Improving LLM-based Verilog Code Generation with Data Augmentation and RL

Kyungjun Min*, Seonghyeon Park*, Hyeonwoo Park, Jinoh Cho, and Seokhyeong Kang
Department of Electrical Engineering, Pohang University of Science and Technology
Pohang, Republic of Korea
{kj.min, seonghyeon98, jimmy0709, jinoh.cho, shkang}@postech.ac.kr

Abstract—**Large language models (LLMs) have recently attracted significant attention for their potential in Verilog code generation. However, existing LLM-based methods face several challenges, including data scarcity and the high computational cost of generating prompts for fine-tuning. Motivated by these challenges, we explore methods to augment training datasets, develop more efficient and effective prompts for fine-tuning, and implement training methods incorporating electronic design automation (EDA) tools. Our proposed framework for fine-tuning LLMs for Verilog code generation includes (1) abstract syntax tree (AST)-based data augmentation, (2) output-relevant code masking, a prompt generation method based on the logical structure of Verilog code, and (3) reinforcement learning with tool feedback (RLTF), a fine-tuning method using EDA tool results. Experimental studies confirm that our framework significantly improves syntax and functional correctness, outperforming commercial and non-commercial models on open-source benchmarks.**

I. Introduction

Large language models (LLMs) offer significant advantages in text generation tasks due to their ability to understand and generate human-like text [2]. Specifically, this capability extends to electronic design automation (EDA) workflows, such as converting natural language descriptions into design flow scripts and digital systems verification [23]. Mainly, leveraging LLMs to automate Verilog [21] code generation represents a groundbreaking advancement in design methodologies. Using LLMs, developers can reduce the manual effort required for coding, thereby accelerating the design process and minimizing human error [23]. Furthermore, this approach can democratize the design process by making it accessible to a broader range of developers with varying levels of expertise.

Despite the promising capabilities of LLMs, existing studies on LLM-based Verilog generation face several limitations. In particular, relying on commercial LLMs such as GPT [26] may raise data privacy concerns and limit in-depth research [18]. Therefore, some studies [1], [4], [11], [12], [17] attempt to fine-tune local LLMs; however, Verilog codes are often patented or copyrighted, making it challenging to obtain diverse training datasets [9]. Furthermore, most studies primarily focus on fine-tuning models with labeled datasets composed of prompts and corresponding completions[1]. To obtain labeled datasets, some studies [11], [12] generate prompts that describe Verilog codes using LLMs, which is time-consuming and resource-intensive. Other studies [1], [17] have introduced methods masking parts of Verilog code and using it as a prompt. Nevertheless, no studies have focused on generating prompts by masking Verilog code to align with its logical structure.

In this study, we address all these challenges, and our primary contributions are outlined as follows:

- **AST-based Data Augmentation.** We propose an effective *AST-based Verilog code augmentation* technique. By representing Verilog code into a hierarchical structure with an abstract syntax tree (AST), we can traverse nodes and generate synthetic code by deleting, inserting, and modifying them. Our experiments demonstrate that this data augmentation technique enhances training performance. Additionally, we make this dataset publicly available on our GitHub repository [31]. (Section III-A and IV-B)
- **Output-relevant Code Masking.** We explore a novel concept for prompt generation, called *output-relevant code masking*. By traversing the AST, we construct a directed graph that captures the logical structure of Verilog code. We then identify and mask the portions of code that influence the output. Our experiments show that this code masking technique enhances the model's performance in generating Verilog code. (Section III-B and IV-C)
- **RL with Tool Feedback.** We introduce *reinforcemet learning with tool feedback* (RLTF) method that optimizes the model by incorporating EDA tools. By evaluating the generated codes with tool feedback and using these evaluations in training, our approach guides the model to produce syntactically and functionally correct codes without labeled datasets. To validate this approach, we conduct experiments that demonstrate its effectiveness in enhancing model performance. (Section III-D and IV-D)
- **Experimental Confirmations.** We validate our framework using open source benchmarks [12], [14]. Compared to the baseline model [6], our methods show significant improvements in both syntax and functionality correctness, measured by $pass@k$ metric [3] (e.g., $k = 1$), with increases of (20.1%, 17.6%) in the RTLLM benchmarks [14] and (16.1%, 8.4%) in the VerilogEval benchmarks [12], for syntax and functionality correctness, respectively. (Section IV-D2)

The rest of the paper is organized as follows. Section II reviews the preliminaries of our study; Section III describes our overall framework; Section IV presents the experimental setup and discusses the experiment results; we conclude the paper in Section V.

*These authors equally contributed to this work. This research was supported by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Science and ICT (MSIT) (No. 2019R1A5A1027055), HRD Program for Industrial Innovation (No. RS-2024-00401466), and LX Semicon.

[1]In this task, a prompt refers to the input text given to LLMs, and completion is Verilog code.

II. Preliminaries

A. Abstract Syntax Tree

An AST represents the logical structure of program code. Each node corresponds to a construct (e.g., variables, expressions, control flow statements), and edges depict relationships between these constructs. By removing nonessential details such as parentheses or punctuation, the AST highlights the code's hierarchical structure. For example, the expression $a + (b * c)$ has a root node representing the $+$ operator, with two child nodes: one for the variable a and another subtree for the $*$ operator. The $*$ operator node then has two children, b and c. This arrangement captures the underlying operations and relationships without extra syntactic details. ASTs are widely used by compilers and interpreters for tasks like code generation, optimization, and static analysis [22]. In Verilog, tools such as Pyverilog [20] can generate ASTs from source code, enabling developers to interact with the code structure more directly and effectively.

