

CirCA: Aligning Credit Assignment with Circuit Semantics via Atomic Logic Blocks for RTL Generation

Lei Huang^{1,3}, Rui Zhang¹, Jiaming Guo¹, Yang Zhang^{1,3},
Di Huang¹, Zidong Du¹, Xing Hu¹, Qi Guo¹, Yunji Chen^{1,2,3}

¹State Key Lab of Processors, Institute of Computing Technology,
Chinese Academy of Sciences, Beijing, China

²Institute of AI for Industries, Chinese Academy of Sciences

³University of Chinese Academy of Sciences

Correspondence: cyj@ict.ac.cn

Abstract

Reinforcement learning (RL) post-training has emerged as a promising paradigm for LLM-based Verilog generation. However, existing RL credit assignment methods are either overly coarse (sequence-level GRPO) or fragmented (line-level methods), leading to severe misattribution as they ignore the characteristics of Verilog designs that are fundamentally structured through concurrent constructs and procedural control flows. To bridge this gap, we propose CirCA, a novel RL framework that performs credit assignment based on Atomic Logic Blocks (ALBs), the smallest semantically complete units of circuit logic. CirCA decomposes Verilog into ALBs, derives reliable rewards from signal-level feedback, and then constructs logic-enhanced signal dependency cones to localize erroneous ALBs, whose localized responsibilities are further incorporated into GRPO through asymmetric advantage correction. Extensive experiments show CirCA establishes a new state-of-the-art across multiple benchmarks among non-reasoning models, and even surpasses some cutting-edge reasoning models on complex tasks.

1 Introduction

Large language models (LLMs) have demonstrated strong capability in automated hardware description language (HDL) generation, such as Verilog (Huang et al., 2026). At the same time, RL-based post-training has become an important paradigm for improving functional correctness beyond supervised fine-tuning (SFT), with GRPO-style methods (Shao et al., 2024) showing strong empirical effectiveness. However, some inherent characteristics of HDL make credit assignment in RL more challenging than in software programming, thereby limiting the full potential of post-training optimization for Verilog generation (Yang et al., 2025a).

Fundamentally, Verilog is governed by strict circuit semantics: it describes hardware behavior

through a set of concurrent local update rules instead of a sequential execution process. Each code block typically implements a distinct logical function and influences outputs through heterogeneous dependency paths (Tan and Rosdi, 2014). Under such a structure, the sequence-level credit assignment adopted by GRPO leads to overly coarse-grained attribution (Figure 1-A): it treats the entire generated code as a single optimization unit and uniformly assigns the same advantage to all tokens. As a result, positive credit that should belong to correct local structures can be simultaneously assigned to code regions responsible for erroneous outputs (Tan et al., 2025). By contrast, the line-level credit assignment adopted by prior works (Zhang et al.) for finer-grained attribution is overly fragmented: a coherent logical function in Verilog often spans multiple lines, yet line-level attribution treats each line as an independent unit, splitting credit that should belong to the same functional block across isolated lines and thus causing misattribution (Figure 1-B). Moreover, the search space for line-level error localization grows combinatorially with code length and fault counts (Sudakrishnan et al., 2008; Abreu and Van Gemund, 2009; Wu et al., 2024; Gharachorlu et al., 2026), further exacerbating incorrect credit assignment. In conclusion, for credit assignment in Verilog RL, the key challenge is to make attribution fine-grained and accurate to optimize units that truly align with its local logics.

To this end, we propose CirCA, an RL framework for Verilog generation that performs credit assignment over atomic logic blocks (ALBs) which are defined as the smallest semantically complete units to represent circuit logics. The core insight lies in regarding Verilog code as a composition of these ALBs rather than a sequence or fragmented lines. Thus, applying ALBs as the basis for RL optimization strictly aligns credit assignment with circuit semantics, thereby guaranteeing the fine-grained and accurate attribution.

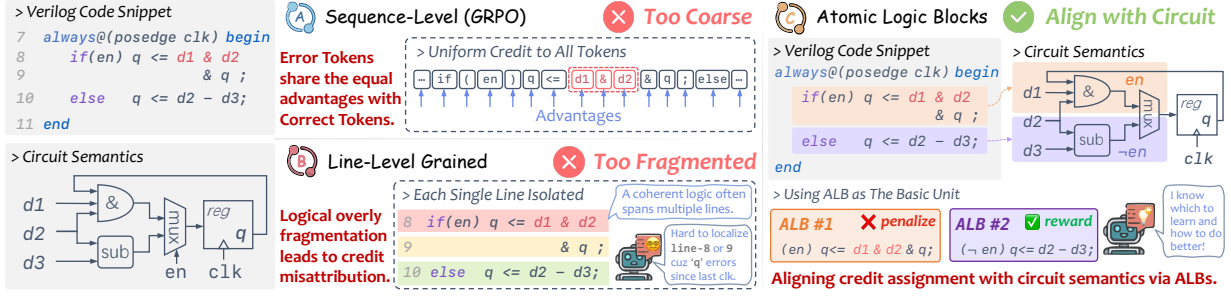


Figure 1: Motivation for CirCA. (A) Sequence-level assignment (GRPO) is too coarse, uniformly sharing advantages across correct and erroneous tokens. (B) Line-level assignment is overly fragmented, breaking coherent multi-line logic and causing severe credit misattribution. (C) The proposed Atomic Logic Blocks (ALBs) align precisely with circuit semantics, enabling accurate penalization of faulty logic while rewarding correct functional units.

Specifically, CirCA comprises four stages. First, *Atomic Logic Block Decomposition* isolates concurrent constructs and flattens procedural control flows to organize Verilog code into ALBs. Then *Semantics-Aware Signal-Level Feedback* constructs testbench-driven simulations with semantics-aware stimuli to provide reliable feedback at the output-signal granularity. Building on this, *Fingerprint-Guided Cone Matching* augments dependency cones with syntax-invariant semantic attributes, performing structural isomorphism checks to accurately localize the ALBs responsible for erroneous outputs. Finally, *Asymmetric Advantage Correction* penalizes faulty ALBs without disrupting the global optimization stability, thus integrating fine-grained attribution into the GRPO framework. Together, these components turn coarse sequence-level optimization into a finer-grained structured credit assignment process aligned with the circuit semantics of Verilog, as shown in Figure 1 (C).

Comprehensive experiments across multiple Verilog generation benchmarks demonstrate that CirCA establishes a new state-of-the-art among non-reasoning general-purpose and code models. Specifically, compared to the previous best non-reasoning model, QiMeng-CRUX (Huang et al., 2026), CirCA achieves an absolute *pass@1* improvement of 4.5% (from 64.6% to 69.1%) on Spec-to-RTL of VerilogEval-v2 and 7.7% (from 62.8% to 70.5%) on RTLLM-v1. Notably, on more challenging tasks, CirCA even outperforms cutting-edge reasoning models. For instance, on the RTLLM-v2 benchmark, it surpasses CodeV-R1 (Zhu et al., 2025) by 1.0% (from 68.0% to 69.0%) and the 671B DeepSeek-R1 (Guo et al., 2025) by 4.3% (from 64.7% to 69.0%) in *pass@1*, respectively. Furthermore, systematic ablation studies validate the essential contribution of adopting Atomic Logic Blocks (ALBs) as the fundamental optimiza-

tion unit, providing a robust and highly efficient paradigm for RL-driven Verilog generation. Code is available at <https://github.com/Taskii-Lei/CirCA>.

2 Related Works

LLM-Based Verilog Generation Automated Verilog code generation using large language models (LLMs) has achieved rapid progress. Foundational works primarily focused on dataset construction and enhancement, utilizing techniques ranging from public code mining and task refinement (Pearce et al., 2020; Liu et al., 2024a; Thakur et al., 2024; Zhao et al., 2025; Liu et al., 2025) to cross-language translation and fine-grained annotation (Hong et al., 2025; Zhang et al., 2024; Li et al., 2026). Beyond static supervision, incorporating execution and verification feedback from downstream EDA tools has emerged as a crucial optimization strategy. One line of work applies such feedback iteratively during inference, using it to identify and revise incorrect outputs to improve the functional correctness of the final generated code (Wang et al., 2025a; Blocklove et al., 2025; Nakkab et al., 2024; Cui et al., 2024; Bai et al., 2025). However, these inference-time methods introduce significant computational overhead and deployment latency, limiting their practical efficiency.

RL-Based Post-Training for Verilog Generation

To overcome inference overhead and inherently align models with hardware-centric objectives, recent studies leverage RL-based post-training (Bai et al., 2022; Shao et al., 2024; Yu et al., 2025) to enhance performance (Le et al., 2022; Dou et al., 2024; Wang et al., 2024). While recent explorations into chain-of-thought (CoT) reasoning via RL (Yubeaton et al., 2025; Qin et al., 2025; Zhu et al., 2025; Wang et al., 2025b) achieve high performance, they suffer from massive training costs and severe inference latency (Yang et al., 2025b).

Efficient, non-reasoning models such as QiMeng-CRUX (Huang et al., 2026) improve sequence-level rewards but still within the naive GRPO framework, while SAL-V (Zhang et al.) explores fine-grained feedback but limited in DPO learning (Rafailov et al., 2023). However, the core challenge of credit assignment in RL remains unresolved, calling for a method that effectively integrates Verilog’s unique circuit semantics with fine-grained attribution.

3 Methodology

Our core insight is that Verilog should be decomposed into atomic logic blocks that reflect its underlying circuit semantics, thus enabling credit assignment to operate on semantically meaningful local logic units. To this end, we build CirCA upon GRPO (Shao et al., 2024), which we review as a preliminary in Section 3.1. Following this, CirCA executes a four-step pipeline as shown in Figure 2. We first derive atomic logic blocks from the Verilog code (Section 3.2). Then we obtain reliable and fine-grained feedback at the output-signal level via testbench with semantics-aware stimuli (Section 3.3). Using these signals, we perform cone matching between the generated and reference designs to localize the ALBs responsible for the erroneous outputs (Section 3.4). Finally, we use the responsible ALBs to assign credit more accurately during policy optimization (Section 3.5).

