ChipSeek: Optimizing Verilog Generation via EDA-Integrated Reinforcement Learning

Zhirong Chen[1,3], Kaiyan Chang[2,3], Zhuolin Li[4], Cangyuan Li[1,3], Xinyang He[3], Chujie Chen[1,3], Mengdi Wang[1,3], Haobo Xu[1,3], Yinhe Han[1,3], Huawei Li[2,3], Ying Wang[1,3]

[1]Research Center for Intelligent Computing Systems,
Institute of Computing Technology, Chinese Academy of Sciences,
[2]SKLP, Institute of Computing Technology, Chinese Academy of Sciences,
[3]University of Chinese Academy of Sciences,
[4]University of Electronic Science and Technology of China

Correspondence: wangying2009@ict.ac.cn

Abstract

Large Language Models have emerged as powerful tools for automating Register-Transfer Level (RTL) code generation, yet they face critical limitations: existing approaches typically fail to simultaneously optimize functional correctness and hardware efficiency metrics such as Power, Performance, and Area (PPA). Methods relying on supervised fine-tuning commonly produce functionally correct but suboptimal designs due to the lack of inherent mechanisms for learning hardware optimization principles. Conversely, external post-processing techniques aiming to refine PPA performance after generation often suffer from inefficiency and do not improve the LLMs' intrinsic capabilities.

To overcome these challenges, we propose ChipSeek, a novel hierarchical reward based reinforcement learning framework designed to encourage LLMs to generate RTL code that is both functionally correct and optimized for PPA metrics. Our approach integrates direct feedback from EDA simulators and synthesis tools into a hierarchical reward mechanism, facilitating a nuanced understanding of hardware design trade-offs. Through Curriculum-Guided Dynamic Policy Optimization (CDPO), ChipSeek enhances the LLM's ability to generate high-quality, optimized RTL code. Evaluations on standard benchmarks demonstrate ChipSeek's superior performance, achieving state-of-the-art functional correctness and PPA performance. Furthermore, it excels in specific optimization tasks, consistently yielding highly efficient designs when individually targeting fine-grained optimization goals such as power, delay, and area. The artifact is open-source in https://github.com/rong-hash/chipseek.

1 Introduction

Large Language Models (LLMs) show immense potential to revolutionize hardware design methodology, particularly in tasks like Register-Transfer Level (RTL) code generation (Cui et al., 2025; Pei et al., 2024). Previous works have improved RTL generation by employing various techniques, including Supervised Fine-tuning (SFT) (Thakur et al., 2023; Chang et al., 2024), Retrieval-Augmented Generation (RAG) (Gao et al., 2024), multi-agent collaboration (Yu et al., 2025b; Ho et al., 2025), and Chain of Thought (CoT) reasoning (Qin et al., 2025). While these approaches successfully enhance functional correctness, they generally neglect critical hardware metrics such as synthesizability and more importantly, Power, Performance, and Area (PPA).

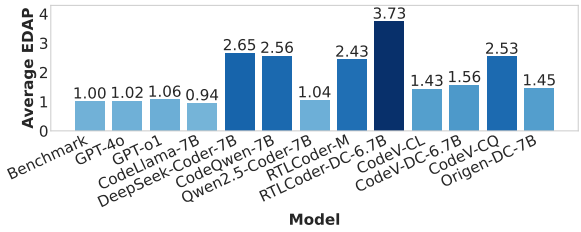

Figure 1: PPA performance comparison between Verilog from models and benchmark RTLLM using the EDAP (Energy-Delay-Area Product). Lower values are better.

However, to truly assist engineers, it is not enough to produce merely functionally correct Verilog. The generated RTL must also be high-quality in terms of PPA metrics. PPA performance is a crucial indicator of Verilog code quality. In practice, design specifications vary widely: edge devices may emphasize area and power constraints, while data center accelerators focus on timing and performance. Existing models, trained predominantly on inconsistent and noisy RTL corpora, lack the inductive bias needed to internalize such nuanced trade-offs, sometimes outcompeted by designs written by expert engineers. Figure 1 illustrates this problem clearly: the Verilog generated by existing models mostly underperforms compared to the RTLLM benchmark, a collection of expert-written

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 25180–25201
July 2-7, 2026

designs. These models fail to match the hardware efficiency of manually crafted RTL and often require post-processing, such as Monte Carlo Tree Search (MCTS) (DeLorenzo et al., 2024) or external optimization pipelines (Yao et al., 2024b), to improve PPA. However, such methods operate externally, introducing computational and manual overhead to take effect without improving the LLM intrinsic ability. Therefore, there is a fundamental gap: current methods lack an inherent mechanism to optimize functional correctness and hardware-specific PPA metrics concurrently. Bridging this gap is essential for making LLMs viable co-designers in practical RTL workflows.

To address this challenge, we introduce **ChipSeek**, a novel reinforcement learning framework that integrates EDA toolchains directly into the training loop. The EDA toolchain offers functional verification to ensure logical correctness and PPA metric measurement to quantify hardware efficiency. Together, these feedback signals serve as rewards to enforce both functional validity and alignment with design specifications. Our framework employs a hierarchical reward system and Curriculum-Guided Dynamic Policy Optimization (CDPO), enabling the LLM to adapt the optimization goals with the training process and optimize RTL code generation according to specific PPA targets. This empowers the LLM to generate RTL codes that not only meet functional correctness requirements but also exhibit high quality in terms of hardware PPA. Our main contributions are as follows:

Hierarchical Rewards from Comprehensive EDA Toolchain: We build a closed-loop RTL generation pipeline that tightly integrates a comprehensive open-source EDA toolchain (compilation, simulation, synthesis, and backend analysis) into reinforcement learning. Based on this toolchain, we derive hierarchical reward signals spanning thinking format, syntax validity, functional correctness, synthesizability, and PPA, together with a strict hierarchical gating mechanism that avoids expensive downstream evaluation on invalid designs. This enables direct tool-verified supervision for functional correctness and provides PPA-aware feedback grounded in physical implementation.

Curriculum-Guided Dynamic Policy Optimization (CDPO): We propose CDPO for multi-objective RTL optimization under two key challenges in managing EDA-derived rewards: *(i) learning-stage mismatch*, where process rewards such as format reward and syntax reward are easier to learn and serve as early-stage scaffolding while functional correctness and PPA are the ultimate goals; and *(ii) scale mismatch*, arises from the disparity between binary discrete signals (0/1) for syntax or function rewards versus continuous, theoretically unbounded PPA rewards, which renders traditional reward aggregation unstable. CDPO addresses these issues with a curriculum-guided weight schedule, advantage-level aggregation, and prompt-conditioned PPA preference weighting, enabling curriculum training and controllable power–performance–area trade-offs.

Automated Data Augmentation Pipeline: We propose a multi-stage automated data augmentation pipeline that systematically processes Verilog codes into richer PPA-aware datasets. This pipeline executes three fundamental tasks: (1) generating reasoning cold-start datasets tailored explicitly for the SFT phase; (2) synthesizing diverse PPA preference vectors and augmenting design descriptions to facilitate comprehensive multi-objective RTL optimization; and (3) producing accurate testbenches and corresponding PPA metrics critical for precise reward computation. Extensive validation and rigorous filtering procedures ensure dataset quality, significantly boosting the robustness and effectiveness of subsequent training processes.

2 Background

Large Language Models (LLMs) have demonstrated significant potential in generating Verilog code directly from natural language specifications. Despite this potential, the quality of the generated Verilog code remains limited due to two fundamental challenges: ensuring functional correctness and achieving optimization in terms of PPA.

Several approaches have been proposed to improve functional correctness in Verilog code generation. For instance, RTLFixer (Tsai et al., 2024) and HDLDebugger (Yao et al., 2024a) utilize Retrieval-Augmented Generation (RAG) to facilitate autonomous debugging by LLMs. Additionally, fine-tuning (Chang et al., 2024) and multimodal techniques (Chang et al., 2025) have been explored to enhance functional accuracy. Regarding performance optimization, RTLRewriter and SymRTLO (Yao et al., 2024b; Wang et al., 2025) applies code analysis combined with RAG to pinpoint redundant structures and potential optimizations. VeriGen-MCTS (DeLorenzo et al., 2024) leverages MCTS

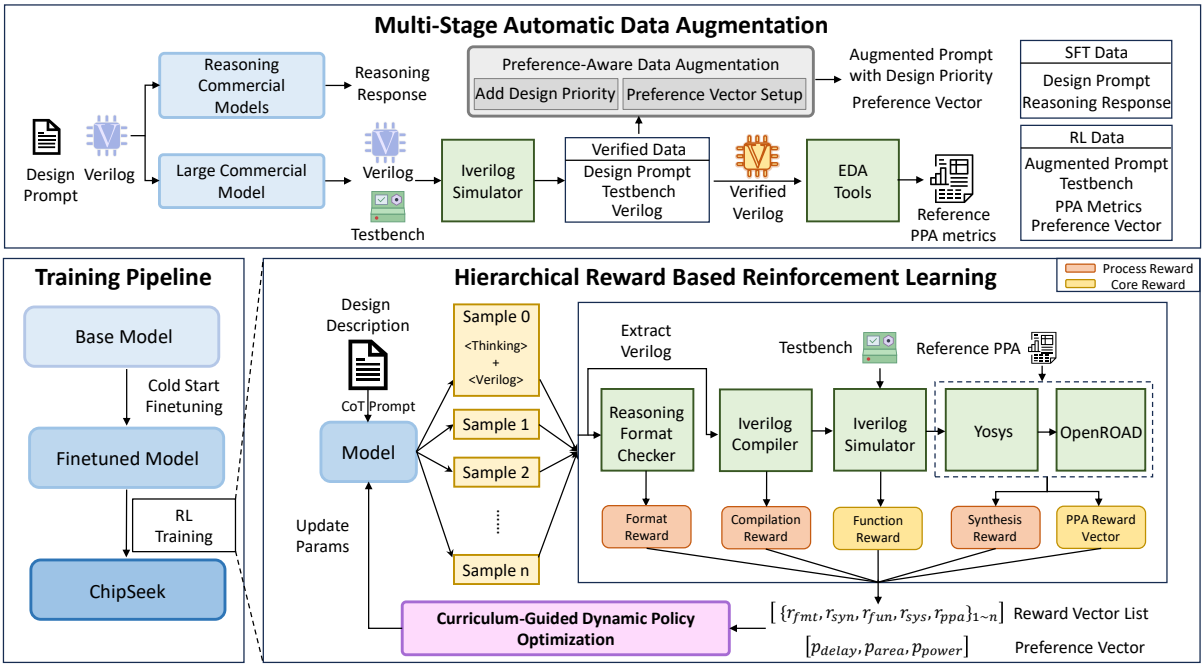

Figure 2: Our Hierarchical Reward based Reinforcement Learning Framework.

to optimize hardware performance metrics.

