535514 RL Team Project Proposal: Enhancing RTL Code Generation with Large Language Models via Reinforcement Learning from Human Feedback

Yang Hsueh Mu-Feng Chua Xing-Han Huang
Department of Computer Science
National Tsing Hua University
{kevin80934, arya59049, delic62773}@gapp.nthu.edu.tw

1 Project Overview

1.1 Profile

Track: 3. Application

Abstract and project goal: Recent advancements in large language models (LLMs) have led to significant progress in code generation. However, generating register transfer level (RTL) code from natural language descriptions remains a challenging task due to the strict syntactic and semantic requirements of hardware description languages.

This project investigates the use of reinforcement learning from human feedback (RLHF) to fine-tune LLMs for RTL code generation. We construct a preference dataset based on 27k examples from the training dataset of RTLCoder, where each example includes a problem description and RTL code generated by GPT-3.5. Positive samples are taken from these high-quality outputs, while negative samples are generated by the target LLM to be fine-tuned. Using this preference dataset, we conducted experiments with Direct Preference Optimization (DPO) to fine-tune the LLM for RTL code generation. While other RLHF algorithms were considered, we focused our evaluation on DPO in this study. Our goal is to enable LLMs to produce more accurate and executable RTL code from simple natural language descriptions, potentially accelerating the hardware design process.

TL;DR ("Too Long; Didn't Read"): Automatically generate RTL code from natural language descriptions by fine-tuning LLMs with RLHF, enabling faster design iterations and reducing manual coding effort.

1.2 Motivation

- **Why is the problem interesting?** If we can effectively reduce the gap between a designer's ideas and the resulting RTL code through automation, hardware development can be greatly improved. By automating the translation process, designers would be able to quickly iterate on their ideas and focus on innovation rather than coding details. Moreover, this automation can speed up the prototyping phase by enabling rapid testing and refinement of different design concepts. This framework not only saves time, but also allows designers to experiment with multiple approaches, ultimately leading to more efficient and tailored hardware solutions.
- **Critical Challenges.**
  - Challenge 1: Generating correct RTL code from free-form text is difficult because hardware design requires strict correctness. LLMs may produce syntactically valid but semantically incorrect code. Ensuring the generated code meets performance requirements, timing constraints, and resource limitations remains a core challenge.

  - Challenge 2: When the generated code does not work as expected, it can be hard to pinpoint the source of error. Understanding why an AI system made certain design choices and how to fix them is not straightforward. This problem becomes more difficult as hardware logic grows in complexity.
- **Justify why the problem remains open or unsolved.** Today's LLMs can already generate small Verilog modules from simple natural language descriptions. However, they still struggle with two key points: (1) guaranteeing full functional correctness, and (2) scaling to large real-world designs. Because of these gaps, automatic RTL generation from natural language is still an open research problem.
  - *BetterV*: Although BetterV [Pei et al., 2024] adds a discriminator that helps an open-source 7B model surpass GPT-4 on a benchmark, the method needs extra token-level feedback and brings noticeable compute cost. It also depends on a specially cleaned and augmented dataset that is hard to build for larger or more diverse circuits, so the approach does not yet solve the problem at industrial scale.
  - *RTLCoder*: RTLCoder [Liu et al., 2024] offers a fully open-source pipeline and passes GPT-3.5 on two public tests, but its training data are filtered only by syntax, not by real functional checks. The authors note that the model still trails GPT-4, and that broader data coverage plus stronger verification are needed. Fine-tuning for Verilog also hurts the model's skills in other languages, showing limited generalization.
- **State-of-the-art methods.** BetterV and RTLCoder are known to be recent competitive baselines and shall be considered state-of-the-art methods in RTL code generation.

2 Problem Formulation

2.1 RTL Code Generation Task

We frame the RTL code generation task as a sequence-to-sequence problem. Given a vocabulary $\mathcal{V}$ and a problem description as an input sequence x, the objective is to generate an output sequence representing the flattened solution program: $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_T)$, where each token $\hat{y}_t \in \mathcal{V}$. Typically, an LLM parameterized by θ generates the solution program autoregressively, sampling each token $\hat{y}_t$ from the conditional distribution $p_\theta(\cdot \mid \hat{y}_{1:t-1}, x)$. The correctness of generated programs is evaluated using unit test cases consisting of input-output pairs $\{(i_j, o_j)\}_{j=1}^{J}$. A generated program $\hat{y}$ is deemed correct if $\hat{y}(i_j) = o_j$ holds for all $j \in \{1, \ldots, J\}$.

