LLM-based Code Generation for Verilog

Anshul Raghav
Redmond High School
Redmond, Washington, USA
anshul.raghav.019@gmail.com

Mark Santolucito
Barnard College, Columbia University
New York City, New York, USA
msantolu@barnard.edu

Abstract

Recent advancements in Large Language Models (LLMs) have transformed code generation across various programming languages. However, there remains a significant gap in research regarding their application to hardware description languages, particularly Verilog. This gap is especially critical as hardware development becomes increasingly complex and automated assistance could substantially reduce development time and minimize errors in system design.

To address this challenge, we conducted a comprehensive ablation study comparing GPT-4o, GPT-4o mini, and GPT-3.5 Turbo for Verilog code generation. Our methodology involved evaluating four distinct approaches: (1) one-shot generation as a baseline, (2) multi-shot generation, (3) an error correction pipeline for handling compilation errors, and (4) a full framework that addresses both compilation and functional errors. Our results demonstrate that the full framework significantly outperforms baseline approaches across all tested models.

The results show that the full framework outperformed all baseline methods, eliminating compilation errors entirely and increasing the overall pass rate by up to 17.65%. In the ablation study, the multi-shot generation and error-handling components demonstrated incremental improvements of 1.96% and 5.88%, respectively, underscoring the contribution of each component to the framework's effectiveness.

Keywords

Verilog, Code Generation, LLMs, Ablation Study

ACM Reference Format:
Anshul Raghav and Mark Santolucito. 2025. LLM-based Code Generation for Verilog. In *Proceedings of International Journal of Secondary Computing and Applications Research (IJSCAR VOL. 2, ISSUE 1).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.5281/zenodo.14988731

1 Introduction

Verilog is a popular hardware description language (HDL) widely used in electronic design automation to model and simulate digital circuits. It allows designers to describe the structure and behavior of digital systems at a high level of abstraction. However, writing functionally correct Verilog code can be challenging. Verilog requires a deep understanding of hardware-specific constructs, which are difficult to understand without a deep domain knowledge. These complexities make human coding prone to subtle errors that can lead to malfunctioning hardware or performance inefficiencies. LLMs offer promising potential in assisting with Verilog code generation by automating parts of the development process. LLMs can quickly generate code based on natural language prompts, reducing development time and aiding engineers in addressing complex design specifications.

Despite the success of LLMs in generating code for a wide range of languages [2, 3, 8, 13], generating Verilog code remains a challenge [7, 15, 16]. Single-shot generation, where the model produces code from a single input prompt, often fails to yield functionally accurate and optimized code [6]. These limitations reveal a clear need for a specialized framework that iteratively refines and validates code outputs, ensuring they meet the strict accuracy and performance standards essential for hardware design. Thus, we focus our work on the ability of LLMs to use compliler feedback and test feedback to iteratively refine code.

To address these challenges, we conducted an ablation study comparing GPT-4o, GPT-4o Mini, and GPT-3.5 Turbo for Verilog code generation. We introduce a novel framework designed to systematically address both compilation and functional errors by identifying specific issues and utilizing the LLM to generate targeted corrections. Our approach incorporates multi-shot generation and iterative refinement techniques to progressively improve code quality.

This paper aims to evaluate the effectiveness of LLMs in generating syntactically and functionally correct Verilog code and proposes a framework to enhance code generation accuracy. In summary, the key contributions of this paper are:

- We provide a comprehensive evaluation of LLMs for Verilog code generation, including an ablation study and a full framework for improved accuracy.
- We identify specific challenges in generating functionally correct Verilog code and provide insights for future research in this area.
- We make our code and datasets publicly available to facilitate further research in this domain.

2 Related Work

The advent of LLMs has markedly transformed the landscape of automated code generation, significantly altering the methodologies and practices employed by developers in the software engineering domain. Among the most compelling applications of LLMs is their capacity to facilitate code generation, which holds tremendous upside and has garnered interest from both academia and industry alike [1–3, 5, 7, 8, 10, 13, 15–17, 19]. General-purpose tools, such as ChatGPT [10], Github Copilot [5], and Claude [1], have surged in popularity in recent years, largely attributable to their user-friendly interfaces and versatility across various programming tasks. According to a survey conducted by GitHub, 92% of developers report

This paper is published under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 International (CC-BY-NC-ND 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
IJSCAR VOL. 2, ISSUE 1
© 2025 The Author(s). Published by IJSCAR.

using AI-based coding tools either in their professional or personal projects, with 70% believing that such tools offer them an advantage in their work [14]. These findings underscore a broader trend of widespread adoption and positive reception of LLM-based code generation. This phenomenon has prompted extensive research aimed at evaluating the performance of domain-specific LLM-based code generation and identifying approaches to enhance their effectiveness.

While general-purpose tools can perform a broad range of tasks, they often struggle with generating code that requires specialized domain knowledge, particularly for less commonly used programming languages. As a result, in parallel with the development of general-purpose applications, there has been a notable increase in the development of research designed to evaluate the quality of code generated for specific programming languages and improve their performances. Research efforts have led to the development of diverse benchmarks as standardized tests to evaluate the accuracy and efficiency of code generation by LLMs across various tasks. Among these, HumanEval has emerged as the most widely recognized benchmark in code generation research [2]. Additionally, it is a common practice to fine-tune certain LLMs for specific programming languages [13]. This fine-tuning process involves training models on language-specific datasets, allowing them to better capture the language's syntax and structure.

