CKTFORMALIZER: AUTOFORMALIZATION OF NATURAL LANGUAGE INTO CIRCUIT REPRESENTATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Hardware infrastructure is a critical bottleneck for LLM-driven circuit design, limiting what agents can express, compile, and iteratively refine within an agentic loop. To address this bottleneck, we introduce CKTLEAN, a typed hardware infrastructure embedded in Lean. It supports hardware description, compilation to SystemVerilog, and interactive type-checking and proof feedback through a persistent read-eval-print loop (REPL). On this foundation, we build CKTFORMALIZER, an agent framework for hardware generation, repair, optimization, and source-level equivalence proving. We evaluate structural correctness through compilation, functional correctness through RTL and gate-level simulation, and formal correctness through proofs relative to stated specifications. Across VerilogEval, RTLLM, ResBench, and CVDP, CKTFORMALIZER with CKTLEAN achieves compilation rates of 91.1%–99.4%. Among designs that pass RTL simulation, 95.4%–100.0% jointly complete synthesis and place-and-route and pass design-rule and layout-versus-schematic checks. In a separate evaluation on 30 VerilogEval problems, interactive proof-state feedback raises kernel-accepted equivalence proof completion from 53.3% to 63.3%. Hardware evaluation feedback also guides architecture exploration and iterative power, performance, and area (PPA) optimization, with synthesis-area reductions of up to 58.6% in the optimization loop. These results suggest that typed representations and explicit proof-state feedback help structure the agentic loop: compiler diagnostics guide targeted repairs, while proof-state feedback helps agents identify what remains to be proved and determine the next proof step. Our code is available anonymously.

1 INTRODUCTION

Large language models (LLMs) can translate natural-language hardware specifications into register-transfer-level (RTL) designs written in SystemVerilog (Committee et al., 2005), but syntactically valid RTL may still fail to implement the intended behavior (Thakur et al., 2024a; Liu et al., 2023). Syntax checks establish conformance to the checked syntax rules, and RTL simulation tests finite traces; neither guarantees equivalence to the intended specification or physical realizability. Preserving the functional behavior is important during iterative power, performance, and area (PPA) optimization, where repeated design changes must remain correct through synthesis and physical implementation.

Higher-level HDLs such as Chisel (Bachrach et al., 2012), Bluespec (Nikhil, 2004), and Clash (Baaij et al., 2010) provide abstractions that compile to RTL. However, addressing these correctness challenges requires more than raising the abstraction level. We therefore use a hardware DSL embedded in Lean 4 (Moura & Ullrich, 2021) to combine hardware description with machine-checked reasoning about circuit behavior. SPARKLE (Hashimoto, 2026) provides a typed hardware representation with SystemVerilog extraction, but leaves Mathlib's reusable lemmas and proof tactics (The Mathlib Community) underexplored for circuit equivalence proofs. We focus on how an LLM agent can use these resources with interactive Lean feedback to construct kernel-checked equivalence proofs. Hardware semantics and equivalence statements must still be defined explicitly. This motivates a hardware representation and interactive feedback interface that support both circuit generation and equivalence proving.

Research question. *Can a typed hardware IR embedded in Lean, with interactive type-checking and proof feedback, help agents generate correct hardware and prove source-level equivalence?*

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.

To answer this question, we present CKTFORMALIZER, an agent framework for hardware generation, repair, optimization, and source-level equivalence proving. Its foundation is our core contribution, **CKTLEAN**, which extends SPARKLE's typed IR, elaboration, and backend lowering. Our analysis (§3) shows that SPARKLE's limited support for parameterized state and memory, dynamic bit operations, signed arithmetic, and multi-clock composition, together with compiler defects, restricts the range of supported designs. CKTLEAN addresses these gaps and provides a persistent Lean read-eval-print loop (REPL). CKTFORMALIZER uses its compiler diagnostics and proof states in a ReAct-style loop to repair hardware descriptions and develop equivalence proofs. Table 1 compares its representation and verification capabilities with direct SystemVerilog generation.

Table 1: SystemVerilog vs. CKTLEAN within the supported generation subset.

Capability	SystemVerilog	CKTLEAN
Width consistency	✗	✓
Register-guarded feedback	✗	✓
Total case handling	✗	✓
Primary diagnostics	Tool logs	Type/macro errors
Kernel-checked proofs	✗	✓
Auto SV extraction	N/A	✓

We assess correctness along three dimensions: *structural correctness*, assessed through type and pattern-match exhaustiveness checking; *formal correctness*, established at the source level relative to stated formal specifications and assumptions through proofs accepted by Lean's kernel; and *functional correctness*, evaluated through RTL simulation, gate-level simulation, and RTL equivalence checks (§4.1). Our contributions are as follows:

- Our core contribution is **CKTLEAN**, extending SPARKLE to support more complex hardware designs and providing a persistent Lean REPL for interactive type-checking and proof feedback.
- We introduce CKTFORMALIZER, which uses this infrastructure for type-guided hardware repair, synthesis-guided PPA optimization, and source-level equivalence proof generation.
- Across four benchmarks, all RTL-simulation-passing designs synthesize successfully, and 95.4–100% complete place-and-route. Among RTL-simulation-passing designs, CKTFORMALIZER with CKTLEAN achieves higher gate-level simulation pass rates after synthesis and place-and-route than SystemVerilog generation using in-context learning or Verilog Agent on RTLLM and ResBench.

2 BACKGROUND AND RELATED WORK

Typed Hardware Representation. Higher-level HDLs such as Chisel (Bachrach et al., 2012), Bluespec (Nikhil, 2004), and Cλash (Baaij et al., 2010), along with C/C++ HLS systems (Cong et al., 2011; Canis et al., 2011), provide abstractions that compile to RTL. SPARKLE (Hashimoto, 2026) embeds hardware descriptions in Lean 4 using `Signal` combinators: `Signal dom` α represents a synchronous stream $\mathbb{N} \to \alpha$ in clock domain `dom`. Its `BitVec` N values encode width in the type; for example, `Signal dom (BitVec 8)` produces an 8-bit value each cycle (Figure 1). SPARKLE uses Lean's type and exhaustiveness checks to reject ill-typed width combinations and incomplete pattern matches. However, SPARKLE has limited support for parameterized state and memory, dynamic bit operations, signed arithmetic, and multi-clock designs.

```
-- Example 1: 8-bit counter (sequential circuit)
def counter {dom : DomainConfig} : Signal dom (BitVec 8) :=
  Signal.circuit do
    let count <- Signal.reg 0#8
    count <~ count + 1#8
    return count

-- Example 2: Bit-width mismatch caught at compile time
def bad_add (a : Signal dom (BitVec 8))
            (b : Signal dom (BitVec 16)) : Signal dom (BitVec 8) :=
  a + b  -- ERROR: type mismatch (BitVec 8 vs BitVec 16)
```

Figure 1: **Typed hardware representation in Lean.** An 8-bit counter uses explicit register feedback, while adding signals with incompatible bit widths is rejected by Lean's type checker.

Lean Circuit Proofs and RTL Verification. Generated RTL is validated through simulation, model checking, or equivalence checking to assess whether it satisfies specified behavioral properties or

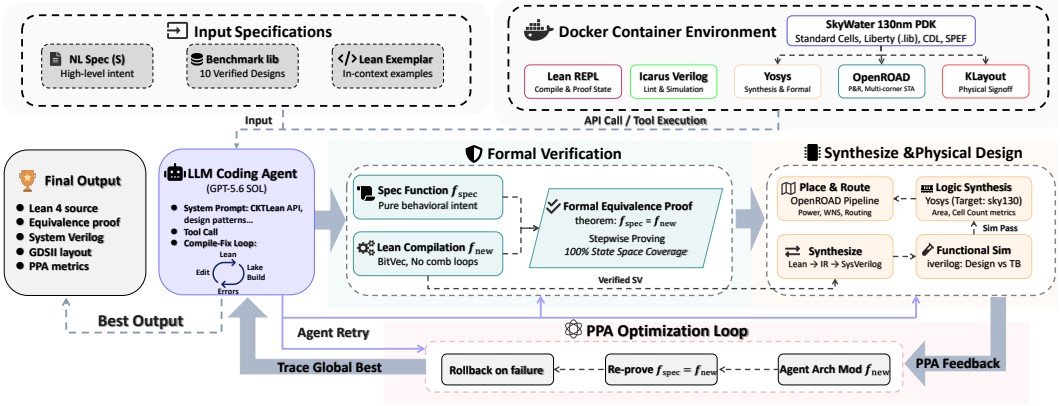


Figure 2: Overview of the CKTFORMALIZER pipeline. Given a natural-language specification and examples, the agent uses compiler diagnostics and Lean REPL feedback to generate hardware and construct source-level equivalence proofs relative to stated specifications. Extracted SystemVerilog is evaluated through simulation, synthesis, and place-and-route using the SkyWater 130 nm PDK. PPA feedback guides design revisions, with proof and evaluation feedback supporting acceptance or rollback.

matches a reference implementation (Committee et al., 2005; Clarke et al., 2011; Biere et al., 2009; Brayton & Mishchenko, 2010). Proof assistants have also been applied to hardware verification: Kami and Silver Oak use Coq to formally verify hardware designs, while ACL2 has been used in industrial hardware verification (Choi et al., 2017; Team, 2023; Hunt Jr et al., 2017). SPARKLE supports typed circuits and SystemVerilog extraction, but lacks a general proof bridge between circuits and specifications and a proof of extraction correctness (Hashimoto, 2026). CKTLEAN provides typed circuit representations that enable CKTFORMALIZER to formalize selected properties from natural-language specifications and prove these properties and equivalence between circuit implementations before and after optimization through iterative agent interaction with the proof environment and proof-state feedback. Completed proofs are checked by Lean's kernel (Appendix E).

RTL Generation and Agentic Workflows. Work on LLM-based hardware design ranges from English-to-Verilog generation (Pearce et al., 2020) and general or fine-tuned code models (Chen et al., 2021; Roziere et al., 2023; Lozhkov et al., 2024; Thakur et al., 2024b) to benchmarks such as VerilogEval (Liu et al., 2023) and RTLLM (Lu et al., 2024) and multi-step systems such as Chip-Chat (Blocklove et al., 2023) and ChipGPT (Chang et al., 2024). These approaches motivate examining which defects are exposed to the agent during generation. SystemVerilog permits implicit width conversions, incomplete combinational assignments, and feedback loops whose consequences may evade finite RTL simulation and emerge during synthesis, timing checks, or gate-level simulation.

Agent Interactive Proving Workflows. Agent feedback loops allow a model to invoke compilers and testing tools, observe their feedback, and iteratively revise its output. Tool-augmented agents such as ReAct (Yao et al., 2023) and SWE-agent (Yang et al., 2024) support iterative editing and test feedback (Jimenez et al., 2024), while GPT-f (Polu & Sutskever, 2020) and LeanDojo (Yang et al., 2023) demonstrate LLM-guided proof-step generation. CKTFORMALIZER uses CKTLEAN and a persistent Lean REPL for type-guided hardware repair and interactive equivalence proving. The agent unfolds circuit definitions and iteratively proves behavioral equivalence using proof-state feedback.

