figures are those AssertLLM2 reports from formal property verification.

Table 14 reports representative designs pooled figure. Detection ranges from 45.0% on srdy-drdy-library to 78.3% on versatile_counter, and soundness from 41.2% to 98.3%.

Table 13: Pooled detection on AssertLLM2.

Method	Bugs	Assert.	Sound (%)	Detection (%)
SpecGuard	1660	383.7	68.6	57.9
Frontier models (Wu et al., 2026a)	1660	–	–	5.1–11.7

Table 14: AssertLLM2 per design for SpecGuard, each count the mean over three repetitions. Soundness is the share of assertions not broken.

Design	Family	Assert.	Sound (%)	Killed	Detection (%)
spi_core	Communication	32.7	61.2	13.0	65.0
uart	Communication	32.7	42.9	11.0	55.0
versatile_counter	Compute	37.0	88.3	15.7	78.3
ima_adpcm_encoder	Compute	60.7	41.2	10.7	53.3
xgate	Control	43.7	71.8	12.3	64.9
s1_core	Control	42.0	75.4	11.0	57.9
sd_fifo	Memory	24.0	61.1	12.3	61.7
srdy-drdy-library	Memory	41.0	65.0	9.0	45.0
pwm	Peripheral	20.0	98.3	10.3	51.7
reed_solomon_decoder_204188	Peripheral	50.0	95.3	9.3	46.7
Pooled		383.7	68.6	114.7	57.9

Table 15 reports the numbers behind Figure 4, with the instance hierarchy below each top module. Detection stays between 45.9% and 52.1% across the NVDLA blocks, which range from 8 to 435 modules.

Table 15: Detection on the Coral NPU and NVDLA. Modules and depth describe the instance hierarchy below each top module.

Design	Mod.	Depth	Bugs	Assert.	Sound (%)	Detection (%)
Coral NPU						
FifoX	2	1	48	207	65.9	94.4
NVDLA						
Convolution accumulator	17	3	48	45	74.8	52.1
Convolution MAC	8	2	37	40	78.3	45.9
Planar data processor	148	6	28	60	74.4	50.0
Single-point data processor	435	7	38	70	77.1	49.1

D.2 Bugs the suites never detect

The per-design figures say how many bugs a suite catches. They do not say which bugs are out of reach. Table 16 names every injected bug by the edit it makes to the design, and pairs it with the claim a suite would need in order to catch it. The classification is recovered from the benchmark's own mutant files by `analysis/undetected_bugs.py`, which compares each mutant against the mutant closest to the others rather than against the original, since every mutant carries the same macro expansion. A bug counts as detected when any of the three repetitions caught it, hence the table reports what the configuration cannot reach.

Detection divides sharply by what the claim has to state. Every bug that changes an operator in a branch condition is caught, 9 of 9, and every exchanged port connection is caught, 3 of 3. Both require a claim about when a signal updates, which the register case questions of Section 4.3 produce for each register they ask about. A constant in an assigned value is caught in 18 of 23 cases, a claim about the value a register takes.

The two kinds that survive need a claim the pipeline rarely writes. A changed bit index escapes 8 times in 9, and a removed or reordered branch escapes 5 times in 6. An index claim names one bit of a vector, which index grounding refuses unless the specification names that bit for that vector. A

Table 16: Injected bugs by edit, and the claim each one requires.

Edit the bug makes	Claim needed to catch it	Inj.	Det.	Miss
Constant in an assigned value	the value the signal takes	23	18	5
Constant in a condition	the threshold the condition tests	1	1	0
Operator in a condition	when the signal updates	9	9	0
Operator in an assigned value	the value the signal takes	9	7	2
Bit index changed	which bit of the vector is read	9	1	8
Branch removed or reordered	the case the branch handled	6	1	5
Port connection exchanged	the submodule's ports at the boundary	3	3	0
Several sites changed	every site the edit touches	12	7	5
Other edit	unclassified	7	5	2

branch claim needs the case list to be complete, where a suite that states one branch says nothing about the others. Bugs that change several sites at once escape 5 times in 12.

The headline consequence is that soundness and detection answer different questions. Of the 79 injected bugs, 27 are never detected by any repetition. The shape of that residue follows from which claims the questions can express, not from how many assertions each suite ships.