2.2 RLHF Objective

Let the reward model $r_\phi(x, y)$ be learned from human preference data under the Bradley–Terry pairwise preference model, so that for any pair (y^+, y^-) on the same input x,

$$p_\phi\big(y^+ \succ y^- \mid x\big) \;=\; \frac{\exp\big(r_\phi(x, y^+)\big)}{\exp\big(r_\phi(x, y^+)\big) + \exp\big(r_\phi(x, y^-)\big)}.$$

An autoregressive policy π_θ induced by the LLM is defined as

$$\pi_\theta(y \mid x) = \prod_{t=1}^{T} p_\theta\big(y_t \mid y_{<t},\, x\big).$$

The RLHF objective is

$$J(\theta) = \; \mathbb{E}_{y \sim \pi_\theta(\cdot|x)}\big[r_\phi(x, y)\big] \;-\; \tau\, D_{\mathrm{KL}}\big(\pi_\theta(\cdot \mid x) \,\big\|\, \pi_{\mathrm{ref}}(\cdot \mid x)\big),$$

where π_{ref} is the pretrained reference policy and $\tau > 0$ controls the trade-off between reward maximization and policy divergence.

3 Empirical Evaluation

3.1 Performance Metrics

We will follow Liu et al. [2023] and evaluate the code generation performance using the passing rate metric *pass@k*:

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

where n is the generation budget for each query and c is the number of correct generated codes. This metric reflects the probability that at least one of k solutions can pass the unit test for each query.

3.2 Baseline Methods

We will compare with the following state-of-the-art methods in RTL code generation:

- BetterV [Pei et al., 2024]: BetterV fine-tunes some 7B LLMs on processed domain-specific datasets and incorporates another generative discriminators for guidance on particular design demands. It has the ability to generate syntactically and functionally correct Verilog code, which can outperform GPT-4.
- RTLCoder [Liu et al., 2024]: RTLCoder generates 27k synthesized instruction-code pairs as a training dataset and further uses them to fine-tunes some 7B LLMs. It also has the ability to generate high-quality RTL code that can outperform GPT-3.5 and is comparable to GPT-4.

3.3 Benchmark Tasks or Datasets

We will evaluate each method using VerilogEval [Liu et al., 2023] benchmark, which consists of 156 problems sourced from the Verilog instructional website HDLBits, each problem includes natural language description, module header and IO definition. The evaluation set consists of a diverse set of Verilog code generation tasks, ranging from basic combinational circuits to complex finite state machines.

4 Methodology

4.1 Base Model Selection

We first conducted baseline experiments on several open-source models, including CodeLlama-7B [Roziere et al., 2023], DeepSeek-Coder-6.7B [Guo et al., 2024], and CodeQwen1.5-7B [Bai et al., 2023], to evaluate their performance on RTL code generation. As shown in Table 1, DeepSeek-Coder achieves the best overall performance, and therefore, we selected it as our base model.

4.2 Dataset Construction

Our original goal was to fine-tune LLMs using reinforcement learning (RL) algorithms. However, applying RL requires a reliable benchmark to provide appropriate reward signals, and constructing such a benchmark is challenging. As a result, we shifted our focus to RLHF.

For RLHF, we constructed a preference dataset based on 27k examples from the training data of the RTLCoder [Liu et al., 2024]. Each example consists of a problem description paired with RTL code generated by GPT-3.5. Positive samples are taken from these high-quality GPT-3.5 outputs, while negative samples are generated by DeepSeek-Coder.

4.3 Finetuning with RLHF

We initially planned to use the constructed preference dataset to fine-tune DeepSeek-Coder with several common RLHF algorithms, including Proximal Policy Optimization (PPO) [Schulman et al., 2017], Direct Preference Optimization (DPO) [Rafailov et al., 2023], and Group Relative Policy Optimization (GRPO) [Shao et al., 2024]. As a first step, we trained a reward model. However, due to time constraints, we were only able to complete fine-tuning with DPO in this study, using LoRA [Hu et al., 2022] with a rank of 16 during the fine-tuning process.

Model	# Params	VerilogEval		
		pass@1	pass@5	pass@10
GPT-3.5	N/A	26.7	45.8	51.7
GPT-4	N/A	43.5	**55.8**	**58.9**
CodeLlama	7B	18.2	22.7	24.3
DeepSeek-Coder	6.7B	30.2	33.9	34.9
CodeQwen	7B	22.5	26.1	28.0
RTLCoder	6.7B	41.6	50.1	53.4
BetterV	7B	**46.1**	53.7	58.2
DeepSeek-Coder w/ DPO	6.7B	32.1	46.0	51.2

Table 1: Performance comparison on the VerilogEval benchmark benchmark.

