LASSO: LEARNING AGENTIC SELF-SYNTHESIZED ORACLES FOR GOLDEN-FREE RTL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic RTL generation succeeds when a trusted testbench supplies feedback, but a specification-to-silicon flow rarely starts with one: the agent must construct its own verifier, and a lenient verifier silently approves incorrect hardware. We argue that learning in this setting should follow the structure of hardware verification. We first introduce LASSO, a golden-free agentic flow whose stages are inspired by formal-verification practice: specification-linked conditions, reconciliation of independently generated reference models through minimized distinguishing traces, vacuity checks, mutation-based adequacy qualification, and counterexample triage that separates design faults from verifier faults. With the same DeepSeek V4 backbone, LASSO improves pass@1 over direct generation and over the golden-free ChipMATE flow on all four benchmarks we evaluate; with Qwen3.8-27B on VerilogEval-v2, it closes most of the gap to golden-testbench feedback. A final-outcome reward does not directly evaluate verifier quality: when the final design is faulty, a verifier that falsely accepts it receives the same credit as one that correctly rejects it. On held-out CVDP bug-fix tasks, outcome-reward training does not improve Qwen3.8-27B (62.9% vs. 65.7% pass@1). We therefore propose Verification-Contract Policy Optimization (VCPO), in which each artifact of the flow is learned against its own verification contract—soundness and adequacy for the verifier, correct, evidence-backed verdicts for diagnosis, and counterexample elimination for the design—and compared only with alternatives for the same artifact on the same specification. On the same tasks, VCPO raises pass@1 to 71.4%, closing 40% of the gap to golden-testbench feedback, and cuts the false self-acceptance rate from 14.3% to 5.7%.

1 INTRODUCTION

Generating register-transfer-level (RTL) hardware from natural-language specifications requires more than producing syntactically valid code. An implementation must preserve the specified behavior across input combinations and sequences of clock cycles, including reset, state transitions, and boundary conditions. Agentic systems address this by coupling generation with simulation and iterative repair: VerilogCoder combines planning with waveform-guided debugging, MAGE coordinates specialized agents, VeriTrace and a structured human-like agentic flow expand the debugging action space through Agentic Temporal Exploration, and HORIZON organizes hardware design as repository-level code evolution (Ho et al., 2025; Zhao et al., 2025; Liu & Yu, 2026; Liu et al., 2026b; Yu et al., 2026a). These systems show that, **given trustworthy feedback, agents can repair designs to near-perfect benchmark accuracy.** The feedback, however, is typically a golden testbench supplied with the benchmark.

A real design flow does not begin with a golden testbench. Verification artifacts are developed alongside the implementation from the same specification (Bergeron, 2003; Wile et al., 2005), account for a large share of project effort, and are themselves fallible (Foster, 2024). ChipMATE addresses this setting by cross-simulating independently generated Verilog and Python models on random stimuli (Yu et al., 2026b). Agreement, however, is conditional on both models and on the exercised inputs: two models can share a misunderstanding, random stimuli can miss the distinguishing behavior, and a mismatch does not say which side is wrong. Timing introduces additional failure modes. A checker that samples one cycle early produces a counterexample against a correct design, and a

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.

checker that never tests a corner case involving reset priority accepts an incorrect one. *A golden-free agent is only as reliable as the verifier it builds for itself.*

Hardware verification has established techniques for these problems: vacuity detection, mutation adequacy, miter-based comparison with minimized counterexamples, and spurious-counterexample analysis. LASSO draws on these ideas, but not their guarantees, to design an agentic flow (Section 2). With a fixed backbone, the resulting self-built feedback outperforms prior golden-free agentic feedback (Section 4.2) and approaches golden-testbench feedback (Section 4.4).

The flow then raises a learning question that does not arise with a golden testbench: *how should an agent learn to construct and use its own verifier?* Golden labels are available when training on benchmark problems, so the question is how to turn them into a learning signal for a flow that produces many artifacts. Rewarding each rollout with the golden pass/fail outcome does not directly evaluate verifier quality. Many calls in a LASSO rollout construct verification artifacts rather than RTL, yet an outcome reward is invariant to what the verifier did: when the design fails, a verifier that falsely accepted it and one that correctly rejected it receive the same zero (Appendix B). In our experiments, outcome-reward training of Qwen3.8-27B does not improve the self-verified flow; it reduces held-out CVDP bug-fix pass@1 from 65.7% to 62.9%.

We therefore organize learning around the stages of the verification flow. Each stage of LASSO produces an artifact whose correctness is defined by a verification contract: a checker should be sound and non-vacuous and should detect relevant faults; a diagnosis should report a correct verdict backed by simulation evidence; a design revision should eliminate genuine counterexamples. Verification-Contract Policy Optimization (VCPO, Section 3) uses golden labels to grade these contracts rather than only the final outcome, and compares each artifact only against alternatives for the same artifact on the same specification. At inference, the learned policy runs the unchanged golden-free flow. Related work is discussed in Appendix A.

Contributions.

- **LASSO**, a golden-free agentic RTL flow whose stages are inspired by formal verification: specification-linked conditions, reference reconciliation, mutation qualification, and counterexample triage. With the same backbone, it outperforms direct generation and ChipMATE on four benchmarks.
- **VCPO**, a learning method that combines artifact-specific rewards with contract-conditioned group-relative advantages. On held-out bug-fix tasks, it raises pass@1 from 65.7% to 71.4%, whereas outcome reward lowers it to 62.9%.
- **An analysis of self-built feedback.** Mutation qualification lowers both false acceptance and false rejection, but a qualified checker still accepts a faulty design on one in five CVDP cid003 tasks. This residual false acceptance is the error that VCPO reduces.

2 LASSO: A Verification-Inspired Golden-Free Flow

2.1 Problem Setting and Overview

Given a natural-language specification s and a module interface I, the goal is an RTL implementation D that satisfies s. *Golden-free* means that generation may use compilers and simulators but not the benchmark's reference implementation or golden testbench; an independent evaluator scores the returned design only after generation.

Figure 1 shows the two sides of LASSO. The testbench (TB) side plans specification-linked checks (Section 2.2), builds and reconciles reference models (Section 2.3), builds a testbench (Section 2.4), and qualifies it before deployment (Section 2.5). The RTL side generates and simulates D against the deployed testbench; failures are triaged, and genuine design bugs are debugged by waveform exploration (Section 2.6). The two sides share only the plan: TB-side agents never see D, and RTL-side agents cannot read or modify the references or the testbench. Each stage runs within a configurable budget and can be disabled for ablation.