D.3 Ablation study

Two ablations support the claims of Section 5: removing one piece of the information the model receives, and removing the deterministic repair of Appendix D.7. Both hold the model, the bug set, and the number of repetitions fixed, and change nothing else between arms.

Information ablation. Table 17 removes one piece of information at a time from the FifoX configuration and leave everything else in place. The table's Δ column gives detection relative to the full SpecGuard configuration, prompt tokens are counted per design, and its last two rows vary the selector lane alone. The first row runs in the opposite direction and adds the module's source to the prompt, which is what a generator conditioned on the RTL does. On FifoX the selector lane alone detects 85.4% and the property lane alone 10.4%, and asking one question per lane rather than per pair drops the selector to 35.4%. The RTL-conditioned arm detects 20.8%. No arm raises detection by adding prompt tokens: the arm that adds the most tokens RTL source, detects the least.

Table 17: Information ablation on FifoX.

Configuration	Prompt tok.	Sound (%)	Detection (%)	Δ
SpecGuard	50k	65.9	94.4	–
RTL source added to the prompt	1.05M	55.0	20.8	-73.6
Drive kinds removed from the card	49k	58.0	89.6	-4.8
Module scope removed, no routing	50k	64.0	92.3	-2.1
Property scope filter disabled	50k	60.3	94.4	0.0
Array notation disabled	50k	64.0	94.4	0.0
One question per lane, not per pair	45k	86.0	35.4	-59.0
Gate folded into the case question	48k	53.7	27.0	-67.4

Deterministic repair. Figure 6 reports the same kind of measurement for the rules of Appendix D.7: each assertion is re-scored with one rule disabled and everything else unchanged, and no model call differs between the two arms. Across 12 AssertLLM2 generations, 494 formalized assertions entered repair, and Table 21 in Appendix D.8 lists every rule with the structure it consults. Retiming a register claim to `$past`, that is, moving it one cycle later, has the largest measured effect. It raises kills from 10 to 14 of 20 on versatile_counter with the assertions otherwise unchanged. Two further rules, correcting a reset guard's presence and its polarity, each make one more bug newly detected. The remaining six rules fix the form of an assertion without changing whether it detects anything on this measurement. Figure 6 keeps each as an empty bar rather than omitting it, since a rule can matter for

compilation or soundness while moving no kill; Appendix D.8 reports both. Across the whole repair set, 6 bugs become newly detected and none become no longer detected.

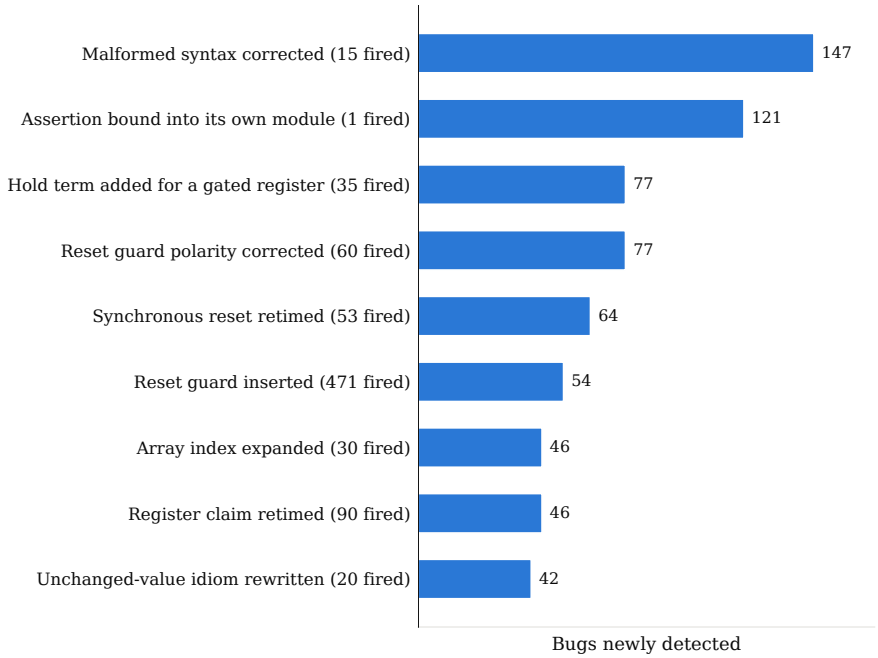

Figure 6: Bugs newly detected per deterministic repair rule, each measured by disabling one rule and re-scoring the same assertions. Labels give how often each rule fired.