To further improve the syntactical and functional accuracy of code generated by LLMs, researchers have introduced comprehensive frameworks. SynCode [17] exemplifies this approach, enabling LLMs to produce syntactically correct code in various formal languages by integrating user-defined context-free grammars. Tools such as RustAssistant [3] and RTLFixer [16] also contribute to this trend, leveraging techniques such as iterative refinement, retrieval-augmented generation (RAG) [4], and ReAct prompting [19] to address and improve upon initial code outputs. By systematically refining the generation process, these frameworks enhance the reliability and effectiveness of LLM-assisted coding.

Narrowing the focus further, there exists a burgeoning interest in the application of LLMs specifically for hardware description languages (HDLs), such as Verilog. The domain-specific knowledge needed for Verilog along with its unique syntax and semantics presents a unique challenge for LLMs, necessitating the development of targeted models that can reliably generate correct and functional Verilog code. Previous studies have proposed benchmarking frameworks to evaluate LLM-generated code [7], while also fine-tuning and testing older models [15], but there is a lack of recent and comprehensive research into tracking the efficacy of LLM-based generation of Verilog code and developing a framework to improve the correctness of generated code.

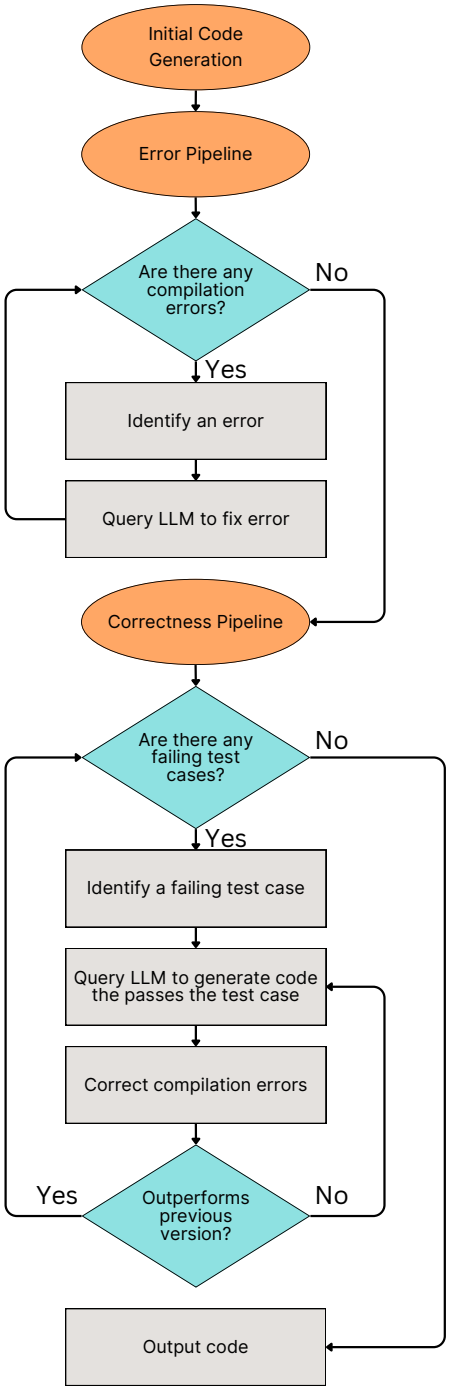


Figure 1: Code generation framework

3 System

3.1 Framework Overview

As shown in Fig 1, the framework is divided into three components that build off each other to generate functionally correct code at the highest possible rate while consuming a minimal amount of resources. At a high level, the pipeline is designed to generate initial code and continuously address specific issues, providing targeted fixes until either the code is fully functional or the maximum resource capacity has been reached. This framework leverages the OpenAI API to implement the targeted corrections, specifically utilizing the GPT-4o-2024-08-06, GPT-4o-mini-2024-07-18, and GPT-3.5-turbo-0125 models, the latest versions at the time of writing.

It is important for this tool to be able to be subjected to realistic constraints, the most important of which being a limited number of API queries. Queries incur both time and financial costs, so it is important to be able to control the maximum number of queries to the LLM while retaining optimal performance. By allowing the LLMs to build on their previous responses with feedback from the compiler or the test cases, we found that the model utilized the given resources effectively to improve upon the initially generated code.

Initial Generation: The goal of this module is to generate a starting point that we can build off of in the following components. We query the LLM with a general template to give it context into the actions it needs to take and the specified problem, and output this result directly to the error pipeline.

Error pipeline: The purpose of the error pipeline is to remove any compilation errors from the code while still maintaining the initial functionality. At a high level, this pipeline includes two main sections - error identification and error correction. The framework will initially identify any compilation errors in the code, advancing directly to the functional pipeline if there are no errors. Once a specific error is identified, it will transition to the error correction phase and query the LLM to provide a targeted fix. Then, this process will repeat from the beginning, with another issue being identified and corrected until the code compiles successfully, at which point it will advance to the correctness pipeline.

