2025-01-13

Investigating Automatic Bug Repair Using Large Language Models for Digital Hardware Design

Elnaggar, Abdelrahman

UNIVERSITY OF CALGARY

Investigating Automatic Bug Repair Using Large Language Models for Digital Hardware Design

by

Abdelrahman Elnaggar

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING

CALGARY, ALBERTA

JANUARY, 2025

Abstract

Register-transfer level (RTL) bugs present critical challenges, impacting the functional correctness, security, and performance of System-on-Chip (SoC) designs. Detecting and repairing RTL bugs is traditionally a time-consuming process requiring skilled engineers, which significantly prolongs SoC development cycles and reduces vendor competitiveness. Given this complexity, there is a strong need for automated repair solutions capable of efficiently addressing RTL bugs to accelerate development timelines. In this thesis, we propose an automated framework leveraging a large language model (LLM) to repair RTL functional bugs. We explore various prompting techniques, including zero-shot, few-shot, and feedback approaches. Zero-shot relies solely on the LLM's pretrained knowledge, few-shot provides specific examples of RTL bug repairs, and feedback iteratively refines the LLM's responses using outputs from prior iterations. Additionally, we investigate six prompting strategies, each incorporating varying levels of context to guide the LLM in the repair process. Our proposed framework operates on benchmarks without requiring prior knowledge of the bug's type, location, or specific repair steps, better reflecting real-world scenarios than previous approaches. Results demonstrate the potential of LLM-driven automation, with the feedback approach achieving the highest repair success rate by fixing 26 out of 32 benchmarks (81.25%), while the best zero-shot and few-shot strategies repaired 23 out of 32 benchmarks (71.88%). These findings highlight the ability of current LLMs to consistently address RTL functional bugs, offering significant promise for streamlining SoC development by reducing the time and effort required for RTL bug detection and repair.

Preface

Material from Chapters 3 and 4 of this thesis is part of a manuscript under peer review at the time of writing, authored by A. Elnaggar and B. Tan, titled "Adding Context to LLM-Guided Verilog Repair," for the 26th International Symposium on Quality Electronic Design (ISQED'25). The rest of this thesis is original, unpublished, independent work by the author, Abdelrahman Elnaggar.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Benjamin Tan, for his invaluable guidance and support throughout my master's studies over the past two years. His mentorship has been instrumental in helping me broaden my knowledge and develop valuable insights in my field of study.

I am deeply thankful to my family and friends for their unwavering support and encouragement throughout this journey. I also extend my appreciation to CMC Microsystems for providing the EDA tools essential for my thesis work, as well as to Alberta Innovates for their support.

Lastly, I am grateful to the University of Calgary for offering the financial assistance and facilities that made this research possible.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

SoC System-on-Chip

RTL register-transfer level

CWE Common Weakness Enumeration

LLM large language model

HDL hardware description language

AST abstract syntax tree

FC Fix Correctness

ASIC Application-Specific Integrated Circuit

Chapter 1

Introduction

In the fast-paced realm of chip manufacturing, companies are under constant pressure to accelerate their product development cycles in order to maintain competitiveness in the market. This race to bring innovative products to market demands high levels of functional correctness, security, and performance, all of which must be ensured in hardware systems such as System-on-Chip (SoC) designs. As hardware design complexity increases, bugs can inadvertently be introduced, significantly impacting both the functionality and reliability of the final product [2]. Detecting and fixing these bugs is a crucial part of the hardware design process. However, the inherent complexity of modern hardware designs often makes this a difficult and time-consuming task, thus motivating the search for automated bug repair solutions [3, 1, 4]. Such solutions are especially useful in earlier stages of code development, such as RTL design, where changes to code in hardware description languages (HDLs) are more affordable compared to the repercussions of addressing hardware issues post-production.

The design of hardware systems is a meticulous, iterative process. As shown in Fig. 1.1, it begins with defining clear requirements that specify the functionality, performance, and constraints that the hardware must meet. The next step is to translate these requirements into a RTL design. RTL serves as a high-level abstraction that describes how data moves between registers and how logical operations are performed, providing a functional blueprint that will ultimately be transformed into a physical chip.