3 Method

3.1 Overview

Our framework, ChipSeek, aims to align LLMs with the rigorous constraints of hardware design. As illustrated in Figure 2, the framework operates in a closed loop where the LLM acts as a policy π_θ, generating Verilog code y given a design specification x. The generated code is evaluated by a comprehensive EDA toolchain, providing feedback signals ranging from basic syntax compliance to complex PPA metrics.

To effectively navigate this complex optimization landscape, we propose **Curriculum-Guided Dynamic Policy Optimization**. Unlike traditional RLVR methods that aggregate rewards into a scalar before optimization, CDPO disentangles distinct feedback signals, normalizes and aggregates them in the advantage space, and dynamically modulates their influence via a curriculum schedule. This schedule guides the model to first master reasoning structures and syntactic compliance, laying the groundwork for subsequent optimization of functional correctness and PPA performance.

3.2 Hierarchical Rewards Definition

We partition the rewards into *Process Rewards* (Format, Syntax, Synthesis) and *Core Rewards* (functional correctness and PPA). To optimize computational resources, we implement a strict gating mechanism where downstream metrics are only evaluated if upstream constraints are satisfied.

Process Rewards ($\mathcal{R}_{pro}$): These serve as binary prerequisites for a valid Verilog code.

Format Reward (r_{fmt}) enforces the CoT structure `(think)...(answer)...` to promote reasoning stability.

Syntax Reward (r_{syn}) is verified by the Icarus Verilog compiler, where $r_{syn} = 1$ indicates successful compilation.

Synthesis Reward (r_{sys}) is verified by Yosys and OpenROAD (Ajayi et al., 2019), where $r_{sys} = 1$ confirms the design is physically synthesizable into a netlist, ensuring hardware realizability beyond logical syntax.

Core Rewards ($\mathcal{R}_{cor}$): These evaluate the semantic quality and performance of the design.

Function Reward (r_{func}) is determined by testbench simulation, where $r_{func} = 1$ only if all test cases pass.

PPA Reward Vector($\mathbf{r}_{ppa}$) quantifies Power, Delay, and Area rewards. For each metric $m \in \{power, delay, area\}$, the reward is the relative improvement over a reference design: $r_m = \text{ref}_m/\text{gen}_m$, where ref_m and gen_m are the metric values of the reference design and the generated design.

Gating Mechanism: We impose a hierarchical de-

pendency chain to avoid expensive simulations on invalid code: **Syntax** $\rightarrow$ **Function** $\rightarrow$ **Synthesis** $\rightarrow$ **PPA**. Specifically, a higher-level reward is computed only if the immediate lower-level reward is positive (e.g., r_{func} is triggered only if $r_{syn} = 1$). If a stage fails, the evaluation terminates, and all subsequent rewards are assigned a value of 0.

3.3 Curriculum-Guided Dynamic Policy Optimization (CDPO)

Given the hierarchical rewards in Section 3.2, we optimize the policy with two design choices: (i) *decoupled advantage estimation* for each reward, and (ii) *dynamic policy optimization* that apply self-adaptive weights of rewards over training steps and conditions PPA optimization on prompt-level preferences. Finally, we aggregate all objectives at the *advantage level* and update the policy with the corresponding loss.

3.3.1 Decoupled Advantage Estimation

For each prompt q, we sample a group of G completions $\{o_i\}_{i=1}^{G}$. Each completion yields a multi-objective reward vector $\mathbf{r}_i$ containing component metrics $r_{k,i}$, where each metric corresponds to the process ($\mathcal{R}_{pro}$) and objective ($\mathcal{R}_{cor}$) definitions detailed in Section 3.2.

For a specific component k, the decoupled token-level advantage is defined as:

$$\hat{A}_{i,t}^{(k)} = \frac{r_{k,i} - \mu_k}{\sigma_k + \epsilon}, \qquad \forall t \in \{1, \ldots, |o_i|\}, \tag{1}$$

where μ_k and σ_k denote the mean and standard deviation of the k-th reward component across group G. This results in a set of advantages that are subsequently aggregated in the dynamic policy optimization phase. Such normalization brings reward components of different scales onto a common scale before aggregation, effectively mitigating cross-component interference caused by reward scale mismatch.

3.3.2 Dynamic Policy Optimization

We dynamically weight Process rewards via curriculum annealing, and weight Objective rewards via fixed function reward weight plus preference-conditioned PPA reward weights. All objectives are combined *after* advantage estimation.

Adaptive Curriculum for Process Rewards. We utilize a curriculum weight schedule mechanism to stabilize the optimization trajectory. This ensures that the policy initially focuses on easier process rewards (e.g., syntax and format) and gradually shifts its emphasis toward the core objectives (e.g., functional correctness and PPA) as it masters basic Verilog programming patterns, enabling an easy-to-hard learning progression. For each process objective $k \in \mathcal{R}_{pro}$ at training step s, we compute the global success rate $\bar{\mu}_k^{(s)}$ over the entire training batch of B prompts and G completions:

$$\bar{\mu}_k^{(s)} = \frac{1}{B \times G} \sum_{j=1}^{B} \sum_{i=1}^{G} r_{k,ji}^{(s)}, \tag{2}$$

where $r_{k,ji}^{(s)}$ is the reward for the i-th completion of the j-th prompt. We first compute an instantaneous curriculum signal

$$\hat{\alpha}_k^{(s)} = \max\left(0,\, 1 - \bar{\mu}_k^{(s)}\right), \tag{3}$$

and then apply an exponential moving average to obtain a smooth curriculum coefficient:

$$\alpha_k^{(s)} = \beta\, \alpha_k^{(s-1)} + (1 - \beta)\, \hat{\alpha}_k^{(s)}. \tag{4}$$

This ensures the gradient contribution of each process reward changes smoothly across steps, reducing batch-to-batch oscillation. The aggregated process advantage is:

$$A_{i,t}^{pro} = \sum_{k \in \mathcal{R}_{pro}} \alpha_k^{(s)} \hat{A}_{i,t}^{(k)}. \tag{5}$$

Preference-conditioned Weighting for PPA Rewards. We maintain a fixed weight for functional correctness while steering PPA optimization via prompt-dependent preference templates. This design allows the policy to prioritize the corresponding PPA component according to the optimization preference specified in the prompt. Let $\mathbf{p}(q) = (p_P, p_D, p_A)$ denote the preference vector extracted from prompt q (e.g., via tags like `[low_power]`), where $\sum p_m = 1$. For the i-th completion, the aggregated core advantage is:

$$A_{i,t}^{cor} = w_{func} \hat{A}_{i,t}^{(func)} + \sum_{m \in \{P,D,A\}} p_m(q) \hat{A}_{i,t}^{(m)}. \tag{6}$$

The final token-level advantage for optimization is formed by summing the adaptive process component and the preference-weighted core component:

$$A_{i,t}^{total} = A_{i,t}^{pro} + A_{i,t}^{cor}. \tag{7}$$

For each prompt q, we sample a group of G rollouts $\{o_i\}_{i=1}^{G}$ from the behavior policy $\pi_{\theta_{\text{old}}}$. We maximize the expected objective:

$$\mathcal{J}(\theta) = \mathbb{E}_{\{o_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}} \left[\frac{1}{\sum_{i=1}^{G} |o_i|} \sum_{i=1}^{G} \sum_{t=1}^{|o_i|} \mathcal{O}_{i,t}(\theta) \right], \tag{8}$$

where the token-level objective $\mathcal{O}_{i,t}(\theta)$ prevents excessive policy updates via decoupled clipping:

$$\mathcal{O}_{i,t}(\theta) = \min\Big(r_{i,t}(\theta)A_{i,t}^{total}, \\ \text{clip}\big(r_{i,t}(\theta), 1-\varepsilon_{\text{low}}, 1+\varepsilon_{\text{high}}\big)A_{i,t}^{total}\Big). \tag{9}$$

The probability ratio $r_{i,t}(\theta)$ is defined as in Eq. (10).

$$r_{i,t}(\theta) = \frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})}. \tag{10}$$

3.4 Multi-Stage Data Generation Framework

As depicted on the upper half of Figure 2, our multi-stage automated data augmentation framework comprises three main stages. The first creates a dataset for supervised fine-tuning using CoT reasoning. The subsequent two stages generate data tailored for reinforcement learning stage.

For the initial supervised training, we curate Verilog code from online repositories, perform syntax checking, and then use the DeepSeek-R1 model to enrich the samples with natural language descriptions and CoT reasoning chains. This foundational dataset fosters the model's initial reasoning and Verilog generation capabilities.

To generate data for accurate rewards computation, we first address the **functional reward**. We use GPT-5 to generate multi-case testbenches for our Verilog codes. A verification pipeline then ensures that only functionally correct code-testbench pairs are included in the RL training set, providing significant validation feedback.

Next, to provide the **PPA performance reward**, we use an automated backend pipeline to extract power, area, and delay metrics via NanGate45 process simulations. Validated designs with their PPA metrics are integrated into the dataset.

To enable multi-objective optimization, we employ a specialized data augmentation strategy consisting of two key steps: prompt augmentation and preference vector generation.

For **Prompt Augmentation**, we augment each instruction in our RL dataset with a fine-grained design priority, such as "Focus on minimizing the hardware area." This guides the model to optimize for specific PPA metrics.

To complement these prompts, we assign a corresponding **Preference Vector** $\vec{p}(q) = (p_P, p_D, p_A)$ according to a predefined set of templates. This vector is used to compute the preference-weighted advantage in Eq. 6. Under the constraint $\sum_{m\in P,D,A} p_m = 1$, the template ensures the preferred metric receives the largest weight.

