RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models

YunDa Tsai*
yundat@nvidia.com
NVIDIA

Mingjie Liu*
mingjiel@nvidia.com
NVIDIA

Haoxing Ren
haoxingr@nvidia.com
NVIDIA

ABSTRACT

This paper presents **RTLFixer**, a novel framework enabling automatic syntax errors fixing for Verilog code with Large Language Models (LLMs). Despite LLM's promising capabilities, our analysis indicates that approximately 55% of errors in LLM-generated Verilog are syntax-related, leading to compilation failures. To tackle this issue, we introduce a novel debugging framework that employs Retrieval-Augmented Generation (RAG) and ReAct prompting, enabling LLMs to act as autonomous agents in interactively debugging the code with feedback. This framework demonstrates exceptional proficiency in resolving syntax errors, successfully correcting about 98.5% of compilation errors in our debugging dataset, comprising 212 erroneous implementations derived from the VerilogEval benchmark. Our method leads to 32.3% and 10.1% increase in pass@1 success rates in the VerilogEval-Machine and VerilogEval-Human benchmarks, respectively. The source code and benchmark are available at https://github.com/NVlabs/RTLFixer.

ACM Reference Format:
YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2024. **RTLFixer**: Automatically Fixing RTL Syntax Errors with Large Language Models. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657353

1 INTRODUCTION

Large language models (LLMs) present great promise in automating hardware design, especially in their capacity to understand design intentions and produce Verilog code from natural language [8]. Recent efforts, such as VeriGen [17] and VerilogEval [9], have primarily focused on zero-shot code generation. However, akin to numerous complex programming tasks, generating flawless code in a single attempt poses a significant challenge, with a high likelihood of errors. Consequently, there exists a clear need for robust debugging and refinement capabilities in Verilog code generated by Large Language Models. This need arises from that like human programmers, achieving precision often requires multiple iterations.

Most importantly, it is evident that Large Language Models encounter challenges in generating fully syntactically correct Verilog code. Surprisingly, our analysis reveals that a substantial 55% of the errors generated by LLMs for Verilog code are comprised of syntax errors, surpassing the occurrence of logic errors detected through simulation. Rectifying syntax errors not only enhances the overall accuracy of LLM-generated code but also holds the potential to alleviate manual efforts for human engineers engaged in Verilog coding. The recognition and mitigation of syntax errors stand as imperative steps, not only for refining LLM capabilities but also for streamlining the coding process for human practitioners in the domain of Verilog development.

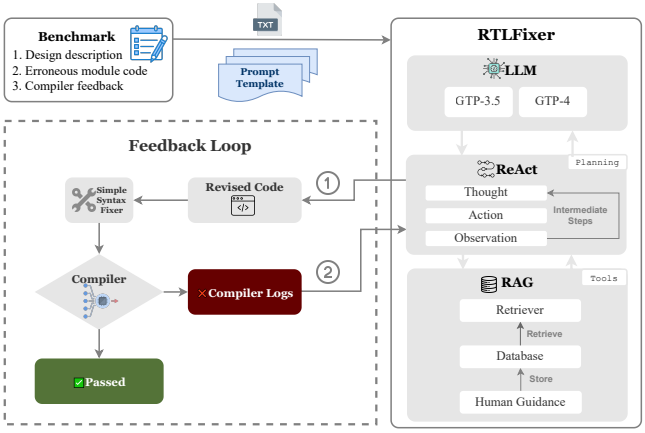

Figure 1: Overview of RTLFixer. The Autonomous Language Agent fixes the syntax error via a feedback loop. ReAct handles the iterative code refinement with intermediate reasoning and action steps. Human expert guidance is incorporated through RAG.

Despite such challenges, LLMs have showcased remarkable capabilities in reasoning and enhancing action plans to address exceptions. The ReAct framework [20] integrates reasoning and action synthesis in language models, demonstrating LLM's capability to engage in reasoning processes and refine decision-making through interactive feedback. Similarly, SelfDebug [3] and SelfEvolve [6] illustrate the model's ability to self-identify mistakes by scrutinizing execution results and articulating generated code in natural language. It is essential to highlight that prior works do not explicitly address the correction of syntax errors and primarily center on improving the accuracy of generated code, particularly in Python, where language models excel syntactically.