The RTL design is written using hardware description languages (HDLs), with Verilog being

among the most widely used. Recently, there has been a trend to generate RTL code using generative AI models [5] [6], which assist in automating parts of the RTL design process and proposing optimizations. Once the initial RTL code is developed, it undergoes a rigorous verification process through simulations and testbenches to ensure functional correctness. This stage often involves unit tests, integration tests, and sometimes formal verification to catch edge cases, significantly reducing bugs and errors early in the design flow.

Following verification, the RTL code moves to the synthesis stage, where it is converted into a gate-level netlist. Optimization is frequently applied here to improve speed, area, and power efficiency, tailoring the design for target performance metrics. The gate-level netlist then proceeds through place-and-route, where it is mapped onto the chip's physical layout [7]. Once layout and timing constraints are verified, the design is fabricated as an Application-Specific Integrated Circuit (ASIC).

RTL workflows are becoming more productive and innovative with generative AI, although challenges remain, particularly in ensuring the correctness and efficiency of AI-generated code in complex designs. Nevertheless, this evolving trend promises a future where RTL design may be increasingly automated and optimized, enhancing hardware design workflows.

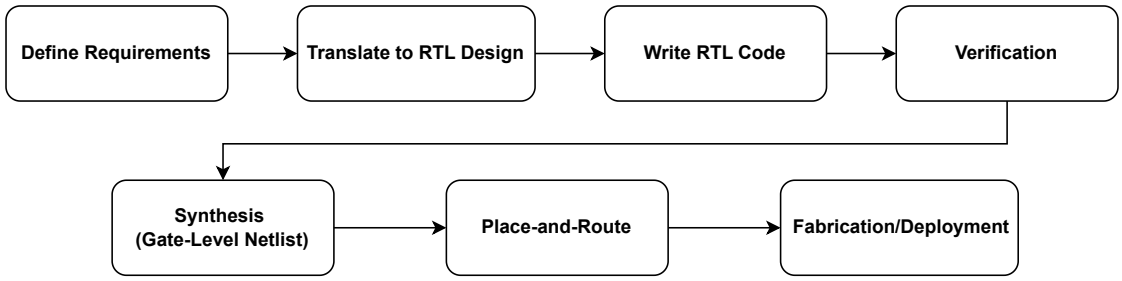

Figure 1.1: Hardware Design Workflow.

1.1 Motivation

The motivation for this work stems from the growing complexity of hardware systems and the need for more efficient ways to ensure correctness early in the design process. Traditional methods for bug detection and repair in hardware are labor-intensive, often requiring expert engineers to manually diagnose and fix issues in the RTL code.

In prior work, Ahmad et al. [1] demonstrated the use of genetic algorithms for hardware bug repair. However, despite their often time-consuming nature, genetic algorithms can sometimes fail to find a solution and struggle to scale with complexity.

LLMs are transformer-based neural networks that have revolutionized the ability of neural networks to understand the context and relationships in text. They have shown powerful capabilities in software and hardware code generation [8] [9]. Thus, there has been growing interest in utilizing LLMs for software bug repairs [10] [11] and hardware bug repairs [4]. The work in [4] explored using LLMs for repairing RTL code, specifically addressing functional bugs in a single-shot approach. Their results suggest that LLMs, which excel in understanding and generating language, could expedite the repair of RTL functional bugs, ultimately enhancing designer efficiency.

However, how best to use LLMs in hardware repair remains an open question, particularly as designers must perform some level of "prompt engineering" to make the most of their capabilities. In Ahmad et al.'s approach [4], bug localization was generally assumed, and their study focused primarily on eliciting repaired HDL code. However, if an LLM were to be integrated into an end-to-end hardware repair system, could additional context (such as simulation outputs) be useful for repairing RTL functional bugs? If so, how should it be incorporated into a prompt?

In this thesis, we seek answers to these questions by proposing and investigating different techniques and strategies for incorporating context to provide guidance to the LLM for repairing RTL functional bugs. We integrate an LLM into the CirFix framework [1] and evaluate the repair success.