3 FROM SPARKLE LIMITATIONS TO CKTLEAN

We analyze the original SPARKLE toolchain using 17 representative hardware tasks that expose five gaps in design support and compilation (Table 2). Appendix D, Table 8, lists all task IDs and the corresponding capability limitations. The tasks cover stateful designs, memories, bit operations, signed arithmetic, and multiple clocks. These tasks help identify limitations in SPARKLE's hardware representation and compilation pipeline.

Stateful designs. Car parking and restoring division require derived widths, parameters, loops, and tuple state to remain consistent during compilation. The original elaboration path could lose

symbolic width and parameter dependencies when processing stateful circuits. This limits the ability to represent configurable architectures faithfully: a circuit's state and module interfaces must retain the width relationships specified by its parameters throughout RTL generation.

Memories. Sync LIFO illustrates the need to handle memory in stateful circuits in which registers store the state used to compute the next state, including reset, initialization, and read/write behavior. Memory width and depth must remain available during lowering. Premature specialization of these dimensions or incomplete handling of memory operations limits parameterized memory support and requires separate checks of the emitted RTL's behavior.

Table 2: Representative tasks by capability gap (n: task count; total 17).

Design class	**Example tasks**	n
Stateful designs	Parking controller; restoring divider	2
Memories	Synchronous LIFO	2
Bit operations	Hamming encoder and decoder	2
Signed arithmetic	Complex multiplier; low-pass filter	8
Multiple clocks	Pulse synchronizer; asynchronous FIFO	3

Bit operations. Hamming-code encoder and decoder tasks combine indexed bit construction, dynamic selection, and parity reduction. In the original compilation path, symbolic dimensions and local aliases could be lost, incorrectly collapsing vector expressions into scalar or fixed-width forms. Consequently, a well-typed description need not retain its intended bit structure in the emitted RTL.

Signed arithmetic. QAM, comparators, complex multipliers, filters, Sobel, and adder tasks require consistent sign extension, arithmetic shifts, truncation, and saturation. Gaps in the representation and lowering of these operations restrict the range of arithmetic designs that can be expressed and compiled faithfully. In particular, correct widths alone do not establish that signed operations have the intended numerical behavior across backends.

Multiple clocks. Clock-domain-crossing pulse synchronizers and an asynchronous FIFO require distinct clock domains, explicit clock/reset mappings, and cross-domain structure. A single-clock representation cannot express these top-level relationships adequately. This capability gap concerns circuit composition rather than width parameterization and limits support for designs with multiple synchronous domains.

To address the five capability gaps identified above, CKTLEAN extends SPARKLE's typed IR and compilation pipeline to preserve parameter dependencies, bit structure, signedness, and clock-domain information. These extensions broaden support for complex hardware logic; Appendix D details their implementation. Passing Lean's type checks and generating SystemVerilog do not guarantee functional correctness: in the recorded parameterized-design run, the restoring divider and synchronous LIFO passed these checks but failed simulation, with a timeout in the former and incorrect full-flag behavior in the latter. These failures motivate CKTFORMALIZER's agent workflow, which uses compiler feedback to iteratively diagnose and repair generated designs.

4 METHOD

Figure 2 summarizes CKTFORMALIZER's workflow built on CKTLEAN. To clarify what the workflow verifies at each stage, we first distinguish three dimensions of correctness, then describe how the agent generates, evaluates, and optimizes hardware.

4.1 THREE DIMENSIONS OF CORRECTNESS

We distinguish three dimensions of correctness: structural correctness of hardware descriptions, formal correctness of circuit models, and functional correctness of generated RTL relative to natural-language specifications. CKTLEAN extends the supported hardware representation and compilation pipeline, while CKTFORMALIZER uses compiler diagnostics, proof-state feedback, and RTL verification to assess these dimensions during generation and repair.

Structural correctness means that a design is well formed within the supported CKTLEAN subset: bit widths, ports, and clock-domain parameters are well typed, and pattern matches are exhaustive (Hashimoto, 2026; Moura & Ullrich, 2021). Lean checks these properties during compilation (also supported by SPARKLE within its supported subset), and CKTFORMALIZER feeds compiler diagnostics back to the agent through the persistent Lean REPL to guide repair.

Formal correctness means constructing a lemma that states equivalence between a CKTLEAN circuit implementation and its formal specification, or between two circuit implementations, and proving it in Lean. The lemma expresses equality of their outputs for all inputs and execution steps covered by its assumptions, and the proof is checked by Lean's kernel (De Moura et al., 2015; Moura & Ullrich, 2021). Compiling a CKTLEAN circuit does not automatically establish its behavioral correctness. For selected candidates, CKTFORMALIZER guides the agent to develop these proofs iteratively through the persistent Lean REPL provided by CKTLEAN, including equivalence proofs between implementations before and after optimization. Only completed proofs accepted without proof placeholders (`sorry` or `admit`) qualify. These proofs establish source-level correctness relative to formal specifications. We separately assess the functional behavior of the generated RTL, as described below.

Functional correctness (evaluated by CKTFORMALIZER*)* requires the generated RTL to implement the intended behavior described by the natural-language specification. CKTFORMALIZER evaluates this behavior through reference-testbench simulation and configured Yosys equivalence checks (Wolf et al., 2013). The evidence provided by these checks depends on how faithfully the reference testbenches and RTL implementations capture the original specification. Simulation covers only the exercised inputs and time horizon, while RTL equivalence remains tool- and design-dependent. Proofs of CKTLEAN circuits do not automatically guarantee the behavior of generated RTL, so we separately verify the emitted RTL within the agent loop.

4.2 AGENTIC HARDWARE GENERATION AND OPTIMIZATION LOOP

The workflow combines compiler-guided hardware generation and repair with evaluation, architecture exploration, and PPA optimization. When enabled, interactive proof-state feedback supports source-level property and equivalence proofs, while the generated RTL is evaluated separately.

Coding Agent. The coding agent $\mathcal{A}$ is an LLM with tool use that iteratively writes, type-checks, and repairs Lean HDL. We instantiate $\mathcal{A}$ with a Codex-style coding harness (GPT-5.6-sol in the reported experiments). The agent is initialized with a compact system prompt covering CKTLEAN idioms, common pitfalls, and a short generate-then-check workflow. The prompt requires explicit registers to store the state used to compute the next state in stateful circuits. The workspace includes generic Lean HDL example files covering combinational logic, sequential circuits, and finite-state machines. Before generating a candidate, the agent may consult at most two generic Lean HDL example files as in-context examples of coding patterns and API usage. The user message contains the public natural-language specification and, when available, the required port interface, but not a reference RTL body or the test harness.

Iterative Compilation and Evaluation. The agent generates a CKTLEAN implementation from the natural-language specification in an isolated workspace and iteratively repairs it using Lean compiler diagnostics returned through the persistent Lean REPL. Generation and repair share a budget of 100 turns per problem, where each turn comprises one LLM inference and its tool calls. Initial generation is capped at 40 turns, with the remaining budget available for compile-only repair. This repair stage targets structural well-formedness and successful RTL extraction; it does not use behavioral test failures to repair functional errors. The loop ends when compilation and SystemVerilog extraction succeed or the budget is exhausted. An automated, containerized evaluator then checks the emitted RTL through linting, reference-testbench simulation, synthesis, and place-and-route to assess functional behavior and obtain PPA metrics. Simulation assesses functional correctness on the exercised traces, while synthesis and place-and-route assess backend realizability. In the reported generation runs, testbench results are not returned to the agent, and synthesis, place-and-route, and proof-state-guided equivalence proving remain outside the generation-and-repair loop.

Architecture Exploration and Metrics. For designs with a large optimization space, the agent generates C architectural candidates (default $C{=}3$), exploring choices such as parallel versus serial datapaths and tree versus chain reductions. Eligible candidates are compared using the evaluation and acceptance procedure below, and the best candidate provides the starting point for iterative refinement.

We describe design quality using the metric vector $\mathbf{m}_i = \big(A_i, P_i, W_i, N_i\big)$, whose components denote area, power, WNS, and cell count, respectively. Area is measured in μm^2 from Yosys synthesis statistics and includes sequential and combinational contributions. Power is measured in μW using OpenROAD analysis across PVT corners. Worst negative slack (WNS) is obtained from static timing analysis across SS, TT, and FF corners, with WNS $= \min_p(T_{r,p} - T_{a,p})$, where p indexes timing

paths, $T_{r,p}$ and $T_{a,p}$ denote required and arrival times, and WNS ≥ 0 indicates timing closure. Cell count serves as a proxy for design complexity and routing congestion.

Synthesis-Feedback-Guided PPA Optimization. Starting from the selected design, optimization runs for K iterations (default K=3), preserving the agent's conversation state. At iteration i, the agent receives the available metrics, their history and changes from the previous iteration, and guidance targeting the measured bottlenecks. It then modifies the Lean source, for example by simplifying multiplexer logic, removing redundant registers, or restructuring arithmetic expressions.

Candidate evaluation comprises recompilation, simulation, and synthesis; place-and-route supplies power and timing metrics when included in the configured evaluation flow. Recompilation checks that the modified design remains well typed and can generate RTL, while simulation checks whether it still passes the testbench. In the reported iterative optimization experiment (§5.3), in-loop feedback uses Yosys area and cell count only; post-route metrics are evaluated separately on the final design. These synthesis metrics provide *synthesis feedback* to guide subsequent Lean source revisions.

For candidates that pass evaluation, the full-metric acceptance rule compares them with the best design found so far. Area, power, and cell count are minimized, whereas WNS is maximized. For metric index $j \in \{1, 2, 3, 4\}$, let $d_j = +1$ for $j \in \{1, 2, 4\}$ and $d_3 = -1$. We define the normalized change Δ_j and accept a candidate if at least one metric improves by more than τ and no metric degrades by more than ϵ, where $\tau, \epsilon \geq 0$:

$$\Delta_j = d_j \frac{m'_j - m^*_j}{\max(|m^*_j|, s_j)}, \qquad \big(\exists\, j : \Delta_j < -\tau\big) \ \wedge\ \big(\forall\, j : \Delta_j \leq \epsilon\big). \tag{1}$$

Here $\mathbf{m}'$ and $\mathbf{m}^*$ denote the candidate and best-so-far metric vectors, with m'_j and m^*_j their jth components. Each $s_j > 0$ is a fixed, metric-specific normalization floor, expressed in the same units as metric j and set before optimization. These floors prevent division by zero and stabilize comparisons when the best-so-far metric is close to zero. With this convention, $\Delta_j < 0$ always indicates improvement. A candidate is retained only if it passes the configured evaluation checks and acceptance criterion; otherwise, its changes are rolled back. After K iterations, the best design found is restored as the final output.

Formal Verification and Equivalence Checking. PPA optimization changes the circuit implementation, motivating equivalence proofs between the original and revised CKTLEAN designs. We express both designs as Lean functions mapping input streams to output streams. The equivalence lemma states that, under explicit assumptions on initialization and reset, their outputs are equal for every input stream x and clock cycle t:

$$\forall x, t, \quad H(x, t) \to \mathrm{orig}(x, t) = \mathrm{revised}(x, t),$$

where H collects the stated assumptions and $\mathrm{orig}(x, t)$ and $\mathrm{revised}(x, t)$ denote the two designs' outputs at cycle t. The persistent Lean REPL supports this proof-development process: in a ReAct-style loop, the agent submits a proof step, inspects the resulting goals and diagnostics, and selects or revises the next step. A completed, kernel-checked proof establishes that the two source-level circuit models produce identical outputs under the stated assumptions.