Correctness pipeline: The objective of the correctness pipeline is to produce functionally correct code, meaning for each input the code produces an output satisfying the specification. It builds on top of the error pipeline, taking in compilable code that may or may not be functionally correct. The pipeline begins by assessing whether any test cases are failing. If the code is functionally correct, it outputs the code as the final solution. In the event of a failure, the pipeline identifies the specific test case that is not passing, queries the LLM to generate a targeted adjustment aimed at resolving the identified issue, and finally subjects the code to the error pipeline to ensure that it is compilable. A prevalent issue observed was the LLM's tendency to inadvertently introduce regressions into the code while attempting to address an issue, causing the code to fail previously passed test cases and rendering it less effective than the prior iterations. In such a scenario, the code would be reverted back to the previous version and the LLM would be queried again in an attempt to resolve the issue. This iterative process of identifying failing test cases and applying corrections occurs until the code successfully passes all test cases. However, in some instances, the model exhibited repetitive behavior, consistently "hard-coding" the same response regardless of the number of queries. To mitigate this issue, we adjusted the temperature parameter of the LLM, increasing response variability and encouraging the generation of alternative solutions. Additionally, we implemented a fixed limit on the number of iterations the model could spend addressing a single issue. This constraint prevented the system from entering infinite loops, thereby optimizing computational efficiency and preventing unnecessary resource consumption.

3.2 Problem Simulation

To effectively showcase the functionality and structure of the framework, we begin by presenting a sample problem. This problem serves as a practical example through which the framework's core processes can be demonstrated. Through a systematic analysis, we will demonstrate the framework's capacity to address challenges in code generation, highlighting its ability to methodically refine the code to produce a fully functional solution.

As shown in Fig 2, the problem outlines a Verilog module for a signed adder that takes two 8-bit 2's complement numbers (a and b) as inputs, produces their sum (s), and detects if a signed overflow occurs during the addition (overflow). The key contribution of the signed adder is to account for the signs of the numbers while adding, with the most significant bit (MSB) representing the sign of the value. A signed overflow occurs when the sum of the numbers exceeds the range for the given number of bits, causing the number to wrap around and produce an incorrect result.

```
// This is a signed adder that adds two 8-bit 2's
     complement numbers. It also captures a signed
     overflow.
module signed_adder(input [7:0] a, input [7:0] b, output
     [7:0] s, output overflow );
```

Figure 2: Problem statement

Initial Generation: The code produced by the initial generation of the framework is illustrated in Fig 3. This code implements the majority of the details correctly, but contains minor compilation and functional errors. Errors during the initial generation phase are anticipated, as this stage serves as a foundational starting point upon which the error and correctness pipelines can build.

```
1  module signed_adder(input [7:0] a,
2      input [7:0] b,
3      output reg [7:0] s,
4      output wire overflow
5  );
6
7      assign s = a+b;
8      assign overflow = a[7]&&b[7]&&(~s[7]);
9
10 endmodule
```

```
.\compilation-error.v:7: error: reg s; cannot be driven
     by primitives or continuous assignment. 1 error(s)
     during elaboration.
```

Figure 3: Initial generation code

Error Pipeline: The initial step in the error pipeline involves verifying the presence of any compilation errors. As identified in Fig 3, the error arises from attempting to drive a variables with type reg (s) using a continuous assignment, which is prohibited in Verilog. To resolve this, the variable s must be declared as wire [20]. This specific error, along with the relevant code, is provided to the LLM, which is tasked with generating a targeted correction to address the issue. The results of this operation can be seen in Fig 4, along with its failing test case. The process subsequently

advances to the initial step once again, attempting to identify if there are any compilation errors. As the code is compilable, the process transitions to the correctness pipeline.

```
module signed_adder(input [7:0] a,
    input [7:0] b,
    output [7:0] s,
    output overflow
);

    assign s = a+b;
    assign overflow = a[7]&&b[7]&&(~s[7]);

endmodule
```

```
Case 3 failed
Input:
a=8'b01000000; b=8'b01000000;
#period;
Correct output: s=8'b10000000 overflow='b1
Given output: s=8'b10000000 overflow='b0
```

Figure 4: Error pipeline code

Correctness pipeline: The first steps in the correctness pipeline are to identify if there are any failing test cases, which is made clearly evident in Fig 4. The failing test case results from a functional error in the code, specifically its failure to detect overflow when both a and b are positive, but their signed sum evaluates to a negative value. This error occurs because overflow is only identified when a and b are positive and s is negative, but not under the inverse conditions. Upon identifying the failing test case, the LLM is queried to generate code that passes the specified test case, while any compilation errors are addressed using the previously covered error pipeline. The output of these operations is illustrated in Fig 5. However, the solution remains functionally incorrect due to a flaw in the overflow detection logic. Specifically, overflow is assigned a value of 1 when the sign of a differs from the sign of the sum (s). This approach is demonstrably incorrect, as evidenced by the failing test case provided. For instance, when a is -5 and b is 50, the sum is 45, and overflow is incorrectly set to 1 due to the mismatched signs of a and s. In this scenario, overflow should instead be 0, as no overflow occurs. This issue is addressed in the subsequent stage of the correctness pipeline, which involves comparing the updated solution with its previous iteration. The LLM is then queried once more to generate code that successfully passes the failing test case. The newly generated code, as shown in Fig 6, incorporates the revised logic: assigning an overflow value of 1 when both a and b are negative and the sum is positive. Upon evaluation, this version is found to outperform the previous iteration and pass all of the test cases, allowing the code to be designated as the final version.

As demonstrated, this process of iterative refinement proves to be effective. By employing a minimal number of queries, it was possible to generate fully functional code. This gradual, targeted approach ensures a more efficient use of resources and increases the likelihood of generating a correct and optimized solution.