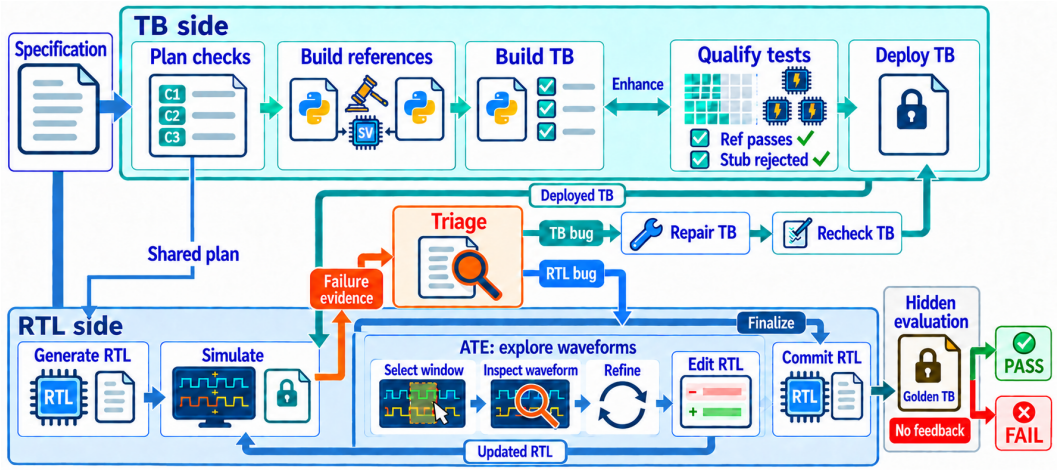


Figure 1: Overview of LASSO. The TB side builds and qualifies a checker; the RTL side refines the design using triaged simulation feedback. The hidden golden testbench scores only the committed RTL.

2.2 Specification-Linked Planning and Stimulus

Verification practice states requirements before writing checks (Wile et al., 2005), and language models have been used to translate requirements into temporal-logic properties and assertions (Appendix A). Following this practice, the planning agent (*Plan checks* in Figure 1) converts s into

$$\mathcal{P} = (I, \chi, \mathcal{C}, \mathcal{S}, \mathcal{A}, \Gamma), \qquad \mathcal{C} = \{(c_j, q_j, t_j)\}_{j=1}^{N_C}, \tag{1}$$

where χ specifies clock and reset semantics, $\mathcal{C}$ the behavioral conditions, $\mathcal{S}$ directed scenarios, $\mathcal{A}$ recorded ambiguities, and Γ optional input constraints. Each condition has a testable statement c_j, a supporting quotation q_j from s, and a type t_j such as reset, transition, latency, or invariant; each scenario lists input sequences and the conditions they target. Unlike property translation, the conditions remain natural-language statements checked by simulation, not formulas checked by a model checker; the quotations make each check traceable to a requirement.

A stimulus builder expands the scenarios, optionally with seeded constrained-random sequences, into a stimulus $X = (x_0, \ldots, x_{H-1})$. For sequential designs, observations distinguish pre-edge from settled post-edge outputs, so that disagreements between references are not confused with different sampling conventions.

2.3 Reference Reconciliation

Equivalence checking compares two designs through a miter whose distinguishing input is a counterexample (Brand, 1993), and a counterexample is most informative once minimized (Zeller & Hildebrandt, 2002; Ravi & Somenzi, 2004). We apply this idea to reference models under finite simulation (*Build references*). Reference agents generate K Python models $R_1, \ldots, R_K$ in separate contexts, with different prompting styles (literal transcription and state-machine modeling) to encourage different interpretations. Let $R_a(X)_{t,\phi,o}$ be the output of R_a on port o at step t and observation phase ϕ, with $\perp$ denoting an unspecified value. The disagreement set of R_a and R_b is

$$\Delta_{ab}(X) = \{(t, \phi, o) : R_a(X)_{t,\phi,o} \neq R_b(X)_{t,\phi,o}, \ R_a(X)_{t,\phi,o} \neq \perp, \ R_b(X)_{t,\phi,o} \neq \perp\}. \tag{2}$$

A nonempty $\Delta_{ab}(X)$ yields a distinguishing trace, which is truncated after the first discrepancy and greedily shortened while the disagreement is preserved.

A judge receives s, $\mathcal{P}$, the reduced trace, and the two output sequences under randomly permuted labels, but not the programs. It either prefers one behavior, citing a quotation from s, or concludes that the specification does not resolve the disagreement. The losing model is regenerated with the ruling; unresolved differences are masked from later comparisons. The surviving model is the reference $R^\star$. In the example of Figure 2a, two references for a decade counter disagree only on the

terminal-count output `tc` when `q`=9 and `en`=0; the judge selects R_1 by quoting the clause that conditions `tc` on `en`.

A separate agent then writes a SystemVerilog reference V from s, $\mathcal{P}$, and $R^\star$. It is accepted if it compiles, matches I, and satisfies

$$V(X)_{t,\phi,o} = R^\star(X)_{t,\phi,o} \qquad \forall\,(t,\phi,o) \in \Omega_X, \tag{3}$$

where Ω_X contains the observations of X that are neither unspecified nor masked. V is the mutation target of Section 2.5 and is distinct from the candidate design D.

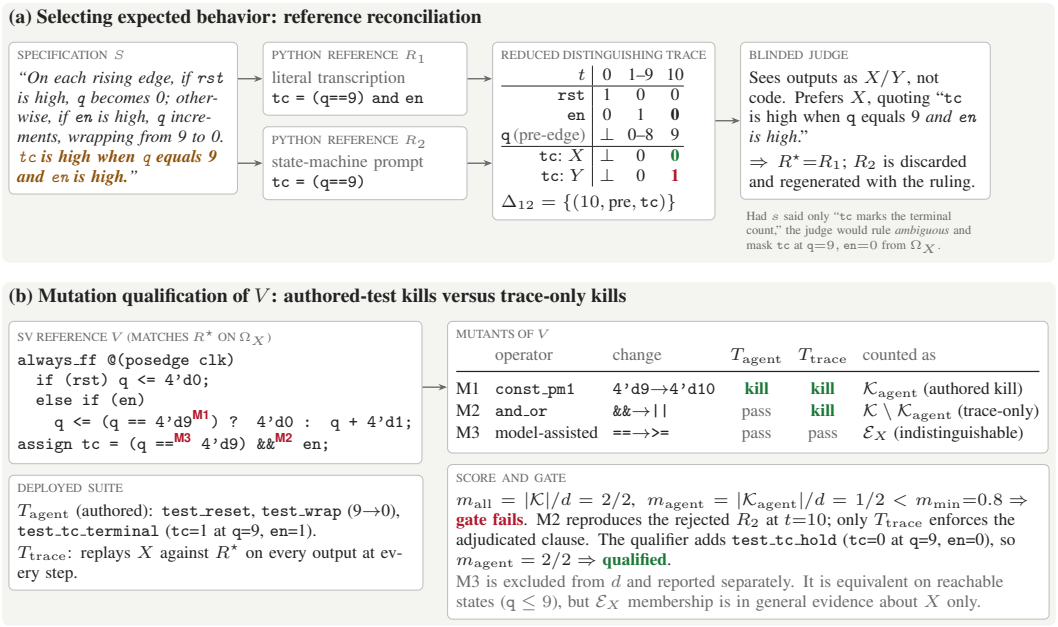


Figure 2: An example on a decade counter. (a) Reference reconciliation. (b) Mutation qualification of the SV reference V.