5 EXPERIMENTS

To answer our research question, we evaluate: *(i)* structural, formal, and functional correctness, comparing with direct Verilog baselines where applicable (§5.2); *(ii)* architecture exploration and iterative PPA optimization using synthesis feedback (§5.3); and *(iii)* source-level equivalence proof generation with interactive Lean feedback (§5.3).

5.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on VerilogEval (Liu et al., 2023), comprising 156 HDLBits problems from wire assignments to complex FSMs; RTLLM (Lu et al., 2024), with 50 arithmetic, memory, and control designs; ResBench (Guo & Zhao, 2025), with 56 arithmetic, signal-processing, control, and datapath problems; and a 168-problem evaluation subset of Comprehensive Verilog Design Problems (CVDP) (Pinckney et al., 2025) covering FSM/control, arithmetic datapaths, interconnects, and memory, which introduces more challenging tasks reflecting real-world RTL design and verification scenarios. We evaluate generated RTL in Docker containers using OpenROAD (Ajayi et al., 2019) and the SkyWater 130 nm PDK (Google and SkyWater Technology, 2020), checking downstream

Table 3: End-to-end pass rates (%): $100\times$ passing designs / evaluated designs. Denominators: all problems for Compile/Sim Pass; RTL-simulation-passing designs otherwise. Synth/P&R: synthesis/place-and-route completion; DRC/LVS: design-rule/layout-versus-schematic check passes; GLS-Synth/GLS-P&R: gate-level simulation passes after the respective stages. PD Pass: joint Synth, P&R, DRC, and LVS success. Verilog Agent is our compiler-feedback control baseline; it and CKTFORMALIZER each receive 100 turns per problem. These turns comprise initial generation, compile-only repair from compiler diagnostics, and tool use (edits and builds).

Benchmark	Method	HDL	Compile	Sim Pass	Synth	P&R	DRC	LVS	GLS-Synth	GLS-P&R	PD Pass
VerilogEval (156 problems)	RTLCoder	SystemVerilog	82.1%	43.6%	**100.0%**	97.1%	97.1%	97.1%	**100.0%**	**97.1%**	97.1%
	CodeV	SystemVerilog	91.7%	42.3%	98.5%	95.5%	95.5%	95.5%	98.5%	95.5%	95.5%
	Verilog Agent	SystemVerilog	98.1%	**94.2%**	**100.0%**	95.9%	95.9%	95.9%	96.6%	93.9%	95.9%
	ICL	SystemVerilog	98.1%	**94.2%**	**100.0%**	96.2%	96.2%	96.2%	97.0%	94.7%	96.2%
		CKTLEAN	89.7%	40.4%	90.5%	85.7%	85.7%	85.7%	85.7%	85.7%	86.6%
	CKTFORMALIZER	SPARKLE	93.6%	85.9%	**100.0%**	**99.3%**	**99.3%**	**99.3%**	96.3%	97.0%	**99.3%**
		CKTLEAN	**99.4%**	83.3%	**100.0%**	95.4%	95.4%	95.4%	97.7%	94.6%	95.4%
RTLLM (50 problems)	RTLCoder	SystemVerilog	70.0%	22.0%	90.9%	90.9%	90.9%	90.9%	81.8%	81.8%	90.9%
	CodeV	SystemVerilog	80.0%	32.0%	**100.0%**	93.8%	93.8%	93.8%	81.2%	81.2%	93.8%
	Verilog Agent	SystemVerilog	94.0%	**74.0%**	97.3%	97.3%	97.3%	97.3%	81.1%	78.4%	97.3%
	ICL	SystemVerilog	**100.0%**	**74.0%**	97.1%	97.1%	97.1%	97.1%	82.9%	80.0%	97.1%
		CKTLEAN	96.0%	60.0%	93.3%	93.3%	93.3%	93.3%	93.3%	93.3%	93.3%
	CKTFORMALIZER	SPARKLE	82.0%	52.0%	**100.0%**	**100.0%**	**100.0%**	**100.0%**	84.6%	**88.5%**	**100.0%**
		CKTLEAN	**98.0%**	68.0%	**100.0%**	**100.0%**	**100.0%**	**100.0%**	**88.2%**	88.2%	**100.0%**
ResBench (56 problems)	RTLCoder	SystemVerilog	85.7%	51.8%	93.1%	93.1%	89.7%	93.1%	93.1%	93.1%	89.7%
	CodeV	SystemVerilog	**100.0%**	50.0%	92.9%	92.9%	92.9%	92.9%	92.9%	92.9%	92.9%
	Verilog Agent	SystemVerilog	**100.0%**	83.9%	57.4%	57.4%	57.4%	57.4%	55.3%	55.3%	57.4%
	ICL	SystemVerilog	**100.0%**	84.0%	60.5%	60.5%	60.5%	60.5%	58.1%	58.1%	60.5%
		CKTLEAN	85.7%	73.2%	80.5%	80.5%	80.5%	80.5%	80.5%	80.5%	80.5%
	CKTFORMALIZER	SPARKLE	**100.0%**	76.8%	95.3%	95.3%	95.3%	95.3%	93.0%	93.0%	95.3%
		CKTLEAN	98.2%	78.6%	**100.0%**	**100.0%**	**100.0%**	**100.0%**	**100.0%**	**100.0%**	**100.0%**
CVDP (168 problems)	RTLCoder	SystemVerilog	85.7%	7.7%	**100.0%**	92.3%	92.3%	92.3%	0.0%	0.0%	92.3%
	CodeV	SystemVerilog	50.0%	4.2%	85.7%	85.7%	85.7%	85.7%	0.0%	0.0%	85.7%
	Verilog Agent	SystemVerilog	98.8%	**50.0%**	89.3%	88.1%	88.1%	88.1%	29.8%	29.8%	88.1%
	ICL	SystemVerilog	98.8%	**50.0%**	89.7%	88.5%	88.5%	88.5%	**33.3%**	**33.3%**	88.5%
		CKTLEAN	75.6%	16.1%	55.6%	55.6%	55.6%	55.6%	29.6%	25.9%	55.6%
	CKTFORMALIZER	SPARKLE	77.4%	25.6%	67.4%	67.4%	67.4%	67.4%	27.9%	27.9%	67.4%
		CKTLEAN	91.1%	40.5%	**100.0%**	**97.1%**	**97.1%**	**97.1%**	30.9%	27.9%	**97.1%**

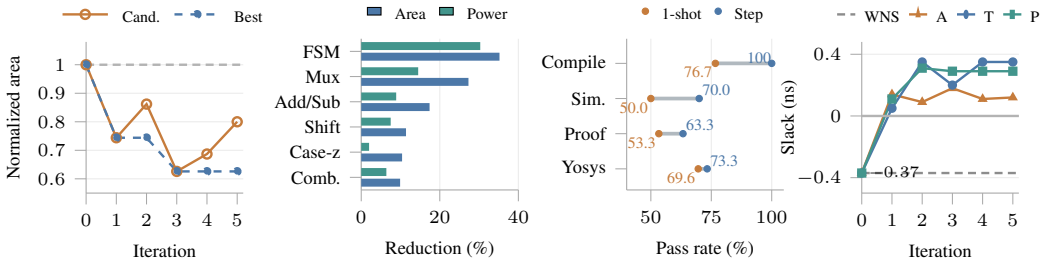
realizability through synthesis, place-and-route, design-rule checking (DRC), layout-versus-schematic (LVS) verification, and gate-level simulation.

Baselines. We compare CKTFORMALIZER with four baselines: RTLCoder (Liu et al., 2024), using the 7B RTLCoder-DeepSeek-v1.1 model; CodeV (Zhao et al., 2025), using CodeV-All-QC; ICL and Verilog Agent, both of which generate SystemVerilog directly. ICL uses in-context learning (ICL) in a single model call. Verilog Agent is our control baseline, which directly generates SystemVerilog and uses an agent harness to iteratively repair the code with compiler feedback. It uses the same model and 100-turn budget as CKTFORMALIZER. ICL, Verilog Agent, and CKTFORMALIZER all use `gpt-5.6-sol`. Whereas Verilog Agent repairs SystemVerilog directly, CKTFORMALIZER generates Lean HDL and uses Lean type-checking and extraction diagnostics for iterative repair before emitting SystemVerilog (§4.2). For the CKTFORMALIZER runs on VerilogEval and RTLLM, reference implementation bodies are removed from the generation prompts. All methods are evaluated using the same physical-design flow with the SkyWater 130 nm HD standard-cell library.

5.2 MAIN RESULTS

Table 3 summarizes the end-to-end results on VerilogEval, RTLLM, ResBench, and CVDP. Downstream pass rates can exceed Sim Pass because they are conditioned on passing RTL simulation, rather than evaluated over all problems.

Structural Correctness. Lean compilation checks the supported structural invariants, including bit-width consistency and exhaustive pattern matching (§4.1). *(i) Compilation.* CKTFORMALIZER enables LLMs to generate compilable hardware through CKTLEAN's typed, high-level representation, achieving compilation rates of 99.4%, 98.0%, 98.2%, and 91.1% on VerilogEval, RTLLM, ResBench, and CVDP, respectively. These results demonstrate that high compilation coverage is achievable while enforcing the supported structural constraints, making typed hardware generation a practical alternative to direct SystemVerilog generation. *(ii) Backend realizability.* All RTL-simulation-passing designs from CKTFORMALIZER with CKTLEAN synthesize successfully across the four benchmarks.



(a) Optimization. (b) Area–power gains. (c) Proof generation. (d) Tight-clock slack.

Figure 3: (a) Measured Yosys synthesis-area trajectory: orange is the candidate at each iteration; blue is the accepted best-so-far (running minimum). Iterations 2, 4, and 5 are rejected worse candidates; the best area first appears at iteration 3 and is held through iteration 5. The gray dashed line marks the seed area ($= 1$). (b) Area and power reductions for the six simulation-passing examples in Table 4. (c) One-shot vs. stepwise proof generation on a 30-problem VerilogEval pilot. (d) Post-route setup slack over $K{=}5$ iterations under a tight clock. The dashed line is the seed WNS (-0.37 ns); orange/blue/teal are area, timing, and power feedback. The gray line at 0 is timing closure.

Among those same designs, P&R, DRC, and LVS pass rates are each 95.4%, 100.0%, 100.0%, and 97.1% on VerilogEval, RTLLM, ResBench, and CVDP, respectively. The largest synthesis gap relative to the SystemVerilog baselines occurs on ResBench, where ICL with SystemVerilog and Verilog Agent reach 60.5% and 57.1%. *(iii) Infrastructure contribution.* Within the same CKTFORMALIZER workflow, replacing SPARKLE with CKTLEAN raises synthesis pass rates from 95.3% to 100.0% on ResBench and from 67.4% to 100.0% on CVDP. PD Pass also rises from 95.3% to 100.0% and from 67.4% to 97.1%, respectively. These results support CKTLEAN's contribution to backend realizability on these benchmarks.

