RTL-INTERACT: TRAINING EDA-INTERACTIVE MODELS FOR MULTI-TASK RTL DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models offer a promising route to register-transfer-level (RTL) design, but reliable implementations require checking and revision based on electronic design automation (EDA) tools' feedback. Existing RTL-specialized models often lack learned interaction with such EDA tools or rely on separate generation and repair components. We introduce RTL-INTERACT, a multi-task post-training framework that trains a single RTL-specialized model to generate RTL, construct task-appropriate EDA checks, and revise implementations using tool feedback. The framework combines automatically constructed multi-task demonstrations with staged supervision from chain-of-thought to tool-integrated reasoning. We further introduce Pattern-Aware Policy Optimization (PAPO) for multi-task reinforcement learning. PAPO augments Group Relative Policy Optimization with correctness-gated rewards that favor concise successful checking while rewarding effective revision, and uses task grouping to preserve coverage across objectives. On CVDP, RTL-Interact-32B achieves 48.7% weighted Pass@1 and 68.9% Pass@5, surpassing its teacher model, DeepSeek-V3.2, while RTL-Interact-8B achieves 44.9% weighted Pass@1. Additionally, we show that PAPO can reduce average tool calls by 63.8% while improving Pass@1. Both models also outperform all RTL-specialized baselines on VerilogEval v2 and RTLLM v2, demonstrating the effectiveness of our method.

1 INTRODUCTION

Large language models (LLMs) offer a promising route to automating register-transfer-level (RTL) design. Generating usable RTL requires not only syntactically valid code but also compliance with the functional requirements and implementation constraints stated in a specification. Electronic design automation (EDA) tools provide feedback on whether candidate RTL compiles, behaves as intended in simulation, and meets task-specific design objectives. Such feedback allows a model to check and revise its design before submitting a solution.

Existing RTL-specialized models improve code generation through domain supervision or EDA-derived training rewards, but do not explicitly learn to use EDA feedback to revise implementations during inference (Liu et al., 2023b; Zhao et al., 2024a; Zhu et al., 2025). Tool-assisted workflows organize generation, checking, and revision through prompting and agent coordination, but rely on the underlying models' existing RTL generation and repair capabilities (Thakur et al., 2023; Zhao et al., 2024b). Other approaches train models to revise RTL using EDA feedback, but require separate generation and repair adapters or cooperating models, adding deployment and coordination complexity (Cui et al., 2024; Yu et al., 2026). These limitations motivate training a single RTL-specialized model for both RTL generation and feedback-based revision, without separate adapters or cooperating models. Figure 1 contrasts the solution processes of RTL generation and RTL reasoning models with those of EDA-interactive RTL models. Achieving this across multiple RTL tasks requires learning task-appropriate checks and effective revisions while avoiding redundant tool calls.

To this end, we propose RTL-INTERACT, a post-training framework for training a single RTL-specialized model to generate RTL, construct task-appropriate EDA checks, and revise candidate implementations using EDA feedback. The framework consists of multi-task data construction, CoT-to-TIR staged supervision, and reinforcement learning (RL) with pattern-aware policy optimization. Starting from the specification-RTL pairs in the open-source dataset, we automatically construct tasks

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

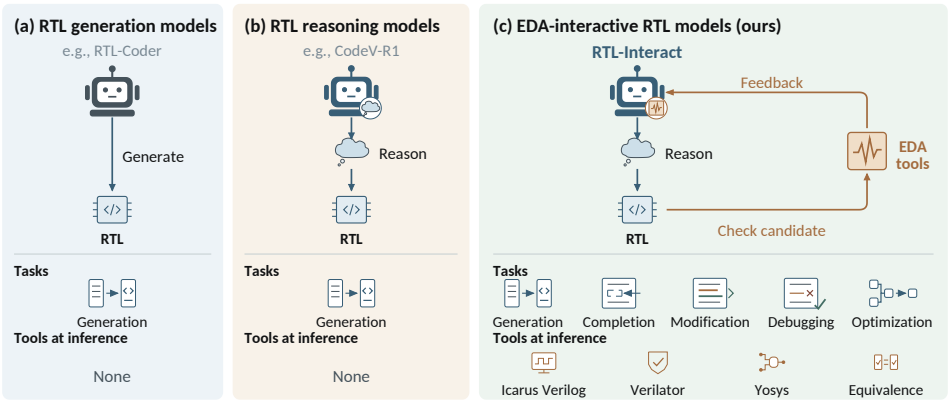

Figure 1: Representative solution processes of (a) RTL generation models, such as RTL-Coder (Liu et al., 2023b); (b) RTL reasoning models, such as CodeV-R1 (Zhu et al., 2025); and (c) our EDA-interactive RTL model.

with different code contexts and design objectives, then generate and filter teacher demonstrations using task-specific checks. Each tool-interaction demonstration records the teacher's candidate implementations, EDA requests, returned feedback, subsequent revisions, and final implementation.

To develop both RTL reasoning and effective use of EDA feedback, we apply CoT-to-TIR staged supervision to improve RTL generation and revision using EDA feedback within the same model. Tool-free chain-of-thought (CoT) examples and tool-integrated reasoning (TIR) demonstrations supervise different solution processes, making their organization an important training choice. We first train on CoT examples from the open-source dataset, covering RTL syntax, hardware semantics, and reasoning from specifications to implementations. We then train on multi-task TIR demonstrations to teach the model to construct EDA requests and use the returned feedback. At a matched template-token budget, this staged approach improves CVDP (Pinckney et al., 2025) Pass@1 by 1.0% over TIR-only supervision. Separate evaluations also show improvements in both RTL generation and revision using EDA feedback over CoT-only supervision.

Multi-task RL further optimizes these behaviors through interaction with the EDA environment. A reward based only on final correctness does not distinguish how a correct implementation was checked and revised. We introduce Pattern-Aware Policy Optimization (PAPO), which incorporates correctness-gated rewards for EDA-checking patterns and task grouping into Group Relative Policy Optimization (GRPO) (Shao et al., 2024). For solutions that pass final verification, the reward favors concise, successful checking while retaining positive rewards for successful revision. To maintain training coverage across different objectives, task grouping keeps prescribed task-group proportions among the samples retained for each policy update. Across the full RL stage, the 8B model improves CVDP Pass@1 by 5.4% while reducing the average number of tool calls by 63.8%.

On CVDP, RTL-Interact-32B surpasses its tool-enabled teacher, DeepSeek-V3.2, achieving 48.7% weighted Pass@1. RTL-Interact-8B achieves 44.9%, with both models outperforming other RTL-specialized models of comparable size. These gains extend to VerilogEval v2 (Liu et al., 2023a; Pinckney et al., 2024) and RTLLM v2, where both model sizes outperform other RTL-specialized baselines in Pass@1 across code completion and specification-to-RTL tasks.

Our contributions are as follows:

- We develop a multi-task post-training framework that uses task-specific teacher demonstrations and CoT-to-TIR staged supervision to train a single RTL-specialized model for RTL generation and revision using EDA feedback.
- We introduce PAPO, which incorporates correctness-gated rewards for EDA-checking patterns and task grouping into GRPO to improve multi-task RTL performance with fewer tool calls and shorter response length, achieving joint gains in both RTL correctness and EDA interaction efficiency.

- Our RTL-Interact-32B achieves 48.7% Pass@1 and 68.9% Pass@5 on CVDP, surpassing its teacher model DeepSeek-V3.2 on both metrics. Our models also outperform all RTL-specialized baselines with large gains in Pass@1 on VerilogEval v2 and RTLLM v2. We will release both models, together with the training data and code.

2 Related Work

RTL generation model training. RTLCoder and CodeV use domain data and supervised fine-tuning; BetterV adds discriminative guidance, and CraftRTL constructs synthetic data and repair examples (Liu et al., 2023b; Zhao et al., 2024a; Pei et al., 2024; Liu et al., 2024a). CodeV-R1 combines reasoning distillation with testbench-based verifiable rewards, while ChipSeek uses hierarchical EDA-derived rewards to optimize functional correctness and hardware efficiency (Zhu et al., 2025; Chen et al., 2026b). These methods primarily use EDA tools to evaluate generated designs. Our RTL-INTERACT instead learns to use EDA tools during inference to check and revise candidates based on feedback before returning the final RTL, thereby improving RTL task performance.

Tool-assisted RTL agents and workflows. AutoChip organizes iterative generation using compilation and simulation feedback, while RTLFixer combines interactive repair with retrieved guidance (Thakur et al., 2023; Tsai et al., 2023). MAGE, VerilogCoder, and LocalV organize generation and debugging through role coordination, graph-based planning, and information locality, respectively (Zhao et al., 2024b; Ho et al., 2025; Lyu et al., 2026). ACE-RTL combines a specialized generator with a general-purpose reflector (Deng et al., 2026). These approaches primarily organize model capabilities through inference-time prompting and workflow design. Our RTL-INTERACT instead strengthens the model's own ability to generate RTL and revise it using EDA feedback through domain-specific training, complementing these inference-time approaches.