2.4 Testbench Construction

Vacuity detection flags properties that hold without constraining the design (Beer et al., 1997; Kupferman & Vardi, 1999); we apply an analogous check to executable tests (*Build TB*). The deployed suite has two parts: a tool-generated trace test T_{trace}, which replays X against $R^\star$, and agent-written cocotb tests (cocotb contributors) T_{agent}, which implement condition-linked checks and coverage bins. After each revision, the suite runs on V and on an interface-compatible all-zero stub. The intended outcomes are that V passes and the stub is rejected (*Ref passes*, *Stub rejected* in Figure 1); a stub pass is reported as a weakness rather than an error, since some specifications legitimately produce all-zero outputs. The tool also reports vacuous tests, which execute without recording any check; conditions whose checks never execute; and the bin coverage $c_{\text{bin}} = |\mathcal{B}_{\text{hit}}|/|\mathcal{B}|$ over declared bins $\mathcal{B}$ (one if none are declared). These checks show that the suite runs and agrees with V; qualification then tests whether it rejects faults.

2.5 Mutation Qualification and Deployment

Mutation adequacy judges a check by whether it detects faulty variants of the system (DeMillo et al., 1978; Chockler et al., 2001; Kupferman et al., 2008). LASSO generates up to N_{mut} mutants of V (*Qualify tests*) with deterministic operators on arithmetic and relational operators, constants, reset polarity, clock edges, and case arms, optionally adding model-generated faults. Mutants that fail to compile or run are discarded. Among the remaining mutants, let $\mathcal{K}$ be those rejected by at least one test, $\mathcal{K}_{\text{agent}} \subseteq \mathcal{K}$ those rejected by an authored test, $\mathcal{L}$ the scored survivors, and $\mathcal{E}_X$ the survivors that are indistinguishable from V under X. Mutants in $\mathcal{E}_X$ are excluded from scoring.

With $d = |\mathcal{K}| + |\mathcal{L}|$,

$$(m_{\text{all}}, m_{\text{agent}}) = \begin{cases} \big(|\mathcal{K}|/d,\ |\mathcal{K}_{\text{agent}}|/d\big), & d > 0, \\ (1, 1), & d = 0 \text{ and } |\mathcal{E}_X| > 0, \\ (0, 0), & \text{otherwise.} \end{cases} \tag{4}$$

The gate scores $m_\star = m_{\text{agent}}$ by default, so that kills by the trace test alone do not qualify the authored checks:

$$g = g_{\text{ref}}\, g_{\text{mut}}\, \mathbf{1}[m_\star \geq m_{\min}]\, \mathbf{1}[n_v \leq v_{\max}]\, \mathbf{1}[n_u = 0],$$
$$g_{\text{mut}} = \mathbf{1}\big[|\mathcal{K}| > 0 \ \vee\ (|\mathcal{L}| = 0 \wedge |\mathcal{E}_X| > 0)\big], \tag{5}$$

where g_{ref} indicates that V passes the suite, n_v and n_u count vacuous tests and unexecuted conditions, and $m_{\min}$ and $v_{\max}$ are thresholds; a threshold on c_{bin} can optionally be added. Figure 2b illustrates the default score. Mutant M1 is rejected by an authored test. M2 is rejected only by T_{trace}: it reproduces the discarded behavior of R_2, which no authored test checks, so $m_{\text{agent}} = 1/2$ and the gate fails until the qualifier adds a test for the adjudicated clause. M3 is indistinguishable under X and is excluded from d.

If the gate fails, a qualifier repairs the tests and repeats the checks for a bounded number of rounds. The deployed checker is then labeled *qualified* if the gate passes, *degraded* if it fails but the deployment policy permits its use (the default), *qualification skipped* if no V was accepted, and *unavailable* if no usable testbench exists.

2.6 Triage and RTL Refinement

In counterexample-guided abstraction refinement, a counterexample is checked for spuriousness before it is used for refinement (Clarke et al., 2000). LASSO applies an analogous step to simulation failures (*Triage*). When the deployed suite rejects D, the failure evidence is classified as a design bug, a testbench bug, or a specification ambiguity, using deterministic rules for recognizable tool errors and a triage agent otherwise. A testbench-bug ruling reopens the testbench for repair and recheck (*Repair TB*, *Recheck TB*) without exposing D; an ambiguity ruling makes the affected conditions non-blocking; unresolved cases default to a design-bug ruling. Design bugs go to RTL debugging with Agentic Temporal Exploration (Liu & Yu, 2026; Liu et al., 2026b): the agent selects signals and time windows, inspects waveforms, and edits D, without reading the verification programs. A repaired testbench is rechecked against V but not re-qualified, so a qualification label refers to the initially deployed suite.

3 Learning from Verification Contracts

3.1 Verification Contracts as Rewards

Let $x = (s, I)$ and let τ_i be a rollout that returns design D_i. During training, a golden evaluator supplies $y_i \in \{0, 1\}$ and mismatch statistics after the rollout; these are never shown to the agents. Let $v_i \in \{\text{accept}, \text{reject}\}$ be the last verdict of the deployed verifier on D_i. An outcome reward $R_i = y_i$ is constant in v_i given y_i, so it cannot distinguish a sound verifier from an unsound one (Appendix B). VCPO instead rewards each group of calls against the criterion that motivated its stage in Section 2. Calls are grouped by the contract they serve: *verifier* (planning, reference reconciliation, testbench construction, and qualification), *design* (RTL generation and repair), *diagnosis* (the debugging agent's pass/fail reports), *orchestration*, and *fallback verification*. Golden labels grade contracts during training only. Coefficient values are listed in Appendix C.

Verifier contract: deployable, adequate, sound, complete. Verifier calls receive

$$R_i^{\text{ver}} = \lambda_d \delta_i + \lambda_g g_i + \lambda_a a_i - \lambda_{\text{FA}} f_i^{\text{FA}} - \lambda_{\text{FR}} f_i^{\text{FR}}, \tag{6}$$

where δ_i indicates that a checker was deployed, g_i that it passed the gate (Eq. 5), $a_i = \mathbf{1}[v_i = \text{accept} \Leftrightarrow y_i = 1]$ that its verdict agrees with the golden result, $f_i^{\text{FA}} = \mathbf{1}[v_i = \text{accept},\ y_i = 0]$ marks false acceptance, and $f_i^{\text{FR}} = \mathbf{1}[v_i = \text{reject},\ y_i = 1]$ false rejection; a_i, f_i^{FA}, and f_i^{FR} are zero when no checker is deployed. Unlike the outcome reward, equation 6 distinguishes correct verdicts

from false acceptance and from false rejection; true acceptance and true rejection both receive the agreement credit λ_a and are not distinguished from each other. We set $\lambda_{\mathrm{FA}} > \lambda_{\mathrm{FR}}$ to prioritize soundness: an escaped design bug is costlier than a false alarm, which triage or repair can still resolve.

