Enhancing LLM Performance on Hardware Design Generation Task via Reinforcement Learning

Yifang Zhao*, Weimin Fu†, Shijie Li*, Yi-Xiang Hu*, Xiaolong Guo†, Yier Jin*‡
*University of Science and Technology of China, {zhaoyifang, shijie_li, yixianghu}@mail.ustc.edu.cn, jinyier@ustc.edu.cn
†Kansas State University, {weiminf, guoxiaolong}@ksu.edu

***Abstract*—Integrated circuit design is a highly complex and time-consuming process. Leveraging large language models (LLMs) for automating hardware design generation is receiving increasing attention. A prominent challenge is that the inherent structure of the text is overlooked during the training process. Existing efforts focus on supervised fine-tuning LLMs to acquire specialized knowledge in hardware design, without considering the conflict between LLMs' linear data processing and the structural nature inherent in hardware design. In this work, we propose a novel LLM-based reinforcement learning (RL) framework that integrates Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs). Our approach enhances the accuracy of generated hardware code by capturing the syntactic and semantic structures of hardware designs. Experimental results show that the SFT-RL model integrated with Text, AST, and DFG achieves notable improvements: a 12.57% increase on VerilogEval-Human and a 5.49% increase on VerilogEval-Machine, outperforming GPT-4; a 14.29% improvement on RTLLM, approaching GPT-4.**

***Index Terms*—Hardware code generation, Reinforcement learning, Large language model.**

I. Introduction

The integrated circuit (IC) design process is a complex workflow, including IC specification, circuit design, physical design, physical verification, and final sign-off [1]. Advances in fabrication technology have dramatically increased the scale of digital chips, with transistor counts often exceeding tens of billions. Simultaneously, the market demand for accelerated product updates compels design engineers to accurately manage more modules, interfaces, and logic layers within compressed timeframes. Since hardware cannot be modified after fabrication, ensuring a correct design from the outset is critical.

With the widespread use of ChatGPT [2] around the world, researchers from various fields have turned their attention to the study of large language models (**LLMs**). In the hardware domain, researchers are exploring ways to utilize LLMs as automated tools, including hardware debugging [3]–[5], hardware design generation [6]–[12], assisting in hardware verification by assertions [13] and aiding in formal verification [14]. These studies leverage hardware-specific datasets and fine-tune general-purpose LLMs to lower the barrier to entry for hardware design and development.

However, existing models often produce low-quality code that is unsuitable for real-world applications. First, supervised fine-tuning (SFT) heavily depends on labeled data, often not rigorously screened, leading to biased outputs or hallucinated responses [15]. Second, hardware design is highly structured and requires the capture of complex interactions between different components. However, the traditional LLM training process primarily focuses on the semantics for token prediction, resulting in a linear understanding of the text, which hinders the accurate generation of hardware design.

To address these challenges, a promising approach for hardware code generation is to align LLM outputs with human experts through Reinforcement Learning with Human Feedback (RLHF) [16], a strategy proven to substantially improve LLM performance [2], [17]–[19]. While collecting high-quality data from experts is often impractical, prior research indicates that reward functions can effectively guide LLMs by selecting high-quality outputs without requiring extensive human evaluation [20]. Therefore, designing a robust reward model that incorporates structural consideration is critical for guiding LLMs to generate high-quality hardware code.

The AST (Abstract Syntax Tree) and DFG (Data Flow Graph) representations in hardware design are widely used for code analysis, optimization, and verification [21]–[23]. These studies demonstrate that graphical representations effectively capture the logic of hardware designs, thereby enabling comprehensive code analysis. Our work introduces a reward approach integrating two critical analytical perspectives: AST and DFG. The AST captures the hierarchical syntactic structure of hardware designs, representing relationships between components such as modules, control structures, and operations. The DFG models the flow of data within the design, reflecting operational dependencies and interactions between hardware components. Incorporating both AST and DFG as rewards in reinforcement learning (RL) enables the model to capture the functional and architectural characteristics of hardware designs, assess output quality, and progressively refine decision-making during content generation. This approach ensures that the LLM produces code that is not only syntactically accurate but also structurally coherent, effectively addressing the limitations of traditional token-based LLM training.