1.2 Research Questions

The fundamental question guiding this thesis is: **How can large language models (LLMs) be best utilized for automatic hardware repair?** This question stems from the **hypothesis that LLMs can effectively assist in repairing Verilog bugs by leveraging their ability to understand syntax, semantics, and contextual patterns in hardware description languages.** To explore this overarching question and validate the hypothesis, the research focuses on addressing the following sub-questions:

RQ1: How effective are LLMs at performing automatic hardware repairs without prior examples (zero-shot)?

This question investigates the capacity of LLMs to identify and correct hardware code issues independently, assessing their zero-shot performance when provided with different types of information derived from the verification process output.

RQ2: How does providing LLMs with example repairs and additional information influence their ability to perform automatic hardware repairs (few-shot)?

This question explores how including examples and extra guidance impacts the accuracy and effectiveness of LLMs in generating correct repairs, focusing on the few-shot approach.

RQ3: How well do LLMs perform when feedback and multiple queries are involved?

This question assesses the effectiveness of using iterative queries and feedback mechanisms to refine LLMs-generated repairs.

Existing literature suggests that LLMs have potential for hardware repair tasks [4]; however, the optimal way to utilize them remains unclear. This thesis aims to fill that gap by identifying and testing different kinds of contextual information and instructions to determine which combination yields the best repair outcomes.

1.3 Contributions

Our contributions are as follows:

- Development of an automated, end-to-end LLM-based RTL repair framework that extends prior work [1] by integrating LLMs with simulation tools for enhanced bug repair.
- Exploration of multiple prompt strategies and variations for incorporating *context* from simulation results with instrumented testbenches.
- A systematic evaluation of leading commercially-available LLMs to investigate the impact of incorporating different types of context in prompts for repairing RTL code.

- Detailed studies on the use of:
  - **Zero-shot and few-shot techniques**: Examining their effectiveness in automating RTL bug repair.
  - **Feedback from simulation results**: Analyzing how integrating iterative feedback affects the performance of automatic RTL fixes.
- Open-sourcing the code for our automated LLM-based RTL repair framework, made available on GitHub [12].

1.4 Thesis Layout

The remainder of this thesis is structured as follows. Chapter 2 provides the background of the hardware design workflow, including an overview of RTL (Register Transfer Level) and Verilog. It offers insights into the processes of bug detection, localization, and fixing within hardware systems. The chapter also covers essential aspects of testing and verification, and introduces LLMs and prompt engineering techniques, which serve as the foundation for the automated bug repair framework explored in this thesis. The second part of the chapter reviews prior work in the field of automatic program repair. It highlights the recent success of LLMs in tasks such as program generation and bug fixing, demonstrating how LLMs are increasingly being adopted for bug fixing in both software and hardware domains.

Chapter 3 outlines the experimental methodology, detailing the framework developed for this research, the metrics used in the experiments, and the experimental setup. It describes how the framework was implemented and the various techniques used to evaluate the effectiveness of LLM-based bug repair.

Chapters 4, 5, and 6 present the experimental results for different prompting techniques: zero-shot, few-shot, and feedback approaches. In these chapters, we explore how incorporating different types of information available from simulations affects the performance of the bug fixes. This includes using the difference between the simulation output and an oracle (which, in this setting, is a bug-free reference design used to verify the correctness of a system under test) and fault localization information. Additionally, we investigate the effect of using structured versus unstructured prompts.

Chapter 4 focuses on experiments and results related to the zero-shot technique, where the LLM is provided with no examples and relies solely on its general knowledge. Chapter 5 examines the few-shot technique, where the LLM is provided with a limited number of examples to aid in understanding and repairing bugs. Chapter 6 investigates the feedback technique, where the LLM iteratively receives feedback to refine its bug fixes. Each chapter discusses the experimental setup, results, and analysis for the respective technique while considering how these additional simulation-based details and prompt structures influence the outcomes.

Chapter 7 discusses the findings from Chapters 4, 5, and 6, providing an overview of the results and drawing conclusions about the effectiveness of the various prompting techniques. It also examines the implications of the results and their potential impact on automatic bug repair in hardware domains. Additionally, it includes a comparison with prior studies that utilized the same benchmark, highlighting similarities, differences, and improvements.