Diagnosis contract: report a supported verdict. Diagnosis calls report whether the current design passes and whether that report is supported by execution evidence. Let $\hat{y}_i \in \{0, 1\}$ be the diagnosed pass/fail verdict and ν_i indicate that it is backed by execution evidence. Then

$$R_i^{\mathrm{diag}} = \begin{cases} \beta_+, & \nu_i = 1,\ \hat{y}_i = y_i, \\ -\beta_{\mathrm{FA}}, & \nu_i = 1,\ \hat{y}_i = 1,\ y_i = 0, \\ \beta_{\mathrm{rep}}, & \nu_i = 1,\ \hat{y}_i = 0,\ y_i = 1, \\ -\beta_u, & \nu_i = 0, \\ 0, & \text{no verdict reported}, \end{cases} \tag{7}$$

with $0 < \beta_{\mathrm{rep}} < \beta_+$. The third case is not necessarily an error: y_i grades the final design, whereas a diagnosis concerns the revision simulated at that time, which may have been faulty and later repaired. Because rollout-level labels cannot distinguish a correct report on an earlier revision that was later repaired from a false rejection, this case receives partial rather than full credit. The second case is unambiguous, since a diagnosed pass ends refinement, so the diagnosed design is the final one; it is penalized.

Design contract: eliminate genuine counterexamples. RTL generation and repair share a design reward. Let $r_i^{\mathrm{c}} \in [0, 1]$ be dense correctness credit, equal to one for a passing design and otherwise computed from compilation status and mismatch fraction; $\mathbf{p}_i$ bounded counts of simulations that fail fewer checks than their predecessor after an edit or a waveform inspection; u_i inspection of existing RTL before modification; and $\mathbf{e}_i \in [0, 1]^2$ the tool-error and inactive-turn rates. Then

$$\widetilde{R}_i^{\mathrm{rtl}} = r_i^{\mathrm{c}} + \eta(y_i)\, \boldsymbol{\lambda}_p^\top \mathbf{p}_i + \lambda_u u_i - \boldsymbol{\lambda}_e^\top \mathbf{e}_i, \qquad R_i^{\mathrm{rtl}} = \begin{cases} \widetilde{R}_i^{\mathrm{rtl}}, & y_i = 1, \\ \min\{\widetilde{R}_i^{\mathrm{rtl}}, \kappa_{\mathrm{fail}}\}, & y_i = 0, \end{cases} \tag{8}$$

where $\eta(0) < \eta(1) = 1$ discounts progress in unsuccessful rollouts. Since every passing rollout receives at least $1 - \|\boldsymbol{\lambda}_e\|_1$, choosing $\kappa_{\mathrm{fail}} < 1 - \|\boldsymbol{\lambda}_e\|_1$ ensures that shaping never ranks a failing design above a passing one.

Orchestration and fallback. The orchestrator receives the dense correctness credit r_i^{c}, minus λ_o times its tool-error rate and λ_{uv} for terminating on an unverified claim. Calls that build a fallback testbench receive μ_x for executing it, and $+\mu_\pm$ or $-\mu_\pm$ for an evidence-backed pass of a design that the golden evaluator accepts or rejects.

3.2 Contract-Conditioned Group-Relative Optimization

A shared policy π_θ executes all stages through their prompts and tool interfaces. Write a rollout as $\tau_i = \{(h_{i,t}, a_{i,t}, k_{i,t})\}_{t=1}^{T_i}$, where $h_{i,t}$ is the context of the active stage, $a_{i,t}$ its response, and $k_{i,t}$ the contract it serves. The calls serving contract k form

$$\tau_i^k = [(h_{i,t}, a_{i,t})]_{t \in \mathcal{T}_i^k}, \qquad \mathcal{T}_i^k = \{t : k_{i,t} = k\}, \tag{9}$$

which keeps the original contexts and order, including information received from other stages. The agents still execute the same coupled flow; only credit assignment changes.

For each training task, we sample G complete rollouts under π_{old}. Following GRPO (Shao et al., 2024), advantages are estimated from sampled rewards without a critic, but the comparison group is conditioned on both task and contract: $\mathcal{G}_{x,k}$ contains the rollouts in which contract k occurs, and

$$\widehat{A}_i^k = \frac{R_i^k - \mu_{x,k}}{\sigma_{x,k} + \varepsilon}, \qquad \mu_{x,k} = \frac{1}{|\mathcal{G}_{x,k}|} \sum_{j \in \mathcal{G}_{x,k}} R_j^k, \tag{10}$$

with $\sigma_{x,k}$ the within-group standard deviation; groups with $\sigma_{x,k} = 0$ are dropped. Every response token in τ_i^k receives $\widehat{A}_i^k$. A verifier is thus compared with other verifiers for the same specification, and a design revision with other revisions. As in assume–guarantee reasoning (Pnueli, 1985;

Henzinger et al., 1998), each component is judged against its own guarantee given the context it received, so one rollout can reinforce a sound verifier while discouraging a failed repair.

For token j of response $a_{i,t}$, let $\rho_{i,t,j}(\theta) = \pi_\theta(a_{i,t,j} \mid h_{i,t}, a_{i,t,<j})/\pi_{\text{old}}(a_{i,t,j} \mid h_{i,t}, a_{i,t,<j})$, and let $\mathcal{M}$ contain the valid response-token indices of the batch. We maximize the token-averaged dual-clipped surrogate (Schulman et al., 2017; Ye et al., 2020)

$$\mathcal{J}(\theta) = \mathbb{E}\Big[\tfrac{1}{|\mathcal{M}|}\textstyle\sum_{(i,t,j)\in\mathcal{M}} \ell\big(\rho_{i,t,j}(\theta), \widehat{A}_i^{k_{i,t}}\big)\Big], \quad \ell(\rho, A) = \begin{cases} \max\{\ell_{\text{clip}}(\rho, A), c_{\text{dc}}A\}, & A < 0, \\ \ell_{\text{clip}}(\rho, A), & A \geq 0, \end{cases} \tag{11}$$

where $\ell_{\text{clip}}(\rho, A) = \min\{\rho A, \text{clip}(\rho, 1-\epsilon_-, 1+\epsilon_+)A\}$ and $c_{\text{dc}} > 1$. Only model-generated tokens contribute; prompts and tool observations are context.