D.4 Bug injection and observability screening

The AssertLLM2 designs ship the benchmark's own 20 single-bug mutants per design, which this work does not author (Wu et al., 2026a). The Coral and NVDLA designs carry no mutant set, and bugs are therefore injected here. This subsection states how, since a fault set that is not drawn from a stated population measures whatever that set happens to contain.

Operators. A mutant applies one single-point change to one line of RTL, hence a detected mutant names one defect rather than an interaction of several. Table 18 lists the seven operators and what each rewrites. They cover the operator, constant and statement classes that mutation testing reports as sufficient, in the sense that a reduced set of operators yields test sets nearly as strong as the full set does (Offutt et al., 1996), adapted to RTL in the manner functional qualification applies mutation analysis to a hardware description (Huang et al., 2015; DeMillo et al., 1978). Operators are applied exhaustively at every site each matches, never at sites chosen by hand, hence no bug is placed where an assertion was known to be weak.

Screening. A mutant whose outputs never diverge from correct RTL under the testbench cannot be killed by any assertion, and scoring it measures the stimulus rather than the suite (Huang et al., 2015). Every candidate is therefore simulated against correct RTL and kept only when some output diverges. Table 19 reports the counts. FifoX keeps 149 of 200 mutants on its ports and 161 when internal registers are watched as well. On NVDLA, 71 of 800 accumulator, 37 of 532 MAC, 28 of 800 planar and 38 of 800 single-point candidates are observable, and every kept bug lies below the top file.

Stratification. Operators do not match equally often, and the observable FifoX population is 113 operator replacements, 28 constant replacements, 6 assignment deletions, 1 sensitivity-list change and 1 condition negation. The 48 bugs are drawn from that population in its own proportions, by largest-remainder allocation under a recorded seed, giving 37, 9 and 2 of the first three operators. Signal swap and bit-slice mutation yield no observable FifoX mutant and consequently appear in no fault set for it.

Stratification is what keeps the measurement honest, and the first FifoX fault set shows why. It was built by screening a pool and keeping the first 12 results, which truncates rather than samples. Constant replacement was 50% of those 12 against 18.8% of the observable population, a 2.7-fold enrichment, and the write-address bugs the selector lane targets are constant replacements. That set therefore over-represented the class the strongest mechanism addresses. Re-scoring the same assertions on a proportional sample lowered detection from 58.3% to 35.4%, and every FifoX figure reported in this paper comes from the stratified set.

Table 18: Mutation operators. Each applies one change at one site, and every site an operator matches yields a candidate mutant.

Operator	Rewrites	Example
Operator replacement	A logical, arithmetic or comparison operator by another of its class	<= becomes <
Constant replacement	A literal's last digit, and a one-bit literal by its complement	`4'hA` becomes `4'hb`
Signal swap	Exchanges two signals of the same width that both appear in the line	`a & b` becomes `b & a`
Bit-slice mutation	Shifts both bounds of a part select by one	`x[7:4]` becomes `x[6:3]`
Condition negation	Adds a negation to a branch condition, or removes an existing one	`if (v)` becomes `if (!v)`
Assignment deletion	Comments out one assignment, leaving its target to hold	`q <= d;` is dropped
Sensitivity-list change	The edge of an event control	`posedge` becomes `negedge`

Table 19: Injected bugs per design. A bug is kept only when the design's outputs diverge from correct RTL under its testbench, since an unobservable bug measures the testbench rather than the suite.