Finally, Chapter 8 concludes the thesis, summarizing the key findings and contributions of the research. It also identifies areas for future work, suggesting potential avenues for further exploration and improvement in the field of automatic bug repair using LLMs.

Chapter 2

Background and Prior Related Work

2.1 Background

2.1.1 Understanding RTL and Verilog

RTL is crucial in bridging the gap between high-level design concepts and physical hardware implementation. It abstracts the hardware into a series of registers and combinational logic, capturing the flow of data and control signals within a circuit. The RTL code, often written in Verilog, provides a detailed description of how the hardware should behave.

Verilog is a popular HDL used to write RTL descriptions. It is known for its simplicity and C-like syntax, making it accessible to those familiar with software programming. Verilog allows hardware designers to define the structure and operation of circuits in a way that can be easily simulated and synthesized into actual hardware components.

The process of writing RTL in Verilog is somewhat analogous to software development, where the goal is to create a precise set of instructions that define the behavior of a system. However, unlike software that runs on general-purpose processors, RTL code written in Verilog directly specifies the architecture and functionality of custom hardware circuits.

Hardware Bugs

Throughout the RTL design process, the possibility of introducing bugs is of significant concern. RTL bugs can be classified into the following categories: functional bugs [1], compilation and

Listing 2.1: Example Verilog RTL Code from the Flip-Flop (T Flip-Flop) Project, Highlighting the Functional Bug in Red and the Correct Fix in Blue

```
module tff (   input clk, input rstn, input t, output reg q);

 always @(negedge clk) begin // Buggy code (functional bug)
 always @(posedge clk) begin // Correct code

    if (!rstn)
      q <= 0;
    else
      if (t)
          q <= ~q;
      else
          q <= q;
  end
endmodule
```

syntax-related bugs [13], and security bugs [14]. Functional bugs occur when the RTL code does not accurately implement the intended behavior as specified by the design requirements. These bugs are especially problematic because they can allow the code to compile successfully but result in incorrect design outputs [1]. Compilation and syntax-related errors generate errors during the compilation phase and thus prevent the RTL code from being properly synthesized into functional design [13]. Hardware security bugs violate security assumptions and features of the design. These bugs can be exploited by attackers to modify the functionality of the design, violate the integrity of the design output and data, leak secret information or cause denial-of-service of the design [14]. A taxonomy of hardware security bugs and examples of historic hardware security bugs inspired by industry work are documented in [15]. While compilation and security bugs are critical, in this thesis, our focus is specifically on fixing functional bugs in RTL designs.

Functional hardware bugs can be introduced by unintentional errors during the translation of design specifications into RTL code. These bugs may arise from human error or through the use of generative AI-based code assistants [16]. An example of a functional hardware bug is shown in Listing 2.1, which features RTL Verilog code for a flip-flop. This example highlights a bug in the sensitivity list, where a `negedge` is incorrectly used instead of a `posedge`, demonstrating how such errors can affect the intended functionality.

2.1.2 Bug Detection, Localization, and Fixing

Bug Detection

To eliminate bugs in final hardware products, they must first be detected. Bug detection indicates the presence of a bug but does not necessarily identify its exact location. This can be achieved through various methods, including expert manual inspection of the code, static RTL security scanners [17], formal verification [18], and simulation [13].

Manual inspection of code can be effective, but it requires experts with extensive experience to be reliable. The manual review process, however, is time-consuming and incurs high costs, as expert reviewers typically have high hourly rates. Additionally, if less experienced engineers perform the review, many bugs may go undetected.

Static RTL scanners are effective at detecting well-known patterns that can be represented by predefined rules [19]. They can also be applied to submodules and incomplete RTL implementations. However, they may fail to detect new bugs for which no predefined rules exist. Furthermore, bugs that span multiple modules or files may be overlooked if the scanners are applied only to submodules. Additionally, static analysis scanners may produce a high rate of false positives [17].

Formal verification mathematically models the design and exhaustively proves that it meets the specifications for all possible inputs and design states. Assertions generated from the design specifications help detect bugs. These assertions can be manually created or generated using LLMs. However, formal verification struggles with scalability as the design becomes more complex [20], becoming a time and resource-intensive methodology for large designs. Moreover, accurate assertions and well-written design properties are crucial for its success.