On the other hand, Large Language Models are acknowledged for their inclination to produce factual errors, a phenomenon termed hallucination [5]. To mitigate this challenge, the Retrieval-Augmented Generation (RAG) paradigm [7] has been introduced, integrating retrieval mechanisms to improve the precision of generated content by incorporating information from external knowledge sources.

*equal contribution

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3657353

In this paper, we introduce **RTLFixer**, a innovative debugging framework that utilizes LLMs as autonomous language agents in conjunction with RAG. We exclusively focus on addressing the challenge of rectifying syntax errors in RTL Verilog code—an essential problem with potential benefits for both LLMs and human engineers. As shown in Figure 1, our framework combines established human expertise stored in a retrieval database for correcting syntax errors while simultaneously harnessing the capabilities of LLMs as autonomous agents for reasoning and action planning (ReAct). Through incorporating human expertise, our approach provides explicit guidance and explanations when LLMs face challenges in error correction. The stored compiler messages and human expert guidance function as a persistent external non-parametric memory database, enhancing results through RAG. By empowering LLMs with ReAct, LLMs serve as autonomous agents adept at strategically planning intermediate steps for iterative debugging. We also create VerilogEval-syntax, a Verilog syntax debugging dataset, derived from VerilogEval [9], containing 174 erroneous implementations.

Our contributions are summarized as follows:

- Our framework demonstrates an impressive 98.5% success rate in resolving syntax errors, resulting in a noteworthy 32.3% and 10.1% improvement in the pass@1 metrics achieved solely by addressing syntax errors in VerilogEval-Machine and VerilogEval-Human benchmarks, respectively.
- Our framework also improves the syntax success rate from 73% to 93% on the RTLLM benchmark [11], demonstrating the generalizability of this approach.
- Compared to One-shot generation, ReAct enhances syntax success rates by 25.7%, 26.4%, and 31.2% with iterative feedback from Simple, iverilog, and Quartus, respectively.
- RAG with human guidance significantly improves syntax success rates, up to 31.2% and 18.6% with feedback from Quartus for One-shot and ReAct prompting, respectively.

The remainder of this paper is structured as follows. In Section 2, we present preliminary works on LLMs for Verilog code generation, ReAct, and RAG. Our debugging framework **RTLFixer** is elucidated in Section 3, where we empower LLMs as autonomous agents with ReAct and innovatively provide human guidance through RAG. Section 4 details our experimental results, showcasing the effectiveness of our method in correcting syntax errors and improving the pass rate. Finally, Section 6 summarizes and concludes the paper.

2 PRELIMINARIES

In this section, we begin by briefly exploring the advancements and applications of LLMs for Verilog code generation in Section 2.1. We then delve into the synthesis of reasoning and action in LLMs, elaborated in Section 2.2. Finally, we discuss Retrieval-Augmented Generation in Section 2.3.

2.1 LLMs for Verilog Code Generation

Large Language Models exhibit the capability to generate code, with Codex [2] standing as an early exemplar. GitHub Copilot [4], building upon such groundwork, played a crucial role in pioneering LLM-based code completion engines, contributing significantly to the domains of auto-completion and conversational code generation. DAVE [14] emerged as an early study of LLM tailored for hardware design. VeriGen [17] further expanded the dataset scope and experimented with open-sourced models. Following suit, ChipChat [1], leveraging GPT-4, demonstrated the extensive potential of LLMs in collaboratively generating processors and other hardware designs. In parallel, benchmarks such as VerilogEval [9] and RTLLM [11] played a pivotal role in advancing the application of LLMs in Verilog code generation.

2.2 Reasoning and Action Synthesis of LLMs

Large Language Models (LLMs) have demonstrated proficiency in both reasoning and planning across various tasks. In terms of reasoning, methods like chain-of-thought [18] empower LLMs to break down intricate problems into logical steps, significantly enhancing their problem-solving abilities. LLMs also excel in interactive decision-making and formulating action plans, effectively leveraging digital tools, as shown in works such as ToolLLM [15].