3.3 Framework Implementation

This section offers a detailed examination of the framework's logic and structure. Designed as a command-line tool, the framework

```
module signed_adder(input [7:0] a,
    input [7:0] b,
    output [7:0] s,
    output overflow
);

    assign s = a + b;
    assign overflow = s[7] != a[7];

endmodule
```

```
Case 2 failed
Input:
a=8'b11111011; b=8'b00110010;
#period;
Correct output: s=8'b00101101 overflow='b0
Given output: s=8'b00101101 overflow='b1
```

Figure 5: Regressed solution code

```
module signed_adder(input [7:0] a,
    input [7:0] b,
    output [7:0] s,
    output overflow
);

    assign s = a+b;
    assign overflow = a[7]&&b[7]&&(~s[7]) || (~a[7])&&(~b
     [7])&&(s[7]);

endmodule
```

```
All cases passed
```

Figure 6: Full solution code

accepts input through a text file and generates output stored in a Verilog file. Several details are abstracted or slightly modified in this section to better explain the overall structure of the code, the most important being the underlying functions such as checkErrs(), checkCorrectness(), and passingCases(). Despite not being included in the pseudocode, these are covered in detail and available as separate files in the GitHub repository [12]. Furthermore, the prompts provided to the LLM to enhance its understanding of both and functional errors are embedded as variables, such as errMessage and correctnessMessage. While these variables are not explicitly visible during the model's decision-making process, they have been systematically fine-tuned on a diverse set of sample problems to maximize comprehensibility. This careful tuning ensures that the LLM receives clear and structured feedback, allowing it to develop a form of "chain of thought" reasoning [9]. Empirical observations indicate that as the clarity and specificity of these prompts increased, the model's overall performance in resolving errors exhibited a corresponding improvement.

Initial Generation: The queryLLM function utilizes the prompt and queries the chosen LLM a certain of times, returning the best result, i.e., the one with the least errors and highest number of test cases passed (line 38 of pseudocode in Fig 7). The number of LLM queries is determined by the user-defined parameter numShots, which can be adjusted based on design specifications. The input

prompt is constructed as startMessage + prompt, where startMessage provides contextual information and specifies the required output format, while prompt defines the specific problem to solve.

Error Pipeline: The purpose of the error pipeline is to remove all errors from the code while still maintaining the intended functionality (line 40). It takes in two parameters, the code and the problem statement and outputs a revised version of the code that is error free. The central concept of this pipeline is that the LLM can process feedback in the form of error messages and use it to correct the code, eliminating the identified errors. One particularly intriguing aspect of Verilog is that its compilation errors often lack clarity, presenting a significant challenge for human developers during the debugging process. This raises the question of whether LLMs would encounter similar difficulties when tasked with interpreting and resolving such errors.

The algorithm starts by checking the code for errors and storing those in a set (line 5). This approach ensures that duplicate errors are eliminated and facilitates rapid error lookups. In line 6, we establish a separate set for errors deemed irreparable. Failure to resolve an error may arise from multiple factors, such as a lack of domain specific knowledge, ambiguity in the error, or misunderstanding the intent. The consequence is that the LLM is unable to resolve the issue, resulting in a potential waste of valuable time and resources if the model is continuously queried without success. The consequence is that the LLM is unable to resolve the issue, resulting in a potential waste of time and resources if the model is continuously queried without success. Consequently, errors that the LLM cannot address are categorized into failedSet on line 17. Notably, these unresolved errors are often rectified by the subsequent resolution of other underlying errors that initially contributed to their emergence.

Lines 7-20 are where the body of the error-fixing process occurs. We continuously iterate through the loop as long as there are remaining errors in the code, selecting a specific error for resolution with each iteration. Given that modifications to the code can lead to the emergence of new errors during the correction of existing ones, we implement an abstraction termed an error group to effectively manage this complexity (line 8). As we proceed with the resolution process, any newly encountered errors that arise will be incorporated into the error group (line 15). This iterative process continues until the error group is completely emptied. As previously noted, there are instances in which the LLM is unable to rectify certain errors. We classify the error as unfixable if it persists despite revisions aimed at resolving the issue (line 16-19). Consequently, should we encounter this unresolvable error again during our examination of the error group, we will simply disregard it and move onto the next error (lines 12-13).

We implement several heuristics to ensure that the code remains within a specified query limit. For clarity, these heuristics are not detailed in the pseudocode. We impose a maximum limit for the number of times the outer loop can run (line 7) and allow the user to define a fixed number of iterations for the inner loop (line 10). Additionally, when the LLM is queried, it executes a user-defined number of queries, represented by numShots, and selects the optimal response from among them (line 14).

Correctness Pipeline: This pipeline accepts error-free code along with the problem statement and produces functionally correct code that successfully passes all test cases (line 42). Prior to initiating the pipeline's core processes, three key concepts are incorporated. First, the pipeline maintains a record of all previously passed test cases, enabling verification that the current code performs at least as well as prior iterations (line 24). Since the LLM occasionally generates revisions that inadvertently reduce code performance, causing earlier test cases to fail, this measure serves as a safeguard against regression. Second, we incorporate a version control mechanism, adapted from the error pipeline (line 26). This allows us to revert the code to its prior state if earlier test cases begin to fail. Third, we track the current failing test case, which is later provided to the LLM as feedback, guiding it in refining the code to pass additional test cases (line 25).