Our Contribution: This work proposes a novel approach incorporating ASTs generated by parsers and DFGs produced by dataflow analyzers into the LLM training process. This allows the model to maintain syntactic integrity (via ASTs) and semantic integrity (via DFGs), addressing the limitations of LLMs' linear data processing when applied to hardware design tasks. Additionally, we provide a more comprehensive reward mechanism during training by combining multiple scoring criteria—textual similarity, syntactic structure, and dataflow consistency. Experiments show that this method significantly improves the functional correctness and syntactic accuracy of the generated hardware code, enhancing the applicability of LLMs in hardware design.

II. Background and Related Work

A. LLM-based hardware design Generation

Prompt-based methods focus on designing prompts to guide general-purpose LLMs in generating hardware code, such as [11] and Chip-Chat [6] based on ChatGPT, ChipGPT [10] based on Claude [17], Autochip [12] based on ChatGPT, Claude, PaLM [24] and CodeLlama [25]. However, due to the lack of professional knowledge, general-purpose LLMs often misunderstand terminology, making it difficult to generate high-quality, standards-compliant designs, such as inserting syntax logic that conforms to software development into hardware designs. This prompt engineering method demands substantial domain-specific expertise post-processing to meet hardware design requirements.

‡Corresponding author.

To overcome these limitations, SFT is employed, utilizing datasets comprised of functional descriptions and corresponding code. VerilogEval [8] demonstrated that SFT can enhance the Verilog code generation capabilities of pre-trained models and provided two comprehensive evaluation benchmarks, VerilogEval-machine and VerilogEval-human. RTLCoder [7] introduced an automated training dataset generation flow that leveraged GPT-3.5 and a syntax checker, and performed SFT based on quality scoring, demonstrating superior performance over GPT-3.5. ChipNeMo [26] utilized a substantial amount of internal hardware design-related data from NVIDIA, outperforming the state-of-the-art GPT-4 in the cases of engineering assistant chatbots and EDA script generation. However, its closed-source nature restricts its broad application. Although these fine-tuning models show promise for general hardware generation tasks, they overlook the contradiction between LLM's linear data processing and hardware design structural characteristics, resulting in limited management of the structural complexity of RTL design.

B. Conflict Between Linear Data Processing and RTL Structure

Based on the 1D Transformer [27], LLMs process data linearly, token-by-token. The model processes an ordered sequence of input tokens one by one and sequentially generates each output token until a complete sequence is produced. It excels at sequential tasks such as coherent text generation, information summarizing, language translation, and other NLP tasks [28]. However, this linear method becomes less effective when applied to domains with highly structured or parallel data [29], such as hardware design or programming languages. Unlike natural language, many operations in hardware design are executed concurrently, with different modules interacting in complex, hierarchical ways. Traditional transformer models struggle to capture these characteristics effectively. RTL code demands a more structured approach, which the current Transformer architecture cannot handle.

C. Addressing Structural Design with Graph Level Representation

Hardware designs involve complex structures that require precise modeling of both control flow and data dependencies. Graph-level representation provides an effective means. For example, [23] applies ASTs and CFGs to convert synchronous RTL models into asynchronous designs, with ASTs capturing syntax and CFGs managing control flow and concurrency. RTL-FSMx [21] extracts finite state machines from RTL code using ASTs and CFGs, aiding in hardware security and design verification by providing a graph-level view of control flow. RTSEC [22] employs ASTs to integrate security features like watermarking and logic locking into RTL designs, ensuring these enhancements align with the hierarchical structure of the code.

D. Reinforcement Learning

Reinforcement learning (RL) originates from trial-and-error learning and optimal control. The formalization of RL through Markov Decision Processes (MDP) provides a robust framework for finding optimal strategies. Unlike traditional LLM training methods, which typically rely on labeled data and extract patterns from large datasets in a linear manner, RL enables learning through interaction with an environment. An agent learns from experience, receiving rewards or penalties based on the outcomes of its actions, thereby refining its decision-making over time [30].