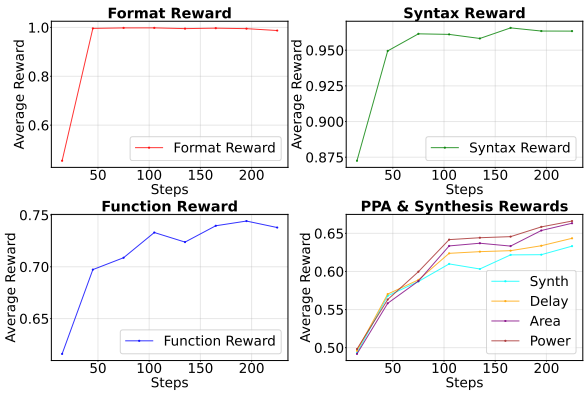

Figure 3: The Verilog code rewards and format reward increase during reinforcement learning. Each plotted point is computed as an average over a window of 30 neighboring steps to smooth the curves.

4 Experiment

4.1 Implementation Details

We adopt the CodeLlama-7B, CodeQwen-7B, DeepSeek-Coder-v1.5-7B and Qwen2.5-Coder-7B-Instruct model as our base models. During the cold-start fine-tuning stage, we use *29,127* data samples to impart preliminary reasoning and Verilog generation capabilities to the model. Subsequently, during reinforcement learning stage, we further train the model using *8,453* data samples to enhance its capabilities. Both training datasets are strictly filtered to exclude any cases from the benchmark. We conduct all training processes on a cluster of 4 NVIDIA A100 80GB GPUs, leveraging the DeepSpeed distributed training framework and the vLLM inference framework to accelerate training.

Furthermore, we employ a multi-faceted strategy to accelerate the computationally intensive reward calculation process. First, we utilize a thread pool to concurrently run multiple simulation and synthesis tasks. Additionally, we implement a hierarchical gating technique based on a "fail-fast" principle. In this pipeline, any Verilog sample that fails an early-stage evaluation is immediately discarded and not passed to subsequent more costly stages. For example, a design that fails the functional correctness simulation will not proceed to the PPA evaluation. This multistage filtering dramatically reduces computation time by avoiding full evaluations for infeasible candidates, significantly accelerating the overall reward calculation process.

Type	Model	Size	VerilogEval Machine (%)			VerilogEval Human (%)			RTLLM v1.1 (%)	
			pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	Syntax@5	Pass@5
Foundational Models	GPT-4o	-	65.9	71.4	72.7	57.1	63.9	66.7	93.9	65.5
	GPT-o1	-	67.4	79.2	81.8	58.5	68.0	71.2	93.1	72.4
Base Models	CodeLlama	7B	43.1	47.1	47.7	18.2	22.7	24.3	62.6	29.9
	DeepSeek-Coder	6.7B	52.2	55.4	56.8	30.2	33.9	34.9	64.4	29.3
	CodeQwen	7B	46.5	54.9	56.4	22.5	26.1	28.0	65.8	34.0
	Qwen2.5-Coder	7B	51.3	76.3	81.8	27.8	43.6	48.7	86.2	48.3
Origen	DeepSeek-Coder	7B	74.1	82.4	85.7	54.4	60.1	64.2	-	65.5
ReasoningV	Qwen2.5-Coder	7B	73.6	83.4	85.3	57.8	69.3	72.4	-	62.2
RTLCoder	Mistral	7B	62.5	72.2	76.6	36.7	45.5	49.2	73.7	37.3
	DeepSeek-Coder	7B	61.2	76.5	81.8	41.6	50.1	53.4	83.9	40.3
CodeV	CodeLlama	7B	78.1	86.0	88.5	45.2	59.5	63.8	89.2	50.3
	DeepSeek-Coder	6.7B	77.9	88.6	90.7	52.7	62.5	67.3	87.4	51.5
	CodeQwen	7B	77.6	88.2	90.7	53.2	65.1	68.5	89.5	53.3
CraftRTL	CodeLlama	7B	78.1	85.5	87.8	63.1	67.8	69.7	93.9	52.9
	DeepSeek-Coder	6.7B	77.8	85.5	88.1	65.4	70.0	72.1	92.9	58.8
	Starcoder2	15B	81.9	86.9	88.1	**68.0**	72.4	74.6	93.9	65.8
ChipSeek	CodeLlama	7B	85.7	88.8	89.5	63.4	70.1	72.4	93.1	**82.8**
	Deepseek-Coder	7B	83.3	88.9	90.2	64.3	71.1	73.7	93.1	72.4
	CodeQwen	7B	**87.2**	90.3	90.9	63.8	69.4	70.5	89.7	75.8
	Qwen2.5-Coder	7B	84.1	**90.6**	**92.3**	62.2	**73.7**	**76.9**	**96.6**	**87.2**

Table 1: Evaluation Results on VerilogEval (Cui et al., 2025) and RTLLM v1.1 (Liu et al., 2024b). We compare our models with GPT series, 4 coding language models, and several Verilog specific models including RTLCoder (Liu et al., 2024a), CodeV (Zhao et al., 2024), Origen (Cui et al., 2025), CraftRTL (Liu et al., 2025) and ReasoningV (Qin et al., 2025).

As shown in Figure 3, the rewards designed in Section 3.2 steadily increase with training steps. The rise in format reward indicates that the model has learned to apply chain of thought before code generation. The increase in Verilog code reward reflects the model's improved chip design capability during reinforcement learning, including enhanced syntax correctness, functional correctness, and PPA performance. This upward trend in rewards provides preliminary evidence for the effectiveness of our method.

4.2 Results

Functional Correctness: In Table 1, we compare the Verilog functional correctness of our proposed model against several baseline models on RTLLM v1.1 and VerilogEval benchmarks. On the VerilogEval benchmark, our model ChipSeek achieves the best performance for *pass@1*, *pass@5*, and *pass@10* in the Machine track. In the Human track, it attains state-of-the-art results for *pass@5* and *pass@10*, and its performance for *pass@1* is comparable to the previous best. On the RTLLM v1.1 benchmark, our generated code achieves the highest *pass@5* rate in both functionality and syntactical correctness, surpassing the previous best by 21.4% and 2.7% respectively. In Table 1, we highlight the best scores in boldface.

Overall Performance Evaluation: To compare the core design capabilities of different models, we first conduct an overall performance evaluation on RTLLM v2.0 without any specified optimization objectives, with the results shown in Table 11. We focus our comparison on two key metrics: pass@5 and the comprehensive Energy-Delay-Area Product (EDAP). We evaluate the original benchmark, GPT series, RTLCoder series, CodeV series, Origen, Verigen-MCTS and our ChipSeek series. For each design prompt, every large language model generates 10 candidate solutions. We then select the functionally correct designs and report the maximum, mean, and minimum EDAP scores among these candidates. Our ChipSeek models achieve state-of-the-art results, with our best configuration improving functional correctness (pass@5) by 11.4% and reducing the worst, average and best EDAP by 20%, 18% and 9% respectively compared to the previous best-performing models.

Performance Evaluation With Design Priorities: To investigate the model's fine-grained design capabilities, we augmented the RTLLM v2.0 bench-

Model	Func. ↑	EDAP ↓		
	pass@5	max	avg	min
GPT-4o	70.45	1.13	1.02	0.84
GPT-o1	72.73	1.14	1.06	0.99
CodeLlama	43.18	1.04	0.94	0.87
DeepSeek-Coder-7B	56.82	3.00	2.65	1.91
CodeQwen	38.64	2.76	2.56	2.41
Qwen2.5-Coder	56.82	1.38	1.04	0.89
Verigen-MCTS	45.45	0.96	0.93	0.87
RTLCoder-M	50.00	2.97	2.43	0.83
RTLCoder-DC-6.7B	52.27	5.83	3.73	1.41
CodeV-CL	59.09	2.12	1.43	1.02
CodeV-DC-6.7B	63.64	2.30	1.56	1.28
CodeV-CQ	61.36	3.35	2.53	1.03
Origen-DC-7B	65.91	3.17	1.45	0.85
ChipSeek-CL	77.27	0.85	**0.76**	**0.74**
ChipSeek-DC-7B	75.00	0.93	0.84	0.78
ChipSeek-CQ	72.73	**0.84**	0.83	0.81
ChipSeek-QC	**84.09**	0.96	0.88	0.81

Table 2: A comparison of various models and methods on function and overal performance. EDAP values are normalized to the original benchmark. QC: Qwen2.5 Coder, M: Mistral, DC: DeepSeek-Coder, CQ: CodeQwen, CL: CodeLlama

Model	Delay↓	Area↓	Power↓	ADP↓	EDP↓	Error↓
Base Model	0.93	1.02	1.02	1.02	1.02	0.46
Origen	0.88	0.93	0.93	0.85	0.86	0.34
ChipSeek	0.83	0.88	0.85	0.80	0.77	0.25

Table 3: Specific design goals optimization capability of Verilog Coding Models with the DeepSeek-Coder-v1.5-7B as the Base Model.

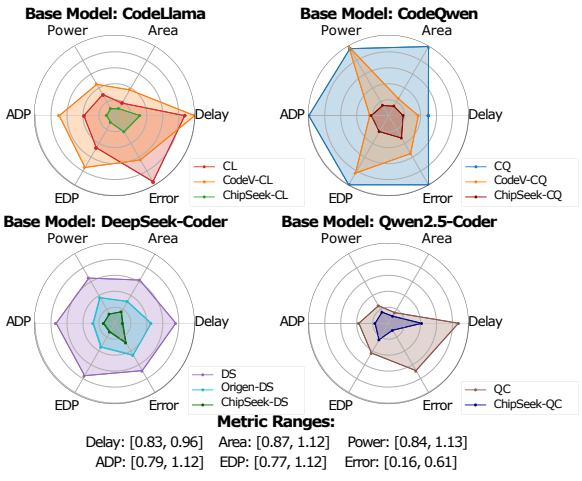

Figure 4: Radar chart comparison of key performance metrics.

mark with specific optimization objectives communicated through targeted prompts. We measure the metrics including power, delay, area, ADP (Area-Delay Product), EDP (Energy-Delay Product) and functional error rate across 11 models grouped by 4 base models, with the comprehensive results visualized in the radar charts in Figure 4. We choose the best metric score across 5 attempts for each problem. Each axis on the charts represents a normalized score, where values closer to the center signify better performance. The visual evidence shows that the polygon representing the ChipSeek model is substantially smaller and more centered than that of its corresponding baseline in every comparison, demonstrating superior results across all metrics simultaneously.

For a quantitative view, Table 3 details the results on the DeepSeek-Coder-7B based models. Compared to the strong Origen baseline, our ChipSeek method significantly improves all metrics compared to the base model, reducing delay by 5%, area by 5%, power by 8%, ADP by 5% and EDP by 9%. This comprehensive improvement highlights the sophisticated trade-off navigation learned through our dynamic preference reinforcement learning framework.