Compilers can detect syntax issues in RTL code, but they cannot identify bugs caused by undefined behavior, functional issues, or security vulnerabilities [13].

Simulation involves applying specific test vectors as inputs to the design, and comparing the outputs with expected results to detect bugs. However, simulation can only capture bugs triggered by those test vectors. While simulation is a widely used bug detection methodology, it does not scale well for large designs due to the significant time required for simulation. Additionally, simulating all possible test cases is infeasible [20].

Since each method has its own strengths and weaknesses, different bug detection techniques

complement each other to provide optimal bug detection coverage.

Bug Localization

Bug localization is the process of identifying the exact location of the bugs. For example, the exact code lines that need to be modified to fix the bug. Identifying the exact location of the bug is still an active area of research [21] [1] [22].

Bug Fixing

Bug fixing involves only the repair of the bugs. Research work in this area assumes that the location of the bugs have been determined by other tools [4] or the bug repair methodology proposed does not require the knowledge of bug location. In the context of hardware, a bug can be considered resolved if the code compiles successfully and performs with correct functionality when tested against a relevant testbench. This dual criterion of successful compilation and functional validation ensures that the fix not only addresses the identified issue but also integrates seamlessly into the overall design, fulfilling expected operational requirements.

Automatic Program Repair (APR)

Automatic Program Repair (APR) automates the identification, fixing, and validation of software bugs. It typically involves three stages: (1) **Bug Localization**, which identifies the source of the issue; (2) **Repair Generation**, where fixes are proposed; and (3) **Repair Verification**, ensuring the fix is correct and functional [23].

APR is gaining attention for its ability to reduce manual debugging and accelerate development, especially in time-sensitive contexts like safety-critical systems. Traditional APR methods often rely on search-based or heuristic approaches, which can be computationally expensive and struggle with complex bugs.

Recent advancements in machine learning, especially LLMs, offer new opportunities for APR by generating syntactically correct fixes. These models, trained on large code datasets, help reduce the search for fixes, but challenges remain in making sure the fixes are semantically correct, particularly in fields like hardware design, where APR is still developing.

Listing 2.2: Verilog Testbench for a 4-to-1 Multiplexer (`mux_4to1`) Design

```
module mux_4to1_tb;

  // Inputs
  reg [3:0] a, b, c, d;  // 4-bit inputs
  reg [1:0] sel;          // 2-bit selector input

  // Output
  wire [3:0] out;         // 4-bit output

  // Instantiate the Unit Under Test (UUT)
  mux_4to1 uut (.a(a),.b(b),.c(c),.d(d),.sel(sel),.out(out) );

  // Initial block to provide test inputs
  initial begin
    // Initialize inputs
    a = 4'b0001; b = 4'b0010; c = 4'b0100; d = 4'b1000;
    sel = 2'b00; // Select input 'a'

    // Apply test vectors
    #10 sel = 2'b01; // Select input 'b'
    #10 sel = 2'b10; // Select input 'c'
    #10 sel = 2'b11; // Select input 'd'
    #10;
  end

  // Monitor output for verification
  initial begin
    $monitor("Time=%0t,a=%b,b=%b,c=%b,d=%b,sel=%b,out=%b",
             $time, a, b, c, d, sel, out);
  end

endmodule
```

2.1.3 Testing and Verification

Ensuring the correctness of RTL designs is critical, and this is achieved through rigorous testing and verification. Testbenches are used to create controlled environments where the RTL design is subjected to simulated inputs, and the outputs are checked against expected results. To facilitate this process, several simulation tools are available, such as `ModelSim`, `VCS (Verilog Compiler Simulator)`, `XSIM`, and `Questa`. These tools allow for comprehensive RTL verification by running simulations, checking timing, and ensuring the logic performs as expected under various conditions. These tools integrate with the testbenches to emulate the hardware behavior and identify potential issues before physical implementation.

Simulation tools play a vital role in this process by executing the RTL code alongside the testbenches to emulate the behavior of the hardware before it is physically manufactured. They help in detecting functional bugs, syntax errors, and timing violations, which are crucial in verifying the correctness of the design. These tools are indispensable for identifying and fixing functional

bugs early in the design process, thus preventing costly errors in the final chip.