The ReAct [20] framework represents a notable advancement in LLM capabilities by seamlessly integrating reasoning and action planning. This framework enables LLMs to generate both reasoning traces and specific actions, facilitating dynamic interaction with external information sources. This integration not only enhances LLM's performance in complex tasks but also renders them more reliable and versatile as autonomous agents, capable of delivering more accurate and context-aware responses.

2.3 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) [7] represents a significant advancement in addressing the limitations of Large Language Models when handling knowledge-intensive tasks. LLMs despite containing extensive factual knowledge, often encounter challenges in accessing and effectively manipulating this information. RAG leverages a combination of LLMs, which serve as parametric memory, and an external knowledge base, such as Wikipedia, functioning as non-parametric memory. This unique approach allows RAG to access and retrieve relevant documents or passages from the external knowledge base based on the input query. Consequently, this enriches the context available to the text generator, resulting in outputs that exhibit improved accuracy and factual consistency. In the context of code generation, several works such as ReACC [10] and RepoCoder [21] have successfully harnessed the capabilities of RAG to enhance the code generation proficiency of LLMs, showcasing its transformative potential.

3 RTLFIXER: RESOLVING SYNTAX ERROR WITH LLM AGENTS AND RETRIEVAL

In this section, we explain the details of **RTLFixer**, which utilizes Autonomous Language Agents enhanced with ReAct and Retrieval-Augmented Generation (RAG). The framework's structure is outlined in Figure 1 (Section 3.1), and the application of ReAct is thoroughly discussed in Section 3.2. Furthermore, Section 3.3 explores the integration of human expert guidance using RAG. Finally, we explain the curation process for our VerilogEval-syntax error dataset.

3.1 Overview of RTLFixer

RTLFixer comprises an LLM for code generation, RAG for accessing human expert guidance, and ReAct for improved task decomposition, tool use, and planning. Our approach starts by formulating

an input prompt integrating a benchmark dataset problem into a template, followed by the agent utilizing RAG and ReAct, revising erroneous Verilog code. If syntax errors persist, error logs from the compiler as well as retrieved human guidance from the database are provided as feedback. This interactive debugging loop can be repeated multiple times until all errors are resolved.

3.2 Reasoning and Action Planning through ReAct Iterative Prompting

We enable Large Language Models to function as autonomous agents for reasoning and action planning through the **ReAct** prompting mechanism [20]. In ReAct, LLMs generate both reasoning traces and task-specific actions in an interleaved manner. The input prompt, along with the ReAct instruction prompt, is provided to an LLM. Subsequently, the LLM initiates the generation of ReAct steps, each consisting of Thought, Action, and Observation components. An example of a ReAct instruction prompt is depicted in Figure 2b, while Figure 2c illustrates the self-prompting process, showcasing the intermediate steps within each iteration of ReAct.

During this process, the LLM prompts itself for thoughts on how to address the error and selects the next action. Potential actions include generating an explanation for the error, searching for a solution in the human expert guidance database, revising the code, and submitting the revised code to the compiler, among other possibilities. The output of the chosen action becomes the observation in the prompt. The agent continues prompting until the compilation is successful, selecting the Finish action to output the final response. If unsuccessful, the process iterates up to n times, where n is a user-selected hyperparameter. Our objective is to assess the effectiveness of a fully automated feedback-driven solution.

We employ **One-shot** prompting, illustrated in Figure 2a, as the baseline for comparing with ReAct. One-shot prompting involves only a single-turn of feedback from the compiler message, but it excludes ReAct prompting, which decomposes syntax fixing through reasoning and action planning, and the iterative multi-round interactions with the compiler.

3.3 Retrieval Augmented Generation (RAG)

We leverage Retrieval-Augmented Generation (RAG), a potent technique that notably enhances Large Language Models' capabilities by incorporating human expert guidance through a retriever. A key distinction from traditional RAG lies in our curated database, enriched with human instructions and demonstrations.