Baseline barrel_shifter

```
module barrel_shifter (in, ctrl, out);
  input  [7:0] in;
  input  [2:0] ctrl;
  output [7:0] out;
wire [7:0] x,y;
mux2X1 ins_27 (.in0(in[7]),.in1(1'b0),
.sel(ctrl[2]),.out(x[7]));//4 bit shift
……
mux2X1 ins_17 (.in0(x[7]),.in1(1'b0),
.sel(ctrl[1]),.out(y[7]));//2 bit shift
……
mux2X1 ins_07 (.in0(y[7]),.in1(1'b0),
.sel(ctrl[0]),.out(out[7]));//1 bit shift
……
endmodule
module mux2X1 ( in0,in1,ssel,out);
input in0, in1, sel;
output out;
assign out=(sel)?in1:in0;
endmodule
```

ChipSeek barrel_shifter

```
module barrel_shifter (
    input  [7:0] in,
    input  [2:0] ctrl,
    output [7:0] out
);
wire [7:0] shift_4;
wire [7:0] shift_2;
wire [7:0] shift_1;
assign shift_4 = ctrl[2] ?
                 (in >> 4) : in;
assign shift_2 = ctrl[1] ?
                 (shift_4 >> 2) : shift_4;
assign shift_1 = ctrl[0] ?
                 (shift_2 >> 1) : shift_2;
assign out = shift_1;
endmodule
```

Clarity: 63.9% lines of code reduction
Performance: 13.3% area reduction
Optimization Strategy: Operator Inference

Figure 5: Case Study of ChipSeek.

Case Study: The barrel shifter is a crucial component in high-performance computing. As shown on the left in Figure 5, the baseline design uses a traditional implementation with explicitly instantiated multiplexer (MUX) sub-modules. In contrast, shown on the right, our model found that describing only the high-level barrel-shift behavior—without employing MUX sub-modules—allows backend EDA tools to perform the optimization of operator inference. This process, combining a large-scale generation of candidate circuits with rapid RL-based iteration and EDA feedback, yields a powerful co-optimization of front-end and back-end design stages.

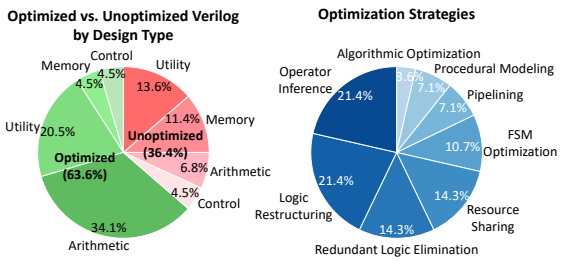

Figure 6: Optimization Analysis of ChipSeek.

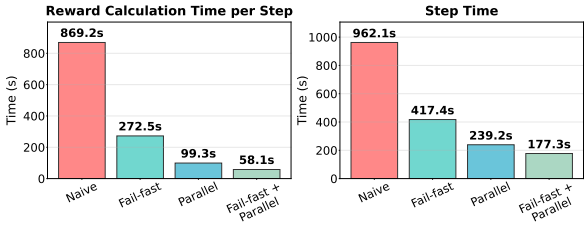

Figure 7: Effect of Acceleration on Training Time.

Method	Syntax↑	Pass@1↑	Synthesis↑	EDAP↓
ChipSeek	91.36%	71.36%	70.91%	0.81
w/o Format	-7.27%	-10.00%	-10.23%	+0.08
w/o Syntax	-2.73%	-0.45%	-0.68%	+0.02
w/o Function	-0.91%	-14.31%	-14.55%	+0.56
w/o Synthesis	+0.45%	-0.22%	-3.18%	+0.05
w/o PPA	-0.68%	-1.36%	-1.36%	+1.10

Table 4: Ablation Study of Rewards. We use Qwen2.5-Coder-7B as the base model to conduct the experiment.

Method	Delay↓	Area↓	Power↓	ADP↓	EDP↓
GRPO	0.89	0.90	0.91	0.85	0.85
DAPO	0.92	0.90	0.89	0.87	0.86
CDPO	0.84	0.87	0.86	0.79	0.77

Table 5: Ablation Study of Reinforcement Learning Methods (Shao et al., 2024; Yu et al., 2025a) on different RTL Performance Optimization goals.

4.3 Ablation Study

We conducted an ablation study on the RTLLM v2.0 to evaluate the contribution of each component within our framework, as detailed in Table 4. In this study, we systematically removed each reward and measured the impact on a suite of metrics: syntax pass rate, pass@5, synthesis pass rate, and EDAP. The results in Table 4 show that the removal of any single component leads to performance degradation to varying degrees. Furthermore, we compared our CDPO algorithm with prior optimization methods. As shown in Table 5, CDPO consistently outperforms both GRPO and DAPO across all RTL performance goals, highlighting its superior capabilities in multi-objective optimization tasks for critical chip design metrics.

4.4 Runtime Analysis

Compact and fast-inference RTL generation models are essential for efficient chip design work flows, particularly given the computational intensity and resource constraints of large-scale designs. Utilizing the acceleration techniques detailed in Section 4.1, we significantly reduce computational overhead, achieving marked improvements in runtime performance. Specifically, as demonstrated in Figure 7, we achieve a substantial reduction of 93.5% in reward calculation time and 81.6% in step update time. To ensure a rigorous and fair evaluation, all experiments were executed with the same training config on an isolated, dedicated server without concurrent processes.

5 Optimization Analysis

Shown in Figure 6, the optimization profile of ChipSeek demonstrates a strong competency in gate-level and structural refinement. The model predominantly leverages techniques such as Operator Inference and Logic Restructuring (both at 21.4%) to enhance design efficiency. This proficiency is particularly impactful on Arithmetic modules, which constitute the largest portion of optimized designs. However, despite these strengths, the model exhibits a comparatively weaker optimization capability on Control and Memory modules. We hypothesize that this stems from a relative scarcity of these circuit types within the training dataset, which may have limited the model's exposure to their unique optimization patterns.

6 Conclusion

We introduce ChipSeek, a reinforcement learning framework for superior RTL generation. By using direct feedback from the EDA toolchain as reward signals within our CDPO algorithm, ChipSeek simultaneously optimizes for both functional correctness and specific PPA goals. On standard benchmarks, this approach allows our model to significantly outperform previous models, demonstrating superior results in both functional correctness and PPA performance. ChipSeek thus enables more efficient automated hardware design by bridging the gap between the training process and EDA toolchain feedback.

Limitations

Our approach still has several practical limitations. First, although the closed-loop design enables direct optimization against compilation, simulation, and synthesis feedback, it remains computationally heavy due to repeated EDA invocations; in particular, reward evaluation still takes a long time even with the proposed gating mechanism to skip unnecessary tool calls. Second, while the optimization profile indicates strong competency in gate-level and structural refinement, the gains are not uniform across circuit categories: consistent with our optimization analysis, ChipSeek exhibits comparatively weaker optimization capability on *Control* and *Memory* modules than on *Arithmetic* and *Utility* modules. We hypothesize that this gap is partly driven by a relative scarcity of control- and memory-centric designs in the training/evaluation data, which limits the model's exposure to their distinct optimization patterns (e.g., stateful control flow and memory access behaviors) and makes it harder to consistently discover effective refinements under the same feedback budget.

Acknowledgements

This work was supported in part by the National Key Research and Development Program of China under Grant Number 2023YFB4404400 and in part by the National Natural Science Foundation of China under Grant Number 92473205. The corresponding author is Ying Wang.

References

Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, and 2 others. 2019. Invited: Toward an open-source digital flow: First learnings from the openroad project. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4.

Kaiyan Chang, Zhirong Chen, Yunhao Zhou, Wenlong Zhu, Kun Wang, Haobo Xu, Cangyuan Li, Mengdi Wang, Shengwen Liang, Huawei Li, Yinhe Han, and Ying Wang. 2025. Natural language is not enough: Benchmarking multi-modal generative ai for verilog generation. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '24, New York, NY, USA. Association for Computing Machinery.

Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, Zhuoliang Zhao, Yuan Cheng, Yudong Pan, Yiqi Liu, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2024. Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, page 1–6. ACM.

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, and Dahua Lin. 2025. 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*, ICCAD '24, New York, NY, USA. Association for Computing Machinery.

Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, Shailja Thakur, Ramesh Karri, Siddharth Garg, and Jeyavijayan Rajendran. 2024. Make every move count: Llm-based high-quality rtl code generation using mcts. *Preprint*, arXiv:2402.03289.

Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. 2024. Autovcoder: A systematic framework for automated verilog code generation using llms. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, pages 162–169.

Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2025. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(1):300–307.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

Cangyuan Li, Chujie Chen, Yudong Pan, Wenjun Xu, Yiqi Liu, Kaiyan Chang, Yujie Wang, Mengdi Wang, Huawei Li, Yinhe Han, and Ying Wang. 2025. Autosilicon: Scaling up rtl design generation capability of large language models. *ACM Trans. Des. Autom. Electron. Syst.*, 30(6).

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. *Preprint*, arXiv:2409.12993.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024a. Rtlcoder: 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(invited). In *Proceedings of 2024 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM.

Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. 2024. Betterv: controlled verilog generation with discriminative guidance. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

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. *Preprint*, arXiv:2504.14560.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1279–1297, New York, NY, USA. Association for Computing Machinery.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2023. VeriGen: A large language model for verilog code generation. *Preprint*, arXiv:2308.00708.

Yunda Tsai, Mingjie Liu, and Haoxing Ren. 2024. Rtlfixer: Automatically fixing rtl syntax errors with large language model. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA. Association for Computing Machinery.

Yiting Wang, Wanghao Ye, Ping Guo, Yexiao He, Ziyao Wang, Yexiao He, Bowei Tian, Shwai He, Guoheng Sun, Zheyu Shen, Sihan Chen, Ankur Srivastava, Qingfu Zhang, Gang Qu, and Ang Li. 2025. Sym-RTLO: Enhancing rtl code optimization with llms and neuron-inspired symbolic reasoning. *Preprint*, arXiv:2504.10369.

Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free verilog synthesis suite.

Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024a. Hdldebugger: Streamlining hdl debugging with large language models. *Preprint*, arXiv:2403.11671.