Design	Modules	Injected	Observable	Obs. (%)	Below top (%)
Coral NPU FifoX	2	200	149	74.5	15.0
Convolution accumulator	17	800	71	8.9	100.0
Convolution MAC	8	532	37	7.0	100.0
Planar data processor	148	800	28	3.5	100.0
Single-point data processor	435	800	38	4.8	100.0

D.5 Assertion binding across the hierarchy

On FifoX none of the 622 assertions shipped across three repetitions binds below the top module, since those runs predate module routing. On AssertLLM2, routing bound 1 of uart's 39 assertions into its `has` submodule. On the NVDLA blocks, 35–45% of shipped assertions bind below the top, at depths of 2 to 4. An assertion bound into a submodule is the only kind that can constrain logic the top module's ports do not expose.

D.6 Questions of the selector lane

Table 20 lists the questions the selector lane asks, the unit each is asked once for, and the claim code builds from the answer.

D.7 Deterministic repair

Repair implements R_σ (Algorithm 1, lines 24–26). By Lemma 2, the signals an assertion may name, its temporal form and its binding follow from the inventory rather than from the design's intent, so the code stage recovers all three without a model call (h), as Table 21 in Appendix D.8 lists.

The drive kind determines timing: a claim that `mem_4` equals `d0` in the write cycle is moved to the next cycle and compares `mem_4` with `$past(d0)`, the value `d0` held one cycle earlier. The reset polarity determines the guard, a `disable iff` clause that suspends an assertion during reset:

Table 20: Questions the selector lane asks.

Question	Asked once per	Asks for	Code builds
Gate	design, from configuration	condition under which any write occurs	conjunct of every pair assertion
Pair	reachable lane and pointer pair	when the lane is written through the pointer	one assertion per entry
Update	register that may hold	condition under which it takes a new value	hold-tolerant claim with the condition
Next value	register the specification names	condition and new value	next-cycle claim
Flag	one-bit conditionally written register	set and clear conditions	set and clear claims
Phase	phase of a one-hot state vector, per combinational output	value in that phase	one claim per phase
Bounded response	one-bit signal the specification names	response and its bound	response within a number of cycles
Protocol	design	what the environment guarantees about the inputs	assumptions for formal verification

for an active-low reset `rst_n`, the assertion is disabled while `rst_n` is low. The module scope determines the binding: an assertion naming `ipos` is bound into the Slice rather than into FifoX. No rule consults a simulation result, since selecting on the RTL under test would be self-filtering.

Repair fixes the form of an assertion, not the relation it states, as the rule-level counts of Appendix D.8 report. Implementation capture is therefore beyond its reach (Proposition 1). Where the relation is right, repair recovers detection, as the retiming measurement of Section 3.2 shows. Stating a correct relation over the declared signals consequently remains the model's only task.

D.8 Repair rule statistics

Table 21 lists every deterministic repair with the structure it consults, followed by the cases refused rather than rewritten. Table 22 reports how often each rule fires and compiles, and Appendix D.3 reports the detection effect measured by disabling each in turn. On one FifoX repetition, all 17 uncompilable assertions compiled after repair, 16 of them through the unchanged-value rewrite. Of 37 assertions that compiled yet failed on correct RTL, only 3 came to hold, 2 through retiming and 1 through a reset guard.

D.9 Verdict distribution and echo rate

Every assertion receives one verdict per bug, and the mix says what a suite is doing when it is not detecting. Over the seven-design AssertLLM2 pilot, 5,420 verdicts split into 49.2% vacuous, 34.0% broken, 15.1% unknown, 1.0% detects and 0.7% bug-validating. On FifoX the selector lane's echo rate is 0.0% and the property lane's 11.5%. The property lane yields 12% bug-validating verdicts and the selector lane yields none, since only the property lane can restate an assignment. A bug-validating assertion is the failure mode that equivalence invariance predicts, and its share is therefore reported separately from the vacuous one.

D.10 Token consumption

Measured on versatile_counter with every mechanism on, one generation made 92 model calls with 53,141 prompt and 181,860 completion tokens. Formalization took 68 of those calls and 33,587 prompt tokens, formulation 20 and 10,246, the update questions 3 and 6,941, and the protocol question 1 and 2,367. The configurations of Table 17 send 45k to 50k prompt tokens per design, and 1.05M with the RTL source added. The paper's claim concerns which information is sent rather than how much, and these figures record the cost at which each configuration was obtained.

Table 21: Deterministic repairs applied without a model call.