3.1 Preliminary: Group Relative Policy Optimization (GRPO)

GRPO is a typical RL method for LLM post-training that eliminates the value model by estimating advantages via intra-group relative rewards. For each prompt q , the model rollouts G responses $\{o_1, o_2, \dots, o_G\}$, which are scored by a reward model or a rule-based verifier to get rewards $\mathbf{r} = \{r_1, r_2, \dots, r_G\}$. The advantage for each response is computed by normalizing these rewards: $\hat{A}_i = \frac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r})}$. The final optimization objective is to maximize the objective function:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q, \{o_i\}} \left[\frac{1}{G} \sum_{i=1}^G \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \left(\min(\rho_{i,t} \hat{A}_{i,t}, \text{clip}(\rho_{i,t}) \hat{A}_{i,t}) - \beta \mathbb{D}_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}) \right) \right]$$

where $\rho_i(\theta) = \frac{\pi_\theta(o_i|q)}{\pi_{\theta_{\text{old}}}(o_i|q)}$ is the importance sampling ratio between the current policy and

the old policy, and $\text{clip}(\rho_{i,t})$ is a shorthand for $\text{clip}(\rho_{i,t}, 1 - \epsilon, 1 + \epsilon)$. β is the coefficient controlling the strength of the KL penalty.

3.2 Atomic Logic Block Decomposition

The goal of this step is to partition Verilog code into ALBs that align with the underlying circuit semantics. Verilog exhibits a two-tier execution structure from the code perspective. At the macro level, concurrent constructs, including module instantiations, continuous assignments, and procedural blocks, operate concurrently. At the micro level, statements in procedural blocks, such as always blocks, follow a sequential procedural ordering (Meredith et al., 2010; Wei et al., 2025). Therefore, we perform the decomposition progressively (Figure 2-1).

We first employ a regex-based analyzer to identify module/interface declarations and concurrent constructs by matching HDL keywords, such as module, always, and assign, with their corresponding syntactic boundaries, including semicolons, begin/end pairs, and module-level delimiters. Non-procedural constructs, such as continuous assignments and module instantiations, are directly treated as individual ALBs, since they usually describe structurally separable combinational, connectivity, or component-level logic. For procedural blocks, we further decompose their nested control structures recursively. At divergence points, such as if and case statements, the code is split along branch boundaries to isolate branch-specific update paths. To preserve logic context, enclosing Boolean conditions are propagated downwards to inner blocks through conjunction: an inner block guarded by C_{inner} under an outer condition C_{outer} inherits an effective condition of $C_{\text{outer}} \wedge C_{\text{inner}}$. Consequently, each distinct update path functions as an independent hardware logic unit, thereby constituting a separate ALB with its own trigger condition, control context, and local logic.

3.3 Semantics-Aware Signal-Level Feedback

We next derive reliable feedback at the output signal level from testbench-driven simulation. While prior works demonstrate that signal-level feedback is more informative than module-level pass/fail correctness, they typically rely on random input stimulation (Zhang et al.; Zhu et al., 2025; Huang et al., 2026), which is highly inadequate for sequential designs like FSMs or UARTs and often fails to activate meaningful circuit behaviors, thus producing heavily sparse or biased feedback.

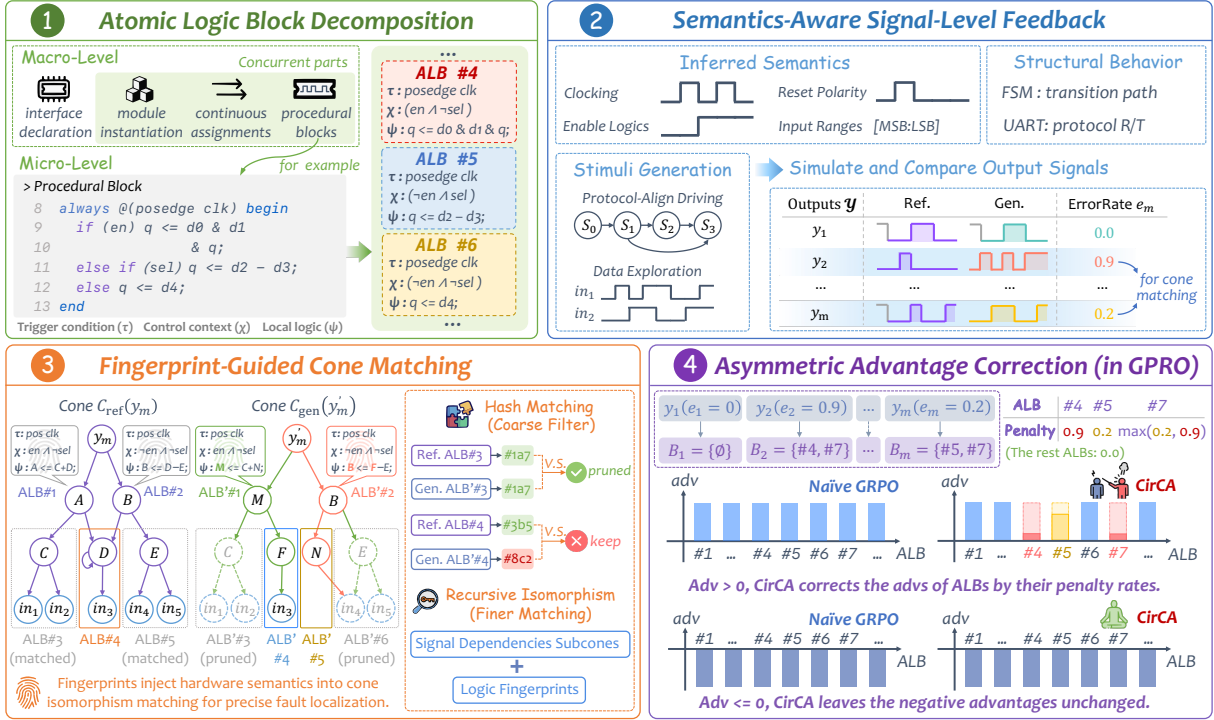


Figure 2: Overview of the CirCA framework, comprising four stages. (1) Atomic Logic Block Decomposition reorganizes Verilog code into macro-level concurrent parts and micro-level ALBs. (2) Semantics-Aware Signal-Level Feedback infers structural behaviors to generate effective testbench stimuli, deriving reliable error rates (e_m) for each output signal. (3) Fingerprint-Guided Cone Matching localizes faulty ALBs by comparing syntax-invariant dependency cones between reference and generated designs via coarse hash matching and recursive isomorphism. (4) Asymmetric Advantage Correction integrates these fine-grained penalties into GRPO, selectively dampening the advantages of responsible ALBs only within globally favored trajectories ($Adv > 0$).

To overcome the limitations of random stimulation, we construct semantics-aware stimuli through rule-based analysis of the reference design. We first extract the top-level module interface and parameters, then identify clock signals from edge-triggered always blocks and resolve reset polarity from sensitivity lists and reset conditions. Enable-like controls are detected using naming and interface patterns. For sequential behaviors, we further analyze state-related registers and parameterized state encodings to recognize FSM-style structures, and match UART-style ports (RX/TX) to generate protocol-valid serial transactions. Based on these extracted semantics, the testbench drives control and protocol-related signals with valid timing and transaction sequences, while leaving unconstrained inputs to generic exploration. Using the semantics-aware stimuli, we co-simulate the reference and generated designs and collect cycle-level mismatches over the output signal set $\mathcal{Y} = \{y_1, \dots, y_M\}$. For each output signal y_m , we define its error rate over time steps T as:

$$e_m = \frac{1}{T} \sum_{t=1}^T \mathbf{1}[\hat{y}_m^{(t)} \neq y_m^{(t)}],$$

where $y_m^{(t)}$ and $\hat{y}_m^{(t)}$ are the reference and generated output values at step t . We then aggregate the correctness scores of all M output signals into a sample-level reward $r = \frac{1}{M} \sum_{m=1}^M (1 - e_m)$. The erroneous output signals ($e_m > 0$) are subsequently used for cone matching, while the e_m and r are used later for the final credit assignment.

3.4 Fingerprint-Guided Cone Matching

To trace the erroneous output signals back to their specific ALBs, we model the dependency cone of a target output signal y as a directed graph $\mathcal{C}(y) = (\mathcal{V}_y, \mathcal{E}_y)$ for both the reference and generated designs. Specifically, we trace backward from y to identify all upstream signals and their driving signal-update events, forming the vertex set $\mathcal{V}_y$. The directed edge set $\mathcal{E}_y$ is naturally established by the dependency flowing from right-hand side (RHS) variables to left-hand side (LHS) signals. To support precise semantic comparison, we augment each node $s \in \mathcal{V}_y$ with logic fingerprint $\phi(s) = \langle \tau(s), \chi(s), \psi(s) \rangle$, where $\tau(s)$ denotes the trigger conditions, $\chi(s)$ the control context inherited from its corresponding ALB, and $\psi(s)$ the local operational logics. To ensure matching ro-

Type	Model	VerilogEval-v1-Machine			VerilogEval-v1-Human			RTLLM-V1		
		p@1	p@5	p@10	p@1	p@5	p@10	p@1	p@5	p@10
Foundation General Models	GPT-4o	67.7	75.5	77.2	60.1	71.4	74.5	41.7	65.9	-
	Deepseek-V3-671B	77.6	86.2	87.4	70.7	77.4	78.8	60.9	74.2	72.4
Foundation & Verilog-Specific Reasoning Models	Deepseek-R1-671B	81.0	87.4	89.5	81.5	87.6	88.5	64.8	82.9	-
	CodeV-R1-Distill-7B	75.8	85.6	87.0	65.7	76.8	79.7	57.4	75.8	81.3
	CodeV-R1-7B	86.1	84.1	85.7	69.9	79.3	81.9	72.9	86.1	88.8
General Non-Reasoning Code Models	Qwen2.5-Coder-7B	50.1	66.5	70.9	22.9	36.0	39.5	32.2	48.2	55.9
	Qwen2.5-Coder-32B	66.6	76.6	79.7	47.6	58.1	61.8	47.9	67.7	-
	Deepseek-Coder-6.7B	52.2	55.4	56.8	30.2	33.9	34.9	30.2	44.0	52.5
Verilog-Specific Non-Reasoning Code Models	RTLCoder-6.7B	61.2	76.5	81.8	41.6	50.1	53.4	35.8	40.3	43.1
	CodeV-Qwen2.5-7B	77.3	87.9	90.1	58.3	66.7	69.7	39.3	63.5	74.2
	OriGen-7B	74.1	82.4	85.7	54.4	60.1	64.2	50.6	68.3	74.3
	QiMeng-SAL-V-7B	79.8	88.0	90.3	60.1	67.6	70.4	62.6	75.1	81.1
	QiMeng-CRUX-7B	82.9	88.3	90.2	65.2	72.0	73.8	62.8	69.0	71.5
	CirCA-7B (Ours)	86.0	87.5	88.2	65.1	70.7	73.7	70.5	79.8	82.4

Table 1: Main Results in VerilogEval-v1 and RTLLM-v1 benchmarks.

bustness, expressions within the fingerprints are strictly canonicalized (e.g., standardizing constant formats and resolving equivalent boolean forms) to capture the hardware semantics and be immune to superficial syntactic variations.