4 Experiments

4.1 Setup

Benchmarks and models. The flow comparison uses VerilogEval-v2 (Pinckney et al., 2025a), RTLLM-v2 (Lu et al., 2024; Liu et al., 2024), ChipBench-SC (Yu et al., 2026c), and CVDP cid003 (Pinckney et al., 2025b); all golden-free systems share the DeepSeek V4 backbone without training, so differences reflect the flow. For learning, we train a shared LoRA adapter (Hu et al., 2022) on Qwen3.8-27B using Veri-Sure (Liu et al., 2026a) (which includes VerilogEval-v2) and RTLLM-v2, augmented with bug-fix and completion variants derived from their golden designs, and evaluate on held-out CVDP cid016 (35 bug-fix tasks). All other Qwen3.8-27B results (Tables 3–5 and Figure 3) use the base model, since VerilogEval-v2 is part of the training data.

Metrics. All designs are scored by the withheld golden harness. *Pass@1* is the fraction of tasks whose final design passes. Verifier reliability compares the checker's last verdict on the final design with the golden result. *FA* and *FR* are the shares of all tasks whose final design the checker accepts but the golden harness rejects, and vice versa. In Table 3, *Draft* and *Final* are the mismatch counts that the golden testbench reports, summed over the benchmark, on the initial RTL draft and on the final RTL, with percentages of all tested signals in parentheses. *Reduction* is the relative decrease from Draft to Final and serves as a rough measure of how much useful feedback each checker provides. Rows marked *golden TB* replace the self-built checker with the benchmark testbench; they provide reference results obtained with golden-testbench feedback, not golden-free results.

4.2 Comparison with Golden-Free Flows

Table 1: Pass@1 (%) of direct generation and golden-free flows. *The released ChipMATE flow covers 33 of the 50 RTLLM-v2 problems; for the 17 its authors deem unsuited, we use one-shot generation with the same backbone (Appendix D). †Other backbones, listed for reference; bold marks the best DeepSeek V4 result.

Method	Model	# of Params	VerilogEval-v2	RTLLM-v2	ChipBench-SC	CVDP-cid003
Generic LLM	DeepSeek V4	1.6T	67.3	58.8	18.0	21.5
ChipMATE	DeepSeek V4	1.6T	84.6	66.0*	26.7	57.1
LASSO (Ours)	DeepSeek V4	1.6T	**90.4**	**74.0**	**36.7**	**65.4**
Generic LLM†	Claude Opus 4.6	N/A	85.9	70.0	40.0	56.4
HORIZON†	GPT Sol-5.6	N/A	95.0	N/A	N/A	62.8

With a fixed backbone, LASSO improves over direct generation and over ChipMATE on all four benchmarks (Table 1). Relative to ChipMATE, which already reconciles independent Verilog and Python models, the largest gains appear on ChipBench-SC (+10.0) and CVDP cid003 (+8.3), where the stages that ChipMATE lacks (specification-linked conditions, adjudicated reconciliation, and qualification) target shared misunderstandings and weak stimulus, not only disagreement. HORIZON (Yu et al., 2026a), run by its authors without golden feedback, reaches 95.0% on VerilogEval-v2 and 62.8% on cid003 with GPT Sol-5.6 (Appendix D).

4.3 Learning from Verification Contracts

Table 2: Learning on held-out CVDP cid016 (bug fixing) with Qwen3.8-27B. Parentheses: change from the base model (green: better; red: worse).

Training reward	Pass@1 (%)	FA (%)↓	FR (%)↓
Base model (no training)	65.7	14.3	20.0
Outcome (golden pass/fail)	62.9 (−2.9)	17.1 (+2.9)	17.1 (−2.9)
VCPO (ours)	**71.4** (+5.7)	**5.7** (−8.6)	**14.3** (−5.7)
Golden TB (reference)	*80.0*	—	—

In this evaluation, training with golden pass/fail outcomes does not improve performance: pass@1 is 62.9% versus 65.7% for the base model, and FA rises from 14.3% to 17.1% (Table 2), consistent with Appendix B. VCPO raises pass@1 to 71.4%. With the golden testbench as feedback, the same base model reaches 80.0%, so VCPO closes 40% of the gap to golden feedback but remains 8.6 points below it. The verifier also improves: FA falls from 14.3% to 5.7% and FR from 20.0% to 14.3%. The largest reduction is in false acceptance, which receives the larger penalty in the verifier contract ($\lambda_{\mathrm{FA}} > \lambda_{\mathrm{FR}}$) and persists even with qualification (Section 4.6). Because cid016 is small, each difference is only a few tasks.

4.4 Ablation: Source of Feedback

Table 3: Feedback source on VerilogEval-v2 with the Qwen3.8-27B base model.

Feedback	Pass@1 (%)	Draft	Final	Reduction
One-shot RTL (none)	78.2	—	222,706 (28.8%)	—
One-shot TB	82.7	264,744 (34.3%)	65,122 (8.4%)	75.4%
LASSO	**93.6**	722,150 (93.4%)	**6,408 (0.8%)**	**99.1%**
Golden TB (reference)	*98.7*	700,715 (90.7%)	710 (0.1%)	99.9%

To isolate the source of feedback, Table 3 keeps the Qwen3.8-27B repair loop fixed and varies only the checker: none (one-shot RTL), an unqualified self-generated testbench (one-shot TB), LASSO, and the golden testbench. A one-shot self-generated testbench adds only 4.5 points over one-shot RTL, whereas the full flow adds 15.4 points and comes within 5.1 points of golden feedback.

Figure 3 breaks these results down by category. About four-fifths of LASSO's gain over one-shot RTL comes from sequential, FSM, and waveform problems, where transitions, reset priority, and cycle timing must be inferred from prose or traces. An unqualified testbench helps on FSM (65.8% to 78.9%) and waveform (63.6% to 90.9%) problems but is worse than no feedback on sequential (75.0% to 67.5%) and arithmetic (100% to 85.7%) problems. This matches the false-rejection failure of Section 4.6: a testbench that samples at the wrong edge or misreads reset timing rejects a correct design and drives the agent to break it. LASSO's largest gain over one-shot TB is on sequential problems (90.0% vs. 67.5%), and it matches golden feedback on all waveform and arithmetic problems. Its remaining gap to golden feedback is consistent with misreadings that all references share, which reconciliation cannot expose.

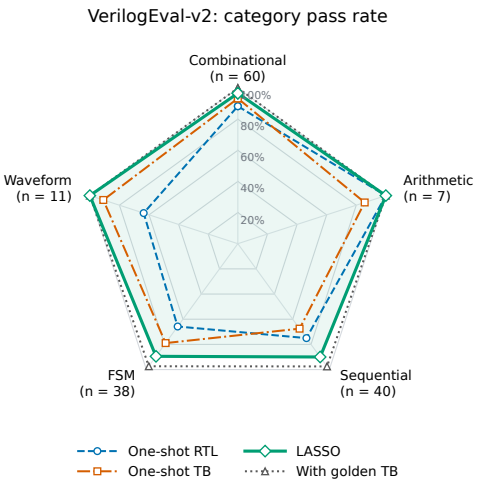


Figure 3: VerilogEval-v2 pass rate by category with the Qwen3.8-27B base model (Appendix D).