The main processing occurs within the while loop, where we iterate through the code body until all test cases are successfully passed, as indicated by the checkCorrectness() function returning 'all cases passed' (line 27). The information from the failing test cases is used to guide the LLM in generating an improved version of the code that addresses the specific failure. We provide the LLM with a structured prompt that includes contextual information on deriving and formatting the solution, as well as the current code, the failing test case, and the original problem statement (line 28). Including the problem statement is crucial, as it helps the LLM remain focused on the intended solution; without it, we found that the LLM tends to 'hardcode' responses, inadvertently causing other test cases to fail. Next, we apply the error pipeline to the newly generated code to ensure it is free from errors (line 29). Following this, we assess whether the code has regressed: if it fails a test case it previously passed, we employ version control to revert to a more functional version (lines 30-32). If no regression is detected, we can confirm that the current code represents the best iteration to date. Consequently, we save this version and update the set of solved test cases (lines 33-35). Finally we return the completed code at the conclusion of the process (line 36).

We employ heuristics similar to those in the error pipeline to constrain the number of queries sent by the code. The primary heuristic is a fixed limit on the number of while loop iterations (line 27), which is set to a constant value configurable by the user.

4 Evaluation

4.1 Results

4.1.1 Experimental Design. To evaluate the effectiveness of our framework for generating Verilog code, we selected a series of problem sets and benchmarks from [15]. These benchmarks offer a thorough evaluation of code generation accuracy, and the problem set is well-suited for examining the performance of LLMs in hardware design tasks.

The problem set in VGen is systematically categorized into three levels of difficulty: basic, intermediate, and advanced, ensuring a progressive evaluation of the LLM's Verilog code generation capabilities. The dataset consists of 17 distinct problems, each designed to cover a broad range of digital design concepts, including combinational and sequential logic, arithmetic operations, and finite state machines (FSMs). This diverse selection ensures a comprehensive

```
Input: Problem description
Output: Code that has no compilation errors and behaves according to the specifications of the problem description

def errPipeline(code, problem):
    errs = checkErrs(code)
    failedErrs = {}
    while errs != {}:
        errGroup = {errs.pop()}
        snap = code
        while errGroup != {}:
            currentErr = errGroup.pop()
            if currentErr not in checkErrs(code):
                continue
            code = queryLLM(errMessage+currentErr+code+problem, numShots)
            errGroup = checkErrs(code)-errSet
            if currentErr in errGroup:
                failedErrs.append(currentError)
                errGroup.remove(currentErr)
                code = snap
        errSet = checkErrs(code)-failedErrs
    return code

def correctnessPipeline(code, problem):
    solvedCases = {}
    failingCase = checkCorrectness(code)
    snap = code
    while failingCase != "all cases passed":
        code = queryLLM(correctnessMessage+code+failingCase+problem, numShots)
        code = errPipeline(code)
        failingCase = checkCorrectness(code)
        if failingCase in solvedCases:
            code = snap
        else:
            snap = code
            solvedCases = passingCases(code)
    return code

code = queryLLM(startMessage+prompt, numShots)

code = errPipeline(code, problem)

code = correctnessPipeline(code)
```

Figure 7: Pipeline pseudocode

assessment of the LLM's ability to generate Verilog across various complexity levels.

Each problem is further subdivided into three distinct prompts, all addressing the same core problem but differing in the level of explicit guidance provided. The prompts range from highly specific instructions, which closely outline the implementation strategy, to open-ended problem descriptions that require the model to independently determine the appropriate approach. This structured variation enables an analysis of how prompt specificity impacts the LLM's performance and reasoning ability in Verilog code generation.

To ensure objective evaluation, each problem is accompanied by a dedicated testbench that verifies the functional correctness of the generated Verilog code. These testbenches consist of multiple test cases, systematically designed to evaluate different edge cases and functional requirements within each problem. The testbenches serve as a benchmark for correctness, allowing for a clear and standardized assessment of the generated code's validity. However, in our methodology, the testbenches are deliberately excluded from assisting the LLM in generating functionally correct code.

A contribution of this work is the introduction of individual test cases for each problem, aimed at enhancing the rate of functionally correct code generation. In real-world applications, users interacting with an interface often lack access to predefined testbenches to verify functional correctness. To simulate this scenario, the LLM was not granted access to the testbenches during the generation process. However, our findings revealed that without test cases available for the LLM to be able to identify issues and provide targeted fixes, the LLM produced functionally incorrect code at an alarmingly high rate. To address this challenge, we developed a program that enables users to create their own test cases efficiently. The program allows users to specify parameters such as the module name, input and output names and values, the number of clock periods, and the period length. Based on these inputs, the program generates a fully functional testbench. This user-generated testbench can then be utilized by the LLM to verify the correctness of its code and iteratively refine it if errors are detected. By streamlining the creation of test cases, this approach addresses the critical need for functional verification while ensuring ease of use for users.

The framework was evaluated on the problem set using three distinct OpenAI models: GPT-4o, GPT-4o mini, and GPT-3.5 Turbo. GPT-4o is designed for complex, multi-step tasks, and is the most resource-intensive out of the three models. GPT-4o mini is a more lightweight and affordable version of GPT-4o, and is better for fast, lightweight tasks. Finally, GPT-3.5 Turbo is a fast, inexpensive model for simple tasks.

4.1.2 Ablation Study. The purpose of this ablation study is to isolate and assess the impact of each component in the code generation framework on the quality and reliability of LLM-generated Verilog code. This approach allows us to understand the role of each component in improving code quality and to identify which elements are most essential for generating functionally correct Verilog code.