As shown in Listing 2.2, here is a simple testbench for a 4-to-1 multiplexer (`mux_4to1`) in Verilog, which demonstrates how a testbench can be set up to test the functionality of an RTL design.

By using simulation tools such as `ModelSim` or `VCS`, the testbench can be simulated to observe the behavior of the multiplexer. If any functional bugs are detected, they can be fixed in the RTL code, and the simulation can be rerun to verify the fix.

2.1.4 Large Language Models (LLMs)

Large language models are a class of machine learning algorithms that have recently demonstrated state-of-the-art solutions to natural language processing problems. LLMs are built on the architecture of transformers[24]. Transformers have shown superior results in predicting future sequences in natural language based on previous sequences. LLMs are characterized by having significantly large number of parameters in the range of billions. LLMs are trained on enormous datasets. LLMs have shown success in multiple disciplines beyond natural language text generation. Examples of applications include code generation [25], [26], software code bug fixing [27][10], RTL bug fixing [13][4][28], System-on-chip design [29], security validation and verification [30] [31].

2.1.5 Prompt Engineering

Prompt engineering is a critical technique that involves carefully crafting the input text (prompt) given to a language model to optimize its responses. By strategically selecting and structuring words, phrases, and questions, prompt engineering guides the model's behavior and encourages specific types of answers. This approach allows researchers to fine-tune the language model's output for complex applications like bug repair, where it can address subtle ambiguities and biases that arise in technical tasks.

When working with large language models like those developed by OpenAI, prompt engineering techniques become essential. OpenAI, for instance, outlines strategies to refine prompts for maximum effectiveness, such as adding contextual detail, simplifying tasks, and iteratively adjusting prompts based on output quality [32]. These techniques are especially valuable in guiding models to accurately interpret and repair issues within specialized domains like digital hardware design. By leveraging

these principles, prompt engineering enhances the model's ability to respond effectively, making it a fundamental component of research in automatic bug repair using language models.

2.2 Prior Related Work

Automatic program repair has been an active research area especially in the software domain for decades [33] [34] [35] [36]. Since LLMs have recently shown success in multiple disciplines including program generation [37] [38] [39] [6] [5], researchers have started to explore LLMs for bug fixes in the software and hardware domains.

2.2.1 Software Bugs Repair

Machine learning-based techniques have been widely used in recent decades to automatically fix software bugs. These techniques include deep learning-based, reinforcement learning-based, and large language model (LLM)-based approaches [40, 41, 10, 27, 42].

Non-LLM-Based Bug Repair

The work in [40] proposes using supervised learning to train encoder-decoder recurrent neural networks (RNNs) to identify the location of syntax errors and automatically fix them. However, the results indicate that the trained neural network fails to correctly fix some complex errors that require a major rewrite of the program. In [41], reinforcement learning (RL) is proposed for fixing syntax errors in software programs. In this approach, the program text and the location of the syntax error are modeled as states in a reinforcement learning environment. The program with the initial syntax error is represented as the initial state, and the edits applied to the program are modeled as actions taken by the RL agent to transition the program from one state to another. The RL agent receives a reward if the number of syntax errors decreases after an edit is applied. While this approach demonstrates effective results for fixing syntax errors in software, deep reinforcement learning (DRL) may not be the best solution for fixing hardware bugs. DRL requires large training datasets to learn from, and the dataset used in [41] consisted of 165K examples. It is not feasible to obtain such large open-source hardware bug datasets. Moreover, DRL requires significant computational power for training and consumes considerable time to learn from the datasets.

LLM-Based Bugs Repair

Recent work has explored the use of different prompts/instructions for LLMs to repair software [10]. The work in [10] studies the effectiveness of Codex for edit [43] to repair software code bugs. While the utilized OpenAI Codex LLMs has been discontinued, such work provides insights into prompting strategies that might be useful for other models. This work evaluates three types of instructions to codex-e to fix the bugs. These instructions are as follows: 1) asking the LLM to fix the bug without specifying details about the location, type of the bug or how to fix it; 2) instructions to fix a particular line number or 3) instructions to fix a particular statement in the code. The results shows that providing the actual statement of the buggy line outperforms all the other instruction formats provided to codex-e. Our work is inspired by [10] in its framing of different levels of detail in the repair instruction (from none through to precise instructions to fix a specific line of code) but we instead examine the hardware context and make no assumptions on defect localization.