Training for tool interaction. ReAct interleaves reasoning and actions through prompting and trajectory fine-tuning, Toolformer learns API use through self-supervision, and ToRA trains reasoning models on tool-interaction trajectories (Yao et al., 2023; Schick et al., 2023; Gou et al., 2023). Within RTL design, OriGen trains separate generation and repair adapters to correct compilation errors, while ChipMATE jointly trains Verilog and Python reference-model agents for cross-verification (Cui et al., 2024; Yu et al., 2026). These methods learn feedback-based correction around compiler diagnostics or cross-language behavioral comparison. Our RTL-INTERACT instead trains a single model to construct task-appropriate EDA checks and revise RTL using their feedback across functional and optimization tasks, reducing the deployment overhead for multiple models or adapters.

3 Method

3.1 Task Formulation and EDA Environment

A task x specifies hardware behavior and may include partial RTL or an existing design. A policy π_θ produces final RTL y through a trajectory $\tau = (z_0, a_0, o_0, \ldots, z_L, y)$ with L tool requests, where z_ℓ, a_ℓ, and o_ℓ denote model-generated reasoning, tool requests, and environment observations. Each request specifies a tool and arguments. The policy conditions on the task and history to reason, request checks, or return RTL satisfying the requirements and implementation constraints. With no requests, this reduces to direct RTL generation. Here ℓ indexes interactions, not tokens.

The environment exposes simulation, lint-checking, synthesis-analysis, and simulation-based equivalence-checking interfaces (Appendix A, Table 6). Icarus Verilog, Verilator, and Yosys provide compilation/simulation, lint-checking, and synthesis functions (Icarus Verilog Contributors, 2026; Verilator Contributors, 2026; YosysHQ, 2026). Simulation-based equivalence checking (hereafter equivalence checking) uses CodeV-R1's rule-based testbench generator to compare candidate and reference outputs under shared stimuli (Zhu et al., 2025). These testbenches serve only equivalence checking and are distinct from model-authored simulation testbenches. Testbenches and execution logs used for independent final verification are not supplied as model context.

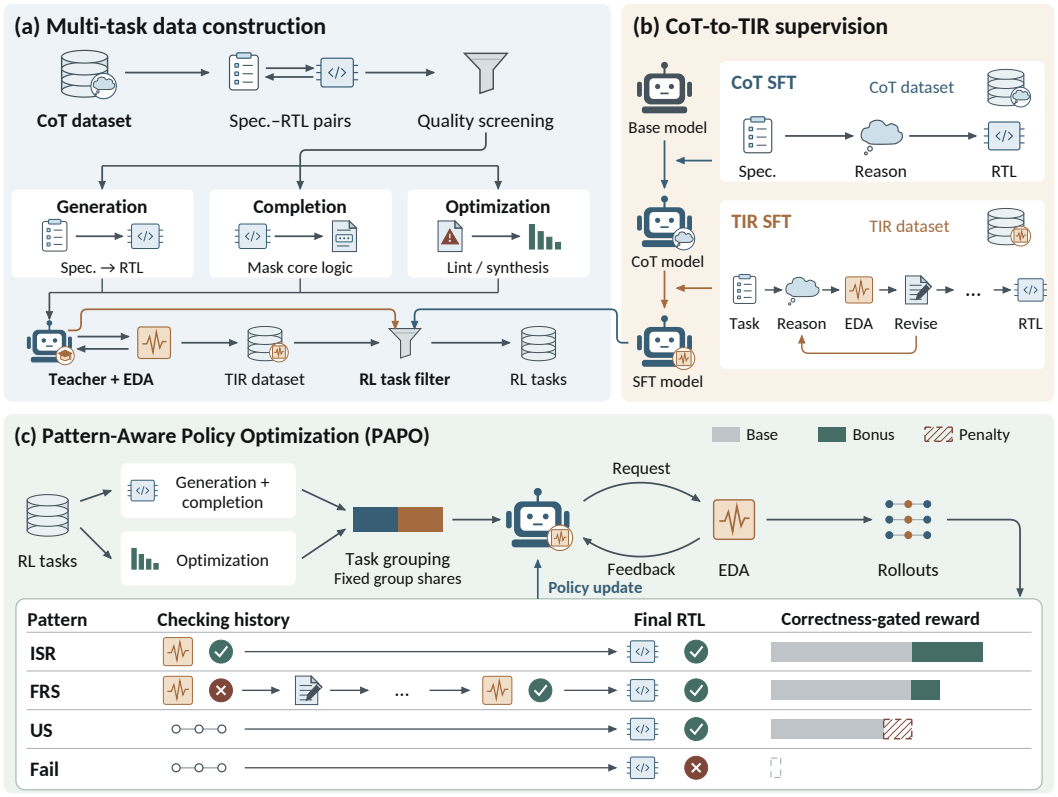

Figure 2: Overview of RTL-INTERACT: (a) multi-task TIR data construction and RL task selection starting from the CoT dataset; (b) CoT-to-TIR staged supervision; and (c) PAPO with pattern-aware trajectory rewards and task grouping. Reward bars are schematic; incorrect final RTL receives zero reward.

3.2 MULTI-TASK DATA CONSTRUCTION

Figure 2(a) summarizes data construction. We use the open-source CodeV-R1-dataset (Zhu et al., 2025) for CoT supervision and its specification–RTL pairs as the source for multi-task TIR data. We use DeepSeek-V3.2 as the teacher for quality scoring and TIR trajectory distillation. The teacher scores pairs along four dimensions: specification–implementation alignment, hardware-semantic precision, behavioral reasoning value, and the ability to derive discriminating checks. Pairs above the quality threshold form a shared source pool (scoring prompt and threshold in Appendix A.1). Each distilled trajectory retains the teacher's reasoning, tool requests, actual observations, subsequent revisions, and final RTL. Below, we describe the data construction and trajectory acceptance criteria for each task.

Generation: constructing an implementation from a specification. Given a specification, the teacher writes candidate RTL and a simulation testbench, then uses simulation diagnostics to revise the design as needed. We retain trajectories whose final RTL passes the teacher-authored testbench and is synthesizable by Yosys.

Completion: recovering logic in an existing design. An LLM identifies core logic in the reference RTL, which we mask while retaining the surrounding design and interfaces as context. The teacher fills in the missing logic, checks the completed design through the simulation interface, and revises it using the returned feedback. We retain a trajectory only if its final RTL passes equivalence checking against the unmasked reference RTL, using the checker described in Section 3.1. Generation and completion thus share a simulation interface but impose different implementation constraints.

Optimization: improving a design while preserving its behavior. We run EDA checks on the reference designs in the shared source pool and select those with lint warnings or relatively high synthesis-based implementation costs. These designs define tasks with explicit lint or implementation-metric objectives. The teacher iteratively modifies each design using the corresponding diagnostics and equivalence-checking feedback. We retain a trajectory only if its final RTL passes equivalence checking against the original design and meets the requested improvement: making a non-clean design lint-clean, or exceeding the prescribed improvement threshold for the target synthesis-based PPA proxy. These task-specific acceptance criteria yield 13,480 TIR trajectories. Appendix A.1 provides further data details.

RL task selection. Let $\mathcal{D}_{\mathrm{TIR}}$ denote the task pool corresponding to the retained TIR demonstrations. We sample five attempts per task using the two-stage SFT model and let $c_{\mathrm{SFT}}(x) \in \{0, \ldots, 5\}$ count successful attempts. Functional correctness is checked by comparing the student's final RTL against the reference RTL with the same equivalence checker used for completion and optimization data. Optimization success additionally requires the specified improvement. We exclude tasks solved in all five attempts:

$$\mathcal{D}_{\mathrm{RL}} = \{x \in \mathcal{D}_{\mathrm{TIR}} : c_{\mathrm{SFT}}(x) < 5\}. \tag{1}$$

This retains approximately 8K tasks for fresh policy rollouts, excluding tasks already solved consistently by the student.

3.3 Staged Supervision for a Single Policy

We use CoT-to-TIR staged supervision (Figure 2(b)). The CoT stage reinforces RTL syntax, hardware knowledge, and reasoning from specifications to implementations through direct specification–reasoning–code examples. Building on this initialization, TIR supervision teaches the same model to construct EDA requests and use the returned feedback to check and revise its implementation.

Table 1: Controlled CoT/TIR supervision on 8B. Tool use is the share of trajectories containing a tool request; Avg. tool turn is the mean number of tool-call turns in trajectories. Token counts denote the total number of chat-template tokens processed across all training epochs; for sequential configurations, the budget is shared across both stages.