The retrieval database curation process involves a meticulous procedure of categorizing syntax errors and developing instructions and demonstrations for syntax error resolution. In the initial step, we categorize various syntax errors into groups using error number tags provided by compilers (such as Quartus) in the compiler logs. During the manual inspection of LLM's struggle cases, it becomes evident that ambiguous error messages present a significant challenge, impeding the model's error resolution capabilities. The inclusion of clear instructions, and demonstrations of possible solutions enables the LLM to adeptly address errors. To facilitate this, human experts offer detailed explanations for compiler logs, serving as human expert guidance. An illustrative example of common errors is showcased in Figure 3. Subsequently, all compiler logs,

System Prompt:
Implement the Verilog module based on the following description. Assume that signals are positive clock/clk edge triggered unless otherwise stated.

Problem Description:
Implement the Verilog module based on the following description. Assume that signals are positive clock/clk edge triggered unless otherwise stated.
Given an 8-bit input vector [7:0], reverse its bit ordering.

```
module top_module (
input [7:0] in,
output [7:0] out
);
```

Erroneous Implementation:

```
assign {out[0],out[1],out[2],out[3],out[4],out[5],out[6],out[8]} = in;
endmodule
```

Feedback:
main.v:5: error: Index out[8] is out of range.
1 error(s) during elaboration.

(a) One-shot Prompt Template

Solve a task with interleaving Thought, Action, Observation steps.
Thought can reason about the current situation, and Action can be the following types:
(1) Compiler[code], which compiles the input code and provide error message if there is syntax error.
(2) Finish[answer], which returns the answer and finished the task.
(3) RAG[logs], input the compiler log and retrieve expert solutions to fix the syntax error.

(b) ReAct Instruction (System Prompt)

Question:
What is the syntax error in the given Verilog module implementation and how to fix it?

Thought 1:
Implement a finite state machine that searches for message boundaries in a byte stream.
Action 1: Compiler

```
always @ (state or in) begin
case (state)
 4'b0001: begin
```

Observation 1:
main.v:15: error: out is not a valid l-value in top_module.
Thought 2:
The code has a syntax error because the "out" signal was declared as a wire instead of a register. I need to correct this and run the compilation.
Action 2: RAG[..not a valid l-value..]
Use assign statements instead of always block if possible.
...
Action n: Finish
The revised code is as follow: ...

(c) ReAct Steps with Thought-Action-Observation.

Figure 2: Prompts used for ReAct. (a) shows the One-shot prompting template with feedback message. (b)-(c) demonstrate the example where LLMs serve as autonomous agents with ReAct to decompose syntax fixing problems with reasoning and planning.

error code segments, and corresponding human guidance undergo systematic storage in the database for future retrieval.

We integrate the human guidance and demonstration database with Large Language Models (LLMs) using Retrieval-Augmented Generation (RAG). RAG enables the retrieval of pertinent documents or data from a source, utilizing this information as context for the original input prompt. This approach allows the language model to access the latest information without the need for retraining, proving especially valuable in enhancing the model's capacity to generate more accurate and reliable outputs. Leveraging RAG

Compiler Logs:
Object 'clk' is not declared. Verify the object name is correct. If the name is correct, declare the object.
Human Expert Guidance:
Check if 'clk' is an input. If not, and if 'clk' is used within the module, make sure the name is correct. If it's meant to trigger an 'always' block, replace 'posedge clk' with '*'.

Compiler Logs:
Index cannot fall outside the declared range for vector
Human Expert Guidance:
Carefully examine the index values to prevent encountering 'index out of bound' errors in your code. When utilizing parameters for indexing, try to use binary strings for performing the indexing operation instead.

Figure 3: Examples of common error categories that LLM constantly could not solve and the corresponding human expert guidance in the retrieval database.

further ensures that the language model has access to the most current and relevant information, including compiler logs and human guidance, facilitating effective error resolution.

Figure 3 illustrates two common error categories along with a demonstration of compiler logs and corresponding human guidance. For this task, common retrievers such as pattern-matching, fuzzy search, or similarity search with a vector database are suitable. In our experiments, we opted for an exact match to error tags for simplicity, given the limited number of error cases. We collected 7 common error categories with 30 entries for iverilog and 11 common error categories with 45 entries for Qaurtus in total.