B. RL with Human Feedback

Reinforcement learning with human feedback (RLHF) integrates human judgment into reinforcement learning (RL) to improve decision-making in complex tasks [16]. In the context of LLM fine-tuning, RLHF increases the model's adaptability to complex tasks by aligning rewards with human expectations. RLHF consists of three main steps: (1) supervised fine-tuning (SFT), (2) reward modeling, and (3) RL algorithm. Initially, LLMs are fine-tuned on a supervised dataset to generate initial outputs that closely match human preferences. Next, a reward model is trained to assess the quality of these outputs generated by LLMs. Finally, LLMs are refined using RL algorithms. During this step, the model generates outputs, evaluates them using scores from the reward model, and adjusts its parameters to maximize expected rewards. However, RLHF struggles with limitations such as dataset generation through human labeling and subjective biases influenced by human perspectives. By replacing human feedback with EDA tools, we can replace the reward model in a cost-effective and objective way.

C. Proximal Policy Optimization

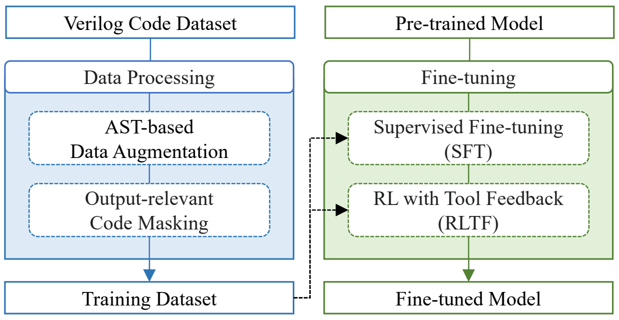
Proximal policy optimization (PPO) is an RL algorithm designed to enable stable learning by limiting policy changes [19]. PPO algorithm updates agent via the clipped surrogate objective function $L^{\text{CLIP}}(\phi)$ at step t as shown in Equation 1.

$$L^{\text{CLIP}}(\phi) = \mathbb{E}_t\left[\min\left(r_t(\phi)\hat{A}_t, \text{clip}(r_t(\phi), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \quad (1)$$

In the equation, the importance sampling ratio $r_t(\phi)$ compares action probabilities under the current and old policies. $r_t(\phi)$ is defined as $\frac{\pi_\phi(a_t|s_t)}{\pi_{\phi_{\text{old}}}(a_t|s_t)}$, where $\pi_\phi(a_t|s_t)$ is the probability of action a_t given state s_t under the current policy ϕ, and $\pi_{\phi_{\text{old}}}(a_t|s_t)$ is the probability of action a_t given state s_t under the old policy ϕ_{old}. This ratio indicates how much the new policy diverges from the old policy for a given action and state. The advantage function $\hat{A}_t$ is given by $Q(s_t, a_t) - V(s_t)$, where $Q(s_t, a_t)$ is the action-value function representing the expected return of taking action a_t in state s_t, and $V(s_t)$ is the value function representing the expected return of being in state s_t. The advantage function helps determine how much better or worse an action is compared to the average action taken from the same state. The clipping range, controlled by ϵ, sets the allowable range for policy updates to prevent large deviations. In this study, we utilize the PPO algorithm to update the model in RLTF.

III. Methodology

Our framework, illustrated in Figure 1, consists of four components: (1) AST-based data augmentation, (2) output-relevant code masking, (3) SFT, and (4) RLTF. The following sections detail each component.



Verilog Code Dataset
Data Processing
AST-based Data Augmentation
Output-relevant Code Masking
Training Dataset
Pre-trained Model
Fine-tuning
Supervised Fine-tuning (SFT)
RL with Tool Feedback (RLTF)
Fine-tuned Model

Fig. 1: Overall framework.

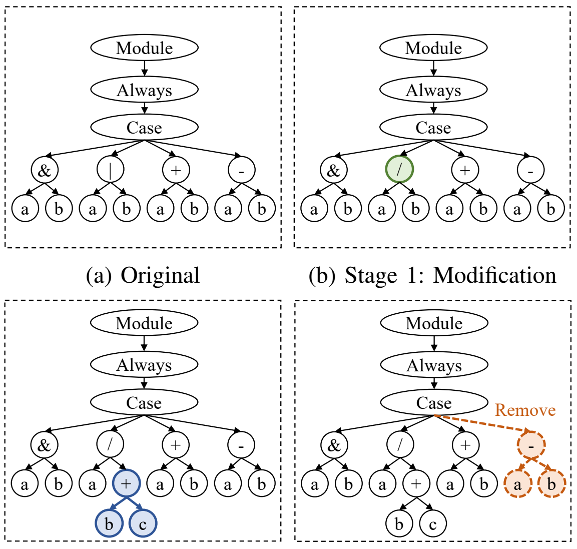
TABLE I: AST Node Classification

Node Type	Elements
Binary Operator	times (∗), divide (/), mod (^), plus (+), minus (−)
Shift Operator	sll (<<), srl (>>), sla (<<\`), sra (>>\`)
Equality	lessThan (<), greaterThan (>), lessEq (<=), greaterEq (>=), Eq (==), notEq (!=), Eql (===), notEql (!==)
Gate Operator	and (&), xor (^), knor (~^), or (|), land (&&), lor (||)
Unary Operator	uplus (+), uminus (−), ulnot (!), unot (~), uand (&), unand (~&), uor (|), unor (~|), uxor (^), uxnor (~^)
Substitution	blockingSubstitution (=), nonblockingSubstitution (<=)
Case	caseStatement, casexStatement