5 Experimental Results

5.1 Evaluation of Baseline Methods

As baselines, we evaluated closed-source LLMs (GPT-3.5, GPT-4), open-source code LLMs (CodeLlama, DeepSeek-Coder, CodeQwen), and specialized RTL code models (RTLCoder, BetterV) on the VerilogEval benchmark. As shown in Table 1, GPT-4 consistently demonstrated the strongest performance on the RTL code generation task. Additionally, the $pass@1$ score on VerilogEval shows that current models have a lot of space for improvement in RTL code generation task.

To illustrate this, we selected an example from VerilogEval, "Prob130_circuit5", as a toy case study (see Figure 1). We sampled about 10 generations from RTLCoder on this problem and observed that its instruction-following ability is still lacking. It often failed to generate valid Verilog and instead expanded the task description. An example is shown in Figure 2. The result highlights that even current state-of-the-art models still have significant space for improvement in RTL code generation task.

Based on the result shown in Table 1, DeepSeek-Coder seems to have a better understanding of RTL code compared to other open-source code LLMs (CodeLlama, CodeQwen). This suggests that it is a great candidate for applying RLHF to further improve RTL code generation.

5.2 Evaluation of Our Method

As shown in Table 1, our method (the DPO-fine-tuned DeepSeek-Coder) improves the $pass@1$ performance of DeepSeek-Coder by about 2%, and achieves significant improvements on $pass@5$ and $pass@10$. However, the performance remains lower than that of the current state-of-the-art models, RTLCoder and BetterV, both fine-tuned with supervised learning. This may be due to the lack of hyperparameter tuning during fine-tuning, or because RLHF-based fine-tuning may currently be less effective than supervised fine-tuning for this task.

6 Conclusion

Our original goal was to fine-tune LLMs for RTL code generation using RL algorithms. However, applying RL requires a reliable benchmark to provide appropriate reward signals, and constructing such a benchmark is challenging. As a result, we shifted our focus to RLHF. Based on the training data of the RTLCoder model, we constructed the first preference dataset for RTL code generation, and fine-tuned DeepSeek-Coder with DPO. The fine-tuned model achieved improvements over the base model, but its performance remained lower than that of the current state-of-the-art models. For future work, we aim to further explore RL-based fine-tuning approaches for RTL code generation, tune the hyperparameters of DPO to enhance model performance, and conduct a comprehensive

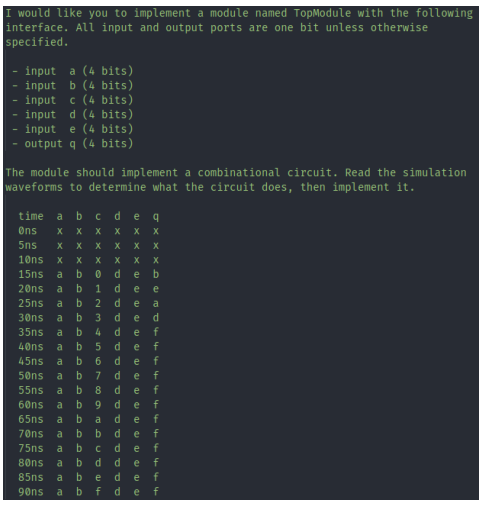

Figure 1: The problem description of "Prob130_circuit5" from VerilogEval

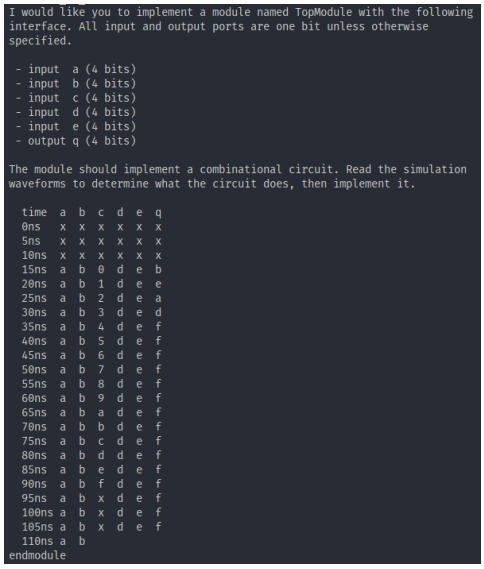

Figure 2: The generated result by RTLCoder with respect to "Prob130_circuit5"

comparison of RLHF methods such as PPO and GRPO, to better understand their impact on RTL code generation performance.

7 Contributions of Each Team Member

- Yang Hsueh: 1/3 (implement our methods, i.e., construct training dataset, train the reward model and fine-tune LLMs with DPO)
- Mu-Feng Chua: 1/3 (implement baseline methods)
- Xing-Han Huang: 1/3 (writing reports, paper survey and reading, poster presentation)

References

Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. Betterv: Controlled verilog generation with discriminative guidance. *arXiv preprint arXiv:2402.03375*, 2024.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8. IEEE, 2023.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741, 2023.

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

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.