Xufeng Yao, Yiwen Wang, Xing Li, Yingzhao Lian, Ran Chen, Lei Chen, Mingxuan Yuan, Hong Xu, and Bei Yu. 2024b. Rtlrewriter: Methodologies for large models aided rtl code optimization. *Preprint*, arXiv:2409.11414.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, and 16 others. 2025a. Dapo: An open-source llm reinforcement learning system at scale. *Preprint*, arXiv:2503.14476.

Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine Lin, Yong Liu, and Haoxing Ren. 2025b. Spec2rtl-agent: Automated hardware code generation from complex specifications using llm agent systems. *Preprint*, arXiv:2506.13905.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Ziyuan Nan, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. 2024. Codev: Empowering llms for verilog generation through multi-level summarization. *Preprint*, arXiv:2407.10424.

A CDPO Algorithm

Algorithm 1 CDPO (Curriculum-Guided Dynamic Policy Optimization)

Require: Policy π_θ, behavior policy $\pi_{\theta_{\text{old}}}$; batch size B, group size G; process rewards $\mathcal{R}_{pro}$, objective rewards $\mathcal{R}_{cor} = \{\text{func}, P, D, A\}$; EMA factor β, ϵ, clip bounds $\varepsilon_{\text{low}}, \varepsilon_{\text{high}}$, w_{func}; curriculum weights $\{\alpha_k\}_{k \in \mathcal{R}_{pro}}$

Ensure: Updated θ and $\{\alpha_k\}$

for training step $s = 1, 2, \ldots$ **do**
  Sample prompts $\{q_j\}_{j=1}^{B}$
  for each prompt q_j **do**
    $\mathbf{p}(q_j) \leftarrow \textsc{ExtractPref}(q_j)$
    Sample $\{o_{j,i}\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q_j)$
    $\{r_{j,i}^{(k)}\} \leftarrow \textsc{EvalToolchain}(q_j, o_{j,i})$
    $\{\widehat{A}_{j,i}^{(k)}\} \leftarrow \textsc{Advantage}(\{r_{j,i}^{(k)}\}_{i=1}^{G}, \epsilon)$
  end for
  for each $k \in \mathcal{R}_{pro}$ **do**
    $\bar{\mu}_k^{(s)} \leftarrow \frac{1}{BG} \sum_{j=1}^{B} \sum_{i=1}^{G} r_{j,i}^{(k)}$
    $\hat{\alpha}_k^{(s)} \leftarrow \max(0, 1 - \bar{\mu}_k^{(s)})$
    $\alpha_k^{(s)} \leftarrow \beta\, \alpha_k^{(s-1)} + (1 - \beta)\, \hat{\alpha}_k^{(s)}$
  end for
  $\mathcal{L} \leftarrow 0$
  for each token $o_{j,i,t}$ in all rollouts **do**
    $A^{pro} \leftarrow \sum_{k \in \mathcal{R}_{pro}} \alpha_k^{(s)}\, \widehat{A}_{j,i}^{(k)}$
    $A^{cor} \leftarrow w_{func} \widehat{A}_{j,i}^{(func)} + \sum_{m \in \{P,D,A\}} p_m(q_j) \widehat{A}_{j,i}^{(m)}$
    $A^{total} \leftarrow A^{pro} + A^{cor}$
    $r(\theta) \leftarrow \frac{\pi_\theta(o_{j,i,t} \mid q_j, o_{j,i,<t})}{\pi_{\theta_{\text{old}}}(o_{j,i,t} \mid q_j, o_{j,i,<t})}$
    $\tilde{r} \leftarrow \textsc{Clip}(r, 1 - \varepsilon_{\text{low}}, 1 + \varepsilon_{\text{high}})$
    $\mathcal{L} \leftarrow \mathcal{L} - \min\big(r A^{total}, \tilde{r} A^{total}\big)$
  end for
  Update θ by minimizing $\mathcal{L}$; set $\theta_{\text{old}} \leftarrow \theta$ *(RL step)*
end for

B Implementation Details

ChipSeek implements a comprehensive multi-stage reward evaluation system specifically designed for Verilog code generation with PPA (Power, Performance, Area) optimization using Dynamic Preference Policy Optimization (CDPO). The reinforcement learning algorithm is built upon VeRL (Sheng et al., 2025). The core architecture consists of a specialized reward manager that orchestrates the evaluation of generated Verilog designs across multiple dimensions including functional correctness, synthesis feasibility, and PPA performance metrics. The system incorporates several critical training speed optimizations including DeepSpeed ZeRO-2 acceleration strategy for efficient distributed training and vLLM for high-throughput inference generation.

The functional correctness evaluation phase utilizes Icarus Verilog simulation within containerized environments to assess both syntactic validity and behavioral correctness of generated designs. Each Verilog code snippet is compiled with comprehensive warning flags and executed against provided testbenches to determine functional accuracy. This phase produces binary rewards for compilation success and testbench passage, establishing a foundational requirement that designs must be functionally correct before proceeding to performance evaluation. To optimize training efficiency, the system implements intelligent filtering mechanisms that remove training samples where all generated responses have identical reward values, thereby focusing computational resources on diverse and informative examples.

For designs that pass functional verification, the system performs comprehensive PPA analysis using Yosys (Wolf et al., 2013) and OpenROAD (Ajayi et al., 2019) in a secondary evaluation phase. The synthesis process extracts detailed power consumption, area utilization, and timing performance metrics from the generated designs. A key innovation lies in the dynamic preference-based evaluation mechanism, where user-specified preference vectors allow flexible weighting of different PPA objectives, enabling the system to optimize for power-efficient, area-efficient, or performance-optimized designs based on application requirements. The reward manager implements efficient parallel processing with ThreadPoolExecutor to handle batch evaluation of multiple design candidates simultaneously, significantly reducing evaluation time during training. All third-party benchmarks/tools are used in accordance with their original licenses, and we will release our code and scripts under an open-source license.

The training configuration employs carefully tuned hyperparameters optimized for Verilog code generation tasks. The system uses asymmetric clipping ratios with clip_ratio_low=0.2 and clip_ratio_high=0.28 to provide more flexibility for positive policy updates while maintaining strict constraints on negative updates. Training operates with a batch size of 32 prompts,

Model	VerilogEval-Machine(%)			VerilogEval-Human(%)		
	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
CodeLlama	43.1	47.1	47.7	18.2	22.7	24.3
+SFT	48.1	60.2	69.9	27.8	53.6	58.9
+SFT+CDPO	85.7	88.8	89.5	63.4	70.1	72.4
DeepSeek-Coder	52.2	55.4	56.8	30.2	33.9	34.9
+SFT	55.7	75.4	78.1	33.8	45.8	52.3
+SFT+CDPO	83.3	88.9	90.2	64.3	71.1	73.7
CodeQwen	46.5	54.9	56.4	22.5	26.1	28.0
+SFT	51.5	78.9	82.3	31.9	49.5	53.8
+SFT+CDPO	87.2	90.3	90.9	63.8	69.4	70.5
Qwen2.5-Coder	51.3	76.3	81.8	27.8	43.6	48.7
+SFT	57.3	75.4	79.0	34.7	49.9	54.5
+SFT+CDPO	84.1	90.6	92.3	62.2	73.7	76.9

Table 6: An ablation study of Training Stage on VerilogEval.

generating 8 responses per prompt, and uses a conservative learning rate of 1e-6 with 10 warmup steps and weight decay of 0.1. The generation process employs high diversity settings with temperature=1.0 and top_p=1.0 during training, while validation uses more conservative top_p=0.7 for stable evaluation. Sequence lengths are configured for max_prompt_length=2048 and max_response_length=8192 tokens to accommodate complex Verilog design descriptions.

Memory and computational efficiency optimizations include FSDP (Fully Sharded Data Parallel) with parameter and optimizer offloading enabled, sequence parallelism with sp_size=2, and dynamic batch sizing to maximize GPU utilization. The system uses vLLM (Kwon et al., 2023) with tensor model parallelism (gen_tp=2) for inference acceleration, chunked prefill for memory efficiency, and gradient checkpointing to reduce memory footprint during training. Additionally, an overlong buffer mechanism with penalty_factor=1.0 discourages excessively long responses that could impact training stability. We use 13 x 4 A100 GPU hours to fully train our models.

Response format enforcement is integrated through structured output validation, requiring generated content to follow a reasoning-then-answer format with explicit thinking and solution sections. The final reward signal combines compilation feasibility, functional correctness, PPA improvement ratios relative to reference implementations, synthesis feasibility, and format compliance into a comprehensive score that guides the CDPO training process. This multi-faceted approach enables ChipSeek to learn nuanced trade-offs between different design objectives while maintaining computational efficiency through strategic filtering and acceleration techniques.

C Evaluation Details

For functionality evaluation, we use the unbiased pass@k metrics to evaluate the functional correctness of the generated designs, calculated as Equation 11. This metric estimates the probability that at least one functionally correct design is generated out of k attempts. n represents the total number of generations and c represents the number of successfully generated code.

$$\text{pass@}k := \mathbb{E}_{\text{task}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \tag{11}$$

For performance evaluation of a design, we use fine-grained metrics including Delay, Area, Power and comprehensive metrics including ADP (Area-Delay-Product), EDP (Energy-Delay-Product) and EDAP (Energy-Delay-Area Product), calculated as Equation 12, 13, 14.

$$\text{ADP}_{\text{gen}} = \text{area}_{\text{gen}} \times \text{delay}_{\text{gen}} \tag{12}$$

$$\text{EDP}_{\text{gen}} = \text{delay}_{\text{gen}} \times \text{power}_{\text{gen}} \tag{13}$$

$$\text{EDAP}_{\text{gen}} = \text{area}_{\text{gen}} \times \text{delay}_{\text{gen}} \times \text{power}_{\text{gen}} \tag{14}$$

Model	RTLLM v2.0-Func (%)		RTLLM v2.0-Performance					
	Syntax	**pass@5**	**Power**	**Area**	**Delay**	**ADP**	**EDP**	**EDAP**
CodeLlama	36.4	34.1	0.93	0.92	0.99	0.95	0.96	0.94
+SFT	78.2	62.3	1.20	1.02	1.04	1.09	1.25	1.47
+SFT+CDPO	89.5	77.3	0.87	0.91	0.91	0.83	0.80	0.76
DeepSeek-Coder	67.5	50.1	1.13	1.13	1.00	1.17	1.17	2.65
+SFT	80.2	64.0	1.05	1.04	1.00	1.06	1.08	1.51
+SFT+CDPO	88.6	75.0	0.91	0.93	0.83	0.90	0.88	0.84
CodeQwen	33.8	33.8	1.18	1.18	0.95	1.20	1.21	2.56
+SFT	81.6	69.1	1.05	1.06	1.02	1.10	1.10	1.42
+SFT+CDPO	87.3	72.7	0.89	0.92	0.94	0.88	0.86	0.82
Qwen2.5-Coder	67.0	53.1	0.96	0.95	1.03	1.01	1.01	1.04
+SFT	79.5	68.5	1.07	1.03	1.05	1.10	1.15	1.31
+SFT+CDPO	91.4	84.1	0.93	0.92	0.95	0.91	0.92	0.88

Table 7: An ablation study of Training Stage on RTLLM-v2.0. Energy, Delay, Area, ADP, EDP, EDAP are all normalized to the benchmark baseline.