Defect	Rewrite	Structure consulted
Unguarded reset window	Insert `disable iff (<in reset>)`	Reset signal
Guard of wrong polarity	`disable iff (rst_n)` becomes `disable iff (!rst_n)`	Reset polarity
Synchronous reset checked a cycle early	`reset |-> X` becomes `reset |=> X`	Reset synchrony
Same-cycle claim about a register	`A |-> (X == E)` becomes `A |=> (X == $past(E))`	Drive kind of `X`
Register that may hold	Append `|| (X == $past(X))`	Whether `X` may hold
Unchanged value as a delay	`A |-> (E == ##1 E)` becomes `A |=> $stable(E)`	None
Index into a flattened array	`mem[p]` becomes one guarded assertion per entry	Signal families and indices
Assertion about a submodule signal	Bind into the declaring module	Module scope
Malformed syntax	Close one missing parenthesis, escape `%` in a message, write a range set as `inside {[0:9]}`	None
Nested implication	Refused: `a |-> (b |-> c)` has two readings	None
Undeclared signal named	Refused: cannot compile	Declared signals
Compound left-hand side	Refused: retiming target is ambiguous	Drive kinds

Table 22: Repair rules in use. Fired counts are over 494 formalized AssertLLM2 assertions. Now detected and no longer detected are measured by re-scoring the same assertions with the rule disabled.

Rule	Fired	Compiled	Now det.	No longer det.
Reset guard inserted	471	450	1	0
Reset guard polarity corrected	60	60	1	0
Synchronous reset retimed	53	50	0	0
Register claim retimed to `$past`	90	90	4	0
Hold term added for a gated register	35	35	0	0
Unchanged-value idiom rewritten	20	20	0	0
Array index expanded	30	30	0	0
Assertion bound into its own module	1	1	0	0
Malformed syntax corrected	15	15	0	0
Refused rather than rewritten	15	–	–	0

D.11 Supplementary documents

When a design provides supplementary documents, such as a PDF manual, they are converted to text with each figure transcribed as a table and each code block removed. Figures are always included, and a section containing an RTL construct is always excluded. The remaining sections are ranked by the distinctive terms they share with the design, with declared signal names weighted more heavily. The highest-ranked sections that fit a word budget are included in document order.

E Sample assertions that detect bugs

Table 23 lists 30 assertions SpecGuard generated on AssertLLM2, each of which detects at least one injected bug. They are drawn from two repetitions of the current pipeline and span every design with a detection. Each holds on the correct design: 16 are proven to hold by bounded model checking, and the other 14 hold in simulation, as each row states. The assertions are verbatim, including the repairs of Appendix D.7, and line breaks are the typesetter's.

Table 23: Sample generated assertions that detect injected AssertLLM2 bugs, with the number of the design's mutants each detects.