4.5 Ablation: Specification-Linked Planning

Removing the planning stage lowers pass@1 from 93.6% to 89.0% and doubles false acceptance, from 3.8% to 7.7% of tasks, while false rejection rises only from 0.6% to 1.3% (Table 4). Without a plan, stimulus and checks are no longer tied to enumerated specification conditions, so behaviors such as reset priority, boundary values, and specific state transitions go unexercised, and a faulty design that differs only there passes the checker. These results suggest that planning reduces false acceptance by helping the checker test more of the specified behavior.

Table 4: Removing specification-linked planning (Section 2.2) on VerilogEval-v2 with the Qwen3.8-27B base model.

Variant	Pass@1 (%)	FA (%)↓	FR (%)↓
LASSO	**93.6**	**3.8**	**0.6**
w/o planning	89.0	7.7	1.3

4.6 Ablation: Mutation Qualification

Table 5: Removing mutation qualification (Section 2.5) on CVDP cid003 with the Qwen3.8-27B base model; without it, the constructed testbench is deployed directly.

Variant	Pass@1 (%)	FA (%)↓	FR (%)↓
LASSO	**61.5**	**20.5**	**3.8**
w/o qualification	58.9	25.6	6.4

We run this ablation on cid003 because on VerilogEval-v2 the checker errs on too few tasks (Table 4) to resolve the effect of qualification. Disabling the gate lowers pass@1 from 61.5% to 58.9% and increases both errors: false acceptance from 20.5% to 25.6% and false rejection from 3.8% to 6.4% (Table 5). A false rejection is a spurious counterexample that pushes the agent to change a correct design, as when a testbench samples at the wrong edge or misreads reset timing; a false acceptance ends the loop on a faulty design. Qualification does not remove false acceptance, however. With the gate enabled, the checker still accepts a faulty final design on 20.5% of tasks, more than five times its false-rejection rate. Mutants are derived from the SV reference, so a checker that inherits a misunderstanding shared by all references can kill every mutant while enforcing the wrong behavior, and reconciliation cannot expose the misunderstanding because the references agree. This residual error is what the soundness term λ_{FA} of the verifier contract targets, and VCPO reduces it (Table 2).

5 Discussion and Limitations

Golden-free applies to inference. VCPO uses golden labels to grade contracts during training; grading them by internal evidence alone remains unexplored.

Inspired by, not equivalent to, formal verification. Qualification and triage rely on finite stimulus and selected mutants, so a qualified checker can still miss faults. References produced by the same model family can also share a misreading of the specification, which reconciliation cannot expose.

Breadth of evaluation. Our evidence for VCPO is still limited. It comes from a single held-out set of 35 tasks, and we do not yet ablate the individual contracts or contract-conditioned grouping. The results in Table 2 provide preliminary evidence of improvement, but larger evaluations are needed to establish its consistency and magnitude.

6 Conclusion

A golden-free RTL agent must build the verifier it depends on. LASSO builds that verifier with stages drawn from formal verification and outperforms prior golden-free flows; VCPO uses artifact-specific rewards to improve held-out bug fixing and reduce false self-acceptance in our evaluation.

AI USE STATEMENT

We used generative AI tools to provide feedback on the research methodology and experiments, assist in implementing methods, support qualitative and thematic data analysis, and help interpret results. Additionally, we used generative AI tools to polish the writing. We did not use generative AI tools to design the research methodology, develop theoretical models or conceptual frameworks, formulate mathematical claims, assist with translation, clean and reformat datasets, or generate synthetic datasets. Assisting with proofs, proposing or refining hypotheses, and providing critical ingredients for proving mathematical claims are not applicable to this work. We have reviewed all AI-assisted work, and we take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Section 2 and Appendices C and D describe the flow, the reward coefficients, and the benchmark and baseline setups; the RTL-side prompts follow Liu et al. (2026b). We will release the training data, model checkpoint, source code, and all prompts.

REFERENCES

Accellera Systems Initiative. *Universal Verification Methodology (UVM) 1.2 User's Guide*. Accellera Systems Initiative, 2015. URL https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in ACTL formulas. In *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pp. 279–290. Springer, 1997. doi: 10.1007/3-540-63166-6_28.

Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.

Daniel Brand. Verification of large synthesized designs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 534–537, 1993. doi: 10.1109/ICCAD.1993.580110.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations (ICLR)*, 2023a.

Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu Fan. NL2TL: Transforming natural languages to temporal logics using large language models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 15880–15903. Association for Computational Linguistics, 2023b. doi: 10.18653/v1/2023.emnlp-main.985.

Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for temporal logic model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pp. 528–542. Springer, 2001. doi: 10.1007/3-540-45319-9_36.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pp. 154–169. Springer, 2000. doi: 10.1007/10722167_15.

cocotb contributors. cocotb: Python verification framework. Project documentation. URL https://www.cocotb.org/. Accessed September 26, 2026.

Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *Computer Aided Verification (CAV)*, volume 13965 of *LNCS*, pp. 383–396. Springer, 2023. doi: 10.1007/978-3-031-37703-7_18.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. doi: 10.1109/C-M.1978.218136.

Chenhui Deng, Zhongzhi Yu, Guan-Ting Liu, Nathaniel Pinckney, and Haoxing Ren. ACE-RTL: When agentic context evolution meets RTL-specialized LLMs. *arXiv preprint arXiv:2602.10218*, 2026.

Lishui Fan, Mouxiang Chen, Tingwei Zhu, Kui Liu, Xin Xia, Shanping Li, and Zhongxin Liu. ZeroCoder: Can LLMs improve code generation without ground-truth supervision? *arXiv preprint arXiv:2604.07864*, 2026.

Harry D. Foster. 2024 Siemens EDA and Wilson Research Group IC/ASIC functional verification trend report. White paper, Siemens EDA, 2024. URL https://resources.sw.siemens.com/en-US/white-paper-2024-wilson-research-group-ic-asic-functional-verification-trend-report/.

Francesco Fuggitti and Tathagata Chakraborti. NL2LTL – a python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 37, pp. 16428–16430, 2023. doi: 10.1609/aaai.v37i13.27068.

Christopher Hahn, Frederik Schmitt, Julia J Tillman, Niklas Metzger, Julian Siber, and Bernd Finkbeiner. Formal specifications from natural language. *arXiv preprint arXiv:2206.01962*, 2022.

Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Computer Aided Verification (CAV)*, volume 1427 of *LNCS*, pp. 440–451. Springer, 1998. doi: 10.1007/BFb0028765.

Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. VerilogCoder: Autonomous Verilog coding agents with graph-based planning and abstract syntax tree (AST)-based waveform tracing tool. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 39, pp. 300–307, 2025. doi: 10.1609/aaai.v39i1.32007.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. ReVeal: Self-evolving code agents via reliable self-verification. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=q56ZI1Co43.

Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. (Security) assertions by large language models. *IEEE Transactions on Information Forensics and Security*, 19:4374–4389, 2024. doi: 10.1109/TIFS.2024.3372809.

Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, volume 1703 of *LNCS*, pp. 82–98. Springer, 1999. doi: 10.1007/3-540-48153-2_8.

Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 1–9, 2008. doi: 10.1109/FMCAD.2008.ECP.29.

Jiale Liu, Taiyu Zhou, and Tianqi Jiang. Veri-Sure: A Contract-Aware Multi-Agent Framework with Temporal Tracing and Formal Verification for Correct RTL Code Generation. *arXiv preprint arXiv:2601.19747*, 2026a. URL https://arxiv.org/abs/2601.19747.

Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. OpenLLM-RTL: Open dataset and benchmark for llm-aided design rtl generation (invited). In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2024. URL https://zhiyaoxie.com/files/ICCAD24_OpenLLM.pdf.

Yu-Tung Liu and Cunxi Yu. VeriTrace: Human-like temporal exploration completes agentic action space. In *Proceedings of the IEEE International Conference on LLM-Aided Design (ICLAD)*, 2026.

Yu-Tung Liu, Zhan Song, Chenhui Deng, Chia-Tung Ho, and Cunxi Yu. Structured human-like agentic flow for RTL design. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2026b.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. RTLLM: An open-source benchmark for design rtl generation with large language model. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 722–727, 2024. URL https://github.com/hkust-zhiyao/RTLLM.

Daniel Mendoza, Christopher Hahn, and Caroline Trippel. Translating natural language to temporal logics with large language models and model checkers. In Nina Narodytska and Philipp Rümmer (eds.), *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pp. 119–129. TU Wien Academic Press, 2024. doi: 10.34727/2024/isbn.978-3-85448-065-5_17.

Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation. *ACM Transactions on Design Automation of Electronic Systems*, 30(6):1–20, 2025a. doi: 10.1145/3718088.

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*, 2025b. URL https://arxiv.org/abs/2506.14074.

Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt (ed.), *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series F*, pp. 123–144. Springer, 1985. doi: 10.1007/978-3-642-82453-1_5.

Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. CorrectBench: Automatic testbench generation with functional self-correction using LLMs for HDL design. In *Proceedings of the Design, Automation & Test in Europe Conference (DATE)*, pp. 1–7, 2025. doi: 10.23919/DATE64628.2025.10992873.

Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pp. 31–45. Springer, 2004. doi: 10.1007/978-3-540-24730-2_3.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017. URL https://arxiv.org/abs/1707.06347.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Fu Teng, Miao Pan, Xuhong Zhang, Zhezhi He, Yiyao Yang, Xinyi Chai, Mengnan Qi, Liqiang Lu, and Jianwei Yin. VeriRL: Boosting the LLM-based Verilog code generation via reinforcement learning. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, 2025. doi: 10.1109/ICCAD66269.2025.11241003.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving LLM coder and unit tester via reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, pp. 143630–143664, 2025. doi: 10.52202/085713-4809.

Yiting Wang, Guoheng Sun, Wanghao Ye, Gang Qu, and Ang Li. VeriReason: Reinforcement learning with testbench feedback for reasoning-enhanced Verilog generation. In *Proceedings of the Great Lakes Symposium on VLSI 2026*, pp. 948–953, 2026.

Bruce Wile, John C. Goss, and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle*. Systems on Silicon. Morgan Kaufmann, 2005.

Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 614–621. ACM, 2025. doi: 10.1145/3658617.3697756.

Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, Qiaobo Chen, Yinyuting Yin, Hao Zhang, Tengfei Shi, Liang Wang, Qiang Fu, Wei Yang, and Lanxiao Huang. Mastering Complex Control in MOBA Games with Deep Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34 (04):6672–6679, 2020. doi: 10.1609/aaai.v34i04.6144.

Cunxi Yu, Chenhui Deng, Nathaniel Pinckney, and Brucek Khailany. Agentic hardware design as repository-level code evolution. *arXiv preprint arXiv:2606.28279*, 2026a.

Zhongkai Yu, Yichen Lin, Chenyang Zhou, Yuwei Zhang, Kun Zhou, Junxia Cui, Haotian Ye, Zhengding Hu, Zaifeng Pan, Ruiyi Wang, Yujie Zhao, Hejia Zhang, Jingbo Shang, Jishen Zhao, and Yufei Ding. ChipMATE: Multi-agent training via reinforcement learning for enhanced RTL generation. *arXiv preprint arXiv:2605.12857*, 2026b.

Zhongkai Yu, Chenyang Zhou, Yichen Lin, Hejia Zhang, Haotian Ye, Junxia Cui, Zaifeng Pan, Jishen Zhao, and Yufei Ding. ChipBench: A Next-Step Benchmark for Evaluating LLM Performance in AI-Aided Chip Design. *arXiv preprint arXiv:2601.21448*, 2026c. URL `https://arxiv.org/abs/2601.21448`.

Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. doi: 10.1109/32.988498.

Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. MAGE: A multi-agent engine for automated RTL code generation. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC)*, pp. 1–7, 2025. doi: 10.1109/DAC63849.2025.11133191.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, Pengwei Jin, Shuyao Cheng, Shengwen Liang, Xishan Zhang, Rui Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. QiMeng-CodeV-R1: Reasoning-enhanced Verilog generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, 2025.

A Related Work

Agentic and learned RTL generation. Agentic RTL systems pair generation with simulation-driven repair but typically assume a supplied testbench (Ho et al., 2025; Zhao et al., 2025; Liu & Yu, 2026; Liu et al., 2026b; Yu et al., 2026a; Deng et al., 2026). Reinforcement learning for Verilog (Zhu et al., 2025; Wang et al., 2026; Teng et al., 2025) rewards a single generator with testbench or equivalence outcomes against reference designs, and testbench generators such as CorrectBench self-correct HDL testbenches (Qiu et al., 2025). Closest to our setting, ChipMATE trains Verilog and Python reference-model agents with multi-agent RL and reconciles them by randomized cross-simulation (Yu et al., 2026b). LASSO instead builds a specification-linked, qualified checker, and VCPO assigns credit separately to verifier construction, diagnosis, and design repair.

Veri-Sure combines design contracts, temporal tracing, dependency slicing, and formal verification in an RTL generation framework (Liu et al., 2026a). Its contracts guide generation and checking; here, verification contracts also define artifact-specific rewards and comparison groups for learning.

Learning to generate and test software. Software work reranks programs by agreement with generated tests (Chen et al., 2023a) and co-trains code and unit-test generation, grading the tester with reference code (Wang et al., 2025; Jin et al., 2026) or with consensus and mutant kills (Fan et al., 2026). These methods target stateless functions judged by input–output tests. RTL verification additionally requires cycle-accurate timing conventions, a reference model reconciled from the

specification, qualification of checker adequacy against hardware faults, and triage between design and verifier faults. An outcome reward credits none of the resulting artifacts separately, whereas VCPO evaluates verifier quality separately from design correctness.