Configuration	Epochs	CVDP P@1 (%)	Tool use	Avg. tool turn
Base model	0	17.2	1.6%	0.0
CoT (450M tokens)	3	23.8	0.0%	0.0
TIR (450M tokens)	3	24.8	96.0%	4.0
CoT (900M tokens)	3	26.5	0.0%	0.0
TIR (900M tokens)	3	28.5	98.0%	3.7
Naive mix (900M tokens)	3	28.2	79.8%	2.7
TIR→CoT (900M tokens)	3	25.5	0.0%	0.0
CoT→TIR (900M tokens)	3	**29.5**	97.4%	4.3

Table 1 compares supervision configurations at matched template-token budgets. The total budgets are 450M or 900M chat-template tokens across all training epochs. For the 900M-token mixed and sequential configurations, CoT and TIR each contribute 450M tokens; sequential configurations differ only in stage order. CoT→TIR yields 97.4% tool use, versus 0.0% for TIR→CoT, supporting this order for retaining tool-use behavior. At the same 900M-token budget, CoT→TIR reaches 29.5% Pass@1, exceeding TIR-only training (28.5%) and naive CoT/TIR mixing (28.2%). These results support CoT-to-TIR staged supervision for direct RTL generation and feedback-guided revision; Section 4.4 evaluates both abilities separately using the full-data checkpoints.

3.4 Pattern-Aware Policy Optimization

We introduce Pattern-Aware Policy Optimization (PAPO) to distinguish trajectories with the same final correctness but different EDA-checking histories. Here, a checking pattern characterizes whether a trajectory succeeds on its first check, succeeds after feedback-driven revision, or lacks a successful task-appropriate check. Starting from the two-stage SFT model, PAPO combines correctness-gated, pattern-conditioned rewards with task-grouped policy updates within GRPO (Shao et al., 2024) (Figure 2(c)).

At update k, let $\mathcal{B}_k^{\mathrm{F}}$ and $\mathcal{B}_k^{\mathrm{O}}$ contain the retained generation/completion and optimization trajectories, respectively. For trajectory τ_i sampled for task x_i, PAPO uses the following objective and reward:

$$\mathcal{J}_{\mathrm{PAPO}}(\theta) = \underbrace{\sum_{g\in\{\mathrm{F},\mathrm{O}\}} \frac{\omega_{g,k}}{|\mathcal{B}_k^g|} \sum_{i\in\mathcal{B}_k^g}}_{\text{task-grouped updates}} \frac{1}{|\mathcal{M}_i|} \sum_{t\in\mathcal{M}_i} c_{i,t} \min\Big\{\rho_{i,t}\widehat{A}_i,\ \mathrm{clip}(\rho_{i,t}, 1-\varepsilon_-, 1+\varepsilon_+)\,\widehat{A}_i\Big\}, \quad (2)$$

$$\widehat{A}_i = \frac{R_{x_i}(\tau_i) - \bar{R}_{x_i}}{\sigma_{x_i} + \epsilon}, \qquad R_x(\tau) = \underbrace{V_x(y)\left[1 + \delta_{p_x(\tau)}\right]}_{\substack{\text{correctness-gated,}\\ \text{pattern-conditioned reward}}}. \quad (3)$$

Equation (3) connects the checking pattern to the advantage used in Equation (2). Definitions of the standard policy-update quantities are provided in Appendix A.2.

Pattern-aware trajectory rewards. Binary final-correctness rewards cannot distinguish EDA-checking histories. We therefore condition rewards on the EDA-checking pattern $p_x(\tau)$ through adjustment $\delta_{p_x(\tau)}$, gated by final verification (Equation (3)). A checking round comprises the checks needed to assess a candidate against its task objective. *Immediate Self-check Resolution (ISR)* succeeds in one task-appropriate checking round; *Feedback-driven Refinement Success (FRS)* uses multiple checking rounds and ends with a successful self-check; *Unsuccessful Self-checking (US)* covers missing, incomplete, or unsuccessful final checks. Self-check success is distinct from independent final verification. Independent final verification requires functional correctness against the reference and any specified optimization improvement for $V_x(y) = 1$; otherwise reward is zero. These rewards distinguish EDA-checking histories among final-correct solutions, favoring concise successful checking while retaining positive rewards for successful revision. Appendix B gives matching rules and reward values.

Task grouping. Random sampling from the imbalanced RL pool may bias training toward more frequent task types. We therefore use task grouping to maintain prescribed proportions among the retained trajectories in each update, continuing sampling until each group meets its quota. Generation and completion share a functional-correctness objective and a simulation interface; optimization adds an improvement objective and uses synthesis-analysis and equivalence-checking interfaces. Sampling quotas keep the retained optimization share $\alpha_k = |\mathcal{B}_k^{\mathrm{O}}|/(|\mathcal{B}_k^{\mathrm{F}}| + |\mathcal{B}_k^{\mathrm{O}}|)$ near the target α, up to integer rounding. In Equation (2), $\omega_{\mathrm{F},k} = 1 - \alpha_k$ and $\omega_{\mathrm{O},k} = \alpha_k$ express these sample proportions in the batch average; they are not applied again.

4 Experiments

This section aims to answer the following research questions:
RQ1 (Overall Performance): How effective is the RTL-Interact training framework across RTL benchmarks?
RQ2 (PAPO Ablation): Do pattern-aware rewards and task grouping improve RTL performance?
RQ3 (Training Stages): How do the training stages affect RTL generation and feedback-guided revision?
RQ4 (Interaction Efficiency): How does RL change tool use and response length?

4.1 Evaluation Setup

Benchmarks. We use CVDP's non-agentic split (Pinckney et al., 2025): 302 problems covering completion, specification-to-RTL, modification, optimization, and debugging. VerilogEval v2 (Liu et al., 2023a; Pinckney et al., 2024) provides 156 problems each for code completion (CC) and specification-to-RTL (SR); RTLLM v2 (Lu et al., 2024; Liu et al., 2024b) contains 50 specification-to-RTL tasks.

Baselines. We select CodeV-R1-7B, ChipSeek-R1-Qwen2.5, SiliconMind-V1-8B, and InCoder-32B as RTL-specialized baselines and include ChipMATE's V+P workflow on CVDP. We include the DeepSeek-V3.2 teacher (DeepSeek-AI, 2025), Qwen3-8B/32B bases (Yang et al., 2025), and our SFT checkpoints. Table 2 also compares seven general-purpose frontier models selected around the teacher's release; Appendix A.3 lists them and their references.

Table 2: CVDP non-agentic code-generation results (%). Entries are Pass@1/Pass@5; Overall is weighted by category size. Size denotes total model parameters; – indicates undisclosed size. Tool marks whether EDA tools are available. Best and second-best results for each metric are shown in **bold** and underlined, respectively; ties share the same formatting. † marks the tool-using distillation teacher; ChipMATE uses its V+P cross-verification workflow.

Model	Size	Tool	Overall	CID002	CID003	CID004	CID007	CID016
General-purpose frontier models								
Claude-Sonnet-4.5	–	✗	46.6/59.6	44.3/53.2	48.2/61.5	49.1/60.0	39.5/62.5	53.7/68.6
		✓	**51.5**/63.6	**46.8**/55.3	**56.9**/**66.7**	**54.2**/61.8	43.5/70.0	57.1/74.3
GPT-5.2	–	✗	41.5/57.6	36.6/50.0	45.1/57.7	42.9/61.8	35.0/57.5	52.0/71.4
		✓	**51.5**/63.9	42.8/55.3	51.8/61.5	50.9/61.8	**63.5**/**82.5**	61.7/74.3
GLM-4.7	355B	✗	40.1/61.3	37.7/56.4	38.2/62.8	34.9/50.9	37.0/65.0	62.3/82.9
		✓	41.5/62.3	38.7/57.4	44.9/**66.7**	38.5/56.4	36.0/57.5	52.6/80.0
LongCat-Flash-Thinking	560B	✗	38.1/57.0	34.7/51.1	38.5/60.3	33.5/49.1	33.5/60.0	58.9/74.3
		✓	39.5/58.6	33.0/50.0	43.3/60.3	34.9/54.5	38.0/67.5	57.1/74.3
MiniMax-M2.1	230B	✗	33.3/53.6	29.8/50.0	39.0/60.3	34.5/56.4	21.0/40.0	42.3/60.0
		✓	37.9/56.3	34.9/53.2	42.8/57.7	36.4/56.4	32.5/57.5	43.4/60.0
Qwen3-Max-Preview	–	✗	34.6/46.4	29.8/40.4	41.0/53.8	34.2/43.6	24.5/40.0	45.7/57.1
		✓	33.7/49.0	29.4/40.4	39.2/55.1	30.5/40.0	28.5/52.5	44.0/68.6
Gemini-2.5-Pro	–	✗	39.1/57.9	35.7/52.1	36.9/55.1	35.6/56.4	32.0/57.5	**66.3**/82.9
		✓	38.2/57.9	33.2/46.8	37.9/60.3	40.0/61.8	27.5/52.5	61.7/82.9
RTL-specialized models								
CodeV-R1 (Zhu et al., 2025)	7B	✗	31.7/46.4	32.3/47.9	42.3/59.0	29.1/45.5	17.0/25.0	26.9/40.0
		✓	23.2/42.7	20.9/40.4	34.6/53.8	27.3/50.9	8.0/22.5	15.4/34.3
ChipSeek-R1 (Chen et al., 2026b)	7B	✗	20.4/33.1	20.4/31.9	27.7/39.7	20.0/36.4	14.0/25.0	12.0/25.7
		✓	21.5/32.8	21.5/29.8	27.7/42.3	24.4/40.0	15.0/22.5	10.9/20.0
SiliconMind-V1 (Chen et al., 2026a)	8B	✗	33.7/51.3	30.6/46.8	41.5/59.0	33.5/54.5	22.0/35.0	38.3/60.0
		✓	33.7/52.0	31.3/46.8	40.5/59.0	34.9/52.7	22.0/40.0	36.6/62.9
InCoder (Yang et al., 2026)	32B	✗	24.4/46.0	25.1/45.7	24.4/48.7	27.3/41.8	14.0/32.5	29.7/62.9
		✓	28.1/47.7	30.0/47.9	31.0/53.8	28.7/47.3	16.0/30.0	29.1/54.3
ChipMATE (Yu et al., 2026)	V: 9B P: 9B	✓	23.8/36.1	23.8/33.0	30.3/43.6	24.4/38.2	14.5/25.0	19.4/37.1
Distillation teacher								
DeepSeek-V3.2†	671B	✗	36.8/57.3	34.5/54.3	38.7/60.3	36.4/49.1	32.0/57.5	45.1/71.4
		✓	46.7/65.2	38.5/55.3	48.2/64.1	44.4/65.5	60.0/80.0	53.7/77.1
Our models								
Qwen3 (base)	8B	✗	18.9/36.1	14.3/30.9	17.4/34.6	22.9/38.2	18.5/37.5	29.1/48.6
RTL-Interact (SFT)	8B	✓	39.5/60.6	33.6/48.9	46.2/64.1	37.1/65.5	44.5/67.5	38.3/68.6
RTL-Interact (RL)	8B	✓	44.9/63.2	37.0/56.4	47.7/61.5	46.5/65.5	52.5/77.5	48.6/65.7
Qwen3 (base)	32B	✗	27.2/47.7	22.8/41.5	31.5/47.4	28.0/49.1	26.5/52.5	29.1/57.1
RTL-Interact (SFT)	32B	✓	42.5/64.6	38.9/57.4	45.1/61.5	39.6/63.6	44.5/**82.5**	48.0/71.4
RTL-Interact (RL)	32B	✓	48.7/**68.9**	40.6/**60.6**	50.5/65.4	48.0/**69.1**	56.0/80.0	58.9/**85.7**