In the hardware domain, RL has been applied to Logic Synthesis Optimization [31], Design Space Exploration [32], Task Scheduling in Hardware Accelerators [33], and Circuit Layout Design [34]. These applications demonstrate RL's ability to tackle complex optimization problems in IC design by exploring various strategies dynamically and improving performance metrics over time.

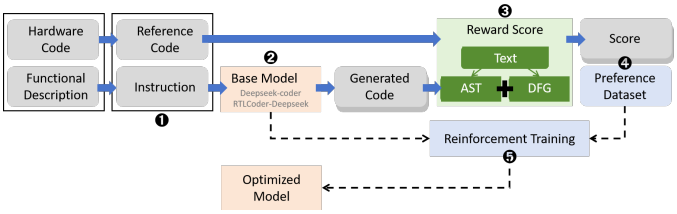

Fig. 1: Enhancing hardware code generation through reinforcement learning incorporated with AST and DFG.

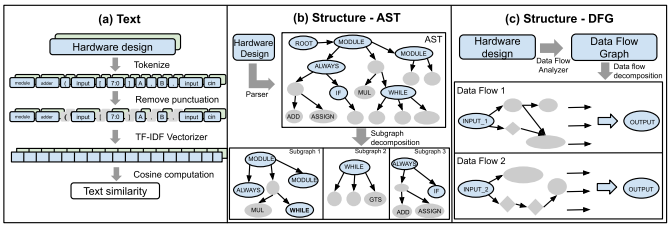

Fig. 2: Reward Score Design: Text, AST, DFG.

III. Methodology

To address the misalignment between the linear data processing of LLMs and the structural characteristics of RTL designs, we conduct reinforcement training across three dimensions: Text, AST, and DFG. These complementary metrics help to capture sequential and structural aspects of hardware design to guide the model's learning, aiming to enhance the accuracy and quality of generated hardware code. The overall framework is illustrated in Figure 1.

Our framework follows these key steps: ❶ **Dataset Construction:** We build a dataset of functional descriptions of hardware designs paired with corresponding codes. The functional descriptions act as *Instructions*, while the original code serves as a *Reference design* for evaluating generated outputs. ❷ **Code Generation:** Using the base model, we generate multiple versions of hardware code based on these descriptions. ❸ **Reward Score:** Each generated code is compared to the reference design from three dimensions—Text, AST, and DFG—to calculate similarity scores as the reward. ❹ **Preference Dataset Creation:** The reward scores rank the generated outputs, creating a preference dataset that identifies the best and worst responses. ❺ **Reinforcement Training:** Finally, we apply RL to fine-tune the model. RL guides the LLMs by providing feedback based on preference signals, enabling the model to learn to align its responses more closely with the desired outcomes, thereby effectively optimizing its performance based on cumulative rewards.

A. AST and DFG Generation

To capture the AST and DFG of hardware designs, we employ the RTL synthesis tool Yosys [35]. The process begins with preprocessing directives such as *include* instructions and macro definitions. The Lexer then conducts lexical analysis, decomposing the Verilog code into a series of tokens that contain keywords, identifiers, constants, and operators. The Parser converts the Verilog code into an AST, where each node embodies a construct within the code, such as an assignment statement, a conditional statement, or a module declaration. We can systematically analyze the design's logical structure, including conditional branches, loops, etc. This process ensures that the design's foundational syntactic information is preserved for further analysis and optimization.

Although Yosys cannot directly generate a DFG, we can use commands *proc* to expand control logic and *show* to export the design

structure. By using Python scripts to parse the exported data, we can identify data dependencies and operations needed to construct a DFG. This semantic view complements the syntactic information provided by the AST, offering insights into the functional behavior and execution dependencies of the hardware design.