A. AST-based Data Augmentation

Compared to other programming languages, the scarcity of Verilog code data poses a limitation in training LLMs, as the volume of data is crucial for their performance [8]. To address this, we propose a novel AST-based Verilog code augmentation technique. First, we convert Verilog code into an AST using Pyverilog [20] and classify the AST nodes based on their functionality. Table I shows the types of nodes where we apply augmentation and the corresponding elements that belong to each type. We generate augmented AST by applying modification, insertion, or deletion for nodes and then convert these augmented AST back into Verilog code to produce synthetic Verilog code data. Examples of each augmentation method are shown in Figure 2, and detailed explanations of each method are described below:

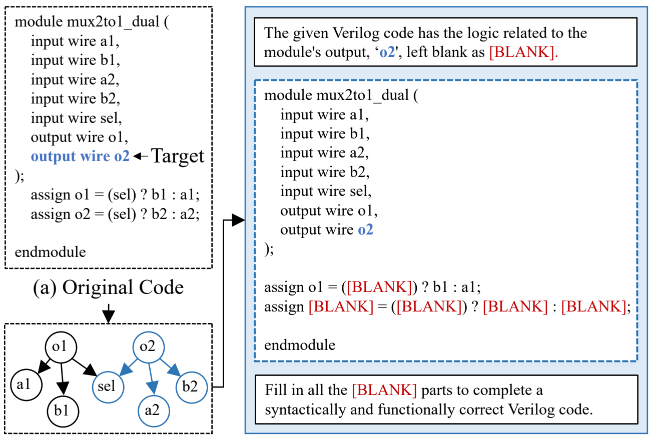
- Modification: The goal of modification is to replace nodes while preserving the structure of the AST. During this process, we replace the target node with another node of the same type. If the selected node is an operator included in



(a) Original (b) Stage 1: Modification

(c) Stage 2: Insertion (d) Stage 3: Deletion

Fig. 2: Stages of AST-based data augmentation.



```
module mux2to1_dual (
    input wire a1,
    input wire b1,
    input wire a2,
    input wire b2,
    input wire sel,
    output wire o1,
    output wire o2
);
    assign o1 = (sel) ? b1 : a1;
    assign o2 = (sel) ? b2 : a2;

endmodule
```

The given Verilog code has the logic related to the module's output, 'o2', left blank as [BLANK].

```
module mux2to1_dual (
    input wire a1,
    input wire b1,
    output wire a2,
    input wire b2,
    input wire sel,
    output wire o1,
    output wire o2
);

assign o1 = ([BLANK]) ? b1 : a1;
assign [BLANK] = ([BLANK]) ? [BLANK] : [BLANK];

endmodule
```

Fill in all the [BLANK] parts to complete a syntactically and functionally correct Verilog code.

(a) Original Code

(b) Logical Relationships

(c) Generated Prompt

Fig. 3: Example of output-relevant code masking. When the target output is o2, Verilog code (a) is converted to a directed graph (b) by identifying the nodes affecting o2, then related nodes are masked to generate the prompt as (c).

the condition of a `for` or `if` statement (e.g., initialization, termination, update), no modifications will be made.

- Insertion: The goal of insertion is to increase the depth of the AST by adding nodes beneath the leaf nodes. After we select the target node, we replace the right child node of the target node with a random operator and then add the children node beneath the operator node. The children nodes are chosen from operator nodes, wires, registers, inputs, or outputs.
- Deletion: The goal of deletion is to remove the target node and its children from the AST. When we delete the target node and its children, any unused wires, registers, etc., generated due to the deletion are also removed. If deletion removes all input or output signals of a module, deletion is not performed.

After converting the augmented AST back into Verilog code, we perform two additional post-processing steps, including error correction and anonymization, both conducted using GPT-4o [26]. First, to prevent unintentional errors caused by structural changes in the AST, we use GPT-4o to identify and correct any syntax errors. Additionally, to ensure the original and augmented code are treated as distinct data, we rename the variable names for modules, inputs, outputs, wires, and registers. We then perform a syntax check on the generated dataset to confirm the validity of the synthetic dataset.

B. *Output-relevant Code Masking*

There are two primary methods for generating prompts for fine-tuning in Verilog code generation. The first method involves using LLMs to generate descriptions of Verilog code functionality, which are then used as prompts. However, this approach requires a large number of queries to resource-intensive LLMs. More importantly, we cannot guarantee that the generated descriptions are well-aligned with the corresponding Verilog code. The second method is code masking, where certain parts of the Verilog code are masked or removed and then used as prompts. Since LLMs need to infer the masked sections based on the surrounding code, the masking process can influence the model's training performance. However, previous approaches have mostly employed random masking techniques.

In this subsection, we introduce output-relevant code masking to create prompts by analyzing the logical structure of the Verilog code and then masking sections that are related to the output signals. First, to analyze the logical structure of Verilog code, we convert the Verilog code into AST. We then traverse from the node corresponding to the output of Verilog code to the root nodes using a breadth-first search (BFS) algorithm to identify the logical relationships between nodes. These logical relationships between nodes are stored in a directed graph structure. After traversing all the nodes in the AST, we select the node corresponding to the output of the Verilog code from the directed graph. We recursively identify the sink nodes of the selected output node and its subsequent sink nodes, designating each identified node to be masked. Once the AST is converted back to Verilog, the code portions that correspond to the masked nodes are replaced with `[BLANK]`.

Figure 3 illustrates the process. First, we convert the Verilog code (Figure 3(a)) into an AST and construct a directed graph based on logical relationships (Figure 3(b)). We then select `o2` as the target node and designate `o2` and its sink nodes, `sel`, `a2`, and `b2`, as nodes to be masked. Finally, we convert the AST back into Verilog code and replace the positions corresponding to `o2`, `sel`, `a2`, and `b2` with `[BLANK]`(Figure 3(c)).

C. *Supervised Fine-Tuning*

A model pre-trained on a large-scale text dataset can handle general tasks but tends to exhibit sub-optimal performance on specific tasks. SFT addresses this by fine-tuning the model on domain-specific data. Specifically, to conduct SFT, a labeled dataset $D_{SFT} = (x_i, y_i)_{i=1}^{N}$ is required, where N is the size of the dataset, x_i (prompt) is the input, and y_i (completion) is the expected output[2]. Both x_i and y_i are sequences of tokens, represented as $x_i = (x_i^0, x_i^1, ..., x_i^{T_{i,x}})$ and $y_i = (y_i^0, y_i^1, ..., y_i^{T_{i,y}})$,

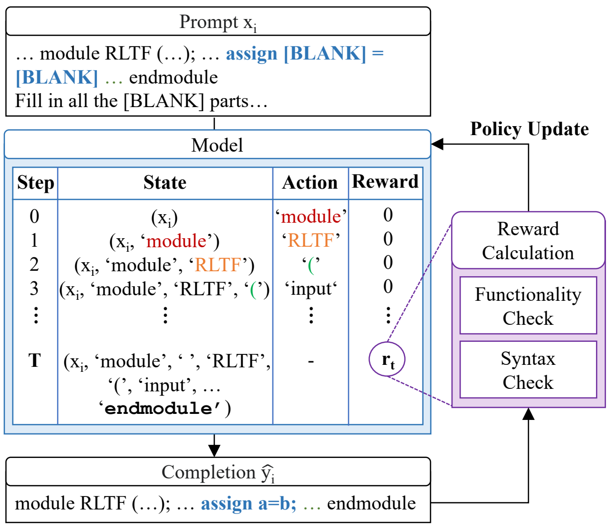
[2]We generate prompts through output-relevant code masking, with the corresponding completions being the original code.

where $T_{i,x}$ and $T_{i,y}$ represent the number of tokens in x_i and y_i, respectively.

At each training step t, for given contexts $(x_i, y_i^0, ..., y_i^{t-1})$, model generates a probability distribution of token candidates $\hat{y}_i^t$. To align the model-generated tokens with our ground-truth tokens, we define cross-entropy loss function $\mathcal{L}(\phi)$ as follows:

$$\mathcal{L}(\phi) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_{i,y}} \log P(\hat{y}_i^t = y_i^t \mid (x_i, y_i^0, ..., y_i^{t-1}), \phi), \tag{2}$$

where $P(\hat{y}_i^t = y_i^t \mid (x_i, y_i^0, ..., y_i^{t-1}), \phi)$ represents the probability that the predicted token $\hat{y}_i^t$ that matches with ground truth token y_i^t for the given context with model ϕ. We maximize this conditional probability through gradient descent.



Prompt x_i
… module RLTF (…); … assign [BLANK] = [BLANK] … endmodule
Fill in all the [BLANK] parts…
Model
Policy Update
Step
State
Action
Reward
0 (x_i) 'module' 0
1 (x_i, 'module') 'RLTF' 0
2 (x_i, 'module', 'RLTF') '(' 0
3 (x_i, 'module', 'RLTF', '(') 'input' 0
T (x_i, 'module', ' ', 'RLTF', '(', 'input', … 'endmodule') - r_t
Reward Calculation
Functionality Check
Syntax Check
Completion ŷ_i
module RLTF (…); … assign a=b; … endmodule

Fig. 4: Example of RLTF flow. Reward is calculated once, after the completion is finished.

D. RL with Tool Feedback

RLTF further optimizes the model fine-tuned through SFT by incorporating RL techniques. While SFT learns from labeled data, RLTF generates multiple code variations based on a given prompt. The quality of the generated code is evaluated using EDA tools, and rewards are assigned based on the results. Through continuous exploration, the model discovers better solutions, progressively improving its performance with each iteration. During RLTF, we utilize prompts generated through output-relevant code masking, as in SFT, but without providing completions. RLTF flow can be represented as a Markov decision process (MDP), as illustrated in Figure 4:

- **State** (s_t): The *state* is defined as the sequence of tokens generated so far. Formally, a state s_t at step t is represented by the tuple $(x_i, \hat{y}_i^0, \hat{y}_i^1 ... \hat{y}_i^{t-1})$. Here, x_i denotes the input prompt, and $\hat{y}_i^t$ corresponds to the token generated at step t.
- **Action** (a_t): Given the current state s_t, the *action* a_t involves selecting the next token $\hat{y}_i^t$ to generate. The action space comprises all possible tokens the model can select.
- **Reward** (r_t): The *reward* r_t evaluates the outcome of action a_t, as formulated in Equation 3. Until step T, which marks the end of the generation, the reward remains 0. At step T, once the generation is done, the reward is calculated based on the syntax and functionality of the generated code. The reward is categorized into the following cases: (1) if the completed code has syntax errors ($fail_{syn}$), the reward is 0, (2) if the code passes the syntax check but fails the functionality check[3] ($fail_{func}$), the reward is 0.5, and (3) if the code passes the functionality check, the reward is 1.

$$r_t = \begin{cases} 0 & \text{if } t \neq T \\ 0 & \text{else if } fail_{syn} \\ 0.5 & \text{else if } fail_{func} \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

To verify functionality correctness, we use formal equivalence checking, mathematical modeling to ensure logical equivalence between Verilog codes [7]. Unlike simulation testing that relies on predefined test vectors [5], formal equivalence checking does not require test vectors and can verify all corner cases. This method allows for automated and reliable functionality checks, enhancing accuracy and ensuring comprehensive coverage without the need for test benches.

IV. Experimental Results

A. Experimental Setup

1) Implementation: Our framework is written in *Python* using the *transformer reinforcement learning library* [30] and the baseline model is *deepseek-coder-6.7b-instruct* [6]. All datasets and scripts are publicly available on GitHub repository [31]. We run all experiments on a server with dual AMD EPYC 7513 32-Core Processors, 945GB RAM, and eight NVIDIA A6000 GPUs. For syntax and functionality checks, we use *Icarus Verilog v11.0* [27] and *Synopsys Formality v21.06* [24], respectively. Using *OpenROAD-flow-script* [29], we execute logic synthesis with *Nangate 45nm* process design kit [28].

2) Hyperparameters:

- **SFT:** We set *learning rate*, *batch size*, *epochs* to 5.0e-05, eight, and three, respectively. We use *AdamW* [13] for the optimizer, with a linear learning rate scheduler and *gradient clipping* set to 1.0 to prevent exploding gradients.
- **RLTF:** We set *learning rate*, *batch size*, *epochs* to 1.41e-05, 16, and four, respectively. The KL reward coefficient η is set to 0.2 to balance updates between the reward and policy models. Additionally, the clipping range ϵ is set to 0.2.
- **Code Generation:** We set maximum of 4096 tokens per completion, *temperature* of 1.0 for prediction randomness, *top_k* value of 50 to limit the sampling pool, and *top_p* of 1.0 to include tokens with a cumulative probability of 100%.

3) Dataset: We initially collect 854,950 Verilog code samples from GitHub repositories [25] and verify their syntax and synthesizability. Using the MinHash algorithm with a Jaccard similarity threshold of 0.9 [10], we deduplicate the dataset and remove any overlap with the test set, resulting in 10,513 unique samples. We then apply AST-based data augmentation, generating 9,880 additional Verilog code samples. For SFT, we use both the original and augmented datasets, while for RLTF, we select 1,051 codes from the original dataset.[4] To ensure

[3]The functionality check verifies whether the generated code operates functionally identical to the original code.

[4]Because the RLTF stage involves comparing the functionality of the generated code with the original code, we choose not to use artificially augmented code during RLTF.

TABLE II: Evaluation of Data Augmentation and Code Masking

Experiment	Model	RTLLM Benchmark [14]						VerilogEval Benchmark [12]					
		Syntax			Functionality			Syntax			Functionality		
		k=1	k=5	k=10	k=1	k=5	k=10	k=1	k=5	k=10	k=1	k=5	k=10
Data (Section 4.B)	D_{real}	**79.8**	**99.1**	**100**	52.8	75.0	80.6	87.2	98.9	99.5	40.4	55.4	60.8
	D_{aug}	75.9	94.6	96.3	49.7	70.4	76.3	89.6	98.8	99.5	**43.0**	55.7	60.6
	D_{both}	79.7	97.8	99.6	**55.9**	**77.5**	**82.6**	**90.8**	**98.9**	**99.8**	**43.0**	**57.0**	**62.3**
Masking-based (Section 4.C.1)	P_{mod}	71.6	93.7	98.0	49.8	73.4	78.4	84.8	98.8	**99.6**	39.2	53.8	57.7
	P_{sen}	77.8	96.2	98.2	52.2	74.4	78.2	**87.5**	98.2	98.9	40.0	54.0	57.3
	P_{tok}	72.4	94.1	97.4	49.3	71.5	75.2	86.8	98.6	**99.6**	39.4	53.1	57.7
	P_{ours}	**79.8**	**99.1**	**100**	**52.8**	**75.0**	**80.6**	87.2	**98.9**	99.5	**40.4**	**55.4**	**60.8**
LLM-based (Section 4.C.2)	P_{high}	70.0	96.8	99.7	45.0	69.8	74.4	85.5	98.0	98.9	39.3	**55.8**	**61.4**
	P_{low}	73.8	98.4	**100**	46.7	73.5	79.6	**88.8**	**99.5**	**99.9**	38.0	54.5	60.3
	P_{ours}	**79.8**	**99.1**	**100**	**52.8**	**75.0**	**80.6**	87.2	98.9	99.5	**40.4**	55.4	60.8

The best result in each column is highlighted in **bold** for each experiment.

the selected codes represent the dataset's characteristics, we convert the dataset into AST format, analyze the max depth distribution, and select 10% of the codes accordingly. To avoid data reuse between SFT and RLTF, we rename the variable names for modules, inputs, outputs, wires, and registers.

We evaluate the model on a total of 185 designs, including 29 from the RTLLM dataset [14] and 156 from the VerilogEval-human dataset (referred to as the VerilogEval) dataset [12], both of which provide human-written prompts paired with Verilog code. For a fair comparison, we use the provided prompts without any modifications.

4) Evaluation Metric: We evaluate the syntax and functionality correctness of the generated code using the $pass@k$ metric [3], which measures the probability of producing a correct solution within k attempts, as defined in Equation 4:

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

Here, n is the total number of generated solutions per problem, and c is the number of correct solutions for the problem. In our experiments, we set $n = 20$ and $k = 1, 5, 10$.

B. AST-based Data Augmentation Evaluation

In this subsection, we conduct ablation studies to show the effectiveness of AST-based data augmentation. To examine the importance of the dataset, we prepare three combinations of training datasets: (1) real Verilog only (D_{real}), (2) augmented Verilog only (D_{aug}), and (3) both real and augmented Verilog (D_{both}). We generate prompts with output-relevant code masking and fine-tune models using those datasets through SFT. The experimental results, presented in Table II, show that using both real and augmented Verilog data (D_{both}) leads to the highest functionality correctness. In the RTLLM benchmark, the model achieves functionality correctness rates of 55.9%, 77.5%, and 82.6% for $k = 1, 5, 10$, respectively, and 43.0%, 57.0%, and 62.3% for $k = 1, 5, 10$ in the VerilogEval benchmark. These findings suggest that AST-based data augmentation improves training performance. This improvement is due to increased coverage from node-level augmentation, which generates a wider variety of structural and functional variations in Verilog code, enabling the model to learn from more diverse cases.

C. Output-relevant Code Masking Evaluation

1) Comparison with Masking-based Prompt Generation: To validate our output-relevant code masking technique, we compare our approach with other code masking approaches: module-level (P_{mod}), sentence-level (P_{sen}), and token-level masking (P_{tok}). In the module-level approach, we mask the entire section between the module declaration and the endmodule keyword, while for the sentence-level approach, we split the sentence based on semicolons (`;`) and mask a randomly selected sentence. Token-level masking takes a more detailed approach by splitting the code into individual tokens and randomly masking one of them. For the experiment, we use the real Verilog dataset and generate prompts corresponding to each code sample, applying the module-level, sentence-level, token-level, and our output-relevant masking (P_{ours}) approaches. Then, we fine-tune the models through SFT.

In Table II, the experimental results show that the P_{ours} consistently outperforms P_{mod}, P_{sen}, and P_{tok} in most cases. For syntax correctness, in the RTLLM benchmark, the P_{ours} achieves 79.8%, 99.1%, and 100% for $k = 1, 5, 10$, respectively and 87.2%, 98.9%, and 99.5% for $k = 1, 5, 10$, respectively in the VerilogEval benchmark. For functionality correctness, in the RTLLM benchmark, the model records 52.8%, 75.0%, and 80.6% for $k = 1, 5, 10$, respectively, and 40.4%, 55.4%, and 60.8% for $k = 1, 5, 10$, respectively in the VerilogEval benchmark. The differences in model performance across various code masking techniques demonstrate that the choice of masking method significantly affects training outcomes. This highlights that masking critical parts of the code enhances the model's contextual understanding, leading to improvements in both syntax and functionality.

2) Comparison with LLM-based Prompt Generation: We further evaluate the effectiveness of our output-relevant code masking by comparing it to alternative prompt generation methods, LLM-based prompts. LLM-based prompts are created by having the LLM read the Verilog code and generate a description of its functionality. For this experiment, we generate two types of prompts using GPT-3.5 [26]: (1) high-level prompt (P_{high}), which describes the functionality of the code in natural language, and (2) low-level prompt (P_{low}), which focuses on the implementation, explaining the code on a line-by-line basis. We generate prompts for real Verilog codes and fine-tune two different models through SFT.

TABLE III: Comparison with other LLMs

Type	Models	Number of Parameters	RTLLM Benchmark [14]						VerilogEval Benchmark [12]					
			Syntax			Functionality			Syntax			Functionality		
			k=1	k=5	k=10	k=1	k=5	k=10	k=1	k=5	k=10	k=1	k=5	k=10
Pre-trained	Baseline [6]	6.7B	60.9	88.4	96.2	39.3	63.0	70.6	74.2	95.0	97.5	38.0	52.0	55.6
Commercial	GPT-3.5-turbo [26]	N/A	64.8	84.7	90.9	50.8	65.0	71.0	68.6	88.4	92.4	39.1	53.4	57.9
	GPT-4 [26]	N/A	61.6	88.2	96.9	48.6	70.4	80.7	76.0	94.6	96.9	50.5	**64.0**	**67.2**
Fine-tuned	VerilogEval [12]	16B	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	28.8	45.9	52.3
	RTLCoder-DeepSeek [11]	6.7B	60.2	74.7	77.8	46.8	62.2	66.7	73.0	81.3	83.2	43.3	47.5	48.5
	BetterV-DeepSeek [17]	6.7B	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	45.9	53.3	57.6
	OriGen [4]	7B	N/A	N/A	N/A	N/A	65.5	N/A	N/A	N/A	N/A	**54.4**	60.1	64.2
	Ours$_{\text{SFT}}$	6.7B	79.7	97.8	99.6	55.9	77.5	82.6	**90.8**	**98.9**	**99.8**	43.0	57.0	62.3
	Ours$_{\text{RLTF}}$	6.7B	**81.0**	**99.2**	**100**	**56.9**	**80.5**	**85.8**	90.3	**98.9**	99.4	46.4	60.0	65.0

The best result in each column is highlighted in **bold** and we use the GPT-4 snapshot from June 13, 2024.

The experimental results, presented in Table II, demonstrate that generating prompts using output-relevant code masking leads to performance that is either comparable to or significantly better than generating prompts using LLMs. Specifically, in the RTLLM benchmark, P$_{\text{ours}}$ outperforms the other two methods in terms of both syntax and functionality. The reason for these results is likely due to the possibility of misalignment between the generated descriptions and the code. Descriptions generated by LLMs tend to follow general natural language patterns, which often fail to capture or accurately represent critical technical details in the code. This misalignment may have hindered the expected performance improvement.

D. RLTF Evaluation

1) Comparison with the SFT Model: To evaluate the fine-tuning flow composed of SFT and RLTF, we conduct ablation studies. Table III details the performance of the model fine-tuned with both SFT and RLTF (Ours$_{\text{RLTF}}$) as compared to the model fine-tuned with only SFT (Ours$_{\text{SFT}}$). In terms of syntax correctness, both Ours$_{\text{SFT}}$ and Ours$_{\text{RLTF}}$ show similar performance across the two benchmarks. However, when it comes to functional correctness, Ours$_{\text{RLTF}}$ outperforms Ours$_{\text{SFT}}$. In terms of fucntionality correctness, Ours$_{\text{RLTF}}$ surpasses Ours$_{\text{SFT}}$ by 1.0%, 3.0%, and 3.2% for $k = 1, 5, 10$, respectively in the RTLLM benchmark and 3.4%, 3.0%, and 2.7% for $k = 1, 5, 10$, respectively in the VerilogEval benchmark.

These results suggest that fine-tuning with RLTF improves functionality while maintaining a high success rate in syntax. We believe that RL enables the model to iteratively learn how to generate functionally appropriate code for blank sections based on the surrounding context, which contributes to the improvement in overall functionality.

2) Comparison with other LLMs: We compare the performance of Ours$_{\text{RLTF}}$ with seven other models in Table III. These models include two commercial models (GPT-3.5-turbo and GPT-4 [26]), four fine-tuning studies (VerilogEval [12], RTLCoder [11], OriGen [4], and BetterV [17]), and our baseline model [6]. The results for VerilogEval, BetterV, and OriGen are sourced from their respective publications, and we exclude Chang et al. [1] from the comparison because they did not cover all designs in RTLLM benchmark, making it difficult to conduct a fair comparison. Except for VerilogEval, which uses *CodeGen* [15], other fine-tuned models use *DeepSeek* as the baseline. We emphasize that RTLCoder, BetterV, and ours are based on a 6.7B model, while OriGen is based on a 7B model.

First, compared to the baseline, Ours$_{\text{RLTF}}$ demonstrates significant improvements in both syntax and functionality correctness. On average, it achieves 18.1%, 7.4%, and 2.8% higher syntax correctness for $k = 1, 5, 10$, respectively, and 13.0%, 12.8%, and 12.3% higher functionality correctness for $k = 1, 5, 10$ across both the RTLLM and VerilogEval benchmarks. When compared to other fine-tuned models, Ours$_{\text{RLTF}}$ consistently outperforms RTLCoder and BetterV, both of which use the same baseline model. Although the baseline model of OriGen is different from ours, Ours$_{\text{RLTF}}$ still shows improvements in functionality, particularly in the RTLLM benchmark for $k = 5$ and the VerilogEval benchmark for $k = 10$. In comparison to commercial models, Ours$_{\text{RLTF}}$ consistently surpasses GPT-3.5-turbo in all cases. Although on the VerilogEval benchmark, GPT-4 shows 4.1%, 4.0%, and 2.0% better functionality correctness for $k = 1, 5, 10$, respectively, on the RTLLM benchmark, Ours$_{\text{RLTF}}$ achieves 8.3%, 10.1%, and 5.1% better functionality correctness for $k = 1, 5, 10$, respectively. In summary, we demonstrate that our fine-tuning framework optimizes LLMs for Verilog code generation with superior performance compared to commercial models and other studies.

V. Conclusion

We have introduced fine-tuning LLMs for the Verilog generation framework. Our framework consists of four components: AST-based data augmentation, representing Verilog code in a hierarchical structure and generates synthetic code by editing the tree structure; Output-relevant code masking, generating prompt by analyzing the logical structure of Verilog code; SFT, training the model with prompt-completion pairs; and RLTF, where the model-generated code is evaluated using EDA tools, and feedback is provided to further improve the model.

Our experimental results demonstrate the following: (1) our AST-based data augmentation significantly improves model training performance compared to models trained with non-augmented data; (2) the model trained with the prompt generated by our output-relevant code masking method outperforms models trained with other existing prompt generation methods; and (3) our RLTF method further enhances the functionality correctness of the code generated by the model through iterative tool feedback. Consequently, we show that our fine-tuning framework optimizes LLMs for Verilog code generation, achieving superior performance compared to commercial models and other studies.

References

[1] K. Chang, K. Wang, N. Yang, *et al.*, "Data is all you need: Finetuning LLMs for chip design via an automated design-data augmentation framework", *Proc. DAC*, 2024, pp. 1-6.

[2] Y. Chang, X. Wang, J. Wang, *et al.*, "A survey on evaluation of large language models", *ACM Trans. Intelligent Syst. Tech.*, vol 15, no 3, 2024, pp. 1-45.

[3] M. Chen, J. Tworek, H. Jun *et al.*, "Evaluating Large Language Models Trained on Code", *arXiv*, 2021, pp. 1-35.

[4] F. Cui, C. Yin, K. Zhou *et al.*, "OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection", *arXiv*, 2024, pp. 1-9.

[5] S. Fine, A. Ziv, "Coverage Directed Test Generation for Functional Verification using Bayesian Networks", *Proc. DAC*, 2003, pp. 286-291.

[6] D. Guo, Q. Zhu, D. Yang *et al.*, "DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence", *arXiv*, 2024, pp. 1-23.

[7] S.-Y. Huang, K.-T. T. Cheng, *Formal equivalence checking and design debugging*, Springer Science & Business Media, 2012.

[8] J. Kaplan, S. McCandlish, T. Henighan, *et al.*, "Scaling Laws for Neural Language Models", *arXiv*, 2024, pp. 1-19.

[9] D. Kim, S. Lee, K. Min, *et al.*, "Construction of realistic place-and-route benchmarks for machine learning applications", *IEEE Transactions on CAD*, vol 42, no 6, 2022, pp. 2030-2042.

[10] K. Lee, D. Ippolito, A. Nystrom *et al.*, "Deduplicating Training Data Makes Language Models Better", *Proc. ACL*, 2022, pp. 8424-8445.

[11] S. Liu, W. Fang, Y. Lu *et al.*, "RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution", *arXiv*, 2024, pp. 1-10.

[12] M. Liu, N. Pinckney, B. Khailany *et al.*, "Invited Paper: VerilogEval: Evaluating Large Language Models for Verilog Code Generation", *Proc. ICCAD*, 2023, pp. 1-8.

[13] I. Loshchilov, F. Hutter, "Decoupled weight decay regularization", *arXiv*, 2017, pp. 1-19.

[14] Y. Lu, S. Liu, Q. Zhang *et al.*, "RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model", *Proc. ASPDAC*, 2024, pp. 722-727.

[15] E. Nijkamp, B. Pang, H. Hayashi, *et al.*, "Codegen: An open large language model for code with multi-turn program synthesis", *Proc. ICLR*, 2023, pp. 1-25.

[16] L. Ouyang, J. Wu, X. Jiang *et al.*, "Training language models to follow instructions with human feedback", *arXiv*, 2022, pp. 1-68.

[17] Z. Pei, H. Zhen, M. Yuan, *et al.*, "BetterV: Controlled Verilog Generation with Discriminative Guidance", *Proc. ICML*, 2024, pp. 40145-40153.

[18] P. P. Ray, "ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope", *Internet of Things and Cyber-Physical Systems*, vol 3, 2023, pp. 121-154.

[19] J. Schulman, F. Wolski, P. Dhariwal *et al.*, "Proximal policy optimization algorithms", *arXiv*, 2017, pp. 1-12.

[20] S. Takamaeda-Yamazaki, "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL", *Proc. ARC*, 2015, pp. 451–460.

[21] D. Thomas, P. Moorby, *The Verilog® hardware description language*, Springer Science & Business Media, 2008.

[22] P. Yin, G. Neubig, "TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation", *arXiv*, 2018, pp. 1-6.

[23] R. Zhong, X. Du, S. Kai *et al.*, "LLM4EDA: Emerging Progress in Large Language Models for Electronic Design Automation", *arXiv*, 2023, pp. 1-15.

[24] Synopsys Formality User Guide, http://www.synopsys.com

[25] Github, https://github.com

[26] OpenAI, "GPT-4 Technical Report", *arXiv*, 2024, pp. 1-100.

[27] Iverilog, https://github.com/steveicarus/iverilog

[28] Nangate45 PDK, https://eda.ncsu.edu/freepdk/freepdk45/

[29] OpenROAD-Flow-Scripts, https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts

[30] Transformer Reinforcement Learning, https://github.com/huggingface/trl

[31] 97kjmin/VeriLogos Github Repository, https://github.com/97kjmin/VeriLogos