To localize the responsible ALBs efficiently, we perform a coarse-to-fine isomorphism matching between $C_{\text{ref}}(y)$ and $C_{\text{gen}}(y)$, the dependency cones of the reference and generated codes. We first employ ALB-level hash matching as a coarse filter to reduce the search space. For each ALB b , we collect and sort the logic fingerprints of its internal event nodes to ensure invariance to the reordering of independent events. If the generated and reference ALBs share the same hash, their logic is considered structurally verified and safely pruned.

For the remaining unmatched ALBs, starting from the output y , we then employ a recursive isomorphism check $\text{Iso}(u, v)$ to compare dependent signals (u, v) pairs from the reference and generated codes, respectively. If u and v have incompatible types or widths, $\text{Iso}(u, v) = 0$. If both are input signals (leaves of the cone), exact name equivalence is required. Otherwise, we retrieve their driving event nodes to compare their respective sub-cones. Because a signal can be driven by multiple events across branches, we construct a bipartite pairing between the reference and generated event nodes based on their logic fingerprint similarity. For each paired event (s_u, s_v) , we verify the strict equivalence of their fingerprints $\phi(s_u)$ and $\phi(s_v)$. The comparison then proceeds recursively on their upstream dependent signal pairs induced by the graph edges $\mathcal{E}$. If any event-level comparison fails, any recursive dependency pair

violates Iso, or any generated event is left unpaired, we add the generated ALBs associated with s_v to the responsible block set. In this way, the behavioral failures of the output signals are accurately attributed to the responsible ALBs for fine-grained credit assignment.

3.5 Asymmetric Advantage Correction

To redistribute credit assignment over semantically meaningful circuit units rather than uniformly over the whole sequence, we apply localized advantage corrections to the specific ALBs responsible for simulation errors. Let $\mathcal{B}_m$ denote the ALB set responsible for erroneous output signal y_m with error rate e_m . Since a single ALB b may be implicated by multiple erroneous output signals, we assign it the maximum penalty rate among them:

$$\gamma(b) = \begin{cases} \max_{m: b \in \mathcal{B}_m} e_m, & \text{if } b \in \bigcup_m \mathcal{B}_m, \\ 0, & \text{otherwise.} \end{cases}$$

Subsequently, given the original sample-level advantage $\hat{A}$ and the set of tokens $\mathcal{T}(b)$ in ALB b , we define the asymmetric corrected token-level advantages as:

$$\tilde{A}_t = \begin{cases} (1 - \gamma(b))\hat{A}, & \text{if } \hat{A} > 0 \text{ and } t \in \mathcal{T}(b), \\ \hat{A}, & \text{otherwise.} \end{cases}$$

This asymmetric correction rule bridges our fine-grained attribution with global RL frameworks like GRPO. When $\hat{A} > 0$, the trajectory is globally favored, yet faulty ALBs should not be blindly rewarded equally to correct components. So we proportionally down-weight their positive credit

Type	Model	VerilogEval-v2-CC					VerilogEval-v2-SR					RTLLM-V2		
		T=0 (p@1)	T=0.8 (p@1)	p@1	T-Cross p@5	p@10	T=0 (p@1)	T=0.8 (p@1)	p@1	T-Cross p@5	p@10	p@1	p@5	p@10
Foundation General Models	GPT-4o	59.0	56.1	57.6	66.1	69.0	62.5	61.8	64.1	73.7	76.2	56.5	70.3	75.2
	Deepseek-V3-671B	68.0	66.1	68.7	76.3	78.2	68.8	66.9	62.4	71.7	75.0	59.1	71.5	73.3
Foundation & Verilog-Specific Reasoning Models	Deepseek-R1-671B	82.7	81.0	79.1	85.1	87.1	83.3	79.8	77.5	84.7	87.4	64.7	75.8	79.7
	QWQ-32B	-	-	64.0	77.8	80.9	-	-	64.2	77.3	80.1	52.9	68.0	71.2
	CodeV-R1-Distill-7B	59.6	66.9	65.5	75.6	78.2	60.9	64.7	65.2	75.2	77.5	57.2	71.9	77.1
	CodeV-R1-7B	48.1	71.1	69.9	78.2	80.9	49.4	68.1	68.8	78.2	81.1	68.0	78.2	81.7
General Non-Reasoning Code Models	Qwen2.5-Coder-7B	30.1	25.6	30.5	46.8	52.0	23.0	22.0	31.3	49.3	54.6	36.1	52.4	57.6
	Qwen2.5-Coder-32B	44.2	41.5	46.6	59.0	62.8	46.8	41.6	47.5	60.7	64.7	47.8	63.9	67.8
	Deepseek-Coder-6.7B	25.0	29.3	25.2	37.8	45.3	21.7	19.5	20.1	35.4	43.7	41.8	53.9	60.3
Verilog-Specific Non-Reasoning Code Models	RTLCoder-6.7B	35.9	31.5	33.70	45.9	49.8	36.8	30.9	31.1	47.8	52.3	33.6	45.3	49.2
	CodeV-Qwen2.5-7B	58.3	49.9	60.4	68.0	73.3	44.8	37.4	42.2	63.9	69.3	41.0	60.1	68.1
	QiMeng-SAL-V-7B	60.2	57.8	62.6	70.4	73.7	56.5	55.5	59.1	69.9	74.0	52.3	68.6	73.4
	QiMeng-CRUX-7B	68.0	66.7	67.6	72.3	75.5	64.7	64.4	64.6	71.5	74.6	63.8	70.6	73.9
	CirCA-7B (Ours)	67.9	67.8	67.9	72.3	74.2	69.2	68.9	69.1	74.1	75.6	69.0	77.0	79.3

Table 2: Main Results in VerilogEval-v2 of various settings and RTLLM-v2 benchmarks.

via penalty rate $\gamma(b)$. In contrast, when $\hat{A} < 0$, the trajectory is already penalized. Since aggressively amplifying or locally redistributing negative advantages often induces training instability and policy collapse (Ren et al., 2026; Deng et al.), we leave the negative advantages unchanged. In this way, CirCA aligns credit assignment with underlying circuit semantics by penalizing faulty ALBs, while preserving the optimization stability and the sample-level relative ranking established by GRPO.

4 Experiments

4.1 Experiment Settings

Datasets We combine data from CRUX (Huang et al., 2026) and CodeV-R1 (Zhu et al., 2025), remove all intermediate CRUX and reasoning steps and retain only the Description-Verilog pairs. In addition, we use Qwen2.5-Coder-7B (Team, 2025) to generate five responses for each sample, then filter out samples that are entirely or mostly correct, resulting in a final dataset of 165k samples for SFT training to match the scale used in previous works. After that, we applied the same filtering strategy on the SFT model to remove fully correct problems, leaving 135k samples for GRPO training.

Benchmarks Consistent with existing works, we evaluate model performance on the most commonly used and classic four benchmarks to obtain a comprehensive assessment: i) VerilogEval-V1 (Liu et al., 2023) contains 143 GPT-synthesized Machine tasks and 156 hand-crafted Human tasks; ii) VerilogEval-V2 (Pinckney et al., 2025) removes the lower-quality Machine tasks, improves the quality of the Human tasks to Code-Completion (CC)

tasks, and further introduces the more challenging Spec-to-RTL (SR) tasks; iii) RTLLM-v1 (Lu et al., 2024) consists of four task categories: Arithmetic, Memory, Control, and Miscellaneous, with a total of 29 problems; iv) RTLLM-v2 (Liu et al., 2024b) further expands the number of problems to 50, featuring higher overall complexity.

Training and Evaluation Setups To ensure a fair comparison with prior works, we also adopt Qwen2.5-Coder-7B as the backbone. The training pipeline comprises two epochs of SFT followed by one epoch of GRPO. During evaluation, we use $pass@k$ to estimate the probability of generating at least one correct solution among k candidates per task, where n is the total number of sampled candidates and c is the number of correct ones:

$$pass@k := \mathbb{E}_{\text{problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]. \quad (1)$$

To further assess the impact of RL-training explicitly, we additionally define the absolute and relative performance gains over the SFT baseline as $G = p@1_{RL} - p@1_{SFT}$ and $G\% = G/p@1_{SFT}$, respectively. We note that the original VerilogEval-V2 study requires reporting $pass@1$ only under two motivated settings: low-temperature setup ($T=0.0$, $top_p=0.01$, $n=1$) and high-temperature setup ($T=0.8$, $top_p=0.95$, $n=20$) (Pinckney et al., 2025). Many prior works select the best results across $T \in \{0.2, 0.5, 0.8\}$ with $n = 20$ (denoted as *Cross*). For comprehensive fairness, we report results under both evaluation paradigms.