Functional Correctness. CKTFORMALIZER exceeds RTLCoder and CodeV in RTL simulation on all four benchmarks, but does not consistently lead direct-generation baselines: Verilog Agent leads on VerilogEval and ResBench, while ICL leads on RTLLM and CVDP (Table 3). Among RTL-simulation-passing designs, CKTFORMALIZER achieves 88.2% gate-level simulation pass rates after both synthesis and P&R on RTLLM, and 100.0% on ResBench, exceeding ICL with SystemVerilog and Verilog Agent at both stages. Its GLS-P&R rate matches Verilog Agent on VerilogEval (94.6%) but falls below both baselines on CVDP (27.9% vs. 33.3%). These results demonstrate stronger backend functional robustness on RTLLM and ResBench relative to the direct SystemVerilog baselines. *With PD Pass rates of 95.4%–100.0% among RTL-simulation-passing designs,* CKTFORMALIZER *demonstrates that hardware generated through* CKTLEAN*'s typed abstractions can translate into physically realizable implementations.*

Findings. CKTLEAN*'s typed abstractions enable hardware generation that extends beyond simulation to physical implementation.*

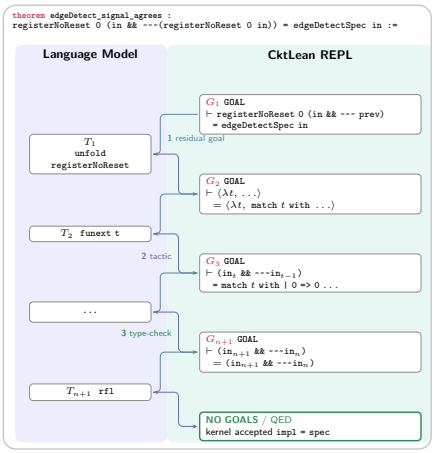


Figure 4: Stepwise proof of the rising-edge detector equivalence lemma: the registered implementation matches its specification for every input stream. Teal: goals (REPL → model); violet: tactics (model → REPL).

5.3 HARDWARE OPTIMIZATION AND VERIFICATION

Equivalence Proof Generation. *As a step toward correctness-preserving PPA optimization, we explore whether stepwise Lean feedback enables LLMs to prove implementation–specification equivalence.* We run a 30-problem VerilogEval pilot to isolate the effect of proof-generation strategy. Figure 3(c) compares the two modes. (i) *One-shot.* The model writes the implementation, specification, and Lean proof in a single attempt without intermediate proof-goal feedback. (ii) *Stepwise.* The

model can query Lean proof states and iteratively refine the theorem. A proof counts as complete only if Lean's kernel accepts the final theorem without `sorry` or `admit`.

Stepwise proof development improves all metrics, raising compile pass from 77% to 100%, simulation from 50% to 70%, kernel-accepted proof completion from 53.3% (16/30) to 63.3% (19/30), and Yosys equivalence from 70% to 73%. These completed proofs establish source-level formal correctness relative to their stated Lean specifications.

Figure 4 illustrates the proof for a registered rising-edge detector whose output must match its specification for every input stream. The model unfolds definitions and refines tactics using REPL feedback until Lean's kernel accepts the proof that the implementation and specification produce identical outputs for every input stream.

Findings. *By exposing intermediate proof states beyond whole-file compiler feedback, the* CKTLEAN *REPL improves proof completion through stepwise refinement.*

Architecture Exploration. We evaluate architecture exploration on the 63-problem VerilogEval subset (`Prob035–Prob156`, the more complex half of the 156-problem benchmark, excluding simple wiring and gate tasks), generating $C{=}4$ candidates per design with different micro-architectural choices. (i) *Coverage.* Of 63 problems, 27 produced multiple successfully synthesized candidates, enabling cross-candidate comparison. (ii) *Overall gains.* Among these 27 designs, 10 (37.0%) achieved synthesis-level area reductions exceeding 5%. (iii) *Illustrative examples.* Table 4 presents six examples with simulation-passing candidates, with area reductions of 9.9%–35.2% and post-route power reductions of 2.0%–30.3%. Appendix F explains the design labels. Figure 3(b) plots the area and power reductions for these six examples.

Findings. *Synthesis feedback guides PPA optimization, while equivalence proofs can support more aggressive changes by verifying source-level behavioral preservation.*

PPA Optimization Loop. We run $K{=}5$ PPA optimization iterations on the 63-problem VerilogEval subset, using the simulation-passing semantic CKTFORMALIZER seeds. 54 designs enter the loop (9 seeds are not `sim_pass`); 31 change Lean and reduce Yosys area, of which 28 exceed $\tau{=}5\%$. Figure 3(a) plots a measured candidate vs. best-so-far trajectory. (i) *Effectiveness.* The largest Yosys-area reduction is -58.6% ($196 \rightarrow 81\,\mu\mathrm{m}^2$). This loop uses synthesis-level area and cell count. (ii) *Non-monotonic trajectories.* Optimization trajectories are non-monotonic—Lean's type system rejects unsafe transformations, limiting the search space but preventing optimization-induced bugs. (iii) *Comparison with architecture exploration.* Architecture exploration generates and compares alternative microarchitectures (Table 4 and Figure 3(b)), whereas this loop iteratively modifies a selected design. (iv) *Tight-clock P&R feedback.* A separate pass feeds post-route setup slack and power to the agent. Figure 3(d) shows one tight-clock case: timing and power arms close WNS (setup slack from -0.37 to $+0.35$ and $+0.31\,\mathrm{ns}$). In the area-focused optimization run, the selected design's setup slack remains at the initial value of $-0.37\,\mathrm{ns}$. *This example shows that post-route feedback can guide optimization toward timing closure, which area-focused optimization alone does not ensure.*

Table 4: Area, cell-count, and power reductions from architecture exploration.

Design	Reduction (%)		
	Area	Cells	Power
FSM (Lemmings)	**35.2**	35.0	**30.3**
9-to-1 Mux	27.3	**45.9**	14.5
Add/Sub Unit	17.4	26.2	8.9
FSM Shift Reg.	11.4	0.0	7.5
Case-z Logic	10.4	0.0	2.0
Comb. Logic	9.9	14.8	6.4

6 CONCLUSION

We presented CKTFORMALIZER, an agent framework built on our core contribution, CKTLEAN, which combines typed hardware representations, compilation, and a persistent Lean REPL. Across four benchmarks, it generates hardware that passes structural checks and supports physical implementation. Proof-state feedback improves kernel-checked equivalence proof completion, while hardware evaluation guides architecture exploration and PPA optimization. Typed infrastructure and feedback thus support LLM-driven circuit generation, optimization, and verification.

AI USE STATEMENT

Generative AI is both the subject of study and an implementation component of the evaluated system: language models generate candidate Lean or SystemVerilog artifacts under the workflows described in Section 4, particularly Section 4.2. These artifacts are checked with the compiler, simulation, proof, synthesis, equivalence, and physical-design stages reported in the paper. An AI coding assistant was additionally used to migrate the manuscript source to the ICLR template and check formatting. The author reviewed the AI-assisted manuscript changes and takes responsibility for the final text, claims, citations, and artifacts.

REFERENCES

Tutu Ajayi, David Blaauw, Timothy Chan, Chien-Yi Cheng, Vidya A Chhabria, Daniel K Choo, Matteo Coltella, Stefan Dobre, Ronald Dreslinski, Mateus Fogaça, et al. OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain. *GOMAC*, 2019.

Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. C? ash: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pp. 714–721. IEEE, 2010.

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th annual design automation conference*, pp. 1216–1225, 2012.

Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Handbook of satisfiability*, 185(99):457–481, 2009.

Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Chip-chat: Challenges and opportunities in conversational hardware design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pp. 1–6. IEEE, 2023.

Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, pp. 24–40, 2010.

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36, 2011.

Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Li, Huawei Song, and Weikang Li. ChipGPT: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. In *ICFP*, pp. 1–30, 2017.

Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pp. 1–30. Springer, 2011.

Design Automation Standards Committee et al. Ieee standard for systemverilog unified hardware design, specification, and verification language standard ieee 1800. *http://www. edastds. org/sv/*, 2005.

Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International conference on automated deduction*, pp. 378–388. Springer, 2015.

Google and SkyWater Technology. The SkyWater open source PDK, 2020. `https://github.com/google/skywater-pdk`.

Ce Guo and Tong Zhao. Resbench: A resource-aware benchmark for llm-generated fpga designs. In *Proceedings of the 15th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, pp. 25–34, 2025.

Junji Hashimoto. Sparkle HDL: A type-safe, formally verifiable HDL compiler in Lean 4. Verilean, GitHub repository, 2026. Version 0.1.0; `https://github.com/Verilean/sparkle/tree/v0.1.0`.

Warren A Hunt Jr, Matt Kaufmann, J Strother Moore, and Anna Slobodova. Industrial hardware and software verification with acl2. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 375(2104):20150399, 2017.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *ICLR*, 2024.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–8. IEEE, 2023.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(4):1448–1461, 2024.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 722–727. IEEE, 2024.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pp. 69–70. IEEE, 2004.

Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. DAVE: Deriving automatically Verilog from English. *MLCAD*, pp. 27–32, 2020.

Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on rtl design and verification. *arXiv preprint arXiv:2506.14074*, 2025.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Silver Oak Team. Silver oak: Hardware software co-design and co-verification in coq. In *Proceedings of the Workshop on Languages, Tools, and Architectures for Formal Hardware Verification (PLARCH)*, June 2023. URL https://pldi23.sigplan.org/details/plarch-2023-papers/23/Silver-Oak-Hardware-Software-Co-Design-and-Co-Verification-in-Coq. Co-located with PLDI 2023.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3):1–31, 2024a.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. VeriGen: A large language model for Verilog code generation. *ACM TODAES*, 29(3):1–31, 2024b.

The Mathlib Community. Mathlib: The mathematics library of Lean 4. https://github.com/leanprover-community/mathlib4. Accessed September 14, 2026.

Verilean. Sparkle HDL: Circuit-DSL development history. GitHub commits, 2026a. `Signal.circuit do` introduced in commit `d22cb7d` and replaced by `circuit do` in commit `104f2cc`; accessed August 10, 2026; https://github.com/Verilean/sparkle/commit/d22cb7da06c6debecfc18094e5998e6064ba5ea4; https://github.com/Verilean/sparkle/commit/104f2cc14d3e80d040c2ee93cf3c5e8945c07147.

Verilean. Sparkle HDL: Circuit-level verification modules. GitHub repository, 2026b. Commit `fa0e48e`; accessed August 9, 2026; https://github.com/Verilean/sparkle/tree/fa0e48e3895f47b3e42ce6fbcb045abff10c38b1/Sparkle/Verification.

Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, pp. 1–6, 2013.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *ICLR*, 2023.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, et al. Codev: Empowering llms with hdl generation through multi-level summarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

A SPARKLE CIRCUIT-LEVEL PROOFS ACROSS VERSIONS

SPARKLE's verification surface differs across releases. Our experimental pipeline pins v0.1.0 (Hashimoto, 2026). That release provides typed `Signal` construction, cycle-level evaluation, and SystemVerilog extraction, and includes Lean lemmas over pure `BitVec` and ISA models; it does not yet contain a dedicated bridge from synthesizable `Signal` terms to those specifications or a theorem for unfolding feedback loops. Later revisions add `SignalDSLProps`, `Equivalence`, `Temporal`, and `LoopProps` modules (Verilean, 2026b). The two examples below use the current, post-v0.1.0 interfaces to illustrate capabilities now available in SPARKLE; they should not be read as capabilities exercised by the v0.1.0 experimental runs. Both examples establish source-level properties only and do not prove semantic preservation of SystemVerilog extraction.