Evaluation protocol. Except for ChipMATE's V+P workflow, baselines are evaluated with and without the same EDA interfaces. We use five responses per problem at temperature 0.7 and top-p 0.95, reporting Pass@1/Pass@5 and category-weighted CVDP scores. All four RTL-specialized baselines cover both tool settings and VerilogEval v2 splits. ChipSeek uses the released Qwen2.5 checkpoint under our shared protocol. Further training and evaluation details are provided in Appendix A.

4.2 Main Results (RQ1)

RTL-Interact-32B surpasses its tool-enabled distillation teacher on CVDP. RTL-Interact-32B achieves 48.7% Pass@1 and 68.9% Pass@5, exceeding the tool-enabled DeepSeek-V3.2 teacher by 2.0% and 3.7%, respectively (Table 2). It also achieves the highest overall CVDP Pass@5 among the compared models. On VerilogEval v2 CC, the 32B model reaches 80.3% Pass@1 and 87.2% Pass@5, exceeding the teacher in both tool settings (Table 3).

Table 3: VerilogEval v2 and RTLLM v2 results (%). CC and SR denote code completion and specification-to-RTL. Tools indicate EDA-tool availability. Size denotes total model parameters. Best and second-best results in each metric are **bold** and underlined; ties at displayed precision share formatting. † marks the distillation teacher.

		VerilogEval v2 CC			VerilogEval v2 SR			RTLLM v2	
Model	**Size**	**P@1**	**P@5**	**Tools**	**P@1**	**P@5**	**Tools**	**P@1/P@5**	**Tools**
RTL-specialized models									
CodeV-R1 (Zhu et al., 2025)	7B	54.7	71.2	✗	53.3	67.3	✗	64.4/78.0	✗
		51.0	73.7	✓	63.3	78.2	✓	74.4/82.0	✓
ChipSeek-R1 (Chen et al., 2026b)	7B	64.4	71.2	✗	64.5	71.2	✗	74.0/82.0	✗
		64.7	70.5	✓	54.4	70.5	✓	71.2/76.0	✓
SiliconMind-V1 (Chen et al., 2026a)	8B	69.9	80.1	✗	69.2	80.8	✗	70.0/84.0	✗
		62.7	80.1	✓	64.7	82.7	✓	54.8/76.0	✓
InCoder (Yang et al., 2026)	32B	62.2	73.1	✗	61.5	74.4	✗	59.2/76.0	✗
		62.1	78.2	✓	66.0	78.2	✓	66.0/84.0	✓
Distillation teacher									
DeepSeek-V3.2†	671B	76.5	85.9	✗	71.5	85.9	✗	75.6/86.0	✗
		67.9	84.6	✓	**78.8**	**87.8**	✓	68.8/82.0	✓
Our models									
Qwen3 (base)	8B	46.5	62.8	✗	53.7	69.2	✗	45.2/62.0	✗
RTL-Interact (SFT)	8B	73.8	86.5	✓	72.4	84.6	✓	76.4/86.0	✓
RTL-Interact (RL)	8B	75.4	83.3	✓	75.1	84.6	✓	**77.6**/86.0	✓
Qwen3 (base)	32B	64.0	77.6	✗	70.3	83.3	✗	64.4/80.0	✗
RTL-Interact (SFT)	32B	76.7	86.5	✓	72.7	86.5	✓	76.0/**88.0**	✓
RTL-Interact (RL)	32B	**80.3**	**87.2**	✓	77.2	86.5	✓	77.2/**88.0**	✓

Both RTL-Interact model sizes outperform the evaluated RTL-specialized baselines of comparable size in Pass@1 across all three benchmarks. On CVDP, RTL-Interact-8B reaches 44.9%/63.2% Pass@1/Pass@5, exceeding the 7B–8B RTL-specialized baselines, while the 32B model exceeds InCoder-32B. Both also lead their corresponding baselines on both VerilogEval v2 splits and RTLLM v2 (Tables 2–3). The 32B model reaches 77.2%/86.5% Pass@1/Pass@5 on VerilogEval v2 SR. On RTLLM v2, 8B and 32B reach 77.6%/86.0% and 77.2%/88.0% Pass@1/Pass@5, respectively, exceeding all evaluated baselines in Pass@1. Relative to SFT, RL yields CVDP Pass@1 gains of 5.4% and 6.2% for 8B and 32B, with Pass@1 gains on VerilogEval v2 and RTLLM v2.

4.3 Ablation of PAPO (RQ2)

Both pattern-aware rewards and task grouping contribute to RTL performance. We examine the two components of PAPO on RTL-Interact-8B (Table 4). The *w/o pattern-aware reward* variant replaces the trajectory reward with binary final-verification reward while retaining task grouping. The *w/o task grouping* variant retains pattern-aware rewards but samples tasks randomly from the RL dataset, without enforcing task-group proportions among the trajectories retained for each update.

Table 4: RL component ablations for RTL-Interact-8B on CVDP. All configurations use tools; entries are Pass@1/Pass@5 (%). Bold marks the best value for each metric, including ties.

Configuration	**Overall**	**CID002**	**CID003**	**CID004**	**CID007**	**CID016**
Full method	**44.9/63.2**	**37.0/56.4**	47.7/61.5	**46.5/65.5**	**52.5/77.5**	**48.6/65.7**
w/o pattern-aware reward	39.7/59.9	36.6/55.3	46.2/64.1	40.7/61.8	34.0/57.5	38.9/62.9
w/o task grouping	41.8/60.3	36.0/52.1	**47.9/65.4**	44.0/**65.5**	41.5/60.0	40.6/62.9

The full method leads in overall Pass@1/Pass@5, with gains of 5.2%/3.3% over *w/o pattern-aware reward* and 3.1%/2.9% over *w/o task grouping*. We use an optimization-task share of $\alpha = 0.3$ among samples retained for each policy update; Appendix C compares task-group proportions.

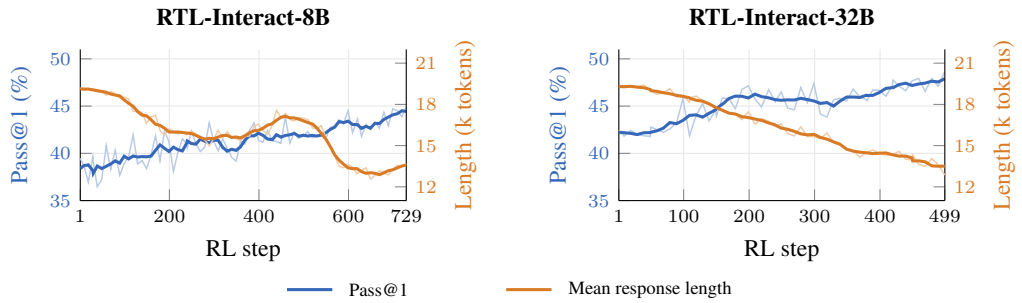

Figure 3: RL training dynamics for 8B and 32B. Pale solid lines show the series before smoothing.