3.4 Debugging Dataset

We created a novel benchmark dataset, VerilogEval-syntax, based on the VerilogEval benchmark [9]. This dataset comprises flawed code implementations sourced from the VerilogEval problem set. Each entry includes the original problem description and erroneous implementation containing syntax errors.

The dataset curation includes sampling, filtering, and clustering. Code samples were selected from VerilogEval problems using One-shot and ReAct prompting methods with *gpt-3.5-turbo* model, retaining only error-inducing samples. In the filtering phase, we focus on code with compile errors and use the following processing and filtering criteria: extraction of code from markdown blocks, validation of module statements, and removal of samples with extraneous language or empty module bodies. The final step involved clustering using DBSCAN [16] with Jaccard distance [12], grouping similar implementations to select representative examples while ensuring a diverse representation of syntax errors. This results in a total of 212 erroneous implementations in the dataset.

4 EXPERIMENTS

In this section, we first present the evaluation metrics in Section 4.1, followed by our primary findings showcased in Section 4.2. Within this section, we show the performance improvements and the impact of ReAct and RAG. Finally, Section 4.3 details ablation studies on the quality feedback message and LLM.

Setup: We conduct all experiments with GPT-3.5 as the LLM through OpenAI APIs [13], except for the ablation experiment on different LLM. We specifically used *gpt-3.5-turbo-16k-0613*. A simple rule-based syntax fixer is applied to every LLM-generated verilog code, which avoids simple errors such as misplaced timescale derivatives. In all experiments, we set the sampling temperature to 0.4. For ReAct prompting, we restrict the LLM to a maximum of 10 iterations of Thought-Action-Observation, where Action might involve interactions with the compiler. We consider the syntax error resolved if any of the generated code passes. To limit test variance, we repeat each experiment 10 times and report the average.

4.1 Evaluation Metric

Compile Fix rate: To demonstrate the debugging capability of our method, we calculate the expectation fix rate, with c as the number of fixed samples out of all $n = 10$ samples.

$$\text{fix rate} = \mathop{\mathbb{E}}_{problems}\left[\frac{c}{n}\right] \tag{1}$$

Functional Correctness: We follow recent work in directly measuring code functional correctness with simulation through pass@k metric [2], where a problem is considered solved if any of the k samples passes the tests. We use the unbiased estimator as follow and ensure $n = 20$ is sufficiently large:

$$\text{pass@k} = \mathop{\mathbb{E}}_{problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{2}$$

4.2 Main Results

Prompt	RAG	Simple	iverilog	Quartus	GPT-4
One-shot	w/o	0.414	0.536	0.587	0.91
	w/	-	0.800	0.899	0.98
ReAct	w/o	0.671	0.731	0.799	0.92
	w/	-	0.820	0.985	0.99

Table 1: Fix rate for One-shot vs. ReAct, w/ and w/o RAG, ablation on feedback quality and LLMs on VerilogEval-syntax.

Dataset VerilogEval	Set	pass@1 original	pass@1 fixed	pass@5 original	pass@5 fixed
Human	All	0.267	0.368	0.458	0.506
	easy	0.521	0.666	0.808	0.847
	hard	0.053	0.120	0.164	0.221
Machine	All	0.467	0.799	0.691	0.891
	easy	0.568	0.833	0.782	0.892
	hard	0.367	0.771	0.601	0.890

Table 2: Pass@k for simulation pass rate on VerilogEval dataset after fixing syntax errors.

The main results in Table 1 show the effectiveness of ReAct and RAG which each provided performance gain in a large margin. We note that One-shot generation includes only a single-turn interaction with either simple or compiler message as feedback. Table 2 showed the improvement of pass@{1,5} on VerilogEval dataset after fixing syntax errors with visualizations in Figure 4. The VerilogEval-Human benchmark statistics show that syntax errors constitute a significant 55% of errors in GPT-3.5 generated Verilog code, surpassing simulation errors. With just addressing syntax errors using our approach (shown in the inner circle), the pass rate increases from 26.7% to 36.8%. Table 3 shows that our approach can generalize across different benchmarks. We further detail our findings below.