A.1 CONNECTING A SIGNAL CIRCUIT TO A PURE SPECIFICATION

Listing 1 separates a pure functional specification from the synthesizable `Signal` implementation. The theorem quantifies over arbitrary input streams and every cycle `t`; evaluating the circuit at `t` yields the same value as applying the pure specification to the inputs at `t`. Lean accepts the proof by definitional reduction, so its kernel checks the connection to the actual `Signal` term rather than to an unrelated test model. This deliberately minimal example demonstrates the semantic bridge, not a complex functional property.

Listing 1: A combinational `Signal` circuit and its pure functional specification.

```
import Sparkle

open Sparkle.Core.Domain
open Sparkle.Core.Signal

namespace AppendixProofs

def addCircuit
    (a b : Signal defaultDomain (BitVec 8)) :
    Signal defaultDomain (BitVec 8) :=
  a + b

def addSpec (a b : BitVec 8) : BitVec 8 :=
  a + b

theorem addCircuit_correct
    (a b : Signal defaultDomain (BitVec 8)) (t : Nat) :
    (addCircuit a b).val t = addSpec (a.val t) (b.val t) := by
  rfl

#synthesizeVerilog addCircuit

end AppendixProofs
```

A.2 REASONING ABOUT A REGISTERED FEEDBACK LOOP

Listing 2 is sequential. The implementation closes a feedback path through `Signal.register`, while `counterPure` is an ordinary recursive function over cycles. The proof first establishes strict causality, then applies SPARKLE's kernel-checked `loop_unfold` fixpoint theorem explicitly. The subsequent `loop_iterate` proof connects the feedback circuit to the pure recurrence at every cycle, rather than only over a bounded simulation trace.

Listing 2: An all-cycle refinement proof for a registered feedback counter.

```
import Sparkle
import Sparkle.Verification.LoopProps

open Sparkle.Core.Domain
```

```
open Sparkle.Core.Signal
open Sparkle.Verification.LoopProps

namespace AppendixProofs

def counterBody (s : Signal defaultDomain (BitVec 8)) :
    Signal defaultDomain (BitVec 8) :=
  Signal.register 0#8 (s + 1#8)

def counterLoop : Signal defaultDomain (BitVec 8) :=
  Signal.loop counterBody

def counterPure : Nat -> BitVec 8
  | 0 => 0#8
  | t + 1 => counterPure t + 1#8

theorem counterBody_strict : StrictlyCausal counterBody := by
  apply strictlyCausal_of_step
    (c0 := 0#8)
    (next := fun _ x => x + 1#8)
  · intro s
    rfl
  · intro s t
    simp [counterBody]

theorem counterLoop_unfold :
    counterLoop = counterBody counterLoop := by
  exact loop_unfold counterBody counterBody_strict

theorem counterLoop_correct (t : Nat) :
    counterLoop.val t = counterPure t := by
  apply loop_iterate
    (f := counterBody)
    (c0 := 0#8)
    (next := fun _ x => x + 1#8)
    (st := counterPure)
  · intro s
    rfl
  · intro s t
    simp [counterBody]
  · rfl
  · intro t
    rfl

theorem counterLoop_counts (t : Nat) :
    counterLoop.val t = BitVec.ofNat 8 t := by
  rw [counterLoop_correct]
  induction t with
  | zero => rfl
  | succ u ih =>
      rw [counterPure, ih]
      bv_omega

#synthesizeVerilog counterLoop

end AppendixProofs
```

B EXPERIMENTAL SETUP DETAILS

This appendix records the SPARKLE main-flow protocol used for Table 3: benchmark inputs, model and action-budget settings, the agent tool interface, and the evaluator used after each generation or

compile-only repair session. Architecture exploration, in-the-loop PPA, simulation-mismatch repair, and the extra skill-pack prompt were disabled for these runs.

B.1 Benchmarks and Task Inputs

We evaluate RTL generation on four benchmarks. VerilogEval (Liu et al., 2023) contains 156 HDLBits-derived problems spanning combinational logic, sequential circuits, counters, shift registers, and finite-state machines. RTLLM (Lu et al., 2024) contains 50 natural-language RTL tasks covering arithmetic, memory, control, and miscellaneous modules. ResBench (Guo & Zhao, 2025) contains 56 arithmetic, signal-processing, control, and datapath tasks. CVDP (Pinckney et al., 2025) uses the official 168-problem non-agentic, no-commercial code-generation split, not the full 783-problem release. That split covers single-turn RTL generation without commercial EDA tools or restricted IP; 156 tasks are ordinary fixed-width designs and 12 retain native parameters.

The coding agent sees only the public specification. On CVDP this is the official prompt and context together with the evaluation top-module name. Hidden testbenches, gold RTL bodies, and simulator traces are withheld. VerilogEval, RTLLM, and ResBench already expose public natural-language specifications and use the same agent loop. Reference RTL may be used by the evaluator for wrappers; it is not an agent input. The SPARKLE agent writes one Lean HDL file per problem. The evaluator extracts SystemVerilog and scores it with the hidden harness. Direct-SystemVerilog baselines use the same backbone model but emit Verilog rather than Lean.

The boxes below reproduce the generation templates used for Table 3. Placeholders stand for the public specification and required module name. ICL is a single call and does not insert worked solutions: its only in-context fragment is the required port header. CKTFORMALIZER is the method that uses in-context examples: the workspace mounts 13 generic Lean idiom files (combinational, sequential, and FSM sketches, including `RTLIdioms.lean`), and the compact skill allows at most two reads (three on CVDP). These files are style references, not evaluation gold; their full contents appear in Appendix G.

RTLCoder (pass@1)

```
Please act as a professional verilog designer.

<public natural-language specification>

module TopModule(
  input  clk,
  input  reset,
  output [7:0] out
);
```

CodeV (pass@1)

```
<Verilog>
Generate synthesizable SystemVerilog for the following hardware task.
Output only one complete module, from module declaration to endmodule.
The module name must be TopModule.
Keep the exact external port interface shown below.

Natural language specification:
<public natural-language specification>

Required module interface:
module TopModule(
  input  clk,
  input  reset,
  output [7:0] out
);

SystemVerilog implementation:
```

ICL (one-shot gpt-5.6-sol)

System.

```
You are an expert hardware designer. Generate synthesizable SystemVerilog code.

RULES:
1. The module MUST be named `TopModule` with EXACTLY the ports specified.
2. Output ONLY the SystemVerilog module code (module ... endmodule). No explanation.
3. Use SystemVerilog-2012 syntax compatible with Icarus Verilog (-g2012).
4. For sequential logic, use `always_ff @(posedge clk)` when appropriate.
5. For combinational logic, use `assign` or `always_comb`.
```

User (in-context port header; gold body withheld).

```
Specification:
<public natural-language specification>

Reference port interface (your module MUST be named TopModule with the same ports):
```
module RefModule (
  input clk,
  input reset,
  output [7:0] out
);
```

Generate the complete `TopModule` implementation:
```

Harness (Archon compile-fix, 100 turns)

```
You are an expert hardware designer using an Archon-style agent harness
to implement RTL directly in SystemVerilog.

## Goal
Generate one synthesizable SystemVerilog module that matches the
benchmark contract and passes the external compile/simulation harness.

## Output File
- Write the candidate to cktarchon_work/<id>/candidate.sv.
- The file must contain complete SystemVerilog with module ... endmodule.
- Do not write Lean or Sparkle.

## RTL Rules
- Use SystemVerilog-2012 syntax compatible with Icarus Verilog/cocotb.
- Match the required top-module name, ports, widths, and clock/reset.
- Avoid unsynthesizable constructs in the DUT.

## Workflow
1. Read the problem and benchmark interface contract.
2. Write a complete candidate.
3. If compile-only feedback is provided, repair the same file.
4. Stop after the candidate file is written.

## Problem
- Required top module: `TopModule`
- Natural-language specification: <public specification>
```

CKTFORMALIZER compact skill

```
You are an expert hardware engineer translating natural-language RTL
specifications into Sparkle HDL, a Lean 4 hardware DSL.

## File Template
import Sparkle
import Sparkle.Compiler.Elab
open Sparkle.Core.Signal
def <target> {dom : DomainConfig} (<inputs>) : <output_type> :=
  <implementation>
#synthesizeVerilog <target>

## Workflow
1. Read at most two small examples, preferably Benchmark/RTLIdioms.lean
   plus one similar Benchmark/*.lean.
2. Write a complete candidate to Generated/<id>.lean.
3. Run the harness Lean check and fix compiler/extraction errors.
4. Stop after the Lean check reports success.
```

B.2 Model and Hyperparameters

All SPARKLE main-flow runs use a Codex coding agent with `gpt-5.6-sol`, reasoning effort ultra, the Responses API, and 16,384 maximum output tokens, in an isolated workspace-write sandbox. A turn is one Codex action. Generation and compile-only repair share 100 actions: the initial Lean candidate is capped at 40, and leftover actions are used for repair in at most 9 rounds of at most 10 actions. The evaluator may retry infrastructure failures without spending model turns.

SPARKLE Version. All reported experiments pin SPARKLE v0.1.0 (commit `13458ca`). Upstream v0.1.0 does not contain the imperative circuit notation used in our experimental source; the experimental stack provides a local `Signal.circuit do` front end that lowers to `Signal.loop` and `Signal.register`. SPARKLE subsequently added the qualified `Signal.circuit do` form in commit `d22cb7d` and later replaced it with the unqualified `circuit do` form in commit `104f2cc` (Verilean, 2026a). The circuit-level proof interfaces illustrated in Appendix A use the later commit `fa0e48e` and document SPARKLE's subsequent evolution; these interfaces were not exercised in the reported experimental runs.

System prompt and idioms. Main runs use a compact SPARKLE skill plus harness rules: import `Sparkle` and `Sparkle.Compiler.Elab`, write one Lean candidate per problem, run the prescribed Lean check until SystemVerilog is emitted, and stop. The agent does not run simulation, cocotb, Yosys, or place-and-route. The sandbox mounts 13 generic idiom files, including `RTLIdioms.lean` and small combinational, sequential, and FSM sketches. These files are not evaluation gold; they are listed in Appendix G. The prompt allows at most two reads, preferably `RTLIdioms.lean` and one similar file (at most three on CVDP).

Compile-only repair. Repair uses only pre-simulation failures: Lean elaboration, missing SystemVerilog, parameter-contract errors, and `iverilog` lint or elaboration. On CVDP, a public port check may trigger at most one structural repair. Hidden simulation failures are not fed back. Post-hoc PPA on sky130hd (10 ns clock) does not call the model. Table 5 lists the remaining knobs.

Table 5: SPARKLE main-flow hyperparameters.