4.4 Direct Generation and Feedback-Guided Revision (RQ3)

Staged supervision improves generation and feedback-guided revision, while RL further strengthens revision. Constructing candidate RTL and revising it using EDA feedback are two important capabilities for tool-assisted RTL design. We evaluate these capabilities separately on 100 selected CVDP tasks spanning all five categories. In direct generation, each model receives only the original task description and generates one RTL solution. In feedback-guided revision, all models receive the same fixed failing implementation and its EDA diagnostic for each task, then produce one revised implementation. Neither setting permits tool calls, allowing us to assess generation and revision separately from tool-use decisions. We compare the Qwen3-8B base, full CoT, two-stage SFT, and PAPO checkpoints using the CVDP evaluator. Detailed experimental settings and procedures are provided in Appendix A.3.

Table 5: Direct generation and single-shot revision success (%) on 100 tasks. All checkpoints are 8B.

Checkpoint	Direct generation	With EDA feedback
Base	17.0	14.0
CoT-SFT	23.0	19.0
CoT→TIR SFT	**27.0**	26.0
PAPO	26.0	**35.0**

Two-stage SFT raises generation/revision success from Base's 17.0%/14.0% to 27.0%/26.0%, also exceeding CoT's 23.0%/19.0% (Table 5). Together with the matched-budget study, this supports staged supervision for both abilities. RL further raises revision success to 35.0% while retaining comparable generation performance.

4.5 Interaction Efficiency (RQ4)

RL reduces tool calls and response length while improving RTL performance. Figure 3 shows that mean response length decreases by 28.8% for 8B and 33.5% for 32B over RL training. On CVDP, RTL-Interact-8B's mean tool calls per trajectory decrease from 4.7 to 1.7, a 63.8% reduction, while the truncation rate drops from 33.7% to 15.2%, a decrease of 18.5%. Detailed trajectory statistics are provided in Appendix A.4, Table 7. Together with the accuracy gains in Section 4.2, these results show that improved RTL performance does not require more tool calls or longer responses. As a complementary observation, the first and total thinking lengths increase to $2.8\times$ and $1.5\times$ their SFT values, respectively, indicating that fewer tool calls coexist with more explicit thinking text.

5 Discussion and Conclusion

We present RTL-Interact, a post-training framework that combines multi-task data construction, CoT-to-TIR staged supervision, and PAPO to train a single model for RTL generation, autonomous EDA-tool use, and feedback-guided revision. Both model sizes outperform evaluated RTL-specialized baselines of comparable size in Pass@1 on CVDP, VerilogEval v2, and RTLLM v2; the 32B model also surpasses its tool-enabled teacher on CVDP. However, this study focuses on module-level RTL tasks and does not yet establish a protocol for repository-level development, where cross-module dependencies, build configurations, and system-level verification must be handled jointly. CoT and synthesized TIR data may retain errors, with TIR quality constrained by the teacher's capabilities and the coverage of automated checks. Future work will develop repository-level training environments and benchmarks and strengthen validation of synthesized trajectories.

Reproducibility Statement

API and diagnostic archives retain responses and scores; diagnostic protocols also preserve frozen inputs. Appendix A documents data construction, decontamination, training hyperparameters, evaluation settings, and the information exposed through each tool interface.

AI Use Statement

We used DeepSeek-V3.2 for quality scoring of specification-RTL pairs and for synthesizing tool-integrated reasoning (TIR) trajectories for generation, completion, and optimization tasks, as described in Section 3.2 and Appendix A.1.

AI tools also assisted literature search, writing, translation, LaTeX, and review, as well as parts of prompt design and code implementation. The authors are responsible for experimental records, references, and claims.

References

Anthropic. Introducing Claude Sonnet 4.5, 2025. URL https://www.anthropic.com/news/claude-sonnet-4-5.

Mu-Chi Chen, Yu-Hung Kao, Po-Hsuan Huang, Shao-Chun Ho, Hsiang-Yu Tsou, I-Ting Wu, En-Ming Huang, Yu-Kai Hung, Wei-Po Hsin, Cheng Liang, Chia-Heng Tu, Shih-Hao Hung, and H. T. Kung. SiliconMind-V1: Multi-agent distillation and debug-reasoning workflows for verilog code generation. *arXiv preprint arXiv:2603.08719*, 2026a. URL https://arxiv.org/abs/2603.08719.

Zhirong Chen, Kaiyan Chang, Zhuolin Li, Cangyuan Li, Xinyang He, Chujie Chen, Mengdi Wang, Haobo Xu, Yinhe Han, Huawei Li, and Ying Wang. ChipSeek: Optimizing verilog generation via EDA-integrated reinforcement learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 25180–25201, 2026b. doi: 10.18653/v1/2026.acl-long.1154. URL https://aclanthology.org/2026.acl-long.1154/.

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, and Dahua Lin. OriGen: Enhancing RTL code generation with code-to-code augmentation and self-reflection. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9. ACM, 2024. doi: 10.1145/3676536.3676830. URL https://doi.org/10.1145/3676536.3676830.

DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025. URL https://arxiv.org/abs/2512.02556.

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:2602.10218v1*, 2026. URL https://arxiv.org/abs/2602.10218v1.

Gemini Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025. URL https://arxiv.org/abs/2507.06261.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. Tora: A tool-integrated reasoning agent for mathematical problem solving. *arXiv preprint arXiv:2309.17452*, 2023.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024. URL https://arxiv.org/abs/2401.14196.

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*, 2025. URL https://arxiv.org/abs/2408.08927.

Icarus Verilog Contributors. Getting started with icarus verilog, 2026. URL https://steveicarus.github.io/iverilog/usage/getting_started.html. Official documentation, accessed September 10, 2026.

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

Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. CraftRTL: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair. *arXiv preprint arXiv:2409.12993*, 2024a. URL https://arxiv.org/abs/2409.12993.

Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. *arXiv preprint arXiv:2312.08617v1*, 2023b. URL https://arxiv.org/abs/2312.08617v1.

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)*, 2024b. URL https://zhiyaoxie.com/files/ICCAD24_OpenLLM.pdf.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. RTLLM: An open-source benchmark for design RTL generation with large language model. In *29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 722–727, 2024. URL https://arxiv.org/abs/2308.05345.

Hanqi Lyu, Di Huang, Yaoyu Zhu, Kangcheng Liu, Bohan Dou, Chongxiao Li, Pengwei Jin, Shuyao Cheng, Rui Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. LocalV: Exploiting information locality for IP-level verilog generation. *arXiv preprint arXiv:2602.00704v1*, 2026. URL https://arxiv.org/abs/2602.00704v1.

Meituan LongCat Team. Introducing LongCat-Flash-Thinking: A technical report. *arXiv preprint arXiv:2509.18883*, 2025. URL https://arxiv.org/abs/2509.18883.

MiniMax. MiniMax M2.1: Significantly enhanced multi-language programming, built for real-world complex tasks, 2025. URL https://www.minimax.io/news/minimax-m21.

OpenAI. Introducing GPT-5.2, 2025. URL https://openai.com/index/introducing-gpt-5-2/.

Zehua Pei, Huiling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. BetterV: Controlled verilog generation with discriminative guidance. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 40145–40153, 2024. URL https://proceedings.mlr.press/v235/pei24e.html.

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. *arXiv preprint arXiv:2408.11053*, 2024.

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

Qwen Team. Qwen3-Max-Preview, 2025. URL https://qwen.ai/blog?id=241398b9cd6353de490b0f82806c7848c5d2777d&from=research.latest-advancements-list.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2023.

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.

Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. AutoChip: Automating HDL generation using LLM feedback. *arXiv preprint arXiv:2311.04887*, 2023. URL https://arxiv.org/abs/2311.04887.

Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. RTLFixer: Automatically fixing RTL syntax errors with large language models. *arXiv preprint arXiv:2311.16543*, 2023. URL https://arxiv.org/abs/2311.16543.

Verilator Contributors. Verilator arguments, 2026. URL https://verilator.org/guide/latest/exe_verilator.html. Official documentation, accessed September 10, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Jian Yang, Wei Zhang, Jiajun Wu, Junhang Cheng, Shawn Guo, Haowen Wang, et al. InCoder-32B: Code foundation model for industrial scenarios. *arXiv preprint arXiv:2603.16790*, 2026. URL https://arxiv.org/abs/2603.16790.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

YosysHQ. Yosys open synthesis suite, 2026. URL https://github.com/YosysHQ/yosys. Official project documentation, accessed September 10, 2026.

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*, 2026. URL https://arxiv.org/abs/2605.12857.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, and Xing Hu. CodeV: Empowering LLMs with HDL generation through multi-level summarization. *arXiv preprint arXiv:2407.10424*, 2024a. URL https://arxiv.org/abs/2407.10424.

Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. MAGE: A multi-agent engine for automated RTL code generation. *arXiv preprint arXiv:2412.07822*, 2024b.

Zhipu AI. GLM-4.7: Advancing the coding capability, 2025. URL https://z.ai/blog/glm-4.7.

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. Codev-r1: Reasoning-enhanced verilog generation. *arXiv preprint arXiv:2505.24183v1*, 2025. URL https://arxiv.org/abs/2505.24183v1.