Models	Type	Machine	Human	CC-Low	CC-High	CC-Cross	SR-Low	SR-High	SR-Cross	RTLLM-v1	RTLLM-v2
QiMeng-CodeVR1 (Reasoning Model)	SFT	75.8	65.7	59.6	66.9	65.5	60.9	64.1	65.2	57.4	57.2
	RL	86.1	69.9	48.1	71.1	69.9	49.4	68.1	68.8	72.9	68.0
	$G_{p@1}$ $G\%$	10.3 13.59%	4.2 6.39%	-11.5 -19.30%	4.2 6.28%	4.4 6.72%	-11.5 -18.88%	4.0 6.24%	3.6 5.52%	15.5 27.00%	10.8 18.88%
QiMeng-SALV (Non-Reasoning)	SFT	76.9	56.8	54.5	57.3	57.2	50.0	56.9	57.2	48.3	50.4
	RL	79.8	60.1	60.2	57.8	62.6	56.5	55.5	59.1	62.6	52.3
	$G_{p@1}$ $G\%$	2.9 3.74%	3.3 5.83%	5.7 10.50%	0.5 0.80%	5.4 9.48%	6.5 13.00%	-1.4 -2.49%	1.9 3.30%	14.3 29.61%	1.9 3.77%
QiMeng-CRUX (Non-Reasoning)	SFT	76.8	63.2	64.7	58.9	65.5	59.6	57.0	61.1	49.0	52.3
	RL	82.9	65.2	68	66.7	67.6	64.7	64.4	64.6	62.8	63.8
	$G_{p@1}$ $G\%$	6.1 7.94%	2.0 3.16%	3.3 5.10%	7.8 13.24%	2.1 3.21%	5.1 8.56%	7.4 12.98%	3.5 5.73%	13.8 28.16%	11.5 21.99%
CirCA (Ours) (Non-Reasoning)	SFT	74.6	60.4	62.8	60.7	61.9	60.3	59.5	61.3	52.8	56.3
	RL	86.0	65.1	67.9	67.8	67.9	69.2	68.9	69.1	70.5	69.0
	$G_{p@1}$ $G\%$	11.4 15.25%	4.7 7.78%	5.1 8.12%	7.1 11.70%	6.0 9.69%	8.9 14.76%	9.4 15.80%	7.8 12.72%	17.7 33.52%	12.7 22.56%

Table 3: Absolute ($G_{p@1}$) and relative ($G\%$) $pass@1$ gains from SFT to RL.

4.2 Main Results

Our main results are summarized in Tables 1, 2 and 3. The baselines include both general-purpose code models and Verilog-specialized models with context length comparable to ours (≤ 4096). We additionally report results for several foundation models, as well as general-domain and Verilog-specific reasoning models, and highlight the best performance of reasoning and non-reasoning models separately for clearer comparison.

Overall performance comparison in Tables 1, 2 demonstrates our CirCA surpasses general-purpose foundation models and achieves state-of-the-art results among non-reasoning code models across most benchmarks, especially on the more challenging Spec-to-RTL and RTLLM tasks. Compared to the previous best QiMeng-CRUX, CirCA delivers a consistent $pass@1$ improvement of at least 4.0% across all settings on Spec-to-RTL, while increasing performance by 7.7% (from 62.8% to 70.5%) on RTLLM-v1 and 5.2% (from 63.8% to 69.0%) on RTLLM-v2. Notably, CirCA even outperforms reasoning models on these tasks. It surpasses the 671B DeepSeek-R1 by 5.7% (from 64.8% to 70.5%) on RTLLM-v1 and by 4.3% (from 64.7% to 69.0%) on RTLLM-v2, and marginally beats CodeV-R1-7B on both Spec-to-RTL and RTLLM-v2 benchmarks.

To isolate the contribution of RL algorithms, we further compare the $pass@1$ gains from SFT to RL for methods with available results at both stages. Comparison results in Table 3 show that CirCA yields more significant improvements across nearly all benchmarks. For example, CirCA achieves absolute $pass@1$ gains of +17.7 (33.52%) and +12.7

(22.56%) on RTLLM-v1 and v2 respectively, exceeding the previous non-reasoning best QiMeng-CRUX (+13.8 and +11.5), and the reasoning best CodeV-R1-7B (+15.5 and +10.8). This confirms that our semantic-aware stimuli and circuit-aligned credit assignment provide precise, fine-grained feedback, thereby maximizing the effectiveness of RL post-training for complex hardware design.

We notice that QiMeng-CodeVR1 suffers severe performance degradation under low-temperature settings on VerilogEval-v2, mainly due to massive token repetitions that prevent valid code generation. This suggests that RL-trained reasoning models may depend on diverse trajectory exploration, while overly restrictive decoding can collapse their output distributions into repetitive, sub-optimal modes (Basu et al., 2020; Pipis et al., 2025; Duan et al., 2026), which we leave for future work.

4.3 Ablation Study

To assess the individual contributions of our proposed mechanisms, we conduct comprehensive ablation studies on semantics-aware stimuli, the ALB-level attribution granularity, and the asymmetric credit assignment. All variants are initialized from the same SFT model and trained on the same data with GRPO for equivalent steps (about 1 epoch). Results are shown in Table 4.

Ablation of Semantics-Aware Stimuli (SA-TB)

To evaluate our testbench construction strategy, we compare traditional random inputs (*w/o SA-TB*) (Zhu et al., 2025; Huang et al., 2026; Zhang et al.) with our semantics-aware stimuli (*w SA-TB*) under naive GRPO-based RL post-training. Results show that while random stimuli offer moderate im-

Models	VE-v1-Machine			VE-v1-Human			VE-v2-CC (p@1)			VE-v2-SR (p@1)			RTLLM-v2		
	p@1	p@5	p@10	p@1	p@5	p@10	T=0	T=0.8	T-Cross	T=0	T=0.8	T-Cross	p@1	p@5	p@10
SFT (Baseline)	74.6	89.4	91.0	60.4	71.9	75.4	62.8	60.7	61.9	60.3	59.5	61.3	56.3	67.6	73.3
w/o SA-TB	80.6 (+6.0)	88.1 (-1.3)	88.9 (-2.1)	62.5 (+2.1)	68.4 (-3.5)	70.3 (-5.1)	64.7 (+1.9)	64.3 (+3.6)	64.9 (+3.0)	64.1 (+3.8)	63.4 (+3.9)	63.7 (+2.4)	61.4 (+5.1)	67.8 (+0.2)	70.4 (-2.9)
w SA-TB	82.6 (+8.0)	87.9 (-1.5)	89.8 (-1.2)	63.1 (+2.7)	70.0 (-1.9)	72.9 (-2.5)	66.0 (+3.2)	65.7 (+5.0)	66.2 (+4.3)	65.4 (+5.1)	65.0 (+5.5)	65.8 (+4.5)	63.7 (+7.4)	71.4 (+3.8)	73.7 (+0.4)
Concurrency-Level	78.8 (+4.2)	81.2 (-8.2)	83.9 (-7.1)	58.3 (-2.1)	64.0 (-7.9)	66.5 (-8.9)	62.8 (+0.0)	61.3 (+0.6)	61.2 (-0.7)	55.7 (-4.6)	57.0 (-2.5)	56.6 (-4.7)	54.9 (-1.4)	67.9 (+0.3)	71.8 (-1.5)
Line-Level	79.2 (+4.6)	83.4 (-6.0)	84.6 (-6.4)	60.7 (+0.3)	66.3 (-5.6)	68.4 (-7.0)	63.5 (+0.7)	63.1 (+2.4)	63.6 (+1.7)	59.6 (-0.7)	61.2 (+1.7)	60.9 (-0.4)	50.7 (-5.6)	62.7 (-4.9)	67.3 (-6.0)
Symmetric CA	83.2 (+8.6)	85.4 (-4.0)	86.8 (-4.2)	64.4 (+4.0)	68.8 (-3.1)	71.0 (-4.4)	66.7 (+3.9)	67.1 (+6.4)	67.4 (+5.5)	67.3 (+7.0)	66.7 (+7.2)	67.5 (+6.2)	65.1 (+8.8)	69.4 (+1.8)	71.5 (-1.8)
Asymmetric CA (Ours)	86.0 (+11.4)	87.5 (-1.9)	88.2 (-2.8)	65.1 (+4.7)	70.7 (-1.2)	73.7 (-1.7)	67.9 (+5.1)	67.8 (+7.1)	67.9 (+6.0)	69.2 (+8.9)	68.9 (+9.4)	69.1 (+7.8)	69.0 (+12.7)	77.0 (+9.4)	79.3 (+6.0)

Table 4: Results of the Ablation Study. Numbers in the second line indicate the absolute gain over SFT baseline.

provements over the SFT baseline, incorporating SA-TB consistently provides further gains across all benchmarks. Specifically, it improves *pass@1* from 61.4% to 63.7% on RTLLM-v2, and brings at least 1.3% improvements over *w/o SA-TB* across all VerilogEval-v2 settings. This confirms that SA-TB better captures key hardware behaviors, providing more reliable reward signals for RL post-training.

Ablation of Attribution Granularity We validate the necessity of ALB decomposition by replacing it with coarser Concurrency-Level and finer Line-Level attribution units, while keeping the same Fingerprint-Guided Cone Matching strategy. Here, Concurrency-Level refers to the macro-level concurrent blocks shown in Figure 2. As shown in Table 4, deviating from the ALB granularity substantially degrades performance, even below the SFT baseline. Concurrency-Level attribution reduces *pass@1* on RTLLM-v2 to 54.9%, below the SFT baseline of 56.3%, because treating an entire multi-branch always block as a single unit unfairly penalizes correct branches when bugs are localized elsewhere within the block. In contrast, Line-Level attribution performs even worse, reaching only 50.7% on RTLLM-v2, as overly fragmented units introduce severe noise into credit assignment, ultimately leading to training instability and degraded performance. These results indicate that attribution granularity should be neither too coarse to unfairly penalize correct logic nor too fragmented to inject noise. Compared to these extremes, our results confirm that ALB serves as a highly suitable and effective granularity for Verilog optimization.