Setting	Value
Model / effort / max tokens	`gpt-5.6-sol`, ultra, 16,384
Harness	Codex, workspace-write, Responses API
Shared action budget	100 (generation cap 40)
Compile-only repair	at most 9 rounds $\times$ 10 actions
CVDP structural repair	at most one public port check
Feedback / hidden simulation	compile-only / withheld
CVDP / other dataset specs	public-spec-v2 / public NL specs
Idiom files	13 mounted; read at most 2 (3 on CVDP)

B.3 Component Ablation on VerilogEval Subset

Table 6 reports a component ablation on a fixed 50-problem VerilogEval subset sampled uniformly at random without replacement using seed 20260505. The Full row is filtered from an existing backend-enabled 156-problem VerilogEval run, and the ablation rows are recomputed from per-problem result records on the same subset. All metrics use the 50 sampled problems as the denominator. Each ablation row reports the result after removing one component from the full system. Removing *Iterative Repair* disables the multi-turn repair loop after the initial Lean HDL generation, so the model cannot use compiler or evaluator feedback to edit the design. Removing *Lean REPL Feedback* keeps the outer generation loop but removes interactive Lean REPL state and proof-goal feedback, leaving the agent to rely on file edits and batch compiler diagnostics. Removing *Few-shot Examples* removes the verified Lean HDL examples from the prompt context and blocks access to the benchmark example library during generation.

Table 6: Component ablation on the fixed 50-problem VerilogEval subset. Metrics are recomputed from per-problem JSONL records and reported independently; skipped or missing downstream checks count as failures under the same 50-problem denominator.

Removed Component	Compile	Sim.	Synth	P&R	DRC	LVS	GLS-Synth	GLS-P&R
None (Full)	50/50 (100.0%)	35/50 (70.0%)	50/50 (100.0%)	47/50 (94.0%)	47/50 (94.0%)	47/50 (94.0%)	34/50 (68.0%)	32/50 (64.0%)
Iterative Repair	44/50 (88.0%)	22/50 (44.0%)	32/50 (64.0%)	31/50 (62.0%)	31/50 (62.0%)	31/50 (62.0%)	22/50 (44.0%)	21/50 (42.0%)
Lean REPL Feedback	50/50 (100.0%)	35/50 (70.0%)	50/50 (100.0%)	47/50 (94.0%)	47/50 (94.0%)	47/50 (94.0%)	34/50 (68.0%)	33/50 (66.0%)
Few-shot Examples	49/50 (98.0%)	34/50 (68.0%)	49/50 (98.0%)	47/50 (94.0%)	47/50 (94.0%)	47/50 (94.0%)	34/50 (68.0%)	32/50 (64.0%)

B.4 Agent Tools and Evaluation Pipeline

The Codex agent may read mounted SPARKLE sources and the idiom library, edit the per-problem Lean file, search the repository, and run a Lean check until SystemVerilog is emitted. After each generation or repair session the outer evaluator, which holds the hidden harness, runs the same ordered checks. Lean elaboration (REPL or `lake build`) executes `#synthesizeVerilog` or `#synthesizeParameterizedVerilog`. The harness extracts SystemVerilog, optionally wraps ports onto the benchmark interface, and runs `iverilog` lint (`-t null -g2012`). Native CVDP tasks elaborate every public parameter case. CVDP additionally checks public port names, directions, and numeric widths. Functional simulation then uses the hidden testbench: Icarus Verilog and `vvp` on VerilogEval, RTLLM, and ResBench; cocotb on CVDP. Compile-only repair may use diagnostics from stages before simulation; candidate selection uses compile/extract/lint and public-structure progress, not simulation scores.

Designs that reach `sim_pass` may be sent to a separate OpenROAD Flow Scripts backend on `sky130hd` (Yosys, place-and-route, DRC, LVS, optional gate-level simulation). That backend does not call the model and does not change the simulation-pass numbers in the main table.

B.5 EDA Tools

Table 7 lists tools used after a Lean candidate exists. Lean and Icarus participate in the agent-visible compile path; Yosys, OpenROAD, and KLayout run only in the post-hoc backend. PPA metrics and Yosys `equiv_*` checks are not returned to the SPARKLE coding agent in the main flow.

Table 7: EDA tools used by the automated evaluation and backend implementation flow.

Tool	Pipeline Stage	Role
`Lean 4`	Source compilation	Type-checks Lean HDL and runs the `#synthesizeVerilog` metaprogram to emit synthesizable SystemVerilog.
`Icarus Verilog`	RTL lint and simulation	Runs `iverilog -t null -g2012` for syntax checking and `iverilog -g2012` for testbench compilation.
`vvp`	RTL and gate-level simulation	Executes the simulator output produced by Icarus Verilog and records testbench pass/fail status or mismatch counts.
`Yosys`	Post-hoc logic synthesis	Performs synthesis inside the OpenROAD Flow Scripts backend. Optional `equiv_*` RTL checks are not part of the SPARKLE generation loop.
`OpenROAD Flow`	Backend orchestration	A set of scripts that orchestrate the EDA backend flow (synthesis, place-and-route, DRC, LVS) inside a Docker container, enabling reproducible physical implementation from RTL to GDSII.
`OpenROAD/OpenSTA`	Physical implementation and timing	OpenROAD is an open-source digital design implementation tool; `OpenSTA` is its static timing analysis engine. Together they perform floorplanning, placement, clock-tree synthesis, routing, finishing, and static timing analysis (STA); the evaluator extracts WNS, power, and PPA metrics from OpenROAD reports.
`KLayout`	Physical verification	Runs DRC and LVS checks through the OpenROAD Flow signoff targets.
`Docker`	Reproducible execution	Runs the backend flow in the `openroad/orfs:latest` container with mounted result, log, and report directories.
`SkyWater sky130hd`	Technology target	Provides the `sky130_fd_sc_hd` standard-cell library and PDK configuration used for synthesis, place-and-route, timing, DRC, and LVS.
`volare`	Cell-model management	Fetches SkyWater standard-cell simulation models when gate-level simulation requires local cell libraries.
`netlistsvg`	Optional visualization	Renders Yosys-generated JSON netlists into schematic SVGs for inspection.

B.6 Error Analysis: Case Study

We manually inspect the 31 VerilogEval designs that compile in Lean but do not pass RTL simulation in the main run. The boxes below give representative cases with the archived evaluator signal and the corresponding testbench evidence.

Case 1: Reset timing – `Prob047_dff8ar`

Prompt. Eight D flip-flops with active-high asynchronous reset.

```
log: compile=pass, lint=pass, sim=fail, synth/pnr/gds=pass
     44 mismatches in 436 samples
tb:  reset_test(1);
     Hint: reset should be asynchronous/synchronous.
```

Case 2: Datapath counter logic – `Prob141_count_clock`

Prompt. A 12-hour BCD clock with reset priority, enable gating, carry propagation, and AM/PM rollover.

```
log: compile=pass, lint=pass, sim=fail, synth/pnr/gds=pass
     177947 mismatches in 200000 samples
tb:  Hint: reset value / reset priority
     Minute roll-over; Hour roll-over; PM roll-over
```

Case 3: FSM corner case – `Prob155_lemmings4`

Prompt. A Lemmings FSM with walking, falling, digging, reset, and long-fall splatter behavior.

```
log: compile=pass, lint=pass, sim=fail, synth/pnr/gds=pass
     44 mismatches in 1003 samples
tb:  repeat(21) @(posedge clk);  // splat after falling left
     repeat(24) @(posedge clk);  // long-fall boundary
```

Case 4: Interface artifact – `Prob099_m2014_q6c`

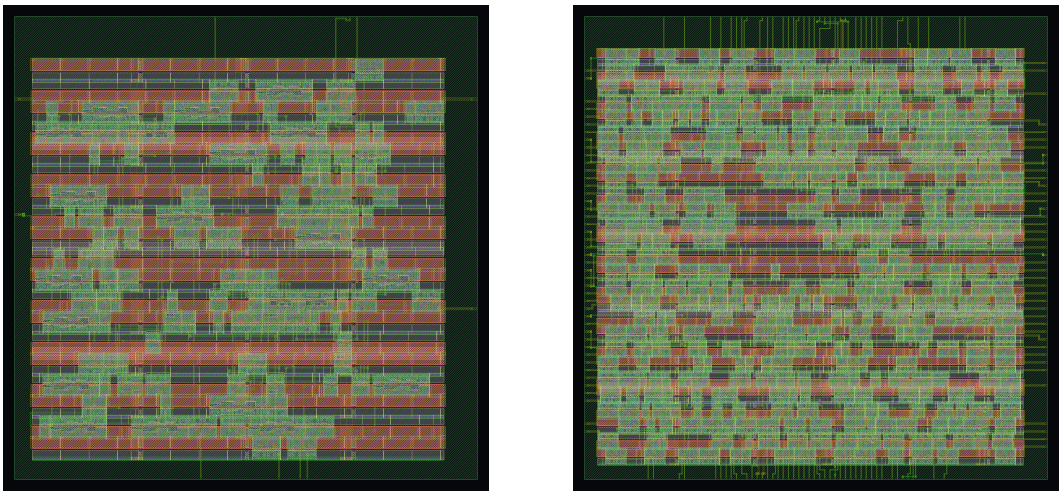
Prompt. One-hot FSM next-state logic with benchmark-level port-name inconsistency.

```
log: compile=pass, lint=pass, sim=error
tb:  Prob099_m2014_q6c_test.sv:71: port 'Y2' is not a port of good1.
     Prob099_m2014_q6c_test.sv:71: port 'Y4' is not a port of good1.
     Prob099_m2014_q6c_test.sv:77: port 'Y2' is not a port of top_module1.
```

These cases show that the residual failures are not backend failures: the designs either reach physical implementation and fail the behavioral oracle, or fail at the benchmark harness interface before simulation can run.

B.7 Example Physical Layouts

Figure 5 shows two raster renderings of OpenROAD-generated GDSII layouts for generated VerilogEval designs: `Prob156` (*fancytimer*), a multi-output sequential timer with BCD display logic, and `Prob097` (*mux9to1v*), a 9-to-1 multiplexer with 16-bit datapath. These examples are included as qualitative illustrations of the layouts emitted by the automated synthesis and place-and-route flow; quantitative backend pass rates are reported in Section 5.



(a) `Prob156` (*fancytimer*) (b) `Prob097` (*mux9to1v*)

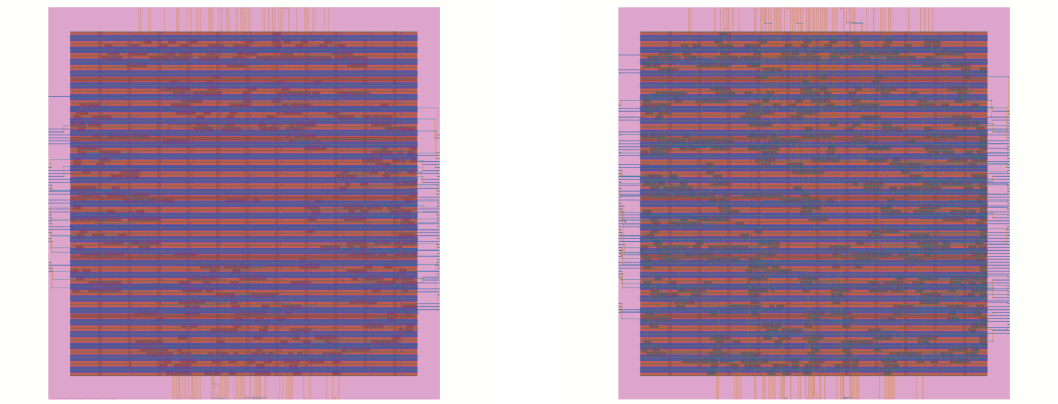
Figure 5: Raster renderings of OpenROAD-generated GDSII layouts for two VerilogEval designs.