To evaluate the impact of each component in our code generation framework, we define a baseline approach and consistent evaluation metrics. The baseline for this ablation study is the pass@1 metric, which we will refer to in this paper as the one-shot generation method. In this, the models respond to each problem with a single prompt, without any error handling or correctness validation. This baseline serves as a benchmark, allowing us to clearly see the relative impact of the other components.

For evaluating each component's contribution to performance, we rely on two key metrics: compilation success rate and functional correctness rate. Compilation success rate measures whether the generated Verilog code is syntactically correct and free of compilation errors. Functional correctness rate, on the other hand, assesses whether the generated code performs as expected in terms of functionality by passing the predefined testbenches. Each subsequent component - multi-shot prompting, error correction, and functional correction—is evaluated relative to this baseline, allowing us to observe how each addition improves or impacts the baseline performance.

The framework comprises of three main components: initial generation, the error correction pipeline, and the correctness pipeline. Each component builds upon the previous one to iteratively enhance the model's output. In the following, we describe each component and its role in generating high-quality Verilog code. Additionally, single-shot generation is included as a baseline for comparison. The initial generation is referred to as multi-shot generation to more precisely capture its function.

- **One-Shot Generation (Baseline):** This approach serves as our baseline and involves presenting each problem to the model once, generating a single response without any follow-up. With no additional context, examples, or error handling, this method captures the model's performance in generating initial Verilog code.
- **Multi-Shot Generation:** This component improves upon the baseline by querying the LLM multiple times and selecting the best result—defined as the output with the fewest compilation errors and the highest number of passed cases. From the perspective of the code generation diagram presented in the System section 1, this corresponds solely to the initial generation component.
- **Error Pipeline:** This pipeline extends the previous approach by iteratively refining its output to eliminate errors, as detailed in the system section. Notably, it does not address functional errors. This encompasses both the initial generation and error pipeline components found in the code generation diagram.
- **Full Framework:** Building upon the error pipeline, the full framework integrates all components discussed in the system section. Its primary advancement lies in its capability to identify and resolve functional errors effectively. This pipeline encompasses all components depicted in the code generation diagram.

With these components defined, we now turn to the results of our ablation study. The following figures provide a comprehensive overview of the framework's overall performance, illustrating how each component influences the models' ability to generate functionally correct Verilog code, as well as comparing the relative success rates of each model.

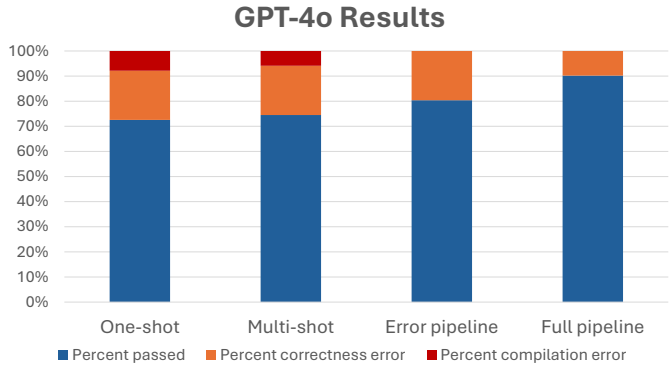


Figure 8: GPT4o Success Rates Graph

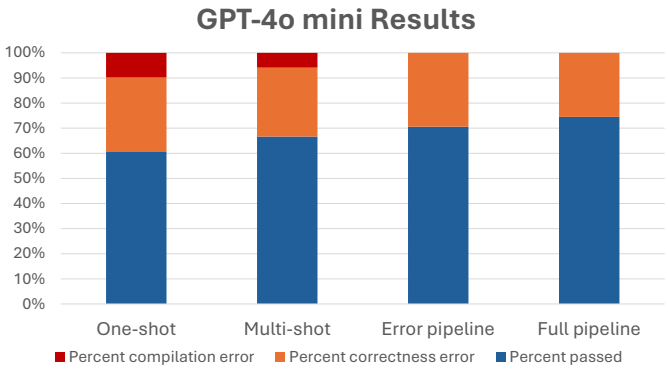


Figure 9: GPT-4o mini Success Rates Graph

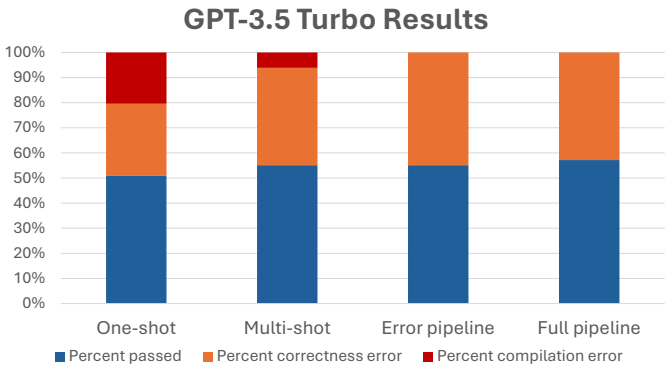


Figure 10: GPT-3.5 Turbo Success Rates Graph

4.2 Discussion

To fully assess the impact and effectiveness of the framework developed in this study, it is crucial to analyze the key findings and their broader implications. The discussion will highlight key takeaways and insights derived from the results, offering a more thorough understanding of the effectiveness of the framework, its components, and the various LLMs utilized.