Method	Syntax	Pass@5	Synthesis	Power	Area	Delay	ADP	EDP	EDAP
ChipSeek	91.4	84.1	70.91	0.93	0.92	0.95	0.91	0.92	0.88
-AdvAgg	88.7	78.1	68.8	0.95	0.94	0.96	0.93	0.94	0.92
-Curriculum	90.8	72.5	63.2	0.96	0.95	0.95	0.94	0.94	0.93
-PreVec	91.2	84.0	70.1	0.95	0.93	0.96	0.94	0.95	0.93

Table 8: Ablation study on the effectiveness of key components in ChipSeek. **AdvAgg** denotes *Advantage-Level Aggregation*, **Curriculum** denotes the *Curriculum Weight Schedule*, and **PreVec** denotes the *Preference Vector*.

For an overall performance evaluation of a model, we first filter the entire set of generated designs for each task, $\mathcal{G}_d$, to retain only those that are functionally correct and pass the testbench. This creates a filtered set of valid designs, shown in Equation 15.

$$S_d = \{s \in \mathcal{G}_d \mid s \text{ passes the testbench}\} \quad (15)$$

For each valid design $s \in S_d$, we then normalize its performance metric, $\text{Perf}(s)$, against the reference design's performance, $\text{Perf}_{\text{ref}}(d)$, for that task, calculating the normalized score as Equation 16.

$$\text{Perf}_{\text{norm}}(s) = \text{Perf}(s)/\text{Perf}_{\text{ref}}(d) \quad (16)$$

where

$$\text{Perf} \in \{\text{Delay}, \text{Area}, \text{Power}, \text{ADP}, \text{EDP}, \text{EDAP}\} \quad (17)$$

From this set of normalized scores for each task, we select the best (minimum), average, and worst (maximum) candidates to characterize the model's performance range on that specific task. The scores for each task d are calculated as follows:

$$\text{Score}_{\text{best}}(d) = \min_{s \in S_d} \{\text{Perf}_{\text{norm}}(s)\} \quad (18)$$

$$\text{Score}_{\text{avg}}(d) = \frac{1}{|S_d|} \sum_{s \in S_d} \text{Perf}_{\text{norm}}(s) \quad (19)$$

$$\text{Score}_{\text{worst}}(d) = \max_{s \in S_d} \{\text{Perf}_{\text{norm}}(s)\} \quad (20)$$

Finally, the overall evaluation score for the model is determined by averaging these individual task scores across all successfully generated tasks $\mathcal{T}$ in the benchmark. This provides a comprehensive view of the model's capabilities, captured in the final metric:

$$\text{Perf}_{\text{method}} = \frac{1}{|\mathcal{T}|} \sum_{d \in \mathcal{T}} \text{Score}_{\text{method}}(d) \quad (21)$$

where the subscript method can be substituted with best, avg, or worst to obtain the corresponding overall performance score.

Model	PDK			Synthesis Opt		Synthesis Tool	
	Sky130	gf180	ihp_sg13g2	Timing	Area	Yosys	DC
RTLCoder	2.33	2.11	2.30	2.72	2.66	2.43	0.97
CodeV	2.52	2.21	2.50	2.91	3.00	2.53	1.78
Origen	1.46	1.37	1.41	1.52	1.59	1.45	0.97
GPT-4	0.97	0.97	0.97	0.99	0.97	1.02	0.99
GPT-o1	1.03	1.04	1.01	1.00	1.07	1.06	1.00
ChipSeek	0.77	0.78	0.81	0.82	0.78	0.76	0.81

Table 9: Results under different PDKs and synthesis configurations. Values are normalized to the benchmark baseline.

D Detailed Results and More Ablation Study

To comprehensively evaluate the effectiveness of our proposed training pipeline, we conduct a series of ablation studies. We begin by analyzing the impact of each training stage on functional correctness using the VerilogEval benchmark. The results, presented in Table 6, demonstrate the contribution of Supervised Fine-Tuning (SFT) and the subsequent reinforcement learning stage (CDPO). Across all four base models, the application of SFT provides an improvement in 'pass@k' metrics. The introduction of the CDPO stage further elevates performance, leading to significant gains in functional correctness.

We then extend our ablation study to assess the impact of the training stages on PPA metrics. As shown in Table 7, while the SFT stage continues to improve functional metrics ('Syntax' and 'pass@5'), it sometimes leads to a degradation in performance metrics. This observation underscores a critical challenge: optimizing for functionality does not guarantee optimization for PPA. However, the subsequent CDPO stage effectively addresses this issue. The results clearly indicate that CDPO not only enhances functional correctness further but also significantly improves all PPA metrics, bringing most values below the 1.0 baseline. This demonstrates the crucial role of our reinforcement learning approach in optimizing for real-world hardware design constraints.

Table 8 presents an ablation study of ChipSeek built on the **Qwen2.5-7B** base model. We compare the full system with three variants: -AdvAgg (removing *advantage-level aggregation*, signals are aggregated in reward level), -Curriculum (removing the *curriculum weight schedule*, process rewards are assigned with the constant weights), and -PreVec (removing the *preference vector*, ppa rewards are assigned with the constant weights). All settings use the same training setup, EDA toolchain, and evaluation pipeline.

We report two types of results. **(i) No-preference evaluation:** *Syntax*, *Pass@5*, *Synthesis*, and *EDAP* are measured on prompts without any PPA preference, reflecting overall correctness and synthesizability. **(ii) Preference-conditioned evaluation:** *Power*, *Area*, and *Delay* (and their composites *ADP/EDP/EDAP*) are measured under the corresponding preference prompts (e.g., *Power* under `power` prompts). All PPA metrics are normalized to the reference, where lower is better.

The full ChipSeek achieves the best overall performance. Removing **AdvAgg** reduces both correctness and PPA quality (e.g., *Pass@5* 84.1→78.1 and *EDAP* 0.88→0.92), indicating that advantage-space aggregation helps stabilize multi-objective optimization. Removing **Curriculum** causes the largest drop in end-to-end success (*Pass@5* 84.1→72.5 and *Synthesis* 70.91→63.2) and also worsens *EDAP* (0.88→0.93), showing the curriculum is important for transitioning from process compliance to PPA optimization. Finally, removing **PreVec** keeps syntax-level capability similar but weakens preference-following and PPA outcomes (e.g., *Power* 0.93→0.95 and *EDAP* 0.88→0.93), confirming that explicit preference conditioning is necessary for controllable PPA optimization.

Next, we present a detailed case-by-case EDAP comparison against the original benchmark designs in Table 11. This table provides raw PPA metrics for a diverse set of hardware modules. The results marked in bold indicate instances where ChipSeek generated a design with superior PPA performance compared to the benchmark. As the data shows, our model successfully optimizes a wide variety of

designs, often achieving significant improvements in one or more PPA metrics.

E Sensitivity Study

We conduct a sensitivity study to assess whether our PPA gains generalize beyond a single toolchain setting. Concretely, we evaluate **average normalized EDAP** on the RTLLM benchmark while varying three production-level factors: *(i)* standard-cell libraries (PDKs), *(ii)* synthesis tools, and *(iii)* synthesis optimization strategies. This analysis probes the robustness of ChipSeek to common sources of variability in EDA flows.

Starting from the same set of RTLLM tasks, we keep the evaluation protocol fixed and only change one component of the synthesis flow at a time. We consider three representative PDK libraries (`Sky130`, `gf180`, `ihp_sg13g2`), two synthesis tools (open-source `Yosys` and commercial `Design Compiler (DC)`), and two typical optimization targets (*timing*-driven vs. *area*-driven synthesis). Following prior RTLLM evaluation, EDAP values are normalized to the benchmark baseline, where smaller values indicate better PPA efficiency.

Table 9 reports the sensitivity results. Across all configurations, ChipSeek consistently achieves the lowest normalized EDAP among all methods, while several baselines exhibit substantial degradation under tool or PDK shifts (e.g., larger variance across libraries and more pronounced sensitivity to timing/area targets). In contrast, ChipSeek maintains stable improvements across *all* tested PDKs, synthesis tools, and optimization strategies, indicating that the model is not overfitting to a single library or a particular synthesis recipe.

These results suggest that ChipSeek learns a *robust Verilog generation style* that transfers across realistic EDA settings. The consistent EDAP gains under different PDK libraries, synthesis tools, and optimization objectives provide evidence that our improvements stem from better structural and micro-architectural coding patterns, rather than artifacts of a specific toolchain configuration.

F Training Analysis

Figure 8 compares the training trajectories of core rewards between CDPO and the DAPO baseline, including the function reward and the three PPA rewards. Across all objectives, CDPO exhibits more stable convergence and reaches consistently higher final reward values. This indicates that CDPO is able to allocate optimization capacity to the truly performance-critical signals, rather than overfitting to shallow constraints.

Model	Pass@5	Avg Normalized EDAP
RTLCoder	5.6%	1.34
CodeV	8.9%	1.27
Origen	6.7%	1.03
GPT-4	27.8%	0.99
GPT-o1	34.4%	1.01
ChipSeek	**53.3%**	**0.86**

Table 10: Scalability results on large-scale designs from AutoSilicon (e.g., CPU/NPU blocks, avg. ∼560 LoC).

A key factor behind this behavior is the curriculum weight schedule in CDPO. As visualized in Figure 9, the weights of process rewards (format, syntax, and synthesis) decay rapidly as training progresses, effectively shifting the learning emphasis from *easy-to-satisfy* constraints to *hard-to-optimize* core objectives. In the early stage, emphasizing process rewards helps the policy quickly learn valid Verilog formatting, syntactic correctness, and basic synthesizability, providing a reliable foundation for downstream EDA evaluation. Once these prerequisites become consistently achievable, their diminishing weights prevent them from dominating the gradient and allow the policy to focus on improving functional correctness and PPA metrics in later stages. This "easy-to-hard" curriculum makes the reinforcement learning process more progressive and better aligned with the ultimate goal of producing high-quality hardware designs.