B.8 AXI4-Lite Physical Layouts

To demonstrate that CktFormalizer scales beyond educational benchmarks to industry-relevant IP, we generate both sides of an AXI4-Lite bus interface, the ARM AMBA protocol used for register access in virtually all modern SoCs. AXI4-Lite requires correct multi-channel handshake logic (AW/W/B for writes, AR/R for reads) with back-pressure and response sequencing; in hand-written Verilog, missing handshake cases and width mismatches across channels are a common source of protocol violations. In Lean HDL, the type system statically enforces channel widths and exhaustive case coverage, eliminating these error classes at compile time.

Figure 6 shows the OpenROAD-generated layouts for the target-side (slave) and initiator-side (master) modules. Both designs pass the full automated pipeline, from natural language specification through

Lean compilation, SystemVerilog extraction, synthesis, and place-and-route, with zero DRC, setup, hold, slew, fanout, and capacitance violations, requiring no manual intervention at any stage.



(a) Target-side module

(b) Initiator-side module

Figure 6: OpenROAD-generated AXI4-Lite target-side and initiator-side physical layouts.

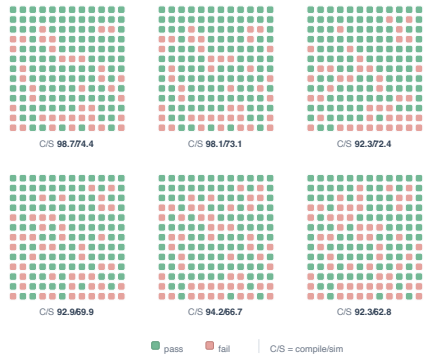


Figure 7: Six compact pass/fail dashboards.

C Compact Result Dashboard

In addition to the aggregate metrics in Section 5, we use compact dashboards to inspect representative CktFormalizer runs at problem granularity. As shown in Figure 7, the embedded view keeps only the audit signals needed for the paper: per-problem simulation outcomes and compact compile/simulation pass-rate summaries. The matrices make it easy to see whether failures are isolated to a few difficult problems or spread broadly across the benchmark.

D CktLean Implementation Details

CktLean extends Sparkle's typed IR, elaboration, and backend lowering to address the five capability gaps in §3. The extensions below follow the same five categories and provide the hardware infrastructure used by CktFormalizer. Table 8 lists the 17 tasks and the capabilities they exercise.

Preserving parameterized state. We extend the typed IR with symbolic width and parameter expressions, preserve derived parameters during elaboration, and propagate their dependencies across module boundaries. Stateful circuits can thereby retain parameter relationships through Lean elaboration, CppSim generation, and SystemVerilog extraction.

Lowering parameterized memories. We preserve symbolic memory dimensions and parameter-dependent read/write structure in the IR and lowering pipeline. This allows memory declarations, accesses, and generated RTL to share the same parameterization rather than specializing memory structure prematurely. Behavioral checks remain necessary for reset, initialization, and read/write interactions.

Compiling generic bit operations. We add compiler support for symbolic per-bit structures and lower them through parameter-aware generate constructs. Symbolic dimensions and local aliases are preserved during elaboration so that indexed layouts are not incorrectly collapsed into scalar or fixed-width forms.

Supporting signed arithmetic. We add signed signal and RTL helpers and extend the IR, elaborator, CppSim backend, and SystemVerilog backend to preserve signedness, extension, shifts, truncation, and parameterized widths consistently.

Representing multiple clock domains. We introduce first-class clock-domain information in the IR and associate signals and state with their corresponding domains. The elaborator and backends propagate this information through multi-domain circuits and support lowering the corresponding CDC structures to CppSim and SystemVerilog.

D.1 Additional Typed Hardware Examples

These examples complement Figure 1. Lean rejects implicit self-reference as an unknown identifier; the supported generation profile expresses feedback using explicit `Signal.reg` constructs. The

Table 8: The 17 CVDP tasks used to analyze capability gaps in the original SPARKLE toolchain. Task IDs omit the common `cvdp_copilot_` prefix. The last column identifies the capability exercised, rather than asserting an identical observed failure for every task.

Task ID	Category	Capability limitation examined
`car_parking_management_0001`	Stateful	Symbolic state widths and parameter dependencies.
`restoring_division_0001`	Stateful	Derived widths, tuple state, and parameterized iteration.
`sync_lifo_0001`	Memory	Parameterized memory dimensions and read/write lowering.
`cache_lru_0022`	Memory	Register-file reset and conditional write behavior.
`hamming_code_tx_and_rx_0009`	Bit operations	Symbolic indexed bit construction and parity reduction.
`hamming_code_tx_and_rx_0011`	Bit operations	Symbolic indexed bit construction and parity reduction.
`16qam_mapper_0001`	Signed arithmetic	Signed/DSP representation and lowering.
`16qam_mapper_0006`	Signed arithmetic	Signed/DSP representation and lowering.
`comparator_0001`	Signed arithmetic	Signed comparison semantics.
`complex_multiplier_0001`	Signed arithmetic	Signed products and intermediate-width handling.
`configurable_digital_low_pass_filter_0001`	Signed arithmetic	Signed filtering arithmetic and width handling.
`configurable_digital_low_pass_filter_0014`	Signed arithmetic	Signed filtering arithmetic and width handling.
`sobel_filter_0011`	Signed arithmetic	Signed intermediate values in filtering arithmetic.
`cont_adder_0023`	Signed arithmetic	Signed addition and width handling.
`cdc_pulse_synchronizer_0004`	Multiple clocks	Distinct clock domains and explicit pulse crossings.
`cdc_pulse_synchronizer_0013`	Multiple clocks	Distinct clock domains and explicit pulse crossings.
`fifo_async_0001`	Multiple clocks	Independent read/write clocks and FIFO crossings.

multiplexer covers every selector value, including a default branch; omitting that branch leaves the pattern match incomplete and causes a compile error.

Listing 3: Implicit self-reference and exhaustive pattern matching in Lean HDL.

```
-- Example 1: Implicit self-reference rejected by Lean
-- Feedback in the supported profile uses explicit Signal.reg
    constructs:
def bad_loop (x : Signal dom (BitVec 8)) : Signal dom (BitVec 8) :=
  let y := x + y  -- ERROR: unknown identifier 'y'
  y

-- Example 2: Exhaustive pattern matching prevents unintended latches
def mux3 (sel : Signal dom (BitVec 2))
         (a b c : Signal dom (BitVec 8)) : Signal dom (BitVec 8) :=
  Signal.match sel fun
    | 0b00 => a
    | 0b01 => b
    | 0b10 => c
    | _    => 0#8  -- must cover all cases; omitting causes compile
    error
```

```
-- Property proofs: counter initial value and increment invariant
theorem counter_starts_at_zero :
    (counter.atTime 0) = 0#8 := by rfl

theorem counter_increments (n : Nat) :
    (counter.atTime (n+1)) = (counter.atTime n) + 1#8 := by
  simp [counter, Signal.register]; rfl

-- Equivalence proof: optimized implementation = specification
theorem equiv (input : Signal dom (BitVec 3)) :
    optimized input = spec input := by
  unfold optimized spec; rfl
```

Figure 8: Formal verification in Lean HDL. Using the counter defined in Figure 1, the top theorems prove behavioral properties (initial value and increment invariant). The bottom theorem establishes equality between an optimized implementation and its specification for all input signals and time steps within the modeled synchronous semantics.

E Formal Verification and Equivalence Checking

Lean HDL supports hardware descriptions and formal proofs within the same language. Signals have a denotational semantics as functions over time (`Signal dom` $\alpha \triangleq \mathbb{N} \to \alpha$), allowing behavioral properties and implementation equivalences to be expressed as Lean theorems. Completed proofs are checked by the Lean kernel. We complement these source-level proofs with RTL equivalence checking and simulation, each providing evidence at a different abstraction level.

Property and Equivalence Proofs. Designers or the LLM agent can state behavioral properties directly over signal definitions, such as a counter's initial value and increment invariant (Figure 8, top). Given a specification `spec` and an implementation `opt`, the agent can also attempt to prove $\forall\, input.\ \texttt{opt}(input) = \texttt{spec}(input)$ as a Lean theorem (Figure 8, bottom).

When formal verification is enabled during architecture exploration or PPA optimization (§4.2), the agent uses a ReAct-style loop to formalize selected properties from the natural-language specification and iteratively prove that the implementation satisfies them. The persistent Lean REPL serves as an interactive interface for stepwise proof construction: the agent submits a proof step, inspects the resulting goals and diagnostics, and uses this feedback to select or revise the next step. This interaction connects the agent's reasoning to the evolving Lean proof state throughout the proof attempt. A completed, kernel-accepted proof establishes the stated source-level property or equivalence, but does not by itself establish that the formal specification faithfully captures the natural-language request. An unfinished proof provides no such certificate, but does not by itself demonstrate a functional defect.

For the signal-equality theorem shown in Figure 8, a completed proof establishes equality for all input signals and all time steps within the modeled synchronous semantics. This guarantee extends beyond the finite traces exercised by simulation, while remaining specific to the formal specification and signal model.

RTL Equivalence Checking. To complement source-level proofs, we use Yosys equivalence checking to compare the generated SystemVerilog against the VerilogEval reference implementation. The checking procedure combines SAT-based combinational reasoning (`equiv_simple`) with sequential induction (`equiv_induct`). Successful checks establish RTL equivalence under the procedure's modeling assumptions, independently of whether a Lean proof is available.

Proof automation remains design-dependent at both levels. Lean tactics such as `simp` and `decide` can discharge suitable obligations, while more complex sequential designs may require additional lemmas or induction arguments that the agent does not always construct successfully. Similarly, Yosys checks may remain inconclusive for designs beyond the capabilities of the configured reduction and induction procedures, including deeply pipelined or memory-heavy designs. These limitations

concern the ability to complete verification, rather than weakening the guarantee of a successfully completed proof.

Table 9: Evidence provided by the checking mechanisms used in the workflow. Proof or equivalence guarantees apply only when the corresponding check completes successfully.

Mechanism	Evidence established	Scope and limitations
Lean type checking	Supported structural invariants, including bit-width consistency and exhaustive case coverage	Does not by itself establish functional correctness
Lean kernel-checked proofs	Stated behavioral properties or source-level equivalence	Applies within the formal signal model; proof automation is design-dependent
Yosys equivalence checking	Equivalence between emitted RTL and the benchmark reference under the checking assumptions	Limited by supported constructs and the configured proof procedures
RTL simulation	Agreement with expected behavior on exercised traces, including supported reset and memory scenarios	Finite test coverage does not establish universal equivalence

Guarantees and Verification Boundaries. Table 9 separates the evidence provided by each checking mechanism from its scope and limitations. Lean type checking establishes supported structural invariants, such as bit-width consistency and exhaustive case coverage. Completed Lean theorems establish the source-level functional properties they state. Yosys checks compare the emitted RTL with the benchmark reference, while simulation assesses behavior on finite testbench traces.

A central verification boundary lies between the Lean source semantics and the emitted SystemVerilog. Lean proofs concern source-level definitions, whereas Yosys checks concern the emitted RTL and a separate reference implementation. Neither directly establishes that the Lean-to-SystemVerilog translation preserves the source semantics. The current pipeline therefore does not provide an end-to-end proof of compiler correctness.