Impact of ReAct: ReAct significantly outperforms One-shot generation. ReAct's ability to iteratively revise code with reasoning and planning results in superior performance. Moreover, even without explicit feedback from the compiler (Simple), the intermediate reasoning steps similar to chain-of-thought can still bring considerable

improvements from 41.4% to 67.1%. When compared with One-shot without RAG, ReAct enhances syntax success rates by 25.7%, 26.4%, and 31.2% with iterative feedback from Simple, iverilog, and Quartus, respectively. We also observe consistent improvement from ReAct, regardless of the compiler and use of RAG.

Impact of RAG: Notably, the application of RAG with human expert guidance boosts the fix rate considerably and substantially enhances the solution's reliability. The results with Quartus compiler in Table 1 show that RAG improves the fix rate by 31.2% (58.7% to 89.9%) for One-shot and 18.6% (79.9% to 98.5%) when using ReAct. We observe consistent improvement with RAG, regardless of the quality of the compiler feedback message and LLM (GPT-4).

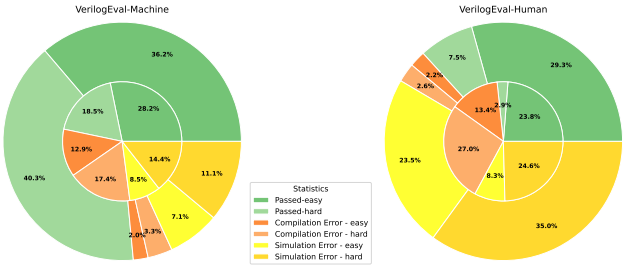

Figure 4: VerilogEval pass@1 results prior (inner) and post (outer) syntax error fixing with RTLFixer.

LLM	Syntax Success Rate	pass@1
GPT-3.5	73%	11%
GPT-3.5 + **RTLFixer**	93%	16%

Table 3: Improvements of Syntax Success rate and simulation Pass@1 on RTLLM benchmark using ReAct and RAG with Quartus compiler.

Simulation Correctness Improvement: Previous research [9, 11] evaluates the performance of LLM-generated Verilog code using the pass@k metric. However, this approach doesn't account for syntax errors in the code samples, which can skew accuracy. In our study, we evaluate functional correctness using the VerilogEval benchmark, specifically addressing fixes to syntax errors in the code samples. The results, displayed in Table 2, show the performance scores on the VerilogEval dataset and the improvements after rectifying syntax errors, with 32.3% and 10.1% improvement on the pass@1 metric for Machine and Human respectively. We further divided the VerilogEval benchmark into two subsets: *easy*, comprising 71 problems, and *hard*, consisting of 85 problems. These subsets have been delineated based on a pass rate threshold of 0.1 on Human. For simple problems in Human and low-level descriptions in the Machine, the correction of syntax errors significantly enhances the pass rate, reaching around 80% for pass@1. When contrasting the pass rate improvements between easy and hard problems in the Human descriptions, we observe a greater improvement for easy problems at 14.5% compared to hard problems at 6.7% for pass@1. This discrepancy suggests that LLMs still face challenges when advanced reasoning and problem-solving skills are required. Discussions on future work to address simulation errors are presented in Section 5.

Generalizability: Our method using ReAct and RAG can be generalized to other benchmark. To account for potential overfitting during the design of the retrieval database, we also tested our method on the RTLLM [11] benchmark without deriving new human guidance for the retrieval database. As shown in Table 3, our framework improves the syntax success rate from 73% to 93%, demonstrating its capability to generalize[1].

4.3 Ablation Studies

Our research includes two ablation studies designed to evaluate the impact of feedback quality and the selection of LLMs on the effectiveness of syntax error correction.

4.3.1 Impact of Feedback Quality. We study the impact of feedback quality with using different feedback messages detailed below.

Simple: We only give an instruct prompt "*Correct the syntax error in the code.*" without any explicit instruction on what the error is about and how to fix it.

Icarus Verilog (iverilog) [19]: Open-source Verilog simulator. The compiler occasionally encounters edge cases where it fails to provide informative logs, outputting messages such as "*I give up.*" Logs lack clarity, making them challenging to decipher.

Quartus[2]: Commercial compiler for FPGAs. In contrast with the open-source counterpart, it delivers well-defined and clear logs, effectively identifying errors and often offering suggestions and validation tips, making it more user-friendly and informative.

We deem Simple, iverilog, and Quartus to have increasing level of feedback message quality and illustrate the difference of the two compilers with an example in Figure 5. Results depicted in Table 1 clearly demonstrate that compiler logs give better feedback than Simple feedback and that the quality of compiler output impacts the performance of LLM debugging. As the quality of the compiler message improves (iverilog v.s. Quartus), the success rate of fixing syntax errors also increases. Intriguingly, the disparity between iverilog and Quartus results is more pronounced when using React with RAG. This discrepancy potentially suggests that high-quality compiler messages enhance the LLM's ability to more effectively utilize the retrieved human expert guidance.

4.3.2 Impact of Different LLMs. In Table 1, we present the results obtained when utilizing GPT-4 as the underlying LLM with Quartus as the compiler. A notable improvement in syntax error resolution is observed when using GPT-4 compared to GPT-3.5, particularly with One-shot prompting with RAG, where the success rate increased from 89.9% to 98%. When comparing the results of GPT-4 between One-shot and ReAct, we observe minor improvements of approximately 1%, suggesting that GPT-4 is already a robust agent proficient in fixing syntax errors without the need for reasoning, action planning, and iterative refinement.

Nonetheless, it is important to highlight that our approach of empowering LLMs with ReAct and RAG can significantly narrow the gap between weaker LLMs and stronger ones, especially beneficial for weaker open-source models [9, 17] that may not be as performant as GPT-4 on programming tasks.

[1]The syntax success rate we collected is different from the original paper because we used the Verilog code sample provided in the their repo for each problem.
[2]https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime/resource.html

Task ID: vector100r

Erroneous Implementation

```
module top_module (
input [99:0] in,
output reg [99:0] out
);
always @(posedge clk) begin
  for (int i = 0; i < 100; i = i +1) begin
    out[i] <= in[99 - i];
  end
end
endmodule
```

iverilog:
vector100r.sv:5: error: Unable to bind wire/reg/memory ‘clk’ in ‘top_module’
vector100r.sv:5: error: Failed to evaluate event expression ’posedge clk’.
2 error(s) during elaboration.

Quartus:
Error (10161): Verilog HDL error at vector100r.sv(5): object "clk" is not declared. Verify the object name is correct. If the name is correct, declare the object. File: /tmp/tmp4u6ib9ig/vector100r.sv Line: 5 Error: Quartus Prime Analysis & Synthesis was unsuccessful. 1 error, 1 warning

Figure 5: Example of compiler log from iverilog and Quartus. Quartus feedback messages are more informative.

5 ANALYSIS AND DISCUSSION

In this section, we delve into a series of analyses and discussions, extracting valuable insights from our discoveries. Specifically, we provide analysis in fail cases and the effect of iterative code refinement. Additionally, we discuss the challenges associated with applying our method to debugging simulation logic errors.

Failure due to LLM's Incapability: Most of the cases where, even with the aid of ReAct and RAG, failed to correct syntax errors is due to the fundamental incapability of the LLM. Figure 6 illustrates one of the failure case where it particularly requires arithmetic index calculations to solve the index out-of-range error. Some other notable failures occurred in cases where LLMs were confident in incorrect syntax, possibly due to it being accepted in C/C++.

Erroneous Implementation (Partial):

```
for (i = 0; i < 16; i = i + 1) begin : ROW
    for (j = 0; j < 16; j = j + 1) begin : COLUMN
        neighbors[0] = q[(i-1)*16 + (j-1)];
        row_above = q[((i-1) & 15)*16 + j];
        ...
```

Compile Error:
Error (10232): Verilog HDL error at conwaylife.sv(23): index -17 cannot fall outside the declared range [255:0] for vector "q"

Figure 6: An example which the agent failed to fix a syntax error. LLM failed to calculate array indices in the for loop and does not recognize the out-of-bound error.