B. Reward Function Design

Given that LLMs process data linearly, we first introduce a Text-based reward to investigate whether focusing solely on sequential patterns could enhance hardware code generation. Building upon this, we incorporate structural features such as AST and DFG to explore the impact of structure on the accuracy of generated hardware designs. Therefore, our reward is constructed from three components:

1) ***Text****:* We evaluate the text similarity between generated and reference codes, focusing on the similarity of word elements. By analyzing the frequency and distribution of these elements in the code (such as variable names, function names, and keywords), we reveal the sequential relationships within the code. We develop a text similarity algorithm employing tokenization, TF-IDF vectorization, and cosine similarity to yield quantitative scores, shown in Fig 2(a).

$$S_{Text} = \cos_{\text{similarity}}(\text{matrix}_{\text{ref}}, \text{matrix}_{\text{gen}}) \tag{1}$$

2) ***AST****:* The AST captures the hierarchical structure of the hardware code, with each node in the tree representing a syntactic element of the design, such as modules, control structures (e.g., IF, WHILE, ALWAYS), and operations (e.g., ADD, ASSIGN). To quantify structural similarity, we use the Graph Edit Distance (GED). Since GED computation is NP-hard and grows exponentially with the number of vertices, we implement a subgraph decomposition strategy to simplify complex ASTs. For instance, in Fig 2(b), the original AST is decomposed into three subgraphs, each focusing on the MODULE structure, WHILE loop, and ALWAYS block, respectively. We apply the AStar algorithm combined with local search (LSa) to compare the query graph (generated design ASTs) with the target graph (reference design ASTs), calculating the GED and checking if it is less than or equal to the specified threshold τ [36]. The final output is the number of $Match$ that meets the condition. The AST-similarity score is defined as:

$$S_{AST\ or\ DFG} = \frac{Match^i - Match_{min}}{Match_{max} - Match_{min}} \tag{2}$$

3) ***DFG****:* Each node in the DFG represents an operation, while edges illustrate the data flow between these operations. To simplify the analysis and improve computational efficiency, we decompose the complex DFG into smaller subgraphs, where each subgraph represents the data flow from a single input to its corresponding output, as shown in Fig 2(c). The DFG-similarity score is defined in Equation 2.

For each input instruction x, multiple distinct candidate outputs y are sampled from the reference policy π_{ref} of the original model. The generated code is scored by combining these factors comprehensively, with weights used to adjust the proportion of each component, $w_T + w_A + w_D = 1$:

$$S_{(x,y)} = w_T \cdot S_{Text} + w_A \cdot S_{AST} + w_D \cdot S_{DFG} \tag{3}$$

The superior output y_w (highest score) and the inferior output y_l (lowest score) are selected. The preference dataset is composed of such triplets: $D = \{(x^{(i)}, y_w^{(i)}, y_l^{(i)})\}_{i=1}^N$, where each candidate's reward is calculated as follows:

$$\hat{r}_\theta(x, y) = \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$$

where $\pi_\theta(y|x)$ denotes the probability of generating output y given input x under the current policy, and $\pi_{\text{ref}}(y|x)$ denotes the probability under the reference policy. This log-ratio formulation enables an assessment of the current policy's relative advantage in generating preferred outputs over the reference policy, with β serving as a scaling factor to control reward intensity.

C. Evaluation Method

The evaluation method comprises two key aspects: Syntax and Function. We use the pass@k [37] standard to assess Syntax and Function accuracy of hardware code generation. The model generates n (where $n > k$) code samples for each problem, and then k samples are randomly selected from these. If at least one of the k samples passes the unit test, the test is considered passed. Here, c represents the total number of samples that pass the unit tests.

$$\text{pass@}k := \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{4}$$

TABLE I: Reinforcement Learning Training Loss and Accuracy

Reward				RL-only		SFT-RL	
Scoring Mechanism	w_T	w_A	w_D	Base Model: Deepseek-coder-6.7b [38]		Base Model: RTLCoder [7], finetuned on DeepSeek	
				Loss	Accuracy	Loss	Accuracy
Text	1	0	0	0.4754	0.7875	0.5322	0.8125
AST	0	1	0	0.2304	0.8999	0.3458	0.8750
DFG	0	0	1	0.2628	0.9625	0.5376	0.8125
Text & AST	1/2	1/2	0	0.5226	0.8345	0.6392	0.7745
Text & DFG	1/2	0	1/2	0.5028	0.8500	0.6210	0.7185
Text & AST & DFG	1/3	1/3	1/3	0.5123	0.8250	0.6355	0.8245

IV. Experiments

In this section, we evaluate the impact of RL models integrated with structural factors on hardware code generation quality.