The work in [27] repairs security vulnerabilities in functions in software code using CodeT5 [38]. In [27], CodeT5 is fine-tuned on a dataset for bug-fixing with inputs that are passed as sequences of Abstract syntax tree (AST) nodes as well as sequences of code tokens. For every vulnerable function that needs to be fixed, the embeddings of the following inputs are passed to the decoder of the fine-tuned CodeT5 to generate the code fix: vulnerable function that needs to be repaired, AST nodes of the vulnerable function, Common Weakness Enumeration (CWE) type of the vulnerable function and examples of relevant vulnerable functions and their corresponding fixes obtained from [15]. This work assumes the knowledge of bug types and relevant vulnerable code snippets and their corresponding fixes at fix time which is not always available in realistic scenarios. Our proposed approach does not rely on these assumptions and thus is more suitable for fixing real-life hardware bugs. Also, this work relies on fine-tuning the model with bug repair dataset which increases the computation overhead, while our proposed approach does not require fine-tuning of the used models.

The work in [42] proposes a tool to detect, localize, classify, and fix software bugs. To fix a bug, the most semantically similar bugs and their related fixes are retrieved from a database of historic bugs. A prompt is formed of the buggy code, lines of code across the entire source code files that are related to the buggy code, and relevant historic bug examples and fixes, and the bug type. A

fine-tuned 12B Codex model generates the patch.

2.2.2 Hardware Bug Repair

Genetic Algorithm-based Bug Repair

In the hardware space, automatic Verilog repair is an emerging area, with prior work such as CirFix [1] exploring the use of genetic algorithms to generate fixes. Genetic algorithms operate by iteratively applying random mutations to the code and evaluating these mutations using a fitness function to determine correctness. Over successive iterations, mutations that improve the design are retained and combined, gradually increasing the probability of finding a valid fix. However, this approach has notable drawbacks. The random mutations often produce RTL code fixes that are syntactically incorrect, necessitating additional iterations to refine them. Moreover, the process relies heavily on repeated cycles of design simulation to evaluate each proposed fix, resulting in significantly high time overhead. This undermines the core motivation of automatic bug repair, which is to expedite the time-to-market for hardware designs.

To address these limitations, we leverage LLMs, which are pre-trained on extensive internet data, including RTL code samples [44] [45]. Unlike genetic algorithms, LLMs generate fixes that are syntactically correct on the first attempt, eliminating the need for repeated cycles of simulation to validate code structure. This approach significantly reduces time overhead while maintaining a high likelihood of producing functionally correct repairs.

Our work builds on the "back-end" part of CirFix (i.e., we make the same assumptions regarding testbench and Oracle (bug-free reference design used to verify the correctness of a system under test) availability), and we investigate the prompting of LLMs as an alternative to the genetic algorithm component.

LLM-based Bug Repair

Prompt Engineering-based Techniques An LLM-based approach was recently proposed by [4], with a specific focus on functional and security hardware bugs. Their work evaluated several models (including some that have since been discontinued) in fixing bugs from the Cirfix [1] and [46] datasets.

The proposed approach assumes the availability of the exact line location of the bug, which is an impractical assumption in many cases. It also requires detailed instructions written in natural language to describe the bug and guide the LLM in fixing it. The descriptions recommended by [4] often include pseudo-code or natural language explanations on how to resolve the bug, tailored to each specific case. This approach depends on human experts to fully understand the bug and know exactly how to fix it, which is unrealistic, especially in commercial designs. The effort needed to pinpoint exact line locations, understand the context, and provide detailed descriptions of the bug is significant and resource-intensive.

Similar to [10], this work assumes defect localization. In contrast, we explore whether it is possible to develop prompting strategies that leverage context, such as simulation feedback, to guide specific bug fixes when precise localization information is unavailable. We focus on providing the LLM with automatically generated information during design simulation—a standard step in the design validation cycle. Additionally, we consider generic guidelines for the LLM to follow while generating bug fixes.