A Training and Evaluation Details

A.1 Data Construction and Filtering

We use 87,321 CoT examples from the open-source CodeV-R1-dataset (Zhu et al., 2025). We use its specification–RTL pairs as the source pool for multi-task TIR construction. DeepSeek-V3.2 performs quality scoring and TIR trajectory distillation.

Quality-scoring prompt. Before constructing any of the three task types, the teacher acts as an LLM judge for the shared instruction–RTL pool. Each pair receives a validity label (PASS, REVIEW, or REJECT) and four integer scores from 0 to 4:

1. *Specification–implementation alignment*: whether the specification defines the observable behavior and the RTL implements it;
2. *Hardware-semantic precision*: whether timing, state, finite width, signedness, and related hardware semantics must be handled precisely;
3. *Behavioral reasoning value*: whether multiple behavioral rules must be coordinated rather than mechanically transcribed; and
4. *Verifiability and error discrimination*: whether checks can be derived from the specification to distinguish correct behavior from plausible implementation errors.

The prompt also requests evidence for every score, concrete specification or code issues, up to two discriminating checks, difficulty, and confidence. Only PASS pairs with a total score above 10 enter the shared candidate pool for generation, completion, and optimization tasks; the score ranks candidates and is not treated as a probability of correctness. The complete prompt is included in the supplementary artifact.

TIR composition. Teacher outputs are executed and filtered according to each task objective. For generation, the final design must pass simulation with the teacher's own testbench and be synthesizable by Yosys. For completion, an LLM identifies core logic in the reference RTL, which is masked while the surrounding design is retained. The teacher's completed design must pass equivalence checking against the unmasked reference. For optimization, EDA tools first screen the source pool for designs with lint diagnostics or comparatively high synthesis cost. A retained trajectory must preserve reference behavior through equivalence checking and either make the design lint-clean or improve the specified synthesis metric beyond the filtering threshold. Both completion and optimization use the reference-based equivalence checker reused from CodeV-R1. The resulting 13,480 TIR trajectories comprise approximately 9K generation, 3K completion, and 1.5K optimization examples.

RL task pool. RL candidates come from TIR tasks whose teacher trajectories satisfy the task-specific acceptance criteria above. We sample five attempts per task from the two-stage SFT model and retain tasks for which fewer than five attempts succeed. Functional correctness is determined by the same equivalence checker, comparing final RTL with the reference design; optimization also requires the specified improvement. The same checker provides the final functional-correctness decision during RL. This yields approximately 8K tasks: about 7K generation/completion tasks and 1K optimization tasks.

Decontamination. Inspired by the n-gram decontamination approach of Guo et al. (2024), we use a 15-gram overlap scan between the training problems and the CVDP, VerilogEval v2, and RTLLM v2 test problems. No overlapping training problems were identified by this procedure.

A.2 Training Configuration

Trajectory serialization and training targets. For training, the trajectory is serialized into model outputs interleaved with tool observations. Tokens in the reasoning, tool requests, and final RTL are model outputs; tool observations provide context for subsequent generation but are not prediction targets. Section 3.4 assigns a reward using the final RTL y and the checking history recorded in the tool requests and observations, then uses this reward to update the same policy π_θ.

The SFT stages use eight NVIDIA A100 GPUs, a batch size of 64, and a learning rate of 10^{-5}. CoT training runs for three epochs, followed by five epochs of TIR training. The base checkpoints are Qwen3-8B and Qwen3-32B.

For multi-task RL, the reported 8B and 32B runs use a constant learning rate of 10^{-6}, 32 prompts per rollout batch, eight samples per prompt, and a global batch size of 256. The retained generation/completion and optimization prompt groups use per-update quotas targeting a 7:3 ratio, rounded to integer group counts. Rollout temperature is 1.0 with a 30,000-token response cap. GRPO uses clipping parameters 0.2/0.28 and no KL loss. Adam uses $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-8}$, weight decay 0.01, and seed 1234. Pattern-aware rewards are enabled for the reported RTL-Interact runs. The binary-reward ablation uses the same initialization, data, optimizer, rollout configuration, and number of training steps; only the pattern-dependent reward weights are disabled.

Policy-update quantities. For trajectory τ_i sampled for task x_i, $\mathcal{M}_i$ contains the indices of model-generated tokens, excluding tool observations. Here t indexes tokens in the serialized trajectory, rather than the tool interactions indexed by ℓ in Section 3.1. Let $u_{i,t}$ be the token at position t and $h_{i,t}$ its preceding context, including the task, previous model outputs, and tool observations. The policy ratio in Equation (2) is

$$\rho_{i,t} = \frac{\pi_\theta(u_{i,t} \mid h_{i,t})}{\pi_{\text{old}}(u_{i,t} \mid h_{i,t})}, \tag{4}$$

where π_{old} is the policy before the update. The parameters ε_- and ε_+ specify the lower and upper clipping thresholds. We retain the fixed rollout correction

$$c_{i,t} = \text{clip}\left(\frac{\pi_{\text{old}}(u_{i,t} \mid h_{i,t})}{\pi_{\text{roll}}(u_{i,t} \mid h_{i,t})}, 0, 2\right), \tag{5}$$

where π_{roll} generates the rollout and no gradient is propagated through $c_{i,t}$.

Let $\mathcal{I}_x$ index the G trajectories sampled for the same task x. The reward statistics used to compute their advantages are

$$\bar{R}_x = \frac{1}{G}\sum_{j \in \mathcal{I}_x} R_x(\tau_j), \qquad \sigma_x = \text{Std}_{j \in \mathcal{I}_x}[R_x(\tau_j)]\,. \tag{6}$$

The constant ϵ stabilizes division by the standard deviation. Reward normalization is performed within the same task, rather than across all tasks belonging to the same task group.

Token losses are averaged over model-generated tokens within each trajectory and then over retained trajectories. The same rollout correction and loss reduction are used in the binary-reward baseline.

A.3 Evaluation Configuration

For the main benchmark evaluations, we cap model-generated responses at 32K tokens, excluding tool outputs. We allow at most 6 tool calls per trajectory for functional tasks—generation, completion, modification, and bug fixing—and 10 for optimization tasks. A response is counted as truncated if it reaches the response-length limit.

Unless stated otherwise, Pass@1/Pass@5 results use five samples per problem, temperature 0.7, and top-p 0.95. CVDP averages are weighted by category sizes $(94, 78, 55, 40, 35)$, totaling 302. With n samples and c passing samples, we use

$$\widehat{\text{Pass@}k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}, \qquad n \geq k. \tag{7}$$

Generation and revision protocol (Section 4.4). We freeze 100 tasks before comparative inference using source-balanced selection with seed 20260909, one task per name-based family, and preference for behavioral/lint failures rather than uniform random sampling. Archived tool-use trajectories from GLM-4.6, MiniMax-M2, DeepSeek-V4-Flash, and a binary-reward 8B RL checkpoint (step 499) supply 25 failing candidates each. The completion/generation/modification/optimization/debugging counts are 24/28/11/20/17. Candidates are intermediate RTL submissions that fail independent CVDP evaluation; diagnostics are reproduced with Icarus Verilog (47 compilation and 33 behavioral failures)

or Verilator (20 lint failures). Behavioral checks must also pass available archived CVDP-passing implementations. Every revision model receives the same candidate, diagnostic, and archived model-authored testbench when present; hidden scoring tests remain unavailable. This diagnostic study uses temperature 1, top-p 1, shared task-derived seeds, and initially one response per task/model/setting with a 32K-token generation budget.

General-purpose baselines. The CVDP frontier baselines are Claude-Sonnet-4.5 (Anthropic, 2025), GPT-5.2 (OpenAI, 2025), GLM-4.7 (Zhipu AI, 2025), LongCat-Flash-Thinking (Meituan LongCat Team, 2025), MiniMax-M2.1 (MiniMax, 2025), Qwen3-Max-Preview (Qwen Team, 2025), and Gemini-2.5-Pro (Gemini Team, 2025). These checkpoints were selected around the teacher's release, reflecting the earlier start of our experiments.

Table 6: Five evaluated RTL task categories and four model-facing EDA interfaces. Checkmarks denote tool access in our environment, rather than restrictions on where an EDA tool can be useful. Equivalence checking uses the original RTL supplied in an optimization task; hidden benchmark tests are never exposed through these interfaces.

Task	Simulation	Lint checking	Synthesis analysis	Equivalence checking
Generation	✓	—	—	—
Completion	✓	—	—	—
Modification	✓	—	—	—
Debugging	✓	—	—	—
Optimization	—	✓	✓	✓

Information boundary. Functional interaction requests contain model-generated RTL and a model-generated testbench. The hidden/reference scoring testbench is not provided by that interface. For optimization, the original design is part of the task and is available for equivalence checking. Workflow systems may use different checking information or budgets, so their published scores are not inserted into our same-interface table.

Controlled supervision comparison. In Table 1, the 450M-token budget corresponds to 150M chat-template tokens per epoch for three epochs; the 900M-token budget doubles this total. The 17.2% Base result belongs to this controlled protocol and is not substituted for the 18.9% tool-free main-table baseline. Avg. tool turn reports the mean number of tool-call turns over all evaluated trajectories, including zero for trajectories without tool calls. Tool-call turns are distinct from the task-specific checking rounds defined below.