In contrast, DAPO lacks an explicit curriculum mechanism to down-weight process-level feedback. As a result, even in later training, relatively simple rewards (e.g., format or syntax compliance) can still contribute non-trivially to the overall optimization signal, introducing interference and reducing the effective learning signal for core objectives. This can lead the policy to become overly satisfied with meeting superficial constraints, slowing down further specialization toward functionally correct and PPA-optimized Verilog generation. Overall, Figures 8 and 9 together support that curriculum-guided reweighting is essential for CDPO to achieve stronger and more stable optimization on the most challenging objectives.

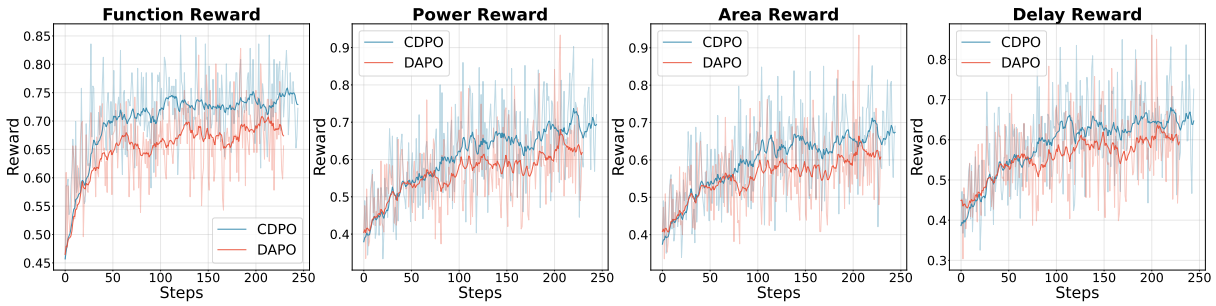

Figure 8: Core reward trajectories comparison between the proposed CDPO and DAPO. CDPO demonstrates superior convergence stability and achieves higher final reward values across all optimization objectives compared to the DAPO baseline. Training steps are mismatched because of the filtering mechanism that omits the equal-reward samples in the training.

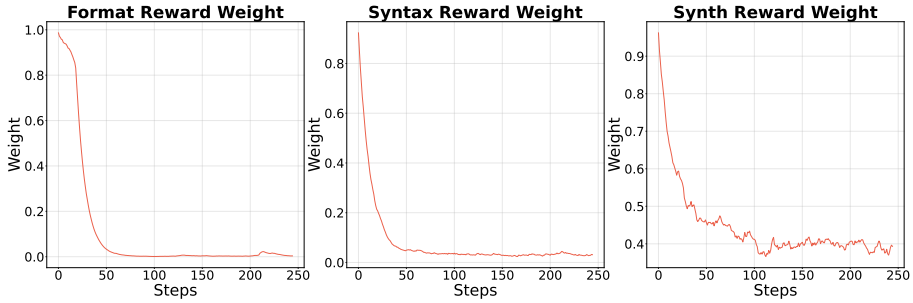

Figure 9: Visualization of the curriculum weight schedule in CDPO.

G Scalability Analysis

To evaluate scalability on more complex RTL generation tasks, we conduct experiments on large-scale designs from AutoSilicon (Li et al., 2025) (e.g., CPU and NPU blocks, *avg.* ∼560 LoC). As shown in Table 10, ChipSeek achieves a substantially higher Pass@5 (53.3%) than the strongest baseline (34.4%), indicating improved functional reliability as program length and structural complexity increase. Meanwhile, ChipSeek also attains the best Avg. Normalized EDAP (0.86), outperforming prior methods and large general-purpose LMs (e.g., GPT-4 at 0.99), which suggests that our EDA-integrated optimization remains effective in steering generation toward better PPA even under a much larger design space. Overall, these results confirm that ChipSeek scales favorably to production-level RTL designs, improving both correctness and hardware quality on long and complex modules.

H Detailed Case Study

In this section, we illustrate the model's reasoning trajectory and show how it improves the design in both functional correctness and PPA.

A barrel shifter is a key building block in high-performance systems. Unlike a conventional shifter, it can perform multi-bit shifts within a single clock cycle. A common textbook implementation uses a hierarchy of multiplexers, where each stage shifts by a power-of-two offset (e.g., 1, 2, or 4 bits), as shown in the golden solution in Figure 11. In this task, the human-provided specification highlights a mux2X1-based construction as a clear structural reference.

Importantly, while preserving the required I/O behavior and functional semantics, our model identifies a more synthesis-friendly realization: it expresses the same shifting logic using a concise behavioral description rather than explicitly instantiating a MUX hierarchy. This formulation exposes the underlying logic more directly to the synthesis engine, enabling the backend EDA toolchain to explore a larger optimization space and select the most efficient gate-level mapping under the target library and constraints.

As a result, compared to a literal mux2X1-instantiated implementation, the model's final design achieves a 13.3% reduction in area and a 63.9% reduction in code size, as shown in Figure 12. This case study highlights the model's ability to satisfy human intent (functional requirements and design constraints) while producing implementa-

Barrel Shifter Design Description

Solve the following coding problem using the programming language Verilog. Output the code between ```verilog and ```.
Please act as a professional verilog designer.
Module name: barrel_shifter
Function:
A barrel shifter for rotating bits efficiently. This 8-bit barrel shifter takes an 8-bit input and shifts or rotates the bits based on a 3-bit control signal.
Input ports:
in [7:0]: 8-bit input to be shifted.
ctrl [2:0]: 3-bit control signal that determines how many positions to shift. The control signal specifies the amount of shift, where each bit indicates a shift by 1, 2, or 4 positions.
Output ports:
out [7:0]: 8-bit shifted output.
Implementation:
The barrel shifter shifts the input ‘in‘ based on the value of the ‘ctrl‘ signal. The shifting happens in stages, where each stage shifts by 4, 2, or 1 position, as controlled by ‘ctrl[2]‘, ‘ctrl[1]‘, and ‘ctrl[0]‘, respectively.
The shifts are achieved using a series of 2-to-1 multiplexers (‘mux2X1‘) that select between the shifted input and the original input. Each stage of multiplexers performs the shift, and the result is passed to the next stage for further shifting if required.
- When ‘ctrl[2]‘ is high, the input is shifted by 4 positions.
- When ‘ctrl[1]‘ is high, the intermediate result is shifted by 2 positions.
- When ‘ctrl[0]‘ is high, the intermediate result is shifted by 1 position.
The ‘mux2X1‘ submodule is used to select between two input values based on the ‘sel‘ signal, determining which bit is used in the shifted output at each stage.
Please generate the Verilog code optimized for area.

Figure 10: Barrel Shifter Design Description

tions that better align with downstream synthesis optimization, ultimately improving hardware metrics.

I Prompt Templates used in Automatic Data Augmentation

This section outlines the structured prompt engineering methodology used in the automatic data augmentation. The approach leverages a series of specialized prompt templates to guide the language model toward specific objectives, including comprehensive function verification and design optimization across various metrics such as delay, area, and power.

I.1 Testbench Generation Prompt

The generation of testbench data for function verification is initiated using the following prompt template. This template instructs the model to act as a Verilog testbench engineer, create a sufficient number of test cases based on design complexity, report the outcome, and terminate the simulation correctly.

Testbench Generation Prompt Template

You are a professional Verilog testbench engineer. You should use at least 10 test cases to verify the design. The number of test cases should be based on the difficulty of the design. If the design passes all the test cases, please display “Design passed". If the design does not pass all the test cases, please display “Design failed with <error count> errors out of <total count> test cases". Remember to use the $finish instruction in the end.
Please generate the testbench code for the Verilog code below:
Verilog description: {instruction}
Verilog code: {output}
Please generate the testbench code below:

I.2 System and Code Formatting Prompts

To enhance the quality and utility of the model's output, two key prompts are prepended to the main task.

I.2.1 Thinking System Prompt

To encourage a detailed and well-reasoned generation process, a "thinking" system prompt is employed. This prompt instructs the model to first articulate its reasoning as an internal monologue before providing the final answer, ensuring a more transparent and logical workflow.

Thinking System Prompt

You are a helpful AI Assistant that provides well-reasoned and detailed responses. You first think about the reasoning process as an internal monologue and then provide the user with the answer. Respond in the following format:
<think>\n...\n</think>
<answer>\n...\n</answer>

I.2.2 Code Guiding Prompt

To ensure the generated Verilog code can be programmatically extracted and parsed, the following code-guiding prompt is used. It specifies a clear demarcation for the code block.

Code Guiding Prompt

Solve the following coding problem using the programming language Verilog. Output the code between ```verilog and ```.

I.3 Design Optimization Prompts

For hardware design generation, the data augmentation system incorporates prompts targeting specific optimization goals. Each data sample is augmented with one prompt randomly selected from a pool corresponding to one of five optimization priorities.

I.3.1 Timing/Delay Priority

When **timing performance** is the primary optimization objective, one of the following prompts is used:

Prompt Templates with Delay as the Design Priority

1. Please generate the Verilog code optimized for timing performance.
2. Focus on minimizing delay and maximizing speed in your Verilog implementation.
3. Optimize your Verilog design for high-speed operation.
4. Please implement the Verilog code with timing performance as the primary goal.

I.3.2 Area Priority

When **area efficiency** is the priority, one of the following prompts is selected:

Prompt Templates with Area as the Design Priority

1. Please generate the Verilog code optimized for area.
2. Focus on minimizing the hardware area in your Verilog implementation.
3. Optimize your Verilog design for minimal silicon area usage.
4. Please implement the Verilog code with area efficiency as the primary goal.

I.3.3 Power Priority

For **low-power design**, the model is guided by one of the following prompts:

Prompt Templates with Power as the Design Priority

1. Please generate the Verilog code optimized for power consumption.
2. Focus on minimizing power consumption in your Verilog implementation.
3. Optimize your Verilog design for low power operation.
4. Please implement the Verilog code with power efficiency as the primary goal.

I.3.4 Delay and Power Priority

To address trade-offs between **delay and power**, the following prompts are utilized:

Prompt Templates with Delay and Power as the Design Priorities

1. Please generate the Verilog code optimized for both delay and power efficiency.
2. Focus on balancing delay and power consumption in your Verilog implementation.
3. Optimize your Verilog design for efficient power usage and high performance.