To assess whether the framework provided a meaningful improvement over single-shot generation, it is necessary to evaluate the difference in the rate of fully functional code generated between the two approaches. At a high level, the improvements observed between the initial code generation and the final code output varied considerably across the different LLM models, yet remained consistently substantial. GPT-4o exhibited the most pronounced improvement, with a 17.65% increase, underscoring its advanced reasoning capabilities and its ability to make precise adjustments to the generated code. The framework also showed a significant improvement when paired with GPT-4o mini, which achieved a 13.73% improvement in success rates from the initial to final outputs. In contrast, GPT-3.5 Turbo exhibited a comparatively poor performance of 6.12%, reflecting its more limited capacity for reasoning and error correction. Overall, the framework proved highly effective, as demonstrated by the substantial improvements across

	One-shot	Multi-shot	Error pipeline	Full pipeline
Percent passed	72.55%	74.51%	80.39%	90.20%
Percent correctness error	19.61%	19.61%	19.61%	9.80%
Percent compilation error	7.84%	5.88%	0%	0%

Table 1: GPT-4o Success Rates Table

	One-shot	Multi-shot	Error pipeline	Full pipeline
Percent passed	60.78%	66.67%	70.59%	74.51%
Percent correctness error	29.41%	27.45%	29.41%	25.49%
Percent compilation error	9.80%	5.88%	0%	0%

Table 2: GPT-4o mini Success Rates Table

	One-shot	Multi-shot	Error pipeline	Full pipeline
Percent passed	51.02%	55.10%	55.10%	57.14%
Percent correctness error	28.57%	38.78%	44.90%	42.86%
Percent compilation error	20.41%	6.12%	0%	0%

Table 3: GPT-3.5 Turbo Success Rates Table

all models. Notably, when paired with more advanced LLMs such as GPT-4o, the framework achieved particularly significant gains.

It is also important to consider the role each component of the framework played in the final improvement. Initially, one-shot generation yielded mixed results, with a significant proportion of cases encountering either compilation or functional errors. This pattern was consistent across all models, although the frequency of these errors diminished as the models' complexity increased, again underscoring the significant influence that model sophistication has on error reduction.

Multi-shot generation produced results similar to those of single-shot generation, but with a notable decrease in both compilation and functional errors. This would have ensured a more structured foundation for the framework, guiding it toward more accurate and reliable solutions and minimizing the risk of diverging into ineffective or irrelevant paths. An interesting observation was the behavior of GPT-3.5 Turbo, which exhibited a unique pattern where the number of compilation errors decreased significantly, only to be replaced by functional errors. This shift suggests that the inherent limitations of GPT-3.5 Turbo in reasoning and functional comprehension may hinder its ability to effectively address functional errors, which demand more advanced reasoning capabilities.

Next, the error pipeline proved highly effective, successfully eliminating all compilation errors across all three models, highlighting its crucial role within the framework. The reduction in compilation errors amounted to approximately 5-6%, with the majority of these errors subsequently transitioning into functional errors. However, in some instances, the generated code passed all test cases and the functional errors were entirely resolved, achieving full functionality. This further underscores the importance of the error pipeline in improving the overall performance and reliability of the generated code.

The correctness pipeline also played a significant role in the overall final improvement. However, the improvement was more varied, at about 2-10% depending on the complexity of the model. As noted earlier, more advanced models, such as GPT-4o, exhibited greater accuracy improvements in comparison to simpler models such as GPT-3.5 Turbo. Our findings indicate that these issues were inherent with GPT-3.5 Turbo, as regardless of the number of queries, GPT-3.5 Turbo consistently struggled to provide effective fixes for the more complex problems within the dataset. However, the correctness pipeline remained a crucial component of the overall framework, making a significant contribution to the observed improvements.

A critical analysis of the performance of various LLM models is essential for understanding the nuances of their effectiveness within the framework. By systematically comparing the results from GPT-3.5 Turbo, GPT-4o, and GPT-4o mini, it is possible to assess the relative strengths and limitations of each model in the context of Verilog code generation. All three models were successful in eliminating compilation errors, demonstrating their ability to generate syntactically correct Verilog code. However, there was significant variation in their performance with respect to functional errors and their capacity for improvement over time. GPT-4o emerged as the most effective model, demonstrating the highest success rate in generating functionally correct code, as well as the greatest improvement in performance. This highlights its superior ability to not only produce accurate code but also to leverage reasoning to refine and enhance the code. In contrast, GPT-3.5 Turbo demonstrated the poorest performance, struggling to achieve consistently accurate results or improve upon the initial code. GPT-4o mini, while less powerful than GPT-4o, struck a balance by offering a competitive performance with a significantly lower computational cost. This makes GPT-4o mini an appealing option for scenarios where resource efficiency is a priority without severely compromising on accuracy.

Success rate was used as the evaluation metric rather than pass@k because the framework allowed for automatic requerying, making

it difficult to identify the top k results. Unlike traditional retrieval tasks, where a fixed set of top results can be ranked and evaluated, this framework operates iteratively, with each requery refining the solution to a compilation or functional error. As a result, there is no clear "top k" set of results; instead, the model continually evolves its responses, providing a series of progressively improved fixes until the issue is resolved. The pass@k metric, which relies on ranking a set of results at a given moment, becomes unsuitable for this dynamic process where the optimal solution is reached through successive refinements. Therefore, success rate provides a more accurate measure of the system's performance, focusing on whether the framework ultimately succeeds in fixing the identified errors rather than how well it ranks intermediate fixes.