Iterative Code Refinement: In Figure 7, we analyze the number of iterations ReAct requires to fix syntax errors. About 90% of problems are resolved in a single revision. For the remaining cases, additional code revisions are necessary, as new errors may surface after addressing the initial ones.

Challenges in Debugging Simulation Errors: While our framework could be readily adapted to employ LLMs for debugging simulation errors, our preliminary studies revealed limited improvements beyond syntax error fixes. Despite our efforts to provide simulation error logs as feedback to LLM agents, including summaries on output error count and text-formatted waveform-like comparisons of error versus solution output, we observed that LLMs

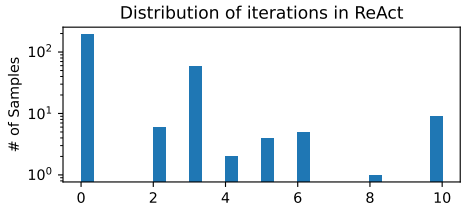

Figure 7: Distribution of iterations required by ReAct to fix syntax errors.

had constrained capabilities to comprehend simulation feedback messages. They only exhibited proficiency in fixing logic implementation errors for simple problems but struggled with more complex questions, especially those involving high-level design functionality descriptions and advanced reasoning. Addressing the challenges associated with LLMs fixing erroneous implementations in such problems, particularly those requiring advanced reasoning and problem-solving skills, remains an exciting area for future research. This also highlights the need to improve LLM's capabilities in reasoning and problem-solving related to hardware design.

6 CONCLUSION

Our framework **RTLFixer** demonstrates the significant impact of employing Retrieval Augmented Generation (RAG) and advanced prompting methods like ReAct in debugging Verilog code with Large Language Models. Key findings indicate that these approaches notably enhance syntax error resolution, achieving success rates as high as 98.5%. This research not only offers a novel autonomous language agent for Verilog code debugging but also introduces comprehensive dataset for further exploration.

REFERENCES

[1] Jason Blocklove, et al. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. *arXiv preprint arXiv:2305.13243* (2023).
[2] Mark Chen, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[3] Xinyun Chen, et al. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).
[4] Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. (2021). https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer
[5] Ziwei Ji, et al. 2023. Survey of Hallucination in Natural Language Generation. *Comput. Surveys* 55, 12 (March 2023), 1–38. https://doi.org/10.1145/3571730
[6] Shuyang Jiang, et al. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *arXiv preprint arXiv:2306.02907* (2023).
[7] Patrick Lewis, et al. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:cs.CL/2005.11401
[8] Mingjie Liu, et al. 2023. ChipNeMo: Domain-Adapted LLMs for Chip Design. arXiv:cs.CL/2311.00176
[9] Mingjie Liu, et al. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. *arXiv preprint arXiv:2309.07544* (2023).
[10] Shuai Lu, et al. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. arXiv:cs.SE/2203.07722
[11] Yao Lu, et al. 2023. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. *arXiv preprint arXiv:2308.05345* (2023).
[12] Suphakit Niwattanakul, et al. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.
[13] OpenAI. 2023. OpenAI models api. (2023). https://platform.openai.com/docs/models
[14] Hammond Pearce, et al. 2020. Dave: Deriving automatically verilog from english. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. 27–32.
[15] Yujia Qin, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789* (2023).
[16] Erich Schubert, et al. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.
[17] Shailja Thakur, et al. 2023. VeriGen: A Large Language Model for Verilog Code Generation. *arXiv preprint arXiv:2308.00708* (2023).
[18] Jason Wei, et al. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:cs.CL/2201.11903
[19] Stephen Williams et al. 2002. Icarus verilog: open-source verilog more than a year later. *Linux Journal* 2002, 99 (2002), 3.
[20] Shunyu Yao, et al. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*.
[21] Fengji Zhang, et al. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).