A.4 Interaction Statistics

Table 7: CVDP interaction statistics for RTL-Interact-8B before and after RL. Tool calls and thinking lengths are means per trajectory; thinking lengths are measured in characters. First and total thinking lengths refer to the first thinking segment and the sum of all thinking segments, respectively. Truncation means reaching the response-length limit. Changes are reported in percentage points (pp) for truncation rate and as a relative percentage for tool calls.

Metric	SFT	RL	Change
Tool calls	4.7	1.7	−63.8%
Truncation rate	33.7%	15.2%	−18.5 pp
First thinking length	11,453	31,619	2.8×
Total thinking length	30,873	44,838	1.5×

B Checking Rounds and Pattern Rewards

Checking rounds. The unit of checking reflects the task objective. For functional tasks, one round consists of simulating the candidate with a model-authored testbench. For optimization, one round combines a task-appropriate optimization check (lint or synthesis analysis) with a self-equivalence check against the original design. Thus two tool calls can constitute one optimization checking round. In both cases, final correctness is determined independently by the reference-based verifier.

Matching rules and weights. Let n be the number of checking rounds and $p_j \in \{0, 1\}$ indicate whether round j succeeds. For functional tasks, success means that the model-authored simulation check passes. For optimization, it requires successful task-appropriate optimization and equivalence checks. Among task-appropriate checking histories, ISR requires $n = 1$ and $p_1 = 1$; FRS requires $n > 1$ and $p_n = 1$. An earlier successful check does not qualify a trajectory for FRS if its final self-check fails. US covers histories that do not meet either condition, including missing or incomplete checks and the use of an optimization tool unrelated to the task objective.

For both functional and optimization tasks, let $p_x(\tau) \in \{\mathrm{ISR}, \mathrm{FRS}, \mathrm{US}\}$ denote the matched checking pattern. The pattern adjustment in Equation (3) is

$$\delta_{p_x(\tau)} = \begin{cases} +0.5, & \mathrm{ISR}, \\ +0.2, & \mathrm{FRS}, \\ -0.2, & \mathrm{US}. \end{cases} \tag{8}$$

Thus $R_x(\tau) = V_x(y)[1 + \delta_{p_x(\tau)}]$ gives final-correct trajectories rewards of 1.5, 1.2, and 0.8, respectively. The value 1.0 is the base correctness reward before the pattern adjustment, not an additional unpenalized US category. For all tasks, a failed final verification gives zero reward regardless of checking history.

Reward ablation. The binary ablation removes the pattern-conditioned successful rewards while retaining the final correctness gate. It tests the whole reward design, not each rule separately. There is no reported ablation against a tuned length penalty, so the paper does not claim superiority to every cost-aware reward.

C Task-Group Proportion Analysis

We vary α, the optimization-task share among samples retained for each policy update, with generation and completion sharing the remaining $1 - \alpha$. We compare $\alpha \in \{0.2, 0.3, 0.4\}$ for RTL-Interact-8B on CVDP. Figure 4 shows that $\alpha = 0.3$ yields the highest overall Pass@1 and Pass@5 among the three tested settings, reaching 44.9% and 63.2%, respectively. We use this proportion in the full method.

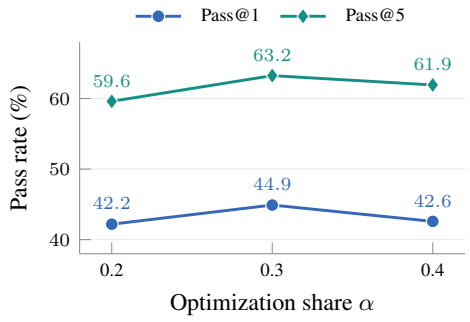

Figure 4: CVDP Pass@1/Pass@5 (%) versus the optimization share α among samples retained for each RL update; the functional share is $1 - \alpha$.

Table 8: Endpoint results for generation-only RL on RTL-Interact-8B. Values are Pass@1/Pass@5.

Task category	SFT	Generation-only RL
CID002	33.4/55.3	38.5/53.2
CID003	40.5/53.8	49.0/62.8
CID004	35.3/58.2	47.3/63.6
CID007	23.0/52.5	26.5/50.0
CID016	38.3/62.9	50.3/71.4
Overall	34.8/56.0	42.6/59.3

D Additional Generation-Only Transfer Results

The generation-only run starts from its own SFT checkpoint. Pass@1 rises from 34.8% to 42.6% overall, with gains of 5.1%, 8.5%, 12.0%, 3.5%, and 12.0% on completion, specification-to-RTL, modification, optimization, and debugging, respectively. This transfer pattern is consistent with benefits across functional tasks sharing an interaction interface. Optimization also changes the objective and correctness conditions, however, so this comparison does not isolate the causal effect of using a different tool. Optimization Pass@5 decreases from 52.5% to 50.0%, and completion Pass@5 decreases from 55.3% to 53.2%. All these endpoints are retained in Table 8; no intermediate category trajectory is inferred from them.

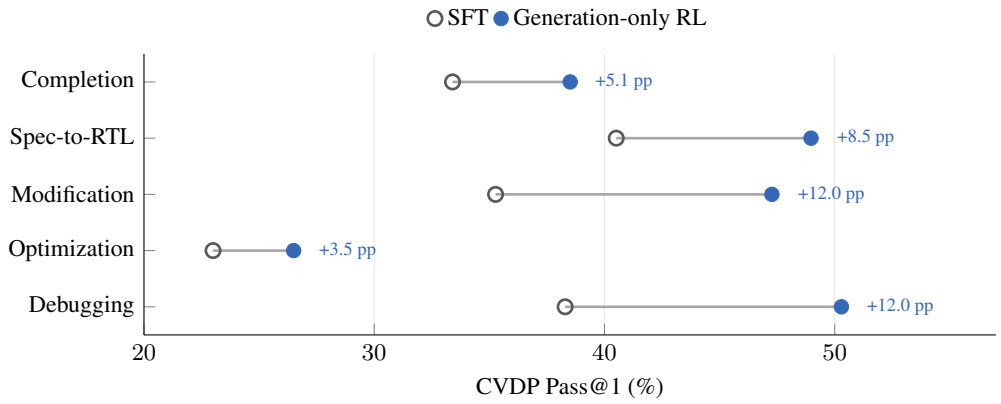

Figure 5: Measured endpoint changes in the separate 8B generation-only run. Segments connect SFT and RL results within a category; they are not training-time curves. Labels show Pass@1 gains in percentage points (pp). This run has a different SFT initialization from the full multi-task run.

E Evaluation Prompts

We provide the common prompt templates used across all evaluated benchmarks, with and without tool access. `{problem_description}` is replaced with the task description, and `{task_specific_tool_definitions}` contains the tool interfaces available for the corresponding task. We display the message contents by role, omitting chat-template delimiters. Line wrapping is adjusted for readability.

E.1 Tool-Free Prompt

Prompt 1: Tool-free evaluation

System

```
You are a Verilog RTL designer that only writes code using correct
Verilog syntax.
```

User

```
Question:
{problem_description}

Enclose your code with [BEGIN] and [DONE]. Only output the code
    snippet
and do NOT output anything else.

Answer:
```

E.2 TOOL-ENABLED PROMPT

Prompt 2: Tool-enabled evaluation

System

```
You are a Verilog coding expert who will write Verilog code after
    careful consideration.
Your goal is to deliver functionally correct, synthesizable, and
    efficient RTL that meets the user's requirements.

Principles:
- Reason carefully about the specification, interfaces, and corner
    cases.
- Validate behavior and iterate as needed using any available tools.
- Iverilog Compatibility: Use robust Verilog-2001 or
    SystemVerilog-2012 constructs.
  - You should prefer `logic`, `always_ff`, and `always_comb`.
  - Avoid complex SystemVerilog features that have limited support
    in iverilog (e.g., specialized interfaces, program blocks, or
    nested struct assignments).
- Prefer clean, portable RTL: separate combinational/sequential
    logic, avoid unintended latches, avoid unsynthesizable constructs
    in design code.
- Use clear resets, parameterize widths when appropriate, and write
    readable, well-commented code.
- When a testbench is requested or required (e.g. by tools), follow
    these rules:
  1. Clearly state the expected behavior/results in the testbench
    (e.g. as comments).
  2. Ensure the testbench checks that simulation outputs exactly
    match the expected results; any mismatch means the design is
    still buggy.
  3. If the module has a reset signal, include proper reset testing
    in the testbench.
  4. Provide sufficient and diverse stimulus: cover normal cases,
    boundary conditions, and corner cases (e.g. 0, maximum values,
    negative numbers, sign-bit cases, etc.) instead of testing only a
    single scenario.
  5. Use $display for textual checks. If you emit errors, surface
    them clearly.
  6. DO NOT use `$monitor` or print signals every clock cycle. This
    will exceed the context limit.
After thinking, when you finally reach a conclusion, enclose the
    final verilog code in verilog code block within code tags. i.e.,
'''verilog
module top_module(in, out, ...);
  ...
endmodule
'''


# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML
    tags:
<tools>
{task_specific_tool_definitions}
</tools>

For each function call, return a json object with function name and
    arguments within <tool_call></tool_call> XML tags:
```

Prompt 2: Tool-enabled evaluation (continued)

```
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>
```

User

```
Design a functionally correct, synthesizable, and efficient Verilog
    module for the following problem.
Use any available tools during the process to validate and refine
    your design if you think it's needed.

The problem is as follows:
{problem_description}
```

F Case Study: Response Trajectories

Below are the original problem and response trajectories of the 8B Base, SFT, and RL models on the RTLLM v2 `freq_divbyeven` task. Input prompts are omitted. Within each response turn, reasoning and tool-call contents are summarized and marked `[Summary]` inside their respective tags. Tool feedback and final answers are reproduced without summarization.