Formal and hardware verification. Our stages draw on verification planning (Wile et al., 2005; Accellera Systems Initiative, 2015), natural-language-to-property translation (Hahn et al., 2022; Fuggitti & Chakraborti, 2023; Chen et al., 2023b; Cosler et al., 2023; Mendoza et al., 2024) and assertion generation (Kande et al., 2024; Yan et al., 2025), miter-based equivalence checking (Brand, 1993), counterexample minimization (Zeller & Hildebrandt, 2002; Ravi & Somenzi, 2004), vacuity detection (Beer et al., 1997; Kupferman & Vardi, 1999), mutation-based coverage (DeMillo et al., 1978; Chockler et al., 2001; Kupferman et al., 2008), counterexample-guided refinement (Clarke et al., 2000), and assume–guarantee reasoning (Pnueli, 1985; Henzinger et al., 1998). We apply these ideas to simulation evidence and credit assignment without claiming their formal guarantees.

B Limitations of Outcome Rewards in a Self-Verifying Flow

We use the notation of Section 3.1. The four combinations of (v_i, y_i) form the verifier's confusion matrix: true acceptance, false acceptance ($v_i = \text{accept}, y_i = 0$), true rejection, and false rejection ($v_i = \text{reject}, y_i = 1$). An outcome reward $R_i = y_i$ assigns the same scalar to every call in the rollout and is therefore constant across v_i given y_i: a verifier that falsely accepts a faulty design and one that correctly rejects it both receive zero, and a vacuous verifier that happens to accompany a correct design receives the same credit as a sound and adequate one. Verifier behavior can influence the outcome reward only indirectly, through the repairs its feedback induces within the budget. This mismatch matters because verification dominates the rollout: in our collected rollouts, most policy calls belong to verifier-construction stages, and only a small fraction to initial RTL generation. Most policy updates therefore affect verifier-construction calls, although the reward does not directly assess the quality of those calls; Section 4.3 shows that, in our setting, it does not improve held-out pass@1.

C Reward Coefficients

Table 6: Implemented reward coefficients.

Contract	Coefficients
Verifier equation 6	$(\lambda_d, \lambda_g, \lambda_a, \lambda_{\text{FA}}, \lambda_{\text{FR}}) = (0.5, 0.3, 0.2, 0.6, 0.2)$
Diagnosis equation 7	$(\beta_+, \beta_{\text{FA}}, \beta_{\text{rep}}, \beta_u) = (1, 0.5, 0.5, 1)$
Design equation 8	$r_i^c = 1$ on pass; on failure $0.1 \cdot \mathbf{1}[\text{compiles}] + 0.4(1 - \text{mismatch fraction})$
	$\boldsymbol{\lambda}_p = (0.1, 0.1)$ (improving simulation after an edit / after a waveform trace), each count capped at 3
	$\eta(1) = 1,\ \eta(0) = 0.5;\ \lambda_u = 0.05;\ \boldsymbol{\lambda}_e = (0.2, 0.1);\ \kappa_{\text{fail}} = 0.69$
Orchestration	$\lambda_o = 0.2,\ \lambda_{\text{uv}} = 0.5$
Fallback verification	$\mu_x = 0.5,\ \mu_\pm = 0.5$

D Implementation, Benchmark, and Baseline Details

RTL-side prompts. The prompts of the RTL-side agents (RTL generation, simulation, and waveform-guided debugging) largely follow those of Liu et al. (2026b).

VerilogEval-v2. In one problem, the module interface given in the specification does not match the interface expected by the golden testbench. We consulted the VerilogEval-v2 authors and adjusted the problem accordingly. For the category breakdown in Figure 3, we assign each problem to one of five categories (Table 7). *FSM* includes problems that ask only for the combinational next-state or output logic of a state machine (e.g., Prob100_fsm3comb, Prob079_fsm3onehot); *Waveform* contains problems whose specification is given as simulation waveforms.

Table 7: VerilogEval-v2 categories used in Figure 3, listed by problem number (ProbNNN).

Category (count)	Problems
Combinational (60)	001, 002, 003, 004, 005, 006, 007, 008, 010, 011, 012, 013, 014, 015, 017, 018, 019, 020, 021, 022, 023, 025, 026, 029, 032, 036, 039, 042, 043, 044, 050, 051, 052, 055, 057, 059, 062, 064, 065, 069, 070, 071, 072, 076, 077, 081, 087, 092, 093, 094, 097, 106, 112, 113, 114, 116, 122, 125, 131, 132
Arithmetic (7)	009, 016, 024, 027, 030, 033, 123
Sequential (non-FSM) (40)	028, 031, 034, 035, 037, 038, 040, 041, 045, 046, 047, 048, 049, 053, 054, 056, 058, 060, 061, 063, 066, 067, 068, 073, 075, 078, 080, 082, 084, 085, 086, 104, 105, 108, 115, 118, 124, 141, 144, 153
FSM (38)	074, 079, 088, 089, 091, 095, 096, 099, 100, 107, 109, 110, 111, 119, 120, 121, 127, 128, 129, 133, 134, 135, 136, 137, 138, 139, 140, 142, 143, 146, 148, 149, 150, 151, 152, 154, 155, 156
Waveform (11)	083, 090, 098, 101, 102, 103, 117, 126, 130, 145, 147

ChipMATE. We run ChipMATE from its public repository (`https://github.com/zhongkaiyu/ChipMATE`, commit `d94087b`), using the benchmark drivers in `paper_repro/` with its default inference framework (Verilog and Python agents with cross-verification). As stated in Section 4.1, both agents use the same DeepSeek V4 backbone as the other golden-free systems, without training. The released RTLLM-v2 driver (`paper_repro/rtllm/`) includes 33 of the 50 RTLLM-v2 problems; the ChipMATE authors consider their flow unsuited to the remaining 17. Following that choice, we use one-shot generation with the same backbone for those 17 problems and report pass@1 over all 50 problems, scored with the same golden harness as every method in Table 1.

HORIZON. To include HORIZON (Yu et al., 2026a) as a golden-free reference, we reached out to its authors and asked what their system can achieve when the golden feedback is removed. The authors ran HORIZON with golden-testbench feedback disabled, using GPT Sol-5.6 as the backbone, and reported pass@1 of 95.0% on VerilogEval-v2 and 62.8% on CVDP cid003; RTLLM-v2 and ChipBench-SC were not run and are marked N/A in Table 1. We report these author-provided numbers as given and did not re-run them. Because the backbone differs (GPT Sol-5.6 vs. DeepSeek V4) and the runs were performed outside our environment, we list HORIZON, together with Claude Opus 4.6, in a separate block of Table 1 for reference.