Ablation of Asymmetric Credit Assignment Finally, we investigate the effectiveness of our asymmetric advantage correction mechanism. In *Asymmetric CA*, token advantages are selectively penalized only in globally favored trajectories ($\hat{A} > 0$), while leaving negative advantages unchanged. To

evaluate this design, we introduce a *Symmetric CA* variant that penalizes responsible ALBs bidirectionally: it applies a dampened advantage $(1 - \gamma(b))\hat{A}$ when $\hat{A} > 0$ and an exacerbated negative advantage $(1 + \gamma(b))\hat{A}$ when $\hat{A} < 0$. Results in Table 4 show that *Symmetric CA* improves over the baseline but consistently underperforms our asymmetric approach. For example, *Asymmetric CA* outperforms *Symmetric CA* by 1.6% on Spec-to-RTL and 3.9% on RTLLM-v2 in *pass@1*. This gap suggests that amplifying penalties in globally unfavored trajectories ($\hat{A} < 0$) is suboptimal: such poor-quality responses already receive negative feedback, and further local penalization yields noisy gradients without constructive guidance (Ren et al., 2026; Deng et al.). By asymmetrically suppressing only localized bugs within globally favored trajectories, CirCA avoids these noisy updates and precisely corrects specific flaws while safely retaining valuable structural patterns. Additional mutation-based diagnostics in Appendix A further support the effectiveness of SA-TB and ALB-level localization.

5 Conclusion

In this work, we propose CirCA, a circuit-semantics-aligned RL post-training framework for Verilog generation. CirCA localizes erroneous output signals to responsible Atomic Logic Blocks (ALBs) through fingerprint-guided dependency cone matching, and uses the resulting block-level feedback to refine token-level credit assignment. Together with semantics-aware testbench construction, CirCA provides more reliable fine-grained feedback for policy optimization. Extensive experiments show that CirCA achieves state-of-the-art performance and outperforms several strong reasoning models on multiple benchmarks, demonstrating the effectiveness of ALB-level credit assignment for hardware code generation.

Limitations

Although CirCA demonstrates strong performance in credit assignment for RL-based Verilog generation, it still has several limitations. The ALB decomposition and signal dependency cone construction rely on static analysis of Verilog, which may introduce non-trivial computational overhead for very large-scale or deep sequential designs. In addition, the overall performance is influenced by the coverage of simulation stimuli, and certain rare or temporally constrained behaviors may not be fully activated under limited simulation budgets. Finally, while CirCA provides a structured and effective attribution mechanism, exploring more efficient and accurate credit assignment strategies for more complex system-level Verilog projects remains an important direction for future work.

References

- Rui Abreu and Arjan JC Van Gemund. 2009. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *SARA*, volume 9, pages 2–9.
- Yunsheng Bai, Ghaith Bany Hamad, Chia-Tung Ho, Syed Suhaib, and Haoxing Ren. 2025. Fvdebug: An llm-driven debugging assistant for automated root cause analysis of formal verification failures. *arXiv preprint arXiv:2510.15906*.
- Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, and 1 others. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*.
- Sourya Basu, Govardana Sachitanandam Ramachandran, Nitish Shirish Keskar, and Lav R Varshney. 2020. Mirostat: A neural text decoding algorithm that directly controls perplexity. *arXiv preprint arXiv:2007.14966*.
- Jason Blocklove, Shailja Thakur, Benjamin Tan, Hammond Pearce, Siddharth Garg, and Ramesh Karri. 2025. Automatically improving llm-based verilog generation using eda tool feedback. *ACM Transactions on Design Automation of Electronic Systems*.
- Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, and Dahua Lin. 2024. Origen: Enhancing RTL code generation with code-to-code augmentation and self-reflection. In *ICCAD*, pages 99:1–99:9. ACM.
- Wenlong Deng, Yushu Li, Boying Gong, Yi Ren, Christos Thrampoulidis, and Xiaoxiao Li. On group relative policy optimization collapse in agent search: The lazy likelihood-displacement. In *ICLR 2026 Workshop on Lifelong Agents: Learning, Aligning, Evolving*.
- Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, and 1 others. 2024. Step-coder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*.
- Zenghao Duan, Liang Pang, Zihao Wei, Wenbin Duan, Yuxin Tian, Shicheng Xu, Jingcheng Deng, Zhiyi Yin, and Xueqi Cheng. 2026. Circular reasoning: Understanding self-reinforcing loops in large reasoning models. *arXiv preprint arXiv:2601.05693*.
- Golnaz Gharachorlu, Mahsa Panahandeh, Lionel C Briand, Ruifeng Gao, and Ruiyuan Wan. 2026. Similar pattern annotation via retrieval knowledge for llm-based test code fault localization. *arXiv preprint arXiv:2605.07957*.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shitong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Charles Hong, Brendan Roberts, Huijae An, Alex Um, Advay Ratan, and Yakun Sophia Shao. 2025. hdl2v: A code translation dataset for enhanced llm verilog generation. *arXiv preprint arXiv:2506.04544*.
- Lei Huang, Rui Zhang, Jiaming Guo, Yang Zhang, Di Huang, Shuyao Cheng, Pengwei Jin, Chongxiao Li, Zidong Du, Xing Hu, and 1 others. 2026. Qimengcrux: Narrowing the gap between natural language and verilog via core refined understanding expression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 31193–31201.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. [Coderl: Mastering code generation through pretrained models and deep reinforcement learning](#). In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.
- Kezhi Li, Min Li, Xiangyu Wen, Shibo Zhao, Jieying Wu, Junhua Huang, and Qiang Xu. 2026. Formal-rtl: Verified rtl synthesis at scale. *arXiv preprint arXiv:2603.08738*.
- Mingjie Liu, Nathaniel Pinckney, Bruce Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8. IEEE.
- Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. 2025. Craftrtl: High-quality synthetic data

- generation for verilog code models with correct-by-construction non-textual representations and targeted code repair. In *ICLR*. OpenReview.net.
- Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024a. Rtl-coder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.
- Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 2024b. Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9.
- Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. Rtlm: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 722–727. IEEE.
- Patrick Meredith, Michael Katelman, José Meseguer, and Grigore Rosu. 2010. [A formal executable semantics of verilog](#). In *8th ACM/IEEE International Conference on Formal Methods and Models for Code-sign, MEMOCODE 2010*, 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2010, pages 179–188. 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2010 ; Conference date: 26-07-2010 Through 28-07-2010.
- Andre Nakkab, Sai Qian Zhang, Ramesh Karri, and Siddharth Garg. 2024. Rome was not built in a single step: Hierarchical prompting for llm-based chip design. In *MLCAD*, pages 26:1–26:11. ACM.
- Hammond Pearce, Benjamin Tan, and Ramesh Karri. 2020. Dave: Deriving automatically verilog from english. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 27–32.
- Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2025. Revisiting verilogeval: A year of improvements in large-language models for hardware code generation. *ACM Transactions on Design Automation of Electronic Systems*.
- Charilaos Pipis, Shivam Garg, Vasilis Kontonis, Vaishnavi Shrivastava, Akshay Krishnamurthy, and Dimitris Papailiopoulos. 2025. Wait, wait, wait... why do reasoning models loop? *arXiv preprint arXiv:2512.12895*.
- Haiyan Qin, Zhiwei Xie, Jingjing Li, Liangchen Li, Xiaotong Feng, Junzhan Liu, and Wang Kang. 2025. Reasoningv: Efficient verilog code generation with adaptive hybrid reasoning model. *arXiv preprint arXiv:2504.14560*.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741.
- Yi Ren, Muchen Li, Danica J Sutherland, Xiaoxiao Li, Christos Thrampoulidis, and 1 others. 2026. On the effect of negative gradient in group relative deep reinforcement optimization. *Advances in Neural Information Processing Systems*, 38:73828–73856.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, and 1 others. 2024. Deepseek-math: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.
- Sangeetha Sudakrishnan, Janaki Madhavan, E James Whitehead Jr, and Jose Renau. 2008. Understanding bug fix patterns in verilog. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 39–42.
- Hongze Tan, Zihan Wang, Jianfei Pan, Jinghao Lin, Hao Wang, Yifan Wu, Tao Chen, Zhihang Zheng, Zhihao Tang, and Haihua Yang. 2025. Gtpo and grpo-s: Token and sequence-level reward shaping with policy entropy. *arXiv preprint arXiv:2508.04349*.
- Tze Sin Tan and Bakhtiar Affendi Rosdi. 2014. Verilog hdl simulator technology: a survey. *Journal of Electronic Testing*, 30(3):255–269.
- Qwen Team. 2025. [Qwen3 technical report](#). *Preprint*, arXiv:2505.09388.
- Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3):1–31.
- Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Nan Guan, and Zhe Jiang. 2025a. Insights from verification: Training a verilog generation llm with reinforcement learning with testbench feedback. *arXiv preprint arXiv:2504.15804*.
- Ning Wang, Bingkun Yao, Jie Zhou, Xi Wang, Zhe Jiang, and Nan Guan. 2024. Large language model for verilog generation with code-structure-guided reinforcement learning. *arXiv preprint arXiv:2407.18271*.
- Yiting Wang, Guoheng Sun, Wanghao Ye, Gang Qu, and Ang Li. 2025b. Verireason: Reinforcement learning with testbench feedback for reasoning-enhanced verilog generation. *arXiv preprint arXiv:2505.11849*.
- Anjiang Wei, Huanmi Tan, Tarun Suresh, Daniel Mendoza, Thiago SFX Teixeira, Ke Wang, Caroline Trippel, and Alex Aiken. 2025. Vericoder: Enhancing llm-based rtl code generation through functional correctness validation. *arXiv preprint arXiv:2504.15659*.