The typical LLM training process comprises three stages: Pre-training, Supervised Fine-tuning, and Reinforcement Learning, which progressively improve the model's ability to specialize and generate high-quality outputs for specific tasks. To assess the impact of RL effectively, we implement and compare two distinct training strategies: 1) **RL-only**, training based on Deepseek-coder-6.7b [38] to explore the potential of RL in optimizing non-proprietary hardware code generation models without additional supervision; 2) **SFT-RL**, training based on RTLCoder-Deepseek-v1.1 [7], which is a supervised fine-tuned version of Deepseek-coder-6.7b by utilizing a specialized dataset to enhance the accuracy of hardware code generation. We evaluate our RL models using the RTLLM [39] benchmark and VerilogEval [8] benchmark, focusing on both syntax and functionality. VerilogEval includes 156 problems from the instructional Verilog website HDLBits, covering a range from simple combinational circuits to complex FSMs, and is divided into two subsets: VerilogEval-human, with handcrafted problem descriptions, and VerilogEval-machine, with LLM-generated descriptions. RTLLM comprises 29 designs encompassing various complexities and scales, with each reference design written by human designers. The experiments are conducted on a workstation with four A100 40GB GPUs.

A. Dataset Generation

Compared to SFT (tens of thousands to millions) and pre-training (billions to trillions), RL usually only requires a few thousand examples. For our RL framework to enhance hardware code generation quality through AST and DFG integration, we randomly sampled 3k functional descriptions from the RTLCoder dataset [40] as Instructions, with the corresponding hardware code serving as the Reference

TABLE II: Evaluation results. **Original** refers to the DeepSeek-Coder-6.7b model. **RL-only** represents reinforcement learning applied directly to the original model with different reward settings. **SFT-only** refers to RTLCoder-Deepseek-v1.1, which applies supervised fine-tuning on the original model. **SFT-RL** indicates RL applied to the SFT-only model (RTLCoder) using different reward settings. Positive percentages (**blue**) indicate stronger accuracy, while negative percentages (**red**) reflect weaker accuracy.