I.3.5 Delay and Area Priority

Similarly, for balancing **delay and area**, the following prompts are used:

Prompt Templates with Delay and Area as the Design Priority

1. Please generate the Verilog code optimized for both area efficiency and timing performance.
2. Focus on balancing area usage and speed in your Verilog implementation.
3. Optimize your Verilog design for efficient area usage and high performance.

I.4 Final Prompt Composition

The final prompt submitted to the model is constructed by systematically concatenating the aforementioned components. Let the primary components be defined as:

- P_{guide}: The *Code Guiding Prompt*.
- D_{desc}: The *Verilog Design Description* (i.e., the instruction).
- P_{opt}: The *Design Priority Prompt*, selected from one of the optimization pools.
- S_{sys}: The *Thinking System Prompt*.

First, the user-facing prompt, P_{user}, is constructed by concatenating the guiding prompt, the design description, and the optimization prompt. Let $\oplus$ denote the string concatenation operation.

$$P_{\text{user}} = P_{\text{guide}} \oplus D_{\text{desc}} \oplus P_{\text{opt}}$$

The final prompt P_{final} is formed by concatenating the system and user prompts with their role tags:

$$P_{\text{final}} = S_{\text{sys}} \oplus P_{\text{user}}$$

where $\oplus$ denotes ordered concatenation with the "system_prompt:" and "user_prompt:" role tags prepended. This structured composition provides the model with clear, role-separated instructions tailored to the target generation task.

Name	Original Benchmark ($\mathrm{ns}, \mu\mathrm{m}^2, \mathrm{W}$)	**ChipSeek** ($\mathrm{ns}, \mu\mathrm{m}^2, \mathrm{W}$)
asyn_fifo	**0.72/1397.032/7.67e-05**	N/A
LFSR	0.14/25.004/2.45e-06	0.14/25.004/2.45e-06
right_shifter	0.08/36.176/4.32e-06	0.08/36.176/4.32e-06
barrel_shifter	0.17/44.688/1.41e-05	**0.17/39.368/1.41e-05**
LIFObuffer	0.38/226.1/0.00027	**0.36/216.79/0.000137**
RAM	0.25/635.74/5.56e-05	**0.19/475.076/3.92e-05**
ROM	0.14/6.65/8.85e-07	0.14/6.65/8.85e-07
alu	1.92/1573.39/0.000751	**1.75/1286.908/0.000433**
pe	1.27/3651.382/0.000224	1.27/3651.382/0.000224
instr_reg	0.14/117.04/1.03e-05	0.14/117.04/1.03e-05
signal_generator	0.37/93.1/7.54e-06	**0.38/74.214/6.08e-06**
square_wave	0.41/100.282/8.39e-06	0.41/100.282/8.39e-06
calendar	0.44/164.92/1.44e-05	0.44/164.92/1.44e-05
parallel2serial	0.2/48.678/4.5e-06	**0.19/47.082/4.36e-06**
pulse_detect	0.18/17.556/1.52e-06	**0.17/16.226/1.47e-06**
serial2parallel	0.4/156.142/1.4e-05	**0.28/157.738/1.39e-05**
width_8to16	0.24/186.732/1.67e-05	**0.21/173.698/1.62e-05**
traffic_light	0.37/149.758/1.31e-05	**0.36/148.162/1.25e-05**
edge_detect	0.12/18.354/1.79e-06	**0.1/17.822/1.75e-06**
freq_divbyfrac	**0.2/48.678/4.67e-06**	N/A
freq_divbyeven	**0.25/40.166/3.63e-06**	N/A
freq_divbyodd	5.17/59.052/6.63e-06	**7.82/82.642/1.33e-6**
sequence_detector	0.15/36.442/3.35e-06	**0.19/25.27/2.27e-06**
ring_counter	0.1/46.816/4.7e-06	**0.1/40.964/4.01e-06**
JC_counter	0.1/340.48/3.54e-05	0.1/340.48/3.54e-05
counter_12	0.25/36.176/3.1e-06	0.25/36.176/3.1e-06
up_down_counter	0.7/217.854/1.74e-05	**0.67/188.86/1.61e-05**
adder_bcd	0.34/46.018/3.85e-05	0.34/46.018/3.84e-05
adder_pipe_64bit	0.75/2534.182/0.000235	**0.15/886.964/6.49e-5**
adder_32bit	0.76/472.15/0.000325	**1.13/191.786/0.00012**
adder_16bit	0.84/89.376/6.49e-05	**0.84/93.632/4.44e-05**
adder_8bit	0.35/51.072/3.14e-05	**0.35/46.816/2.22e-05**
fixed_point_adder	1.69/606.214/0.000565	**1.42/512.313/0.000482**
fixed_point_subtractor	1.09/477.736/0.000381	**0.93/466.8/0.000367**
multi_pipe_4bit	0.34/174.762/1.51e-05	**0.1/154.546/1.28e-05**
multi_pipe_8bit	**0.8/874.608/7.52e-05**	N/A
multi_16bit	2.03/933.394/7.36e-05	**1.97/935.522/7.36e-05**
multi_8bit	1.5/483.854/0.00085	**0.79/373.996/0.000545**
comparator_4bit	0.16/18.886/8.91e-06	**0.13/17.29/7.6e-06**
comparator_3bit	0.1/11.704/5.26e-06	**0.1/11.704/5.25e-06**
radix2_div	**0.59/414.162/3.39e-05**	N/A
div_16bit	5.18/760.228/0.027	**5.57/745.332/0.0234**
accu	**0.47/150.822/1.23e-05**	0.46/210.672/1.83e-05
sub_64bit	2.3/404.586/0.000268	**2.08/400.862/0.000271**

Table 11: Comparison of PPA metrics on RTLLM v2.0 between ChipSeek and benchmark. N/A means generated code are not functionally correct or can't pass the scripts of the EDA tools.

Golden Solution

```
module barrel_shifter (
    input  [7:0] in,
    input  [2:0] ctrl,
    output [7:0] out
);

wire [7:0] x;
wire [7:0] y;

// 4bit shift right
mux2X1  ins_17 (.in0(in[7]), .in1(1'b0), .sel(ctrl[2]), .out(x[7]));
mux2X1  ins_16 (.in0(in[6]), .in1(1'b0), .sel(ctrl[2]), .out(x[6]));
mux2X1  ins_15 (.in0(in[5]), .in1(1'b0), .sel(ctrl[2]), .out(x[5]));
mux2X1  ins_14 (.in0(in[4]), .in1(1'b0), .sel(ctrl[2]), .out(x[4]));
mux2X1  ins_13 (.in0(in[3]), .in1(in[7]), .sel(ctrl[2]), .out(x[3]));
mux2X1  ins_12 (.in0(in[2]), .in1(in[6]), .sel(ctrl[2]), .out(x[2]));
mux2X1  ins_11 (.in0(in[1]), .in1(in[5]), .sel(ctrl[2]), .out(x[1]));
mux2X1  ins_10 (.in0(in[0]), .in1(in[4]), .sel(ctrl[2]), .out(x[0]));

// 2 bit shift right
mux2X1  ins_27 (.in0(x[7]), .in1(1'b0), .sel(ctrl[1]), .out(y[7]));
mux2X1  ins_26 (.in0(x[6]), .in1(1'b0), .sel(ctrl[1]), .out(y[6]));
mux2X1  ins_25 (.in0(x[5]), .in1(x[7]), .sel(ctrl[1]), .out(y[5]));
mux2X1  ins_24 (.in0(x[4]), .in1(x[6]), .sel(ctrl[1]), .out(y[4]));
mux2X1  ins_23 (.in0(x[3]), .in1(x[5]), .sel(ctrl[1]), .out(y[3]));
mux2X1  ins_22 (.in0(x[2]), .in1(x[4]), .sel(ctrl[1]), .out(y[2]));
mux2X1  ins_21 (.in0(x[1]), .in1(x[3]), .sel(ctrl[1]), .out(y[1]));
mux2X1  ins_20 (.in0(x[0]), .in1(x[2]), .sel(ctrl[1]), .out(y[0]));

// 1 bit shift right
mux2X1  ins_07 (.in0(y[7]), .in1(1'b0), .sel(ctrl[0]), .out(out[7]));
mux2X1  ins_06 (.in0(y[6]), .in1(y[7]), .sel(ctrl[0]), .out(out[6]));
mux2X1  ins_05 (.in0(y[5]), .in1(y[6]), .sel(ctrl[0]), .out(out[5]));
mux2X1  ins_04 (.in0(y[4]), .in1(y[5]), .sel(ctrl[0]), .out(out[4]));
mux2X1  ins_03 (.in0(y[3]), .in1(y[4]), .sel(ctrl[0]), .out(out[3]));
mux2X1  ins_02 (.in0(y[2]), .in1(y[3]), .sel(ctrl[0]), .out(out[2]));
mux2X1  ins_01 (.in0(y[1]), .in1(y[2]), .sel(ctrl[0]), .out(out[1]));
mux2X1  ins_00 (.in0(y[0]), .in1(y[1]), .sel(ctrl[0]), .out(out[0]));

endmodule

module mux2X1 (
    input in0,
    input in1,
    input sel,
    output out
);

assign out = (sel) ? in1 : in0;

endmodule
```

Figure 11: Barrel Shifter Golden Solution

ChipSeek Solution

<think>
The implementation prioritizes hardware efficiency by using a logarithmic approach with three cascaded shift stages. Each stage conditionally applies a power-of-two shift based on the corresponding control bit while minimizing area through direct hardware translation without complex multiplexers. This approach eliminates area-intensive components like AND gates and optimizes for area usage.
</think>
<answer>

```
module barrel_shifter (
  input [7:0] in,
  input [2:0] ctrl,
  output [7:0] out
);

// Intermediate wires for shift stages
wire [7:0] shift_4, shift_2, shift_1;

// 4-bit shift stage (controlled by ctrl[2])
assign shift_4 = ctrl[2] ? (in >> 4) : in;

// 2-bit shift stage (controlled by ctrl[1])
assign shift_2 = ctrl[1] ? (shift_4 >> 2) : shift_4;

// 1-bit shift stage (controlled by ctrl[0])
assign shift_1 = ctrl[0] ? (shift_2 >> 1) : shift_2;

assign out = shift_1;

endmodule
```

</answer>

Figure 12: Barrel Shifter ChipSeek Solution