The work in [47] investigates the performance of various models, including `code-cushman-001`, `code-davinci-001`, `code-davinci-002`, `j1-jumbo`, `j1-large`, `polycoder`, and `gpt2-csrc`, in repairing security-related bugs using a custom dataset derived from open-source repositories. This study examines the effects of LLM parameters (e.g., temperature and top_p) and prompt engineering on the quality of bug repairs. Results indicate that no single LLM parameter outperforms others across all types of bugs. However, like other works, this study assumes the availability of bug type, description, and precise location, which is not feasible in real-life scenarios.

RAG-based Techniques The work in [13] focuses on using LLMs to fix RTL syntax errors. [13] uses gpt3.5-turbo model [48] to fix syntax errors. The proposed technique is evaluated on a dataset derived from verilogeval [26]. This work assumes the presence of a database that summarizes human expertise in RTL syntax repair. The prompts used integrate data retrieved from this database using retrieval augmented generation (RAG) [49]. They also integrate feedback from compilers to reasoning and action planning [50] prompts. However, this work does not perform well in fixing simulation errors. Also, the assumption of the availability of a database that summarizes human expertise over the years is not always realistic in real-life scenarios.

The work in [51] utilizes LLMs and RAG framework for functional hardware bug identification and patching. The specifications of the design-under-test are stored in a vector database. The RTL code of a design is split into lines. Each line of code is passed to the RAG framework. The most relevant specification file chunks are retrieved from a vector database and passed to the LLM along with the RTL code line. LLMs would then identify if a line has a functional bug and attempt to fix the bug. If the bug is not fixed, a line before and after each line is considered and a window of selected lines around the original line are passed to the LLM to identify any present bugs and patch them accordingly. This process of adding lines to the window of lines around the original line in the buggy code is repeated till a correct patch is found (at maximum of 3 times). This process is repeated for each line in the buggy code. This work assumes the availability of documents that map high-level specifications to RTL lines of code which is not realistic in industry settings. Also, it assumes that the scope of a single bug spans consecutive lines. However, lines related to a single bug can spread across a single or multiple RTL files and not necessarily be confined to consecutive lines.

RAG-based Techniques combined with LLM Fine-tuning Techniques Similar to [52], the work in [28] utilizes fine-tuned LLM models to debug buggy code and fix them. In [28], CodeLlama-13b is fine-tuned. The difference is the type of data used for fine-tuning. Unlike the buggy codes that are used in [52] which are derived from open-source repositories and have security-related bugs, the buggy codes used in [28] for training focus on code snippets that are generated by modifying high-quality industry-level code to include compilation-related bugs. The data used in fine-tuning also includes historic bug information retrieved from historic databases using retrieval augmented generation. The types of bugs to be inserted and the functions to insert bugs are generated using LLMs. The data used to fine-tune the model includes buggy code, related error messages, relevant historic buggy code snippets, and historic documentation chunks that are relevant to the buggy code-under-fix and its corresponding error message (extracted using retrieval augmented generation). Fine-tuning data also includes LLMs-generated thoughts for fixing the bug and the fixed code. There is a two-step approach used in fine-tuning. First to fine-tune the LLM to predict the correct thought to correct the bug using buggy code, error message, relevant historic document chunks, relevant code snippets and prompts to generate the thoughts to fix the bug. The second stage is to

further fine-tune the model to generate the correct code fix. Similar to [52], the assumption that a dataset of historic bugs and related documentation is not always realistic in industry settings. Also, the high computational cost of fine-tuning limits the wide adoption of this method in academic and industry settings.

2.2.3 Identified Research Gap

Table 2.1: Comparison of our proposed work with prior work in terms of target task type: indicates whether the task is bug repair, bug localization, bug identification or a mix of multiple tasks; bug type: indicates whether syntax errors, functional, or security bugs are fixed; model type; dataset: indicates which dataset is used for evaluation

Paper	Task	Bug Type	Model Type	Dataset
[1]	localization, repair	Functional	genetic algorithm	Cirfix[1]
[4]	repair	Security, functional	different models (GPT4 best)	OpenTITