VeriCoder: Enhancing LLM-Based RTL Code Generation through Functional Correctness Validation

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recent advances in Large Language Models (LLMs) have sparked growing interest in applying them to Electronic Design Automation (EDA) tasks, particularly Register Transfer Level (RTL) code generation. While several RTL datasets have been introduced, most focus on syntactic validity rather than functional validation with tests, leading to training examples that compile but may not implement the intended behavior. We present VERICODER, a model for RTL code generation fine-tuned on a dataset validated for functional correctness. This fine-tuning dataset is constructed using a novel methodology that combines unit test generation with feedback-directed refinement. Given a natural language specification and an initial RTL design, we prompt a teacher model (GPT-4o-mini) to generate unit tests and iteratively revise the RTL design based on its simulation results using the generated tests. If necessary, the teacher model also updates the tests to ensure they comply with the natural language specification. As a result of this process, every example in our dataset is functionally validated, consisting of a natural language description, an RTL implementation, and passing tests. Fine-tuned on this dataset of over 125,000 examples, VERICODER achieves state-of-the-art metrics in functional correctness on VerilogEval and RTLLM, with relative gains of up to 71.7% and 27.4%, respectively. An ablation study further shows that models trained on our functionally validated dataset outperform those trained on functionally non-validated datasets, underscoring the importance of high-quality datasets in RTL code generation.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable performance across natural language processing tasks, spurring growing interest in applying their capabilities to a broad range of Electronic Design Automation (EDA) problems [1–4]. Recent efforts explore LLMs for code generation [5–12], architecture design [13–15], verification [16, 17], tool assistance [18, 19], and debugging [1, 20]. In this work, we focus on generating Register Transfer Level (RTL) code from natural language specifications. Automating RTL code generation has the potential to significantly boost hardware design productivity and reduce the manual effort involved in complex design tasks, making it a timely and impactful area of research.

Developing open-source, lightweight models for RTL code generation is essential for advancing both research and deployment. Proprietary models such as GPT-4o and Claude 3.7 restrict customization and lack transparency, making them unsuitable for in-depth analysis and academic exploration. They also raise privacy and security concerns, especially when handling RTL designs that may contain sensitive intellectual property. In contrast, lightweight models that can run locally offer a secure, privacy-preserving alternative—enabling hardware engineers to integrate AI directly into their design workflows. However, existing open-source models still underperform on RTL tasks, largely due to

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

Prior Work	Strategy	Description	Syntax Checker	Unit Tests
RTLCoder [7]	Keyword-based Generation, Mutation	Prompt LLM with keywords and existing code, followed by iterative mutation to get instruction-code pairs.	✓	✗
OriGen [8]	Code-to-Code, Syntax Error Correction	Applies LLM-driven code-to-code pipeline on existing RTL code and filters them by compiler error feedback.	✓	✗
BetterV [24]	Web Scraping & Cleaning, Alignment with C	Large-scale web-collected Verilog, cleaned and filtered to enforce coding standards; aligns C with Verilog.	✓	✗
VeriGen [26]	Manually Collect Textbook and Open-Source Code	Mines real-world RTL from GitHub and textbooks, manually cleans and organizes them into a structured dataset.	✓	✗
ChipGPT [27]	AST-based Synthesis	Converts Verilog ASTs into natural-language prompts and injects semantic error variants via EDA-tool feedback.	✓	✗
VeriCoder (Our Work)	Feedback-Directed Refinement, Simulation, Unit Test Generation	Iteratively generate unit tests with a teacher LLM, check implementations via compiler and simulator, and refining designs and tests until each design passes.	✓	✓

Table 1: Comparison of Verilog fine-tuning dataset construction approaches.

the absence of high-quality, functionally validated RTL datasets in their training corpora [21, 22]. While training algorithms are readily available, progress is bottlenecked by the lack of open datasets with functional correctness validation.

A key challenge in building such datasets lies in constructing large-scale, high-quality training data that pairs natural language specifications with RTL implementations. Despite efforts to mine RTL code from open-source repositories [23–26], much of the collected data lacks validation and may not align with its intended functionality. To address this, recent work has turned to LLMs-either prompting them to synthesize RTL designs from keyword-based specifications [6, 7] or leveraging them to rewrite existing RTL code and generate matching specifications [8, 24, 26]. In both cases, syntax checkers are often employed to filter uncompilable code or provide feedback for iterative refinement, but these techniques still fall short of validating functional correctness.

As far as we know, all these prior work [6–8, 24, 26] have focused solely on ensuring *syntactic correctness*, overlooking *functional correctness*. As a result, many dataset examples compile successfully but may not implement the behavior described in their natural language specifications. The distinction between syntactic correctness and functional correctness has important implications for model evaluation and real-world deployment. While functionally correct code inherently satisfies syntax constraints, syntactic correctness alone does not guarantee correct functionality. This gap is evident in the results reported by the RTLLM benchmark [10], where GPT-4o attains a high syntax accuracy of 100.0%, yet achieve only 69.0% in terms of functional correctness. Ultimately, in real-world settings, it is functional correctness rather than syntactic validity that truly matters.

In this work, we introduce VeriCoder, a model for RTL code generation fine-tuned on a high-quality dataset consisting of 125,777 examples that has been validated for functional correctness. To construct this dataset, we develop a novel pipeline that combines unit test generation with feedback-directed refinement guided by a teacher LLM (GPT-4o-mini). Given a natural language specification and an

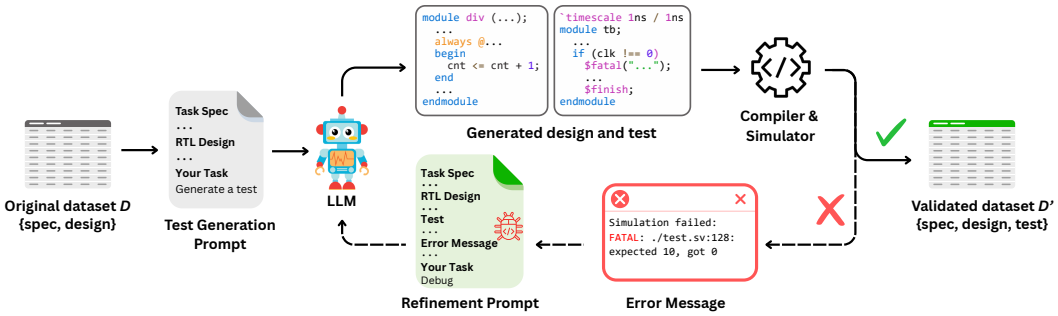

Figure 1: LLM-guided dataset augmentation overview.

initial RTL implementation, the teacher model first generates a unit test. If the RTL code fails the simulation, the model iteratively revises the design based on the observed error messages. When needed, the unit test is also updated to better reflect the intended functionality described by the specification. This process continues until the design passes simulation or a retry limit is reached. The resulting fine-tuning dataset consists of over 125k validated triples: a natural language specification, a correct RTL design, and a self-checking unit test.

We fine-tune VeriCoder from Qwen2.5-14B-Instruct using our curated dataset and evaluate it on two established RTL code generation benchmarks: VerilogEval [9] and RTLLM [10]. VeriCoder achieves new state-of-the-art performance, achieving up to 71.7% and 27.4% relative gains in the pass@k metric over the previous best fine-tuned model OriGen [8].

We conduct an ablation study demonstrating that models trained on our functionally validated dataset outperform those trained on non-validated data, under the same base model and training setup. These results highlight the importance of high-quality, functionally validated datasets for RTL code generation.

Our contributions are as follows:

- We introduce VeriCoder, an RTL code generation model fine-tuned on a dataset validated for functional correctness. On the VerilogEval and RTLLM benchmarks, VeriCoder achieves state-of-the-art performance among open-source fine-tuned models, yielding relative pass@k gains of up to 71.7% and 27.4% over the prior best.
- We develop a dataset augmentation pipeline that combines unit test generation with feedback-directed refinement guided by a teacher LLM. This yields, to the best of our knowledge, the largest fine-tuning dataset to date with functional validation, consisting of over 125k validated triples of natural language specifications, RTL designs, and passing tests.
- We conduct an ablation study showing that functional validation during dataset construction improves model performance, underscoring the importance of using high-quality functionally validated datasets for RTL code generation.

2 Background and Related Work

Progress on open-source RTL code generation is limited by the absence of large-scale, high-quality datasets. To mitigate this, recent efforts have focused on automated data mining and augmentation techniques to enrich existing corpora of RTL examples. Table 1 presents the comparison of different strategies for constructing fine-tuning datasets.

Mining open-source RTL designs is a common strategy for dataset construction. VeriGen [26] compiles Verilog modules from GitHub and textbooks into a structured corpus using automated syntax checks. BetterV [24] collects Verilog modules from the internet and then filters designs based on coding style and syntactic validity. Other works [8, 28, 29] adopt similar methodologies for sourcing and preprocessing RTL code.

Another line of work leverages a commercial LLM for synthetic data generation. RTLCoder [6] prompts GPT-3.5 with domain keywords to generate both task descriptions and corresponding RTL, discarding any outputs that fail to compile. OriGen [8] further employs Claude 3.5 in a two-stage code-to-code pipeline: first turning mined RTL code into natural language specifications, then

Natural Language Specification

Create a Verilog module and3 with three one-bit inputs a, b, c, and a one-bit output y. The output should perform a bitwise AND across all three inputs using a procedural block. The truth table:

a	b	c	y
0	x	x	0
x	0	x	0
x	x	0	0
1	1	1	1

(a) Natural language specification taken from the Origen [8] dataset.

```
module and3(
    input  wire a,
    input  wire b,
    input  wire c,
    output reg  y
);

always @* begin
    y <= a;
    y <= y & c;
    y <= y & b;
end

endmodule
```

(b) Buggy design taken from the Origen [8] dataset. It times out on the generated test shown in Figure 3.

```
module and3(
    input  wire a,
    input  wire b,
    input  wire c,
    output reg  y
);

always @* begin
    y = a & b & c;
end

endmodule
```

(c) Correct design fixed by the teacher model that passes the generated test in Figure 3.

Figure 2: Natural language specification (left) and the corresponding buggy and corrected Verilog designs (middle and right). The specification and buggy design are from the original dataset [8], which lacks tests, while the test (Figure 3) and corrected design are generated by a teacher model (GPT-4o-mini) and included in our validated dataset.

regenerating code from these specifications under compiler guidance, combining the strengths of real-world examples and synthetic generation. ChipGPT [27] transforms Verilog ASTs into natural language specifications.

While most of the existing work listed in Table 1 ensures syntax validity, none of them has any evidence of functional correctness. Without comprehensive unit tests or simulation-based feedback during dataset construction, models fine-tuned on these corpora may produce code that compiles but still fails to meet the intended natural language specification.

A recent work, OpenLLM-RTL [30], explores the idea of using LLMs to generate assertions, producing a functionally verified dataset of 7k examples. While sharing the same goal of improving functional correctness in fine-tuning datasets, our work takes a different approach by generating unit tests for validation. Our final dataset contains over 125k examples, by far the largest functionally validated RTL dataset.

3 Methodology

3.1 Overview

We aim to improve the quality of fine-tuning datasets consisting of natural language specifications paired with syntactically correct Verilog designs, as seen in prior work [6–8, 24, 26]. These datasets, including Origen [8], contain Verilog designs that pass syntax checks but are not validated against unit tests to ensure functional correctness. To address this limitation, we introduce an automated dataset augmentation pipeline that leverages a teacher language model, e.g., GPT-4o-mini, to validate each example through iterative refinement. As illustrated in Figure 1, given a natural language specification and an initial RTL design, the teacher model first generates a unit test. If the RTL design fails the simulation, the model iteratively revises the design based on the error message. When needed, it also updates the unit test to better align with the natural language specification. Although our experiments focus on augmenting the Origen dataset due to its size and quality, the proposed methodology is broadly applicable to any dataset lacking test validation.

The pipeline begins with the original dataset $D = \{(\text{specification}, \text{design})\}$, where each RTL design is intended to implement a corresponding natural language specification. However, because no tests are provided, there is no evidence that the designs exhibit the intended functional behavior. For each pair, we prompt the teacher model, GPT-4o-mini, to generate a unit test for the design. The test is compiled and simulated with the design to check for correctness, where correctness means the design passes the simulation test.

If the simulation fails, we extract the resulting error message and re-invoke the teacher model using a refinement prompt. This prompt includes the specification, the current design and test, and the error message. The model attempts to resolve the failure by making minimal modifications to the design, the test, or both. This refinement process repeats iteratively: each candidate is re-simulated, and the cycle continues until the design passes the test or a maximum number of attempts is reached.

The final output is a validated dataset $D' = \{(\text{specification}, \text{design}, \text{test})\}$, where each triplet contains a natural language specification, a Verilog design, and unit tests. A concrete motivating example is shown in Section 3.2, and the details of the algorithm and prompts are provided in Section 3.3.

3.2 Motivating Example

Figure 2 presents a motivating example taken directly from the Origen dataset [8], highlighting a key limitation of datasets that rely only on syntax checks for validation. Prior work in RTL generation typically assumes that syntactic correctness is sufficient for fine-tuning, without verifying functionality through unit tests. This example demonstrates that a design can compile without errors yet fail to implement the intended behavior. It also illustrates how our method can automatically detect and correct such issues through test generation and iterative refinement.

This example includes a natural language specification (Figure 2a), a buggy RTL design from the original dataset (Figure 2b), and a corrected design produced by our pipeline (Figure 2c). The specification describes a simple combinational module, `and3`, which computes the bitwise AND of three one-bit inputs: `a`, `b`, and `c`.

The original design, though syntactically valid, is functionally incorrect due to several semantic issues. First, it misuses non-blocking assignments (`<=`) inside a combinational `always @*` block, which can lead to counterintuitive synthesis results. Second, if instead used inside a sequential block, the sequence of non-blocking assignments in the design—`y <= a`, then `y <= y & c`, and finally `y <= y & b`—does not correctly compute and store in `y` the bitwise AND of `a`, `b`, and `c`. In particular, non-blocking assignments defer updates until the end of the current timestep, meaning that all assignments operate on the same initial value of `y`, and only the final assignment takes effect. Finally, if the non-blocking assignments were replaced with blocking ones, the code would introduce a combinational feedback loop, which cannot stabilize.

```
`timescale 1ns/1ps
module tb_and3;
  reg  a = 0, b = 0, c = 0;
  wire y;

  // Instantiate the DUT (Design Under Test)
  and3 uut (.a(a), .b(b), .c(c), .y(y));

  initial begin
    // Wait for signals to settle
    #1;

    // Set all inputs to 1; expected y = 1
    {a, b, c} = 3'b111;
    #1;

    // Check output, report error if incorrect
    if (y !== 1'b1)
      $fatal(1, "FAIL: y=%b (expected 1)", y);

    $display("PASS");
    $finish;
  end
endmodule
```

Figure 3: Unit test for the `and3` module. The buggy design (Figure 2b) times out on this test, while the corrected design (Figure 2c) passes successfully. The test is generated by the teacher model GPT-4o-mini using the prompt in Figure A1a, and is used to validate and augment the original dataset, which contains no tests.

These types of errors occur because the RTL code in prior datasets, including Origen [8], is synthetically generated by teacher LLMs such as Claude 3.5 and filtered only through syntax checks. Without simulation or test-based validation, semantic bugs that affect functional correctness remain undetected.

We provide the natural language specification and the buggy RTL design to the teacher model GPT-4o-mini, prompting it to generate a unit test using the template shown in Figure A1a (further detailed in Section 3.3). The resulting test is shown in Figure 3, which sets all three inputs to 1 and checks whether the output `y` evaluates to 1 as expected. When the buggy design (Figure 2b) is simulated with this test, it hangs and ultimately times out. The bug exemplifies a combinational loop. The always @* block is meant for combinational logic and its evaluation is triggered upon changes to any of the variables read inside the block. In this case, an evaluation of the block is triggered when

Algorithm 1 Dataset Augmentation with a Teacher LLM

Input: Original dataset $D = \{(s_i, d_i)\}_{i=1}^{N}$
▷ s_i: NL specification; d_i: RTL design
Maximum attempts T
Define: $GenTestTpl \leftarrow$ prompt template for test generation
$RefineTpl \leftarrow$ prompt template for iterative refinement
Output: Augmented dataset $D' = \{(s_i, d_i, t_i)\}_{i=1}^{M}$
▷ t_i: Generated unit test

$D' \leftarrow \varnothing$
for each $(s, d) \in D$ **do**
  $attempt \leftarrow 0,\ success \leftarrow$ false
  while $attempt < T \wedge \neg success$ **do**
    $attempt \leftarrow attempt + 1$
    if $attempt == 1$ **then**
      $d, t \leftarrow \text{LLMInvoke}(GenTestTpl, s, d)$
    else
      $d, t \leftarrow \text{LLMInvoke}(RefineTpl, s, d, t, err)$
    $success, err \leftarrow \text{RunVerilogTest}(d, t)$
    if $success$ **then**
      $D' \leftarrow D' \cup \{(s, d, t)\}$
return D'

either y, a, b, or c changes. However, y is both read (on the RHS) and written (on the LHS) in the same block. Upon evaluating the block, it schedules an update to y, which causes a change to y. This change retriggers the block, leading to another scheduled update to y, and so on. This loop continues indefinitely, preventing the simulation from converging.

The corrected version replaces the non-blocking assignments with a single blocking assignment (`=`), ensuring that `y` is updated immediately with the result of `a & b & c`, as required by the specification. This version passes the test generated by the teacher model and behaves correctly under simulation.

This example underscores the importance of functional validation in RTL datasets. Syntax checks alone cannot catch subtle but critical semantic errors. Our methodology, through teacher-driven test generation and iterative refinement, ensures that each design in the augmented dataset is not only syntactically valid but also functionally validated with unit tests.

3.3 Algorithm and Prompts

Algorithm 1 presents our automated pipeline for transforming an unvalidated RTL dataset into a functionally validated one. Starting from a dataset $D = \{(s_i, d_i)\}_{i=1}^{N}$, where each example consists of a natural language specification s_i and a corresponding RTL design d_i (e.g., from Origen [8]), the goal is to generate a unit test t_i that validates the functional correctness of the design. If the design fails to pass the test, we invoke an iterative refinement loop that updates the design and test until it passes or a maximum number of attempts T is reached. We set $T = 5$ in our experiments.

The procedure is powered by a teacher model, GPT-4o-mini, which corresponds to the `LLMInvoke` calls in Algorithm 1. While stronger models such as GPT-4o or o3-mini may yield better performance, we use GPT-4o-mini in practice because of the large size of the dataset (217,462 examples in Origen) and the high cost associated with repeated API queries to OpenAI models.

The process begins by prompting the teacher model with the test generation template (Figure A1a), together with a natural language specification and its initial RTL design (e.g., Figure 2a and Figure 2b). The model then produces a candidate unit test (e.g., Figure 3) designed to check whether the design satisfies the intended functionality under simulation.

The design and test are compiled and simulated using standard Verilog tooling. If the test fails, for example due to a timeout, incorrect output, or another runtime error, we construct a refinement prompt (Figure A1b) that includes the specification, the failing design and test, and the simulation

Model Type	Evaluated Model	VerilogEval V1.0 [9] (using pass@k metric)						RTLLM V1.1 [10] (using pass@5 metric)	
		Eval-Machine (%)			Eval-Human (%)			Syntax-VCS (%)	Functional (%)
		k=1	k=5	k=10	k=1	k=5	k=10		
Base Models	o4-mini-2025-04-16	61.9	67.8	68.6	**64.3**	66.4	67.1	86.2	**72.4**
	GPT-4o-2024-11-20	63.7	66.5	67.1	54.3	60.4	62.2	**100.0**	69.0
	GPT-4o-mini-2024-07-18	55.7	62.4	64.3	44.7	51.6	55.1	89.7	65.5
	DeepSeek-R1	65.7	70.9	**72.0**	62.8	**69.1**	69.9	79.3	58.6
	o3-mini-2025-01-31	**66.4**	**71.6**	**72.0**	62.0	68.9	**69.9**	69.0	55.2
	Qwen2.5-14B-Instruct	47.8	54.2	55.2	35.3	40.0	42.3	69.0	41.4
	Gemini-2.0-flash-001	60.3	62.6	63.6	52.1	57.6	59.0	65.5	34.5
	DeepSeek-R1-Distill-Qwen-14B	46.2	64.1	68.5	36.7	51.7	55.1	62.1	34.5
	DeepSeek-Coder-7B-v1.5	44.4	58.9	62.9	25.8	40.2	44.9	48.3	24.1
	LLaMA-2-7B	7.0	15.6	18.9	0.4	2.1	3.8	3.4	0.0
Fine-Tuned Models (Prior Work)	OriGen [8]	35.9	65.1	68.5	22.3	47.5	51.9	51.7	37.9
	RTLCoder-DeepSeek [6]	22.0	51.4	57.3	14.7	35.2	42.3	17.2	10.3
	RTLCoder-Mistral [6]	17.6	46.4	56.6	12.4	31.5	36.5	3.4	0.0
	ChipGPT-LLaMA3.1-8B-SFT [27]	17.6	46.4	56.6	12.4	31.5	36.5	13.8	0.0
	ChipGPT-LLaMA2-SFT-7B [27]	0.9	4.2	7.7	0.6	2.2	3.8	6.9	0.0
Our Work	VeriCoder	55.7	62.9	64.3	38.3	49.2	51.9	79.3	48.3

Table 2: RTL code generation performance across models. To ensure a fair comparison, we use the same input prompts and apply identical post-processing scripts, running inference with model weights released by prior work.

error message (corresponding to the `err` variable in Algorithm 1). This prompt is then passed to the teacher model, which attempts to fix the issue by making edits to the design, the test, or both.

The refinement process repeats until the updated design passes simulation or the maximum number of attempts T is reached. Once a design successfully passes its test, the validated triple (s_i, d_i, t_i) is added to the output dataset D'.

This strategy enables systematic detection and correction of subtle RTL bugs that cannot be identified through syntax checks alone. By integrating LLM-based test generation and iterative refinement into the dataset construction pipeline, we produce a dataset that is not only syntactically valid but also functionally validated through simulation.

While we cannot guarantee that every design in the augmented dataset is functionally correct under all possible inputs, the inclusion of unit tests makes it substantially more robust than prior approaches that rely solely on syntactic checking. We consider this a practical and scalable step toward constructing higher-quality fine-tuning datasets for RTL generation.

4 Experimental Setup

4.1 Dataset

Following the methodology described in Section 3, we construct a fine-tuning dataset comprising 125,777 examples. Each example includes a natural language specification, a corresponding RTL design, and associated unit tests. Table A1 summarizes key statistics: the specifications contain an average of 247 words (ranging from 116 to 549), RTL implementations average 35 lines of code (ranging from 5 to 225), and unit tests average 55 lines (ranging from 6 to 197). We use the specification–solution pairs from this dataset to train our model, VeriCoder. Other details on the experimental setup are discussed in Appendix A.3, Appendix A.4, and Appendix A.5.

5 Results

5.1 Main Evaluation Results

Table 2 shows the results. Our major findings are as follows:

Comparison with prior work VeriCoder achieves state-of-the-art results across two RTL code generation benchmarks, outperforming all previously released open-source fine-tuned models. On VerilogEval-Machine, VeriCoder attains a pass@1 accuracy of 55.7%, representing a 19.8 percentage point improvement over the best prior model, OriGen. On VerilogEval-Human, it reaches 38.3%, exceeding OriGen by 16.0 percentage points. Across all evaluated k-shot settings (k=1, 5, 10),

VeriCoder consistently maintains its lead on the Human split. On the RTLLM benchmark, VeriCoder achieves 79.3% syntax correctness and 48.3% functional correctness, surpassing OriGen's 51.7% and 37.9%, respectively. In conclusion, VeriCoder delivers relative improvements of up to 71.7% on VerilogEval and 27.4% on RTLLM in pass@k accuracy, surpassing the previous state-of-the-art model on both benchmarks.

To better understand the relatively low performance of ChipGPT [27], we examined its outputs in detail. We observed that its generated RTL designs often include module headers that deviate from the given specifications, revealing difficulty in precise instruction following. Moreover, its base model, LLaMA2-7B, performs even worse, suggesting that limitations in the instruction-following capabilities of the underlying pretrained model constrain the effectiveness of the fine-tuned variant. For a fair comparison, we do not apply any of the model-specific customized post-processing scripts that attempt to fix syntax or header issues. Instead, we use a standardized evaluation script for all models, extracting Verilog code as-is to ensure consistency.

Effectiveness of our fine-tuning Starting from Qwen-2.5-14B-Instruct as our base model, VeriCoder delivers substantial gains across VerilogEval. On the VerilogEval-Machine split, pass@1 jumps up by 7.6%, pass@5 by 4.0%, and pass@10 by 2.1%, and VerilogEval-Human reflects the same trend. On RTLLM, functional pass@5 is 7% higher than its base model. Specifically, VeriCoder even marginally outperforms one of the commercial models, Google's Gemini-2.0-flash, on pass@5 and pass@10 metrics of Eval-Machine as well as on RTLLM. Together, these results demonstrate that our fine-tuning process and our validated dataset significantly boost pass@k metrics and semantic correctness in RTL generation.

Model gap remains Despite the observed improvements, a substantial performance gap persists between VeriCoder and the strongest large models. For instance, o3-mini attains 66.4% on VerilogEval Pass@1 compared to VeriCoder's 55.7%. DeepSeek-R1 achieves 69.1% on human-graded Pass@5, versus VeriCoder's 49.2%. Commercial LLMs such as GPT-4o reach perfect 100.0% Syntax-VCS validity and 69.0% functional correctness, while VeriCoder records 79.3% and 48.3%, respectively. Despite the performance gap, open-source lightweight models offer compelling advantages. They provide transparency, allow for local deployment, and ensure intellectual property protection, i.e., capabilities that are particularly important for RTL design workflows where security, customizability, and integration into existing toolchains are critical.

5.2 Ablation Study of Dataset

To assess the impact of dataset quality on RTL code generation, we conduct an ablation study using the same base model, Qwen2.5-14B-Instruct, fine-tuned on two datasets: (1) the unvalidated OriGen dataset from prior work [8], and (2) our newly curated, functionally validated dataset. All factors, including dataset size, fine-tuning hyperparameters, training procedures, and evaluation settings, are held constant to ensure a fair comparison.

Model	**VerilogEval** [9] (Pass@5)	**RTLLM** [10] (Pass@5)	
		Syntax	Func
Qwen2.5-14B-Instruct (base)	46.8	69.0	41.4
Qwen w/ unvalidated data	53.5	75.9	44.8
Qwen w/ validated data	55.8	79.3	48.3

Table 3: We performed fine-tuning on the same base model using a functionally validated dataset and the functionally unvalidated dataset [8]. We report Pass@5 metrics for all models on two benchmarks.

Across all metrics, we observe a consistent improvement as dataset quality increases. On the VerilogEval benchmark, the base model achieves 46.8% Pass@5. Fine-tuning on the unvalidated dataset raises performance to 53.5%, while our validated dataset further improves it to 55.8%. For RTLLM syntax correctness, the trend is similar: 69.0% for the base model, 75.9% for the unvalidated version, and 79.3% when trained on validated data. Functional correctness sees even more significant improvement, rising from 41.4% (base) to 44.8% (unvalidated) and ultimately to 48.3% (validated).

These results demonstrate that functionally validated data provides more effective supervision than existing unvalidated data. This also underscores the importance of dataset quality in fine-tuning LLMs for RTL code generation.

5.3 Test Passing Rates of Non-Validated Datasets

Prior Datasets	# Sampled Examples	Test Passing (%)
RTLCoder [6]	1000	24.4
OriGen [8]	1000	53.5

Table 4: Test passing rates (%) of datasets released by prior work on a randomly sampled set of 1000 examples.

We examine the quality of fine-tuning datasets released by prior work by evaluating their passing rates against our synthetic unit tests generated by the teacher model GPT-4o-mini. For each corpus, we randomly sample 1,000 Verilog implementations and apply the test generation and refinement pipeline described in Section 3. We then run corresponding unit tests against the original design and measure the proportion of the original designs that successfully pass the generated tests. As shown in Table 4, only 24.4% examples of the RTLCoder dataset [6] pass our functional tests, while OriGen [8] reaches 53.5%.

OriGen's higher pass rate aligns with its stronger code generation results in Table 2, hinting at a positive link between dataset validity and downstream performance. These findings highlight the potential value of incorporating functional correctness validation into fine-tuning dataset curation for better RTL code generation.

6 Discussion and Future Work

While VeriCoder, combining unit test generation with feedback-driven refinement, improves the functional correctness of generated RTL code, it does not fully guarantee correctness. Synthetic test cases may fail to capture all possible edge cases. To address this challenge, future work should explore integrating formal verification techniques into the dataset construction pipeline to rigorously ensure the correctness of the generated code. Recent advancements have demonstrated promising results in translating natural language instructions into formal specifications [31, 16], as well as enforcing formal constraints during LLM-based code generation [32].

Moreover, most existing approaches, including VeriCoder, focus on small-scale RTL generation. However, practical hardware development often involves large, repository-level codebases with intricate cross-file dependencies and requirements for long-range context [33–35]. Recent work has begun to address these challenges through techniques such as combining fine-tuning with retrieval-augmented RTL code generation [36, 37]. Extending VeriCoder's unit test generation and feedback-directed refinement components to the repository scale will enable LLMs to handle more real-world RTL tasks.

Furthermore, reinforcement learning (RL) offers a powerful framework for further optimizing large language models' performance beyond what is achievable through supervised fine-tuning alone. Recent studies have demonstrated the effectiveness of RL in enhancing LLM-based code generation by incorporating diverse forms of feedback, such as test case outcomes, compiler diagnostics, and formal verification results [38, 39, 30]. Building on this progress, future work could investigate applying RL techniques to the VeriCoder dataset, using the accompanying test cases as a feedback signal to iteratively improve RTL code generation quality.

7 Conclusion

Recent advances in Large Language Models (LLMs) have opened new possibilities for Electronic Design Automation (EDA), particularly in RTL code generation. However, most existing datasets emphasize syntactic validity while overlooking functional correctness, which limits the effectiveness of fine-tuned models. We introduce VERICODER, a model fine-tuned on a dataset with 125,000 examples that is validated for functional correctness. This dataset is constructed using a feedback-directed refinement pipeline guided by a teacher LLM, which generates and iteratively updates both RTL designs and unit tests until the design passes simulation. The resulting dataset consists of functionally validated triples comprising a natural language specification, an RTL implementation, and a passing test. Fine-tuned on this dataset, VERICODER achieves state-of-the-art results on two established RTL benchmarks, yielding relative improvements of up to 71.7% on VerilogEval and 27.4% on RTLLM. An ablation study confirms the impact of functional validation on model performance, underscoring the importance of high-quality training data. Looking ahead, future work may incorporate formal verification and reinforcement learning to further improve models' performance in AI-assisted hardware design.

References

[1] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[2] L. Chen, Y. Chen, Z. Chu, W. Fang, T.-Y. Ho, R. Huang, Y. Huang, S. Khan, M. Li, X. Li *et al.*, "The dawn of ai-native eda: Opportunities and challenges of large circuit models," *arXiv preprint arXiv:2403.07257*, 2024.

[3] R. Zhong, X. Du, S. Kai, Z. Tang, S. Xu, H.-L. Zhen, J. Hao, Q. Xu, M. Yuan, and J. Yan, "Llm4eda: Emerging progress in large language models for electronic design automation," *arXiv preprint arXiv:2401.12224*, 2023.

[4] Z. He and B. Yu, "Large language models for eda: Future or mirage?" in *Proceedings of the 2024 International Symposium on Physical Design*, 2024, pp. 65–66.

[5] X. Yao, Y. Wang, X. Li, Y. Lian, R. Chen, L. Chen, M. Yuan, H. Xu, and B. Yu, "Rtlrewriter: Methodologies for large models aided rtl code optimization," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–7.

[6] S. Liu, W. Fang, Y. Lu, J. Wang, Q. Zhang, H. Zhang, and Z. 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.

[7] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[8] F. Cui, C. Yin, K. Zhou, Y. Xiao, G. Sun, Q. Xu, Q. Guo, D. Song, D. Lin, X. Zhang *et al.*, "Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection," *arXiv preprint arXiv:2407.16237*, 2024.

[9] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[10] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 722–727.

[11] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language model," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[12] Y. Liao, T. Adegbija, and R. Lysecky, "Are llms any good for high-level synthesis?" in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–8.

[13] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. C. Lin, "Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[14] Z. Yan, Y. Qin, X. S. Hu, and Y. Shi, "On the viability of using llms for sw/hw co-design: An example in designing cim dnn accelerators," in *2023 IEEE 36th International System-on-Chip Conference (SOCC)*. IEEE, 2023, pp. 1–6.

[15] Z. Liang, J. Cheng, R. Yang, H. Ren, Z. Song, D. Wu, X. Qian, T. Li, and Y. Shi, "Unleashing the potential of llms for quantum computing: A study in quantum architecture design," *arXiv preprint arXiv:2307.08191*, 2023.

[16] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: Interactively translating unstructured natural language to temporal logics with large language models," in *International Conference on Computer Aided Verification*. Springer, 2023, pp. 383–396.

[17] C. Sun, C. Hahn, and C. Trippel, "Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions," in *First International Workshop on Deep Learning-aided Verification*, 2023.

[18] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "Chateda: A large language model powered autonomous agent for eda," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[19] Z. Xiao, X. He, H. Wu, B. Yu, and Y. Guo, "Eda-copilot: A rag-powered intelligent assistant for eda tools," *ACM Transactions on Design Automation of Electronic Systems*, 2025.

[20] K. Xu, J. Sun, Y. Hu, X. Fang, W. Shan, X. Wang, and Z. Jiang, "Meic: Re-thinking rtl debug automation using llms," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[21] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[22] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[23] E. Dehaerne, B. Dey, S. Halder, and S. De Gendt, "A deep learning framework for verilog autocompletion towards design and verification automation," *arXiv preprint arXiv:2304.13840*, 2023.

[24] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "Betterv: Controlled verilog generation with discriminative guidance," *arXiv preprint arXiv:2402.03375*, 2024.

[25] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[26] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024.

[27] K. Chang, K. Wang, N. Yang, Y. Wang, D. Jin, W. Zhu, Z. Chen, C. Li, H. Yan, Y. Zhou *et al.*, "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*, 2024, pp. 1–6.

[28] Y. Zhang, Z. Yu, Y. Fu, C. Wan, and Y. C. Lin, "Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[29] E. Goh, M. Xiang, I. Wey, T. H. Teo *et al.*, "From english to asic: Hardware implementation with large language model," *arXiv preprint arXiv:2403.07039*, 2024.

[30] S. Liu, Y. Lu, W. Fang, M. Li, and Z. Xie, "Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[31] D. Mendoza, C. Hahn, and C. Trippel, "Translating natural language to temporal logics with large language models and model checkers," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*, 2024, pp. 1–11.

[32] P. Aggarwal, B. Parno, and S. Welleck, "Alphaverus: Bootstrapping formally verified code generation through self-improving translation and treefinement," *arXiv preprint arXiv:2412.06176*, 2024.

[33] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[34] T. Suresh, R. G. Reddy, Y. Xu, Z. Nussbaum, A. Mulyar, B. Duderstadt, and H. Ji, "Cornstack: High-quality contrastive data for better code retrieval and reranking," in *The Thirteenth International Conference on Learning Representations*, 2025.

[35] N. Jain, M. Shetty, T. Zhang, K. Han, K. Sen, and I. Stoica, "R2e: Turning any github repository into a programming agent environment," in *ICML*, 2024.

[36] P. Wu, N. Guo, J. Lv, X. Xiao, and X. Ye, "Rtlrepocoder: Repository-level rtl code completion through the combination of fine-tuning and retrieval augmentation," *arXiv preprint arXiv:2504.08862*, 2025.

[37] Z. Li, C. Xu, Z. Shi, Z. Peng, Y. Liu, Y. Zhou, L. Zhou, C. Ma, J. Zhong, X. Wang *et al.*, "Deepcircuitx: A comprehensive repository-level dataset for rtl code understanding, generation, and ppa analysis," *arXiv preprint arXiv:2502.18297*, 2025.

[38] N. Wang, B. Yao, J. Zhou, X. Wang, Z. Jiang, and N. Guan, "Large language model for verilog generation with golden code feedback," *arXiv preprint arXiv:2407.18271*, 2024.

[39] J. Wang, Z. Zhang, Y. He, Y. Song, T. Shi, Y. Li, H. Xu, K. Wu, G. Qian, Q. Chen *et al.*, "Enhancing code llms with reinforcement learning in code generation," *arXiv preprint arXiv:2412.20367*, 2024.

A Appendix

A.1 Prompt Templates

Prompt templates are shown in Figure A1.

Prompt Template

System Prompt You are a Verilog design and testing expert. Given a hardware specification described in natural language, your job is to generate both a correct Verilog module and a corresponding unit test that checks its functionality through simulation.

User Prompt
- *Natural Language Specification*: {NL Spec}
- *Initial Implementation*: {design}
- Your task:
  1. Provide the unit tests for the given design.
  2. Revise the Verilog implementation if the original design fails to pass your test cases.
  3. Follow good coding practices, such as using meaningful comments to document key logic and decision points.
  4. Use `$fatal(1, "msg")` to flag incorrect behavior.
  5. Output format: `{"design": "...", "test": "..."}`

(a) Prompt for generating a Verilog module's corresponding test

Prompt Template

System Prompt You are a Verilog design and testing expert. Analyze a failing design and its test, and make minimal yet sufficient edits to correct the issue while preserving the intended behavior specified in natural language.

User Prompt
- *Natural Language Specification*: {NL Spec}
- *Previous Design and Test*: {design}, {test}
- *Simulation Output*: {error message}
- Your task:
  1. Carefully identify the root cause of the failure by analyzing the code and the error message.
  2. Make changes to either the design or the test (or both) to resolve the issue while maintaining correctness.
  3. Output format: `{"explanation": "...", "design": "...", "test": "..."}`

(b) Prompt for refining a failing Verilog design and test

Figure A1: Prompt templates provided to the teacher model for automated Verilog test generation and refinement, ensuring that the final design passes the generated test and matches the original natural language specification.

A.2 Dataset

Category	Count	Length		
		Min	Max	Avg
NL specification (words)	125,777	116	549	247
Design (lines of RTL)		5	225	35
Unit tests (lines of RTL)		6	197	55

Table A1: Dataset statistics: total number of examples and length distributions for natural language specifications, RTL implementations, and unit tests in the VeriCoder dataset.

A.3 LoRA Fine-Tuning Setup

Following standard practices for LLM fine-tuning, we fine-tune the base model of Qwen2.5-14B-Instruct using Low-Rank Adaptation (LoRA), with a rank of 16 and a scaling factor of 32 to all linear projection layers in the transformer. Training is conducted over 3 epochs with a batch size of 40. We adopt a constant learning rate of 1×10^{-5}, paired with a linear decay scheduler and a warm-up ratio of 0.05. The optimizer is used with a weight decay of 1×10^{-4}, and gradient clipping is applied with a maximum norm of 1.

A.4 Benchmarks and Metrics

Following the evaluation protocol established in prior work [7, 8], we benchmark against VerilogEval [9] and RTLLM [10]. For VerilogEval, we report the standard *Pass@k* metric with $k \in \{1, 5, 10\}$, which estimates the expected probability that at least one of the top-k generated programs passes all test cases. The metric is defined as:

$$\text{Pass@}k = \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

where n is the total number of generated programs and c is the number of correct ones. All test cases are manually created by experts who design the benchmarks. In all evaluations, we set $n = 10$. For RTLLM, we report both syntax correctness and functional correctness using Pass@5. This evaluation setup aligns with that used in prior work [8].

A.5 Models for Evaluation

We evaluate two groups of models. The first group consists of pretrained-only base models, including OpenAI's latest releases (o4-mini, o3-mini, GPT-4o, GPT-4o-mini), Google's Gemini 2.0 Flash, DeepSeek's R1 and DeepSeek-Coder-7B-v1.5 (the base model used in prior work [8]), Meta's LLaMA2-7B model, and Alibaba's Qwen2.5-14B-Instruct (our base model for fine-tuning). The second group includes fine-tuned models with released weights from prior work: Origen [8], RTL-Coder [6], and ChipGPT [27].

To ensure a fair comparison, we use identical input prompts and post-processing scripts across all models. For models released by prior work, we do not adopt their model-specific prompts [8] or inference pipelines [27, 6]. Instead, we apply a uniform evaluation script, with the only variable being the model under test. This standardization is critical, as both input formatting and post-processing can significantly affect performance. By controlling these factors, we isolate model capability and enable a fair comparison.