While the results of this study demonstrate promising improvements in LLM-assisted Verilog generation, it is important to acknowledge the limitations of the research. Understanding these constraints provides valuable context for interpreting the findings and identifying areas for future development. One key limitation of this study is the relatively narrow range of LLMs used, which may have constrained the understanding of model performance. Expanding the selection to include additional models, such as OpenAI's o1, could offer a more comprehensive view, especially given its demonstrated success on human exams and ML benchmarks [11]. Additionally, incorporating state-of-the-art models such as Claude 3 Opus, GitHub Copilot, and Grok would enable a more comprehensive comparison, providing deeper insights into their reasoning capabilities and effectiveness in Verilog code generation. This study could also have been extended to include a broader range of programming languages, particularly HDLs closely related to Verilog, such as SystemVerilog and VHDL. Given SystemVerilog's similarities to Verilog, it would likely exhibit comparable behavior. However, its stricter type enforcement and explicit variable declarations could enhance an LLM's ability to debug code by providing more detailed and structured error messages. In contrast, VHDL, which originates from Ada—a less commonly utilized programming language—may pose greater challenges for an LLM due to its limited representation in training data. However, its strong typing system could enhance the debugging process by imposing strict constraints, thereby minimizing ambiguities in error detection and correction. LLMs equipped with advanced chain-of-thought reasoning capabilities are likely to achieve greater accuracy in solving problems in VHDL, as they can leverage knowledge from analogous languages to address novel challenges effectively. Furthermore, incorporating techniques such as ReAct prompting [19], Retrieval-Augmented Generation (RAG) [4], or skeletonization [18] could have enhanced the framework's ability to generate functionally correct Verilog code. These factors suggest that while the framework showed promise, there are several opportunities for further refinement and enhancement.

5 Conclusions

This paper presented an innovative approach to automated hardware design verification through the development of a framework that leverages LLMs to detect and correct both compilation and logical errors in Verilog code. By addressing the critical challenge of hardware design validation, our work demonstrates significant potential for reducing development time and improving code reliability in hardware description languages.

Our primary contribution is the development of an end-to-end framework that successfully identifies and resolves Verilog errors, achieving promising results across a diverse set of test cases. The system not only corrects syntactical errors but also demonstrates capability in addressing functional errors, a traditionally challenging aspect of hardware design verification.

Looking ahead, this research opens several promising avenues for future development. The testing dataset could be expanded to encompass more complex Verilog constructs and edge cases, thereby improving the robustness of the error correction system. In particular, we could explore other benchmark sets of Verilog code to test the system's capabilities, such as VerilogHumanEval [7]. Additionally, exploring the integration of specialized LLMs or fine-tuning existing models could enhance performance specifically for hardware description languages. Another potential enhancement would be incorporating additional debugging tools beyond compiler messages, such as ModelSim, Verdi, and GTKWave. This approach would be particularly novel, as HDLs are unique in that their outputs can be represented visually through waveforms and timing diagrams. Exploring the ability LLMs to analyze and debug HDL designs based on visual representations could provide valuable insights and open new avenues for AI-assisted hardware verification.

The successful implementation of this automated error correction system represents a significant step forward in hardware design verification, potentially transforming how developers approach HDL debugging and validation. As hardware designs become increasingly complex, such specialized tools will be essential for ensuring development efficiency and maintaining high standards of code quality.

References

[1] Anthropic. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[3] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing Rust Compilation Errors using LLMs. arXiv:2308.05177 [cs.SE] https://arxiv.org/abs/2308.05177

[4] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 [cs.CL] https://arxiv.org/abs/2312.10997

[5] Github. 2024. The World's most widely adopted AI developer tool. https://github.com/features/copilot

[6] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2024. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.* (Nov. 2024). https://doi.org/10.1145/3703155 Just Accepted.

[7] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. arXiv:2309.07544 [cs.LG] https://arxiv.org/abs/2309.07544
[8] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE] https://arxiv.org/abs/2402.19173
[9] OpenAI. [n. d.]. Prompt engineering. https://platform.openai.com/docs/guides/prompt-engineering
[10] "OpenAI". 2022. Introducing ChatGPT. https://openai.com/index/chatgpt
[11] OpenAI. 2024. Survey reveals AI's impact on the developer experience. https://openai.com/index/learning-to-reason-with-llms/
[12] Anshul Raghav. 2024. LLM-based Code Generation for Verilog. https://github.com/anraghav/Personal-Project
[13] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950
[14] Inbal Shani and GitHub Staff. 2023. Survey reveals AI's impact on the developer experience. https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/
[15] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. arXiv:2212.11140 [cs.PL] https://arxiv.org/abs/2212.11140
[16] Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. 2024. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models. arXiv:2311.16543 [cs.AR] https://arxiv.org/abs/2311.16543
[17] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. SynCode: LLM Generation with Grammar Augmentation. arXiv:2403.01632 [cs.LG] https://arxiv.org/abs/2403.01632
[18] Xingbo Wu, Nathanaël Cheriere, Cheng Zhang, and Dushyanth Narayanan. 2023. RustGen: An Augmentation Approach for Generating Compilable Rust Code with Large Language Models. https://openreview.net/pdf?id=y9A0vJ5vuM
[19] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629
[20] Jason Yu. 2017. Verilog reg, Verilog wire, SystemVerilog logic. What's the difference? https://www.verilogpro.com/verilog-reg-verilog-wire-systemverilog-logic/#:~:text=Verilog%20rule%20of%20thumb%201,to%20represent%20a%20physical%20connection.

Received 13 January 2025; accepted 18 March 2025