#	Design and claim	Assertion
1	**versatile_counter** Mode-selected next value: while rewinding, the next count is the forward branch's. *Detects 6 of 20; holds under BMC.*	`assert property (@(posedge clk) disable iff (rst) (rew == 1) |-> (q_next == q_next_fw)) else $error("q_next depends on rew when rew is 1");`
2	**versatile_counter** Level flag set: reaching the level-2 threshold, without rewinding past it, sets `level2`. *Detects 7 of 20; holds in simulation.*	`assert property (@(posedge clk) disable iff (rst) (((q_next == level2_value)) && !((q == level2_value) && (rew == 1))) |=> (level2));`
3	**versatile_counter** Register update with hold: when the next count is 0 or 15, `q` takes its Gray encoding or holds. *Detects 3 of 20; holds under BMC.*	`assert property (@(posedge clk) disable iff (rst) ((q_next == 0 || q_next == 15)) |=> ((q == $past(((q_next >> 1) ^ q_next))) || (q == $past(q))));`
4	**versatile_counter** Priority case: `clear` wins over `set`, forcing the next count to zero. *Detects 3 of 20; holds under BMC.*	`assert property (@(posedge clk) disable iff (rst) (clear && set) |-> (q_next == 0)) else $error("When clear and set are both 1, q_next must be 0");`
5	**versatile_counter** Full case of a flag update: set on reaching, cleared on rewinding past, otherwise held. *Detects 2 of 20; holds under BMC.*	`assert property (@(posedge clk) disable iff (rst) ((cke) && ((q_next == level2_value) || (q == level2_value && rew))) |=> (level2 == $past((q_next == level2_value) ? 1 : ((q == level2_value) && rew) ? 0 : level2)));`
6	**versatile_counter** Reset value: the internal count is zero one cycle after reset. *Detects 1 of 20; holds under BMC.*	`assert property (@(posedge clk) (rst |-> ##1 (qi == 0))) else $error("When rst=1, qi must be 0 on next clock edge");`
7	**spi_core** Idle clock polarity: with the bus idle and the core enabled, `sck_o` rests at `cpol`. *Detects 5 of 20; holds in simulation.*	`assert property (@(posedge clk_i) disable iff (!rst_n) ((cyc_i == 0) && ((spe & (sper[7:6] != 2'b11) & (sper[7:6] != 2'b11)))) |=> (sck_o == $past(cpol)));`
8	**spi_core** Counter reload: a pending transfer reloads the clock divider from the rate bits, or holds it. *Detects 4 of 20; holds in simulation.*	`assert property (@(posedge clk_i) disable iff (!rst_n) ((spe && !wfempty)) |=> ((clkcnt == $past((espr ? (2*(spr + 1)) : (spr + 1)))) || (clkcnt == $past(clkcnt))));`
9	**spi_core** Flag hold: before the transfer count is reached, the interrupt flag is cleared or held. *Detects 3 of 20; holds under BMC.*	`assert property (@(posedge clk_i) disable iff (!rst_n) ((tcnt < (icnt + 1))) |=> ((spif == $past(1'b0)) || (spif == $past(spif))));`
10	**spi_core** Counter reload on completion: when the flag rises, the byte counter reloads or holds. *Detects 3 of 20; holds in simulation.*	`assert property (@(posedge clk_i) disable iff (!rst_n) (spif) |=> ((bcnt == $past(icnt + 1)) || (bcnt == $past(bcnt))));`
11	**spi_core** Write-collision flag: a write into a non-full FIFO clears or holds `wcol`. *Detects 1 of 20; holds under BMC.*	`assert property (@(posedge clk_i) disable iff (!rst_n) ((we_i && !wffull)) |=> ((wcol == $past(0)) || (wcol == $past(wcol))));`

continued on the next page

Table 23, continued

#	Design and claim	Assertion
12	**uart** Stability: without new transmit data, the transmitter's busy flag does not change. *Detects 7 of 20; holds in simulation.*	`assert property (@(posedge clock) disable iff (reset) (new_tx_data == 0) |=> ($stable(tx_busy))) else $error("When new_tx_data is low, tx_busy must remain at its previous value");`
13	**uart** Handshake: a received byte clears the receiver's busy flag on the next cycle. *Detects 5 of 20; holds in simulation.*	`assert property (@(posedge clock) disable iff (reset) (new_rx_data) |=> (rx_busy == $past(0)));`
14	**uart** Data hold: while a received byte is flagged, the output register keeps its value. *Detects 4 of 20; holds under BMC.*	`assert property (@(posedge clock) disable iff (reset) (new_rx_data |-> (rx_data == $past(rx_data))) ) else $error("rx_data not stable during new_rx_data");`
15	**uart** Capture: after the eighth bit, the output register takes the shift buffer. *Detects 4 of 20; holds in simulation.*	`assert property (@(posedge clock) disable iff (reset) ((bit_count == 4'd8) && rx_busy) |=> (rx_data == $past(data_buf)));`
16	**uart** Idle line: an idle transmitter drives the serial line high. *Detects 2 of 20; holds under BMC.*	`assert property (@(posedge clock) disable iff (reset) (tx_busy == 1'b0) |-> (ser_out == 1'b1)) else $error("During idle state, ser_out must be high");`
17	**ima_adpcm_encoder** Bounded response: an accepted sample produces a valid output within six cycles. *Detects 3 of 20; holds in simulation.*	`assert property (@(posedge clock) disable iff (reset) (inValid & inReady) |-> ##[1:6] (outValid));`
18	**ima_adpcm_encoder** Back-pressure: once a sample is accepted outside the idle state, input is no longer ready. *Detects 3 of 20; holds in simulation.*	`assert property (@(posedge clock) disable iff (reset) (((inValid & inReady)) && !((pcmSq == `PCM_IDLE))) |=> (!inReady));`
19	**ima_adpcm_encoder** Output capture: a valid output registers the pre-computed code. *Detects 2 of 20; holds under BMC.*	`assert property (@(posedge clock) disable iff (reset) (outValid) |=> (outPCM == $past(prePCM)));`
20	**ima_adpcm_encoder** Output hold: while no output is valid, code and step index keep their values. *Detects 2 of 20; holds under BMC.*	`assert property (@(posedge clock) disable iff (reset) (outValid == 0) |-> (outPCM == $past(outPCM) && outStepIndex == $past(outStepIndex))) else $error("Output stability violated during intermediate state");`
21	**present_cipher_encryption_core** Round counter: loading data without a key restarts the round count at one, or holds it. *Detects 3 of 20; holds under BMC.*	`assert property (@(posedge clk_i) ((data_load && !key_load)) |=> ((round_counter == $past(5'b00001)) || (round_counter == $past(round_counter))));`
22	**present_cipher_encryption_core** Key schedule: loading a key sets the round key to its upper 64 bits. *Detects 2 of 20; holds under BMC.*	`assert property (@(posedge clk_i) (key_load |-> (round_key == key[79:16])) ) else $error("round_key mismatch during key_load");`
23	**present_cipher_encryption_core** Datapath relation: outside a data load, the substitution input is the state XOR the round key. *Detects 1 of 20; holds under BMC.*	`assert property (@(posedge clk_i) (data_load == 0) |-> (sub_per_input == (state ^ round_key))) else $error("sub_per_input mismatch when data_load is low");`