Two further categories remain outside the current formal verification scope. First, the signal model represents behavior within a single synchronous clock domain. Multi-clock interactions and asynchronous reset behavior lack a formal semantics in the current framework; verifying these behaviors requires an extended semantic model or complementary analyses, such as simulation and static clock-domain-crossing checks. Second, the current checks do not establish timing-aware or gate-level equivalence. Post-synthesis transformations, cell timing, and physical parasitics require additional verification beyond the source-level proofs and RTL checks used here.

F Architecture Exploration Data

Table 10 expands the abbreviated design labels used in Table 4. These descriptions explain the labels; establishing the exact benchmark task and variant for each row requires the original per-problem architecture-exploration records. Table 11 provides the absolute synthesis areas, design types, and reduction percentages for the six examples in Table 4. The examples are drawn from simulation-passing candidates and do not represent all problems in the architecture exploration experiment.

G Mounted In-Context Idiom Files

This section lists the 13 Lean files mounted for CktFormalizer as style references (Appendix B.1). Each file begins with `import Sparkle`, `import Sparkle.Compiler.Elab`, and `open Sparkle.Core.{Domain,Signal}`. Gold Verilog from benchmark comments is omitted.

Table 10: Meaning of the design labels used in Table 4. Descriptions identify circuit categories; exact benchmark task IDs are not available in the archived summary.

Table label	Circuit type	Functional interpretation
FSM (Lemmings)	Sequential control	Finite-state controller for a Lemmings task. The summary does not identify which Lemmings variant was evaluated.
9-to-1 Mux	Combinational	Multiplexer selecting one of nine data inputs for its output.
Add/Sub Unit	Arithmetic	Datapath supporting addition and subtraction.
FSM Shift Reg.	Sequential control	Finite-state control associated with shift-register operation.
Case-z Logic	Combinational	Selection logic expressed using wildcard case matching (`casez`).
Comb. Logic	Combinational	Logic whose outputs depend on current inputs; the label does not specify the particular Boolean function.

Table 11: Architecture exploration data for the six simulation-passing examples in Table 4. Initial and candidate areas are in μm^2; the final three columns report reductions.

Design	Type	Init. area	Cand. area	Area	Cell	Power
FSM (Lemmings)	Seq.	198.9	128.9	**35.2%**	35.0%	30.3%
9-to-1 Mux	Comb.	1,818.0	1,321.3	**27.3%**	45.9%	14.5%
Add/Sub Unit	Arith.	483.0	399.1	**17.4%**	26.2%	8.9%
FSM Shift Reg.	Seq.	131.4	116.4	**11.4%**	0.0%	7.5%
Case-z Logic	Comb.	83.8	75.1	**10.4%**	0.0%	2.0%
Comb. Logic	Comb.	226.5	203.9	**9.9%**	14.8%	6.4%

Listing 4: `Prob005_notgate.lean`

```
def prob005_notgate {dom : DomainConfig}
    (input : Signal dom (BitVec 1)) : Signal dom (BitVec 1) :=
  ~~~input
#synthesizeVerilog prob005_notgate
```

Listing 5: `Prob022_mux2to1.lean`

```
def prob022_mux2to1 {dom : DomainConfig}
    (a b : Signal dom (BitVec 1)) (sel : Signal dom Bool)
    : Signal dom (BitVec 1) :=
  Signal.mux sel b a
#synthesizeVerilog prob022_mux2to1
```

Listing 6: `Prob024_hadd.lean`

```
def prob024_hadd {dom : DomainConfig}
    (a b : Signal dom (BitVec 1))
    : Signal dom (BitVec 1 × BitVec 1) :=
  let sum  := a ^^^ b
  let cout := a &&& b
  bundle2 sum cout
#synthesizeVerilog prob024_hadd
```

Listing 7: `Prob027_fadd.lean`

```
def prob027_fadd {dom : DomainConfig}
    (a b cin : Signal dom (BitVec 1))
    : Signal dom (BitVec 1 × BitVec 1) :=
  let sum  := a ^^^ b ^^^ cin
  let cout := (a &&& b) ||| (a &&& cin) ||| (b &&& cin)
  bundle2 sum cout
```

```
#synthesizeVerilog prob027_fadd
```

Listing 8: `Prob031_dff.lean`

```
def prob031_dff {dom : DomainConfig}
    (d : Signal dom (BitVec 1)) : Signal dom (BitVec 1) :=
  Signal.register 0#1 d
#synthesizeVerilog prob031_dff
```

Listing 9: `Prob035_count1to10.lean`

```
def prob035_count1to10 {dom : DomainConfig}
    (reset : Signal dom Bool) : Signal dom (BitVec 4) :=
  Signal.loop fun (q : Signal dom (BitVec 4)) =>
    let atMax := q === 10#4
    let shouldReset := reset ||| atMax
    let nextVal := Signal.mux shouldReset (Signal.pure 1#4) (q + 1#4)
    Signal.register 1#4 nextVal
#synthesizeVerilog prob035_count1to10
```

Listing 10: `Prob054_edgedetect.lean`

```
def prob054_edgedetect {dom : DomainConfig}
    (input : Signal dom (BitVec 8)) : Signal dom (BitVec 8) :=
  let d_last := Signal.register 0#8 input
  let edges := input &&& (~~~d_last)
  Signal.register 0#8 edges
#synthesizeVerilog prob054_edgedetect
```

Listing 11: `Prob086_lfsr5.lean`

```
def prob086_lfsr5 {dom : DomainConfig}
    (reset : Signal dom Bool) : Signal dom (BitVec 5) :=
  Signal.loop fun (q : Signal dom (BitVec 5)) =>
    let shifted := q >>> 1#5
    let lsb := q &&& (1#5 : BitVec 5)
    let fb := lsb === (1#5 : BitVec 5)
    let q_next := Signal.mux fb (shifted ^^^ (20#5 : BitVec 5))
    shifted
    let nextVal := Signal.mux reset (Signal.pure 1#5) q_next
    Signal.register 1#5 nextVal
#synthesizeVerilog prob086_lfsr5
```

Listing 12: `Prob091_fsm_logic.lean`

```
def prob091_fsm_logic {dom : DomainConfig}
    (y : Signal dom (BitVec 6)) (w : Signal dom Bool)
    : Signal dom (BitVec 1 × BitVec 1) :=
  let y0 := Signal.map (fun x => x.getLsb 0) y
  let y1 := Signal.map (fun x => x.getLsb 1) y
  let y2 := Signal.map (fun x => x.getLsb 2) y
  let y4 := Signal.map (fun x => x.getLsb 4) y
  let y5 := Signal.map (fun x => x.getLsb 5) y
  let Y1_bool := y0 &&& w
  let Y1 := Signal.mux Y1_bool (Signal.pure 1#1) (Signal.pure 0#1)
  let states_to_D := y1 ||| y2 ||| y4 ||| y5
  let Y3_bool := states_to_D &&& (~~~w)
  let Y3 := Signal.mux Y3_bool (Signal.pure 1#1) (Signal.pure 0#1)
  bundle2 Y1 Y3
#synthesizeVerilog prob091_fsm_logic
```

Listing 13: `Prob109_fsm1.lean`

```
private abbrev stA : BitVec 1 := 0#1
private abbrev stB : BitVec 1 := 1#1
def prob109_fsm1 {dom : DomainConfig}
    (areset : Signal dom Bool) (inp : Signal dom Bool)
    : Signal dom (BitVec 1) :=
  Signal.loop fun (state : Signal dom (BitVec 1)) =>
    let stateFlipped := ~~~state
    let nextState := Signal.mux inp state stateFlipped
    let nextWithReset := Signal.mux areset (Signal.pure stB) nextState
    Signal.register stB nextWithReset
#synthesizeVerilog prob109_fsm1
```

Listing 14: `Prob127_lemmings1.lean`

```
private abbrev stWL : BitVec 1 := 0#1
private abbrev stWR : BitVec 1 := 1#1
def prob127_lemmings1 {dom : DomainConfig}
    (areset : Signal dom Bool)
    (bump_left bump_right : Signal dom Bool)
    : Signal dom (BitVec 1 × BitVec 1) :=
  let state : Signal dom (BitVec 1) :=
    Signal.loop fun (state : Signal dom (BitVec 1)) =>
      let isWL := state === (Signal.pure stWL)
      let nextFromWL := Signal.mux bump_left (Signal.pure stWR) (
    Signal.pure stWL)
      let nextFromWR := Signal.mux bump_right (Signal.pure stWL) (
    Signal.pure stWR)
      let nextState := Signal.mux isWL nextFromWL nextFromWR
      let nextWithReset := Signal.mux areset (Signal.pure stWL)
    nextState
      Signal.register stWL nextWithReset
  bundle2 (~~~state) state
#synthesizeVerilog prob127_lemmings1
```

Listing 15: `RTLIdioms.lean`

```
def idiomSlicing {dom : DomainConfig}
    (x : Signal dom (BitVec 16))
    : Signal dom (BitVec 8 × BitVec 8 × BitVec 1) :=
  let lo : Signal dom (BitVec 8) := Sparkle.Library.RTL.slice x 0
  let hi : Signal dom (BitVec 8) := Sparkle.Library.RTL.slice x 8
  let nonzero := Sparkle.Library.RTL.boolToBV1 (Sparkle.Library.RTL.
    nonZero x)
  bundleAll! [hi, lo, nonzero]
#synthesizeVerilog idiomSlicing

def idiomReverse8 {dom : DomainConfig}
    (x : Signal dom (BitVec 8)) : Signal dom (BitVec 8) :=
  Sparkle.Library.RTL.reverseBits8 x
#synthesizeVerilog idiomReverse8

def idiomBitScans {dom : DomainConfig}
    (x : Signal dom (BitVec 8)) : Signal dom (BitVec 4 × BitVec 3) :=
  let count := Sparkle.Library.RTL.popCount8 x
  let firstSet := Sparkle.Library.RTL.priorityEncodeLsb8 x
  bundle2 count firstSet
#synthesizeVerilog idiomBitScans

def idiomRegFile {dom : DomainConfig}
    (wen : Signal dom Bool)
    (waddr : Signal dom (BitVec 4))
```

```
    (wdata : Signal dom (BitVec 8))
    (raddr : Signal dom (BitVec 4))
    : Signal dom (BitVec 8) :=
  Sparkle.Library.RTL.regFile1R1W waddr wdata wen raddr
#synthesizeVerilog idiomRegFile

def idiomCounterEnableReset {dom : DomainConfig}
    (rstN en : Signal dom Bool) : Signal dom (BitVec 8) :=
  Signal.loop fun q =>
    let inc := q + 1#8
    let next := Sparkle.Library.RTL.resetLow 0#8 rstN inc
    Sparkle.Library.RTL.dffe 0#8 en next
#synthesizeVerilog idiomCounterEnableReset
```

Listing 16: `TestClkParam.lean`

```
def test_clk_param {dom : DomainConfig}
    (clk : Signal dom Bool)
    (d   : Signal dom (BitVec 8))
    : Signal dom (BitVec 8) :=
  Signal.mux clk d (Signal.pure 0#8)
#synthesizeVerilog test_clk_param
```