Benchmark		VerilogEval												RTLLM-v1.1					
Category		Human						Machine						Syntax			Func		
Metric		$pass$@1	vs Original	vs SFT-only	$pass$@5	vs Original	vs SFT-only	$pass$@1	vs Original	vs SFT-only	$pass$@5	vs Original	vs SFT-only	$pass$@5	vs Original	vs SFT-only	$pass$@5	vs Original	vs SFT-only
Original		30.2	0.00%	-27.40%	42.2	0.00%	-15.77%	54.1	0.00%	-11.60%	63.8	0.00%	-16.60%	89.6	0.00%	-3.76%	34.5	0.00%	-28.57%
RL-only	Text	30.6	1.32%	-26.44%	42.8	1.42%	-14.57%	54.7	1.11%	-10.62%	62.2	-2.51%	-18.69%	93.1	3.91%	0.00%	44.8	29.86%	-7.25%
	AST	29.6	-1.99%	-28.85%	40.3	-4.50%	-19.56%	54.2	0.18%	-11.44%	62	-2.82%	-18.95%	100	11.61%	7.41%	41.4	20.00%	-14.29%
	DFG	31.3	3.64%	-24.76%	42.5	0.71%	-15.17%	58.6	8.32%	-4.25%	68.4	7.21%	-10.59%	100	11.61%	7.41%	34.5	0.00%	-28.57%
	Text&AST	30.4	0.66%	-26.92%	44.3	4.98%	-11.58%	53.8	-0.55%	-12.09%	63.1	-1.10%	-17.52%	100	11.61%	7.41%	44.8	29.86%	-7.25%
	Text&DFG	32.9	8.94%	-20.91%	45	6.64%	-10.18%	54.9	1.48%	-10.29%	64.4	0.94%	-15.82%	93.1	3.91%	0.00%	37.9	9.86%	-21.53%
	Text&AST&DFG	35.8	18.54%	-13.94%	46.2	9.48%	-7.78%	58.1	7.39%	-5.07%	65.7	2.98%	-14.12%	96.6	7.81%	3.76%	48.3	40.00%	0.00%
SFT-only		41.6	37.75%	0.00%	50.1	18.72%	0.00%	61.2	13.12%	0.00%	76.5	19.91%	0.00%	93.1	3.91%	0.00%	48.3	40.00%	0.00%
SFT-RL	Text	42.8	41.72%	2.88%	51.7	22.51%	3.19%	62.7	15.90%	2.45%	77.5	21.47%	1.31%	93.1	3.91%	0.00%	48.3	40.00%	0.00%
	AST	41.6	37.75%	0.00%	50.8	20.38%	1.40%	61.5	13.68%	0.49%	78.7	23.35%	2.88%	96.6	7.81%	3.76%	48.3	40.00%	0.00%
	DFG	42.2	39.74%	1.44%	54.9	30.09%	9.58%	62.3	15.16%	1.80%	79.6	24.76%	4.05%	96.6	7.81%	3.76%	48.3	40.00%	0.00%
	Text&AST	42.6	41.06%	2.40%	54.2	28.44%	8.18%	62.9	16.27%	2.78%	79.2	24.14%	3.53%	96.6	7.81%	3.76%	51.7	49.86%	7.04%
	Text&DFG	43.4	43.71%	4.33%	55.9	32.46%	11.58%	63.1	16.64%	3.10%	79.6	24.76%	4.05%	93.1	3.91%	0.00%	51.7	49.86%	7.04%
	Text&AST&DFG	45.2	49.67%	8.65%	56.4	33.65%	12.57%	63.8	17.93%	4.25%	80.7	26.49%	5.49%	93.1	3.91%	0.00%	55.2	60.00%	14.29%

designs. To construct datasets that reflect preference differences, we utilized two baseline models (DeepSeek and RTLCoder) to generate hardware designs ten times for each Instruction, resulting in a dataset of 30k generated designs. Each generated design is evaluated using Text, AST, and DFG scoring mechanisms. We then examined the contribution of each reinforcement strategy to model performance by designing different combinations of scoring mechanisms as rewards: only **Text**, only **AST**, only **DFG**; combining text similarity with structural and semantic assessments **Text & AST** and **Text & DFG**, applying all scoring mechanisms simultaneously **Text & AST & DFG**. These combinations serve as mutual ablations. We select two hardware designs for each scheme with the highest and lowest scores, collectively forming a preference dataset denoted as Instruction-Chosen-Rejected.

B. Model Training

We employ Direct Preference Optimization (DPO) [41] to train our hardware design generation models, directly optimizing the model's policy to align with RTL structures by leveraging preference data. For fine-tuning, we use Low-Rank Adaptation (LoRA) [42], which significantly reduces the number of trainable parameters, enabling efficient adaptation of large models without the high computational costs associated with full fine-tuning. To balance preference weighting and incorporate feedback effectively, we set $\beta = 0.1$ and train with a learning rate of 5×10^{-6}, using a cosine scheduler with a warmup phase of 0.1 steps. Table I presents the training loss and reward accuracy for various hardware design generation models with our RL framework.

TABLE III: The comparison of our reinforcement learning models with GPT4 and base models. The evaluation metric is $pass$@5.

Evaluated Model		VerilogEval		RTLLM-v1.1	
		Hum.(%)	Mach.(%)	Syn.(%)	Func(%)
GPT3.5		45.8	69.1	89.7	37.9
GPT4		55.8	70.6	**100**	65.0
DeepSeek-Coder		42.2	63.8	89.6	34.5
RL-only	Text & AST & DFG	46.2	65.7	96.6	48.3
RTLcoder-DeepSeek		50.1	76.5	93.1	48.3
SFT-RL	Text & AST & DFG	**56.4**	**80.7**	93.1	**55.2**