- Jiang Wu, Zhuo Zhang, Deheng Yang, Jianjun Xu, Jiayu He, and Xiaoguang Mao. 2024. Knowledge-augmented mutation-based bug localization for hardware design code. *ACM Transactions on Architecture and Code Optimization*, 21(3):1–26.
- Guang Yang, Wei Zheng, Xiang Chen, Dong Liang, Peng Hu, Yukui Yang, Shaohang Peng, Zhengnan Li, Jiahui Feng, Xiao Wei, and 1 others. 2025a. Large language model for verilog code generation: Literature review and the road ahead. *arXiv preprint arXiv:2512.00020*.
- Yiyao Yang, Fu Teng, Pengju Liu, Mengnan Qi, Chenyang Lv, Ji Li, Xuhong Zhang, and Zhezhi He. 2025b. *Haven: Hallucination-mitigated LLM for verilog code generation aligned with HDL engineers*. In *Design, Automation & Test in Europe Conference, DATE 2025, Lyon, France, March 31 - April 2, 2025*, pages 1–7. IEEE.
- Qiyang Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaozhong Liu, Lingjun Liu, Xin Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.
- Patrick Yubeaton, Andre Nakkab, Weihua Xiao, Luca Collini, Ramesh Karri, Chinmay Hegde, and Siddharth Garg. 2025. Verithoughts: Enabling automated verilog code generation using reasoning and formal verification. *arXiv preprint arXiv:2505.20302*.
- Yang Zhang, Rui Zhang, Jiaming Guo, Huang Lei, Di Huang, Yunpu Zhao, Shuyao Cheng, Pengwei Jin, Chongxiao Li, Zidong Du, and 1 others. Qimengsalv: Signal-aware learning for verilog code generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. 2024. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–5. IEEE.
- Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, and 1 others. 2025. Codev: Empowering llms with hdl generation through multi-level summarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.
- Yaowei Zheng, Juntong Lu, Shenzhi Wang, Zhangchi Feng, Dongdong Kuang, and Yuwen Xiong. 2025. Easyr1: An efficient, scalable, multi-modality rl training framework. <https://github.com/hiyouga/EasyR1>.
- Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyang Luo, Zhangchi Feng, and Yongqiang Ma. 2024. *Llamafactory: Unified efficient fine-tuning of 100+ language models*. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.
- Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Pengwei Jin, and 1 others. 2025. Qimeng-codev-r1: Reasoning-enhanced verilog generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

A Appendix

A.1 Additional Diagnostic Analysis

To better understand the mechanisms behind CirCA’s gains, we conduct a controlled mutation-based diagnostic analysis on the 50 designs of RTLLM-v2 benchmark. While the ablation studies demonstrate the contribution of each component in CirCA, this analysis provides a more direct examination of two key questions: whether semantics-aware testbenches can reliably expose functional deviations, and whether the resulting mismatches can be localized to suitable code regions for credit assignment.

Specifically, we inject one local semantic mutation into each reference design, so that the faulty location is known by construction. We first perform a self-consistency test by comparing the reference design with an identical copy, where a reliable testbench should report no mismatch. We then compare the reference design with its mutated version to evaluate whether the generated stimuli can expose the injected fault. Finally, for detected mutations, we compare ALB-level and line-level localization under the same cone-matching procedure, thereby isolating the effect of attribution granularity.

Mutation Construction. We construct the mutation set from 50 reference designs in RTLLM-v2 benchmark. For each design, we use Deepseek-V3 to inject a single local semantic mutation into one selected ALB. The mutation prompt explicitly restricts the modification to a predefined set of local mutation types, including operator mutation, branch-condition mutation, numeric-constant mutation, RHS-operand mutation, and case-item assignment mutation. We also require the model to keep the module interface, signal declarations and clock/reset names unchanged, so that the mutated design remains comparable with the original reference design. The overall distribution of muta-

tion types is shown in Table 5, and representative mutation examples are provided in Table 6.

Testbench Reliability and Effectiveness. We report two diagnostic metrics: self-consistency pass rate and mutation detection rate. The former measures the number and ratio of reference-copy pairs with 100% output correctness, while the latter measures the number and ratio of mutated samples whose output correctness is lower than 100%.

Consistent with the ablation setting, we mainly compare *w/o SA-TB* and *w/ SA-TB*. We additionally include *CodeV-R1 TB* as a representative random-stimulus testbench constructor from prior work (Le et al., 2022). Compared with *CodeV-R1 TB*, *w/o SA-TB* keeps random stimulation but adopts our improved interface parsing and testbench assembly pipeline, while *w/ SA-TB* further introduces semantics-aware stimuli. Thus, *CodeV-R1 TB* vs. *w/o SA-TB* reflects the robustness of testbench construction, and *w/o SA-TB* vs. *w/ SA-TB* isolates the effect of semantics-aware stimulation.

As shown in Table 7, *CodeV-R1 TB* passes the self-consistency test on 42 out of 50 samples, while both *w/o SA-TB* and *w/ SA-TB* achieve 50 out of 50. This shows that our improved parsing and testbench assembly pipeline enhances the robustness of testbench construction across diverse Verilog designs. For mutation detection, *CodeV-R1 TB* detects 35 out of 50 mutations, and *w/o SA-TB* detects 39 out of 50 under the random-stimulus setting. In contrast, *w/ SA-TB* detects 48 out of 50 mutations. The two missed cases require highly specific temporal coverage: one appears only in a half-cycle window of an odd frequency divider, and the other requires a valid reset-release, multi-push, and multi-pop sequence in a LIFO buffer, which is difficult to trigger within our maximum simulation budget of 3000 steps. Overall, these results show that semantics-aware stimuli more effectively activate mutation-affected sequential behaviors and provide more reliable feedback.

Mutation Type	Count	Ratio
Operator mutation	32	64%
Constant mutation	9	18%
RHS operand mutation	6	12%
Condition mutation	3	6%
Total	50	100%

Table 5: Mutation type distribution on RTLLM-v2.

Task	ALB	Type	Local Mutation
adder_8bit	ALB7	Operator	$a + b + \text{cin} \rightarrow a \wedge b \wedge \text{cin}$
adder_bcd	ALB3	Constant	$+ 4'b0110 \rightarrow + 4'b0101$
ring_counter	ALB5	RHS	$\{\text{state}[6:0], \text{state}[7]\} \rightarrow \{\text{state}[0], \text{state}[7:1]\}$
fsm	ALB23	Constant	$\text{IN} == 1 \rightarrow \text{IN} == 0$
counter_12	ALB5	Constant	$\text{out} == 4'd11 \rightarrow \text{out} == 4'd12$
alu	ALB9	Operator	$a_signed - b_signed \rightarrow a_signed + b_signed$
RAM	ALB8	RHS	$\text{RAM}[\text{read_addr}] \rightarrow \text{RAM}[\text{write_addr}]$
synchronizer	ALB19	Condition	$\text{en_clap_two} ? \rightarrow !\text{en_clap_two} ?$
calendar	ALB16	Operator	$\&\& \rightarrow $
radix2_div	ALB8	Condition	$\text{C0} ? \rightarrow \sim\text{C0} ?$

Table 6: Representative mutation examples.

Method	Self-Consistency $\uparrow$	Mutation Detection $\uparrow$
<i>CodeV-R1 TB</i>	42 / 50 (84.0%)	35 / 50 (70.0%)
<i>w/o SA-TB</i>	50 / 50 (100.0%)	39 / 50 (78.0%)
<i>w/ SA-TB</i>	50 / 50 (100.0%)	48 / 50 (96.0%)

Table 7: Diagnostic evaluation of testbench reliability and effectiveness.

Localization Granularity Analysis. In the ablation study, we have shown that ALB-level attribution leads to better final benchmark performance than other granularities. To further examine its localization accuracy under fingerprint-guided cone matching, we evaluate on the mutated codes, where the faulty ALB and changed source lines are known by construction. We compare the ALB or line indices returned by cone matching with the corresponding ground-truth faulty locations.

As shown in Table 8, ALB-level localization correctly identifies the mutated region in 45 out of 50 cases, while line-level localization succeeds in 33 out of 50 cases. This result shows that ALBs provide a more suitable attribution unit for circuit-level localization. Compared with individual lines, ALBs better preserve the semantic completeness of local hardware behavior, reducing fragmented or misaligned attribution.

For the remaining failed cases, beyond the two mutation instances not detected, most errors occur in sequential circuits with complex state transitions. In these cases, the Cone-Matching process may involve deep recursive traversal, which can lead to timeout or degraded localization accuracy. We leave further improvements on highly complex circuit structures as future work. We do not report concurrency-level localization in this diagnostic analysis. Under such a coarse granularity, the faulty block can often be included in the returned region,

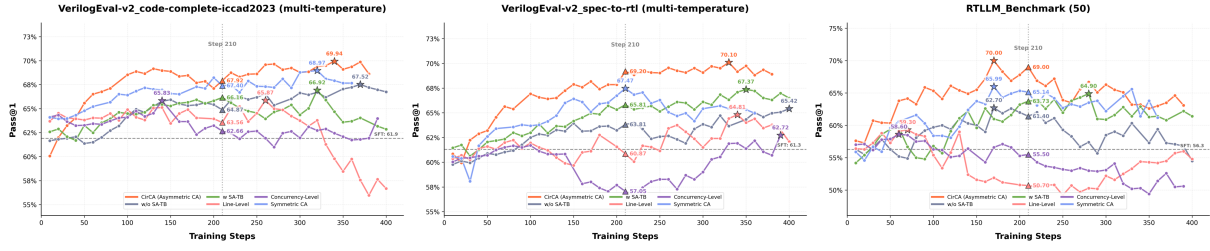


Figure 3: Dynamic pass@1 performance over 400 RL training steps on three benchmarks. Checkpoints are evaluated every 10 steps, with markers indicating the performance at step 210 and the best performance achieved during training.

Granularity	Correct Localization $\uparrow$	Accuracy $\uparrow$
Line-level	33 / 50	66.0%
ALB-level	45 / 50	90.0%

Table 8: Localization accuracy under different attribution granularities.

but the actual error is usually buried inside a large amount of correct code. As a result, the attribution may be technically correct but too imprecise to provide useful fine-grained credit assignment.

A.2 Additional Training and Implementation Details

Training Data Construction Our training data is derived from QiMeng-CRUX (Huang et al., 2026) and QiMeng-CodeV-R1 (Le et al., 2022), both of which are originally built upon CodeV (Zhao et al., 2025) with different processing strategies. CodeV collects open-source Verilog code from the web, removes non-synthesizable designs, and uses GPT to generate corresponding task descriptions, forming *Description-Verilog* pairs. It also removes samples with Rouge-L > 0.5 against evaluation samples in benchmarks to avoid data contamination. QiMeng-CRUX rewrites the descriptions to better match realistic human queries and introduces structured formats such as Karnaugh maps, state-transition diagrams, and timing/state tables. QiMeng-CodeV-R1, adopts a code-to-NL-to-code data construction pipeline: it generates descriptions from the original Verilog snippets, uses DeepSeek-R1 to produce reasoning thoughts and regenerated Verilog code, and filters samples through equivalence checking between the original and regenerated code. All datasets are used in accordance with their original licenses and intended research use.

In our experiments, we remove CRUX intermediate steps and CoT traces, retaining only the final *Description-Verilog* pairs. We then let the LLM rollout five responses for each problem and use

Training Method	SFT	GRPO
epoch	2	1
learning_rate	1.0e-5	1.0e-6
strategy	adam	adamw
warmup_ratio	0.1	0.0
rollout_n	-	5
rollout_batch	-	512
max_length	4096	4096
n_gpu	4 $\times$ A100,80g	4 $\times$ A100,80g
time cost	32h	40h
temperature	-	1.0
top-p	-	0.99

Table 9: Training parameters and computational usage.

testbench-based evaluation to filter out easy samples to keep the final training data scale consistent with prior works. For fair comparison, all methods use the same backbone, Qwen2.5-Code-7B-Instruct (Team, 2025), and the same training hyperparameters. We also report SFT-to-RL performance gains to more directly reflect the improvement from CirCA’s credit assignment mechanism. During GRPO training, generated RTL is compiled with the constructed testbench using Icarus Verilog and simulated with vvp to obtain compilation status and signal-level feedback.

Hyperparameters and Computational Usage

Our training pipeline is implemented with LLaMA-Factory (Zheng et al., 2024) for the SFT stage and EasyR1 (Zheng et al., 2025) for the RL post-training stage. We report the detailed hyperparameters and computational usage in Table 9.

For the complete training process over 165k samples with *rollout_n* = 5, CirCA introduces approximately 7.45 additional hours compared with vanilla GRPO. This overhead mainly comes from test-code ALB decomposition and cone matching during training, rather than from all analysis components in the pipeline. Specifically, testbench

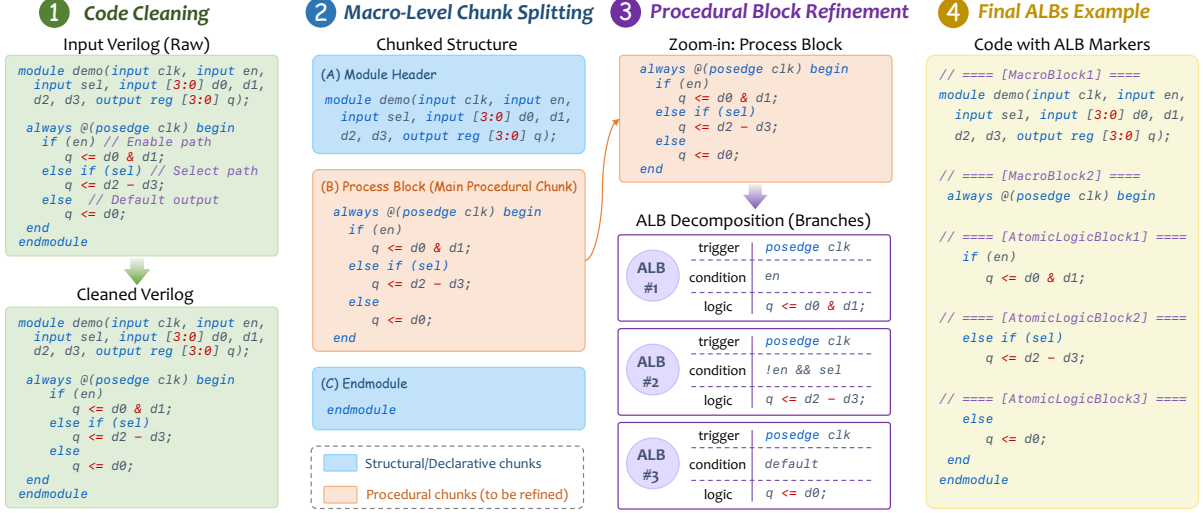


Figure 4: Illustration of Atomic Logic Block (ALB) decomposition. The Verilog code is first cleaned to remove comments and normalize formatting, then split into macro-level chunks such as module headers, procedural blocks, and structural/declarative components. Procedural blocks are further refined according to control-flow branches and guard conditions, producing fine-grained ALBs that preserve both local logic and execution context.

construction and reference-code ALB decomposition can be performed offline and stored, taking approximately 0.99 ms/sample and 1.71 ms/sample on average respectively. Testbench simulation feedback, which is required for reward computation during GRPO training, takes approximately 71.96 ms/sample and is therefore not an additional cost specific to CirCA. Cone matching takes approximately 30.78 ms/sample on average, or 11.72 ms per signal. The overhead of advantage correction is negligible because it only performs simple tensor reweighting based on the existing reward and responsible-ALB mask, without additional simulation or Verilog analysis. Therefore, relative to vanilla GRPO, CirCA’s additional online computation consists primarily of test-code ALB decomposition and cone matching, resulting in an average overhead of approximately 32.49 ms/sample.

Training Dynamics To analyze RL training dynamics, we save checkpoints every 10 steps and evaluate them on the benchmarks. Figure 3 shows the evolution of *pass@1* over 400 training steps. The 210-step checkpoint is used as the final model in the main results, as it provides strong and balanced performance across tasks, though it is not necessarily the best checkpoint on every benchmark. Therefore, Figure 3 reports both the 210-step performance and the best performance achieved during training.

The training dynamics show clear instability in the Line-Level and Concurrency-Level variants. After around 150 steps, both methods degrade on

Spec-to-RTL and RTLLM Benchmark, suggesting that overly fine-grained or coarse-grained credit assignment with advantage penalization may lead to unstable RL optimization. Their improvements on Code-Completion are also limited, with the Line-Level showing a sharp decline in later stages. In contrast, the other variants generally achieve more stable improvements on VerilogEval-v2, with CirCA showing the strongest gains. On the more challenging RTLLM Benchmark, several methods degrade after around 170 steps, possibly due to distribution mismatch between training data and evaluation tasks. Since the same models continue to improve on VerilogEval-v2, the RTLLM degradation is more likely caused by benchmark-specific generalization difficulty rather than a general collapse of RL training.

A.3 Algorithmic Details of CirCA

Atomic Logic Block Decomposition. Algorithm 1 summarizes the procedure for decomposing Verilog code into Atomic Logic Blocks (ALBs). We first clean the code by removing comments while preserving string literals. The cleaned code is then passed to a macro-level splitter, which scans the code line by line and classifies each region using keyword-triggered patterns and boundary matching. Specifically, module headers are collected from the module keyword until the closing `);`, declarations and continuous assignments are grouped until statement-ending semicolons, and procedural blocks are collected by balancing `begin/end`, `case/endcase`,

Algorithm 1 Atomic Logic Block Decomposition

Require: Raw Verilog code x
Ensure: Marked code $\tilde{x}$ and ALB set $\mathcal{B}$

- 1: $x \leftarrow \text{REMOVECOMMENTS PRESERVESTRINGS}(x)$ $\triangleright$ Remove comments without corrupting string literals
- 2: $\mathcal{C} \leftarrow \text{MACROSPLIT}(x)$ $\triangleright$ Split into structural chunks, assignments, generate blocks, and procedural blocks
- 3: $\mathcal{B} \leftarrow \emptyset$
- 4: **for all** chunk $c \in \mathcal{C}$ **do**
- 5: **if** c is an ALB-level non-procedural chunk **then**
- 6: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\text{CREATEALB}(c)\}$ $\triangleright$ Keep standalone executable logic as one ALB
- 7: **else if** c is a procedural block **then**
- 8: $\mathcal{P} \leftarrow \text{REFINEPROCESSBLOCK}(c)$
- 9: **if** $|\mathcal{P}| > 1$ **then**
- 10: **for all** sub-block $p \in \mathcal{P}$ **do**
- 11: $\chi_p \leftarrow \text{NORMALIZECONDITION}(p)$
- 12: $(L_p, R_p) \leftarrow \text{EXTRACTLHSRHS}(p)$
- 13: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\text{CREATEALB}(p, \chi_p, L_p, R_p)\}$
- 14: **end for**
- 15: **else**
- 16: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\text{CREATEALB}(c)\}$ $\triangleright$ Fallback when no meaningful refinement is found
- 17: **end if**
- 18: **end if**
- 19: **end for**
- 20: $\mathcal{B} \leftarrow \text{RENUMBERALBS}(\mathcal{B})$
- 21: $\tilde{x} \leftarrow \text{INSERTMARKERS}(x, \mathcal{B})$ $\triangleright$ Insert markers without changing Verilog semantics
- 22: **return** $\tilde{x}, \mathcal{B}$

generate/endgenerate, and fork/join scopes. In this way, the splitter separates the program into module headers, declarations, continuous assignments, generate blocks, procedural blocks, and other standalone statements.

For procedural blocks, such as always, always_ff, always_comb, and initial blocks, we further refine their internal structures via Algorithm 2. The refiner first removes the outer procedural header and enclosing begin/end wrapper, and then scans the remaining body while tracking nested begin/end and case/endcase depths. The body is split only at top-level sequential boundaries, which avoids breaking assignments inside nested branches or case items. Each resulting segment is then recursively expanded according to its syntactic role. For if/else-if/else branches, the refiner extracts the branch guard as the local control condition. For case statements, it creates one candidate sub-block for each case item using the case selector and item value as the branch condition. For wrapper blocks, the outer wrapper is removed and the inner body is refined recursively. The implementation also separates default pre-assignments before case statements from the following case-specific updates, since such defaults often serve as

Algorithm 2 Refining a Procedural Block

Require: c : a procedural block
Ensure: $\mathcal{P}$: refined assignment-containing sub-blocks

- 1: $\mathcal{S} \leftarrow \text{STRIPOUTERWRAPPER}(c)$ $\triangleright$ Remove procedural header and redundant begin/end wrappers
- 2: **if** $\text{HASDEFAULTPREASSIGNCASE}(\mathcal{S})$ **then**
- 3: $\mathcal{Q} \leftarrow \text{SPLITDEFAULTPREASSIGNCASE}(\mathcal{S})$ $\triangleright$ Separate default update logic before case branches
- 4: **else**
- 5: $\mathcal{Q} \leftarrow \text{SPLITTOPLEVELSTATEMENTS}(\mathcal{S})$ $\triangleright$ Split only at top-level control-flow boundaries
- 6: **end if**
- 7: $\mathcal{E} \leftarrow \text{RECURSIVEEXPANDWRAPPERS}(\mathcal{Q})$ $\triangleright$ Expand nested if/case/wrapper blocks while preserving code spans
- 8: $\mathcal{P} \leftarrow \emptyset$
- 9: **for all** branch $br \in \mathcal{E}$ **do**
- 10: **if** $\text{HASREALASSIGNMENT}(br.\text{lines})$ **then**
- 11: $\chi \leftarrow \text{NORMALIZECONDITION}(br.\text{cond})$
- 12: $(L, R) \leftarrow \text{EXTRACTLHSRHS}(br.\text{lines})$
- 13: $t \leftarrow \text{CLASSIFYCONDITION}(\chi)$ $\triangleright$ e.g., reset, enable, FSM-state, or generic branch
- 14: $idx \leftarrow \text{ADJUSTMARKERPOSITION}(br)$ $\triangleright$ Align marker with branch entries such as else if
- 15: $p \leftarrow \text{CREATESUBBLOCK}(br, \chi, L, R, t, idx)$
- 16: $\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}$
- 17: **end if**
- 18: **end for**
- 19: $\mathcal{P} \leftarrow \text{SORTBYLINEINDEX}(\mathcal{P})$
- 20: **return** $\mathcal{P}$

independent update logic.

During recursive refinement, outer control contexts are merged into inner branches to obtain the effective condition of each candidate ALB. If a procedural block cannot be further split into multiple assignment-containing sub-blocks, we keep it as a single ALB. Finally, we insert lightweight block markers into the original code without changing the code content, which allows downstream modules to map tokens back to their corresponding ALBs during credit assignment 4.

Semantics-Aware Testbench Construction We summarize the implementation details of semantics-aware testbench construction. Given a reference Verilog design and a generated design, we first clean the reference code by removing comments, and rename the top module of the reference design with a golden prefix. This allows the reference design and the generated design to be instantiated in the same testbench without module-name conflicts.

We then extract the top-level interface, including input/output ports and parameters. The implementation first attempts to obtain robust port information through Yosys-based parsing, and falls back to regex-based parsing when tool extraction fails. Based on the extracted inputs and the cleaned reference code, we identify control signals and

design-specific structures. Clock signals are detected from edge-triggered always blocks and naming patterns. Reset candidates are identified from reset-like names, and their polarity is determined from sensitivity lists and reset conditions when possible. Enable-like controls are identified from common naming and interface patterns.

For sequential and protocol-driven designs, we further perform behavioral analysis. FSM-style designs are detected by searching for state-like registers and parameterized state encodings. The implementation also estimates constrained input ranges from comparison predicates, and identifies UART-style ports such as RX, TX, start, done, and tick signals when applicable.

The generated testbench instantiates both the renamed reference design and the generated design with shared inputs. During simulation, it applies valid reset and enable sequences, drives unconstrained inputs using frequency-sweep, range-constrained, or randomized exploration, and generates protocol-valid serial transactions for UART-like designs. At each checking step, the testbench compares every output signal between the reference and generated designs and records per-signal mismatch counts. These counts are later converted into output-level error rates and used for reward computation and cone matching.

Fingerprint-Guided Cone Matching We summarize the implementation of fingerprint-guided cone matching in Algorithm 3. Given the ALB-marked reference and generated Verilog designs, we first construct a signal dependency graph for each design. Each vertex corresponds to a signal or a signal-update event, and each directed edge represents a data dependency from an RHS signal to an LHS signal. Since the ALB markers are preserved during parsing, every update event can be mapped back to its corresponding ALB.

For each output signal identified as erroneous by simulation, we trace backward from the output node to construct its dependency cone in both the reference and generated graphs. Each event node in the cone is annotated with a logic fingerprint, consisting of its trigger condition, control context, and local operation. Constants, parameter names, simple Boolean expressions, and sliced dependencies are canonicalized before fingerprint construction, so that semantically equivalent but syntactically different expressions are less likely to be treated as mismatches.

Algorithm 3 Fingerprint-Guided Cone Matching

Require: ALB-marked reference code x_r , ALB-marked generated code x_g , erroneous outputs $\mathcal{Y}_{err}$

Ensure: Responsible generated ALBs $\mathcal{B}_{err}$

```

1:  $G_r \leftarrow \text{BUILDDependencyGRAPH}(x_r)$ ,  $G_g \leftarrow \text{BUILDDependencyGRAPH}(x_g)$   $\triangleright$  Build signal-event graphs with ALB mappings
2:  $\Phi_r \leftarrow \text{BUILDFINGERPRINTS}(G_r)$ ,  $\Phi_g \leftarrow \text{BUILDFINGERPRINTS}(G_g)$   $\triangleright$  Canonicalize triggers, conditions, operations, and dependencies
3:  $\mathcal{H}_r \leftarrow \emptyset$ 
4: for all ALB  $b_r$  in reference code do
5:    $F_r \leftarrow \{\Phi_r(e) \mid e \in b_r\}$ 
6:    $F_r \leftarrow \text{SORT}(F_r)$ 
7:    $h_r \leftarrow \text{HASH}(F_r)$ 
8:    $\mathcal{H}_r[h_r] \leftarrow \mathcal{H}_r[h_r] \cup \{b_r\}$ 
9: end for
10:  $\mathcal{V} \leftarrow \emptyset$ 
11: for all ALB  $b_g$  in generated code do
12:    $F_g \leftarrow \{\Phi_g(e) \mid e \in b_g\}$ 
13:    $F_g \leftarrow \text{SORT}(F_g)$ 
14:    $h_g \leftarrow \text{HASH}(F_g)$ 
15:   if  $\mathcal{H}_r[h_g] \neq \emptyset$  then
16:      $b_r \leftarrow \text{POPONE}(\mathcal{H}_r[h_g])$ 
17:      $\mathcal{V} \leftarrow \mathcal{V} \cup \{b_g\}$   $\triangleright$  Coarse stage
18:   end if
19: end for
20:  $\mathcal{B}_{err} \leftarrow \emptyset$ 
21: for all  $y \in \mathcal{Y}_{err}$  do
22:    $\mathcal{C}_r \leftarrow \text{TRACECONE}(G_r, y)$ ,  $\mathcal{C}_g \leftarrow \text{TRACECONE}(G_g, y)$ 
23:   if  $\text{CONEHASH}(\mathcal{C}_r, \Phi_r) = \text{CONEHASH}(\mathcal{C}_g, \Phi_g)$  then
24:     continue  $\triangleright$  The whole output cone is matched
25:   end if
26:    $\mathcal{M}_y \leftarrow \text{ISOMATCH}(y, y, G_r, G_g, \Phi_r, \Phi_g)$   $\triangleright$  Fine stage
27:    $\mathcal{B}_{err} \leftarrow \mathcal{B}_{err} \cup (\mathcal{M}_y \setminus \mathcal{V})$ 
28: end for
29: return  $\mathcal{B}_{err}$ 

30: function ISOMATCH( $u, v, G_r, G_g, \Phi_r, \Phi_g$ )
31:   if VISITED( $u, v$ ) then
32:     return  $\emptyset$ 
33:   end if
34:   MARKVISITED( $u, v$ )
35:   if TYPEWIDTH( $G_r, u$ )  $\neq$  TYPEWIDTH( $G_g, v$ ) then
36:     return BLAMESIGNAL( $G_g, v$ )
37:   end if
38:   if ISINPUT( $G_r, u$ ) or ISINPUT( $G_g, v$ ) then
39:     return  $\emptyset$  if  $u = v$ , else BLAMESIGNAL( $G_g, v$ )
40:   end if
41:    $E_r \leftarrow \text{DRIVINGEVENTS}(G_r, u)$ 
42:    $E_g \leftarrow \text{DRIVINGEVENTS}(G_g, v)$ 
43:    $\mathcal{P}, U_g \leftarrow \text{PAIREVENTS}(E_r, E_g, \Phi_r, \Phi_g)$   $\triangleright$  Pair events by fingerprint similarity
44:    $\mathcal{M} \leftarrow \text{BLAMEEVENTS}(G_g, U_g)$   $\triangleright$  Unpaired generated events are suspicious
45:   for all  $(e_r, e_g) \in \mathcal{P}$  do
46:     if  $\Phi_r(e_r) \neq \Phi_g(e_g)$  then
47:        $\mathcal{M} \leftarrow \mathcal{M} \cup \text{BLAMEEVENT}(G_g, e_g)$ 
48:     continue
49:   end if
50:    $\mathcal{D} \leftarrow \text{PAIREDDEPS}(e_r, e_g)$ 
51:   for all  $(a, b) \in \mathcal{D}$  do
52:      $\mathcal{M} \leftarrow \mathcal{M} \cup \text{ISOMATCH}(a, b, G_r, G_g, \Phi_r, \Phi_g)$ 
53:   end for
54: end for
55: return  $\mathcal{M}$ 
56: end function

```

The matching procedure follows the coarse-to-fine strategy described in Section 3.4. We first compute ALB-level hashes by aggregating the fingerprints of all update events within each ALB. Generated ALBs whose hashes match reference ALBs are regarded as structurally verified and are pruned from later localization. For the remaining cones, we recursively compare reference and generated signal pairs starting from the erroneous output. At each recursive step, the matcher checks type and width compatibility, pairs multiple driving events according to their fingerprint similarity, compares the paired event fingerprints, and then recursively verifies their upstream dependencies.

When a mismatch is detected during recursive comparison, the generated-side ALB associated with the mismatched event is added to the responsible block set. Declaration-level mismatches, trigger mismatches, branch-condition mismatches, and assignment-level mismatches are mapped to their corresponding generated ALBs according to the source locations recorded during parsing. The resulting responsible ALB set is then used by the subsequent credit-assignment step.