continued on the next page

Table 23, continued

#	Design and claim	Assertion
24	**ethernet_smii_txrx** Modulo counter: with `speed` low and state bit 10 set, the receive counter advances by one modulo 10. *Detects 1 of 20; holds under BMC.*	`assert property (@(posedge clk) disable iff (rst) ↪ ((speed == 1'b0) && state[10]) |=> (rx_cnt == ↪ $past((rx_cnt + 1) % 10)));`
25	**ethernet_smii_txrx** Pipeline relation: the receive-valid output follows its sampled input, or holds. *Detects 1 of 20; holds under BMC.*	`assert property (@(posedge clk) disable iff (rst) ↪ 1'b1 |=> ((mrxdv == $past(rx_tmp[3])) || ↪ (mrxdv == $past(mrxdv))));`
26	**reed_solomon_decoder_204188** Stage ordering: the error polynomial is ready only after syndromes and locator. *Detects 2 of 20; holds in simulation.*	`assert property (@(posedge clk) disable iff (reset) ↪ (poly_ready |-> (S_ready && L_ready))) else ↪ $error("poly_ready asserted without S_ready ↪ and L_ready");`
27	**reed_solomon_decoder_204188** Completion: decoding done raises the output valid on the next cycle. *Detects 1 of 20; holds in simulation.*	`assert property (@(posedge clk) disable iff (reset) ↪ (DONE) |=> (Valid_out == $past(1)));`
28	**srdy-drdy-library** Protocol field: at the last item identifier, the accepted identifier matches the request. *Detects 6 of 20; holds in simulation.*	`assert property (@(posedge clk) disable iff (reset) ↪ (ip_drdy && (c_itemid == 6'd63)) |-> ↪ (ip_itemid == c_itemid)) else ↪ $error("ip_itemid mismatch when c_itemid is ↪ max and ip_drdy is asserted");`
29	**pwm** Register write: a valid duty-cycle write latches the new duty cycle. *Detects 2 of 20; holds in simulation.*	`assert property (@(posedge i_wb_clk) disable iff ↪ (i_wb_rst) (i_valid_DC) |=> (extDC == ↪ $past(i_DC)));`
30	**xgate** Invariant: the interrupt status mirrors the pending interrupt requests. *Detects 3 of 19; holds in simulation.*	`assert property (@(posedge risc_clk) disable iff ↪ (arst_i) (xgif_status == int_req)) else ↪ $error("xgif_status does not match RISC ↪ interrupt flags");`