Original problem: Even-frequency divider

```
Please act as a professional verilog designer.

Frequency divider that divides the input clock frequency by even
    numbers. This module generates a divided clock signal by toggling
    its output every specified number of input clock cycles.

Module name:
    freq_diveven

Input ports:
    clk: Input clock signal that will be divided.
    rst_n: Active-low reset signal to initialize the module.

Output ports:
    clk_div: Divided clock output signal.

Implementation:
    The frequency divider uses a counter (`cnt`) to count the number
    of clock cycles. The `NUM_DIV` parameter specifies the division
    factor, which must be an even number.
    - When the reset signal (`rst_n`) is low, the counter and the
    divided clock signal (`clk_div`) are initialized to zero.
    - On each positive edge of the input clock (`clk`), if the
    counter is less than half of `NUM_DIV - 1`, the counter increments
     without changing the divided clock output.
    - When the counter reaches the specified limit, it resets to zero
     and toggles the `clk_div` output signal, effectively dividing the
     frequency of the input clock by the even number specified by `
    NUM_DIV`.

Counter:
    - The counter is a 4-bit register (`cnt`) that tracks the number
    of clock cycles.

Give me the complete code.
```

Base-8B

```
===== MODEL RESPONSE t0 =====

<think>
[Summary]
The model considers conflicting interpretations of the counter limit
    and repeatedly confuses the interval between output toggles with
    the full output period. It ultimately chooses to toggle at NUM_DIV
     - 1, incorrectly treating this as division by NUM_DIV.
</think>

[BEGIN]
module freq_diveven (
    input clk,
    input rst_n,
    output reg clk_div
);

parameter NUM_DIV = 4; // Must be even
reg [3:0] cnt;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        cnt <= 4'd0;
        clk_div <= 1'b0;
    end else begin
        if (cnt == NUM_DIV - 1) begin
            cnt <= 4'd0;
            clk_div <= ~clk_div;
        end else begin
            cnt <= cnt + 1;
        end
    end
end

endmodule
[DONE]
```

RTL-Interact-8B (SFT)

```
===== MODEL RESPONSE t0 =====

<think>
[Summary]
The model resolves the half-period interpretation: division by
    NUM_DIV requires toggling every NUM_DIV/2 input cycles. It selects
     CNT_MAX = NUM_DIV/2 - 1 with a 4-bit counter and active-low reset
    , then prepares an initial simulation.
</think>

<answer>
</answer>

<tool_call>
[Summary]
Function: verilog_simulator
Argument code: Submit the DUT using CNT_MAX = NUM_DIV/2 - 1 and a
    testbench for NUM_DIV = 4. Check output timing and reset
    initialization; print ALL_TESTS_PASSED when the error count is
    zero.
```

RTL-Interact-8B (SFT) (continued)

```
</tool_call>

===== TOOL RESPONSE t0_tool0.json =====

Test completed. Error count = 0
ALL_TESTS_PASSED
candidate.sv:70: $finish called at 220 (1s)

===== MODEL RESPONSE t1 =====

<think>
[Summary]
After the first self-check passes, the model plans broader parameter
    coverage and additional reset checks using multiple DUT instances.
</think>

<answer>
</answer>

<tool_call>
[Summary]
Function: verilog_simulator
Argument code: Submit the same divider logic with a testbench
    instantiating NUM_DIV = 2, 4, 6, and 8. Check the expected output
    periods and reset behavior; print ALL_TESTS_PASSED when no errors
    are detected.
</tool_call>

===== TOOL RESPONSE t1_tool0.json =====

Test completed. Error count = 0
ALL_TESTS_PASSED
candidate.sv:77: $finish called at 220 (1s)

===== MODEL RESPONSE t2 =====

<think>
[Summary]
After the multi-parameter self-check passes, the model revisits the
    specification and rechecks the threshold calculation, counter
    width, and reset behavior. It retains the same implementation and
    prepares another verification call before returning the final code
    .
</think>

<answer>
</answer>

<tool_call>
[Summary]
Function: verilog_simulator
Argument code: Submit the divider and another testbench for NUM_DIV =
     4, repeating verification after the previous checks passed. Print
     ALL_TESTS_PASSED when the error count is zero.
</tool_call>

===== TOOL RESPONSE t2_tool0.json =====

Test completed. Error count = 0
ALL_TESTS_PASSED
candidate.sv:64: $finish called at 220 (1s)
```

RTL-Interact-8B (SFT) (continued)

````
===== MODEL RESPONSE t3 =====

<think>
[Summary]
The model prepares the final DUT-only implementation, including
    explanatory comments and omitting the testbench.
</think>

<answer>
```verilog
/**
 * Frequency divider that divides the input clock frequency by an
    even number.
 *
 * @param NUM_DIV  Division factor (must be an even number)
 * @param clk      Input clock signal
 * @param rst_n    Active-low reset (0 = initialize, 1 = normal
    operation)
 * @param clk_div  Divided clock output
 *
 * The module uses a 4-bit counter to count input clock cycles.  When
     the
 * counter reaches (NUM_DIV/2)-1, it resets and toggles the output,
    producing
 * a clock with period = NUM_DIV * Tclk.
 */
module freq_diveven #(
    parameter NUM_DIV = 4   // must be an even number
) (
    input  wire clk,
    input  wire rst_n,
    output reg  clk_div
);

    // Half of the division factor (integer division, NUM_DIV is even
    )
    localparam HALF_DIV = NUM_DIV / 2;
    // Counter maximum value
    localparam CNT_MAX  = HALF_DIV - 1;

    // 4-bit counter as specified
    reg [3:0] cnt;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            // Active-low reset: clear counter and output
            cnt      <= 4'b0;
            clk_div  <= 1'b0;
        end else begin
            if (cnt == CNT_MAX) begin
                // Reached the limit: reset counter and toggle output
                cnt      <= 4'b0;
                clk_div  <= ~clk_div;
            end else begin
                // Normal counting, keep output unchanged
                cnt      <= cnt + 1;
            end
        end
    end

endmodule
```
````

RTL-Interact-8B (SFT) (continued)

```
</answer>
```

RTL-Interact-8B (RL)

```
===== MODEL RESPONSE t0 =====

<think>
[Summary]
The model examines the half-period interpretation and integer
    division, then selects HALF = (NUM_DIV - 1)/2. It develops a
    testbench with multiple DUT instances to check division factors 2,
     4, 6, and 8 in a single simulation call.
</think>

<answer>
</answer>

<tool_call>
[Summary]
Function: verilog_simulator
Argument code: Submit the DUT using HALF = (NUM_DIV - 1)/2 and a
    testbench instantiating NUM_DIV = 2, 4, 6, and 8. Measure output
    periods and compare them with the expected values in one
    simulation; print ALL_TESTS_PASSED when the error count is zero.
</tool_call>

===== TOOL RESPONSE t0_tool0.json =====

ALL_TESTS_PASSED
candidate.sv:88: $finish called at 220000 (1ps)

===== MODEL RESPONSE t1 =====

<think>
[Summary]
After receiving a successful self-check, the model prepares the final
     DUT-only implementation with the same threshold, omitting the
    testbench.
</think>

<answer>
'''verilog
// Frequency divider that divides the input clock frequency by an
    even number.
module freq_diveven #(
    parameter NUM_DIV = 4          // Must be an even number
) (
    input  wire clk,               // Input clock
    input  wire rst_n,             // Active-low reset
    output wire clk_div            // Divided clock output
);
    reg [3:0] cnt;                 // 4-bit counter
    reg       clk_div_reg;         // Internal register for clk_div

    // HALF = (NUM_DIV - 1) / 2  (integer division)
    localparam HALF = (NUM_DIV - 1) / 2;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            // Reset: clear counter and output
```

RTL-Interact-8B (RL) (continued)

```
            cnt        <= 4'b0;
            clk_div_reg <= 1'b0;
        end else begin
            if (cnt < HALF) begin
                // Increment counter, keep output unchanged
                cnt <= cnt + 1'b1;
            end else begin
                // Counter reached the limit: reset it and toggle the
     output
                cnt        <= 4'b0;
                clk_div_reg <= ~clk_div_reg;
            end
        end
    end

    assign clk_div = clk_div_reg;
endmodule
```
```
</answer>
```