C. Evaluation

We evaluate the capabilities of the structure-based RL models in generating RTL code from natural language descriptions using two benchmarks: VerilogEval and RTLLM. Table II presents a performance comparison between RL-only and SFT-RL models against baseline models, shown in the columns labeled *vs Original* and *vs SFT-only*. Both the RL-only vs Original and the SFT-RL vs SFT-only comparisons demonstrate notable improvements. Specifically, the SFT-RL model integrated with Text&AST&DFG (referred to as **SFT-RL-TAD**) achieves optimal performance. Compared to the SFT-only model: On the VerilogEval-Human subset, SFT-RL-TAD improves $pass$@1 by 8.65% and $pass$@5 by 12.57%. On VerilogEval-Machine, it increases $pass$@1 by 4.25% and $pass$@5 by 5.49%. On RTLLM, the functional accuracy improves by 14.29%. Compared to other state-of-the-art models, as shown in Table III: In the RTLLM benchmark, both the RL-only and SFT-RL integrated with Text&AST&DFG enhance the performance of the baseline models, surpassing GPT-3.5. In VerilogEval, the SFT-RL-TAD model further improves the performance of the RTLCoder model, exceeding the results of GPT-4.

Ablation Study: To examine the impact of different reward criteria components, we evaluate multiple model variants: Text, AST, DFG, Text&AST, Text&DFG, Text&AST&DFG, as shown in Table II. Models trained with combined components (Text&AST, Text&DFG, Text&AST&DFG) outperform those relying on a single metric (Text, AST, or DFG). Incorporating structural-level features, which LLMs traditionally find challenging to capture, proves more effective than using the Text-based method alone.

V. Conclusion

This work demonstrates that integrating RL with structural representations, such as ASTs and DFGs, enhances the capability of LLMs in hardware design generation. By addressing the limitations of linear data processing in LLMs, our approach significantly improves both syntax correctness and functional accuracy. The SFT-RL model integrated with Text, AST, and DFG achieves notable improvements: a 12.57% increase on VerilogEval-Human and a 5.49% increase on VerilogEval-Machine, outperforming GPT-4; a 14.29% improvement on RTLLM, approaching GPT-4's performance. This breakthrough enables LLMs to overcome the challenges posed by the structural complexity of hardware descriptions, making them more practical for real-world hardware design tasks.

Acknowledgment

Portions of this work were supported by the National Science Foundation (2340949 and 2419880).

References

[1] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-Chip Verification: Methodology and Techniques*. Springer Publishing Company, Incorporated, 2013.

[2] OpenAI, "Gpt-4 Technical Report," ArXiv, Tech. Rep. abs/2303.08774, 2023.

[3] B. Ahmad, S. Thakur *et al.*, "On hardware security bug code fixes by prompting large language models," *IEEE Transactions on Information Forensics and Security*, vol. 19, p. 4043–4057, 2024.

[4] W. Fu, S. Li *et al.*, "A generalize hardware debugging approach for large language models semi-synthetic, datasets," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 72, no. 2, pp. 623–636, 2025.

[5] W. Fu, K. Yang *et al.*, "Llm4sechw: Leveraging domain-specific large language model for hardware debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2023, pp. 1–6.

[6] J. Blocklove, S. Garg *et al.*, "Chip-chat: Challenges and opportunities in conversational hardware design," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, Sep. 2023.

[7] S. Liu, W. Fang *et al.*, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," 2024.

[8] M. Liu, N. Pinckney *et al.*, "Verilogeval: Evaluating large language models for verilog code generation," 2023.

[9] Z. Pei, H.-L. Zhen *et al.*, "Betterv: Controlled verilog generation with discriminative guidance," 2024.

[10] K. Chang, Y. Wang *et al.*, "Chipgpt: How far are we from natural language hardware design," 2023.

[11] M. Nair, R. Sadhukhan, and D. Mukhopadhyay, "Generating secure hardware using chatgpt resistant to cwes," *IACR Cryptol. ePrint Arch.*, vol. 2023, p. 212, 2023.

[12] S. Thakur, J. Blocklove *et al.*, "Autochip: Automating hdl generation using llm feedback," 2024.

[13] R. Kande, H. Pearce *et al.*, "Llm-assisted generation of hardware assertions," 2023.

[14] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "Using llms to facilitate formal verification of rtl," 2023.

[15] J. Li, J. Chen *et al.*, "The dawn after the dark: An empirical study on factuality hallucination in large language models," 2024. [Online]. Available: https://arxiv.org/abs/2401.03205

[16] L. Ouyang, J. Wu *et al.*, "Training language models to follow instructions with human feedback," 2022. [Online]. Available: https://arxiv.org/abs/2203.02155

[17] Anthropic. (2023) Meet claude. [Online]. Available: https://www.anthropic.com/claude

[18] G. Gemini Team, "Gemini: A family of highly capable multimodal models," 2024.

[19] H. Touvron, T. Lavril *et al.*, "Llama: Open and efficient foundation language models," 2023.

[20] H. Dong, W. Xiong *et al.*, "Raft: Reward ranked finetuning for generative foundation model alignment," 2023. [Online]. Available: https://arxiv.org/abs/2304.06767

[21] R. Kibria, M. Sazadur Rahman *et al.*, "Rtl-fsmx: Fast and accurate finite state machine extraction at the rtl for security applications," in *2022 IEEE International Test Conference (ITC)*, 2022, pp. 165–174.

[22] O. Arias, Z. Liu *et al.*, "Rtsec: Automated rtl code augmentation for hardware security enhancement," in *2022 Design, Automation& Test in Europe Conference& Exhibition (DATE)*, 2022, pp. 596–599.

[23] S. Semba and H. Saito, "Rtl conversion method from pipelined synchronous rtl models into asynchronous ones," *IEEE Access*, vol. 10, pp. 28 949–28 964, 2022.

[24] R. Anil, A. M. Dai *et al.*, "Palm 2 technical report," 2023. [Online]. Available: https://arxiv.org/abs/2305.10403

[25] B. Rozière, J. Gehring *et al.*, "Code llama: Open foundation models for code," 2024.

[26] M. Liu, T.-D. Ene *et al.*, "Chipnemo: Domain-adapted llms for chip design," 2024.

[27] A. Vaswani, N. Shazeer *et al.*, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[28] X. Wang, H. Kim *et al.*, "Human-llm collaborative annotation through effective verification of llm labels," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3613904.3641960

[29] K. Liu, Z. Chen *et al.*, "Educating llms like human students: Structure-aware injection of domain knowledge," 2024. [Online]. Available: https://arxiv.org/abs/2407.16724

[30] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[31] A. Hosny, S. Hashemi *et al.*, "Drills: Deep reinforcement learning for logic synthesis," 2019. [Online]. Available: https://arxiv.org/abs/1911.04021

[32] N. Wu, Y. Xie, and C. Hao, "Ironman-pro: Multiobjective design space exploration in hls via reinforcement learning and graph neural network-based modeling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 3, pp. 900–913, 2023.

[33] A. R. Baranwal, S. Ullah *et al.*, "Relaccs: A multilevel approach to accelerator design for reinforcement learning on fpga-based systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 9, pp. 1754–1767, 2021.

[34] K. Settaluri, Z. Liu *et al.*, "Automated design of analog circuits using reinforcement learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 9, pp. 2794–2807, 2022.

[35] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.

[36] L. Chang, X. Feng *et al.*, "Speeding up ged verification for graph similarity search," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 793–804.

[37] M. Chen, J. Tworek *et al.*, "Evaluating large language models trained on code," 2021.

[38] D. Guo, Q. Zhu *et al.*, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024.

[39] Y. Lu, S. Liu *et al.*, "Rtllm: An open-source benchmark for design rtl generation with large language model," 2023.

[40] "Rtlcoder dataset." [Online]. Available: https://github.com/hkust-zhiyao/RTL-Coder

[41] R. Rafailov, A. Sharma *et al.*, "Direct preference optimization: Your language model is secretly a reward model," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann *et al.*, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 53 728–53 741.

[42] E. J. Hu, Y. Shen *et al.*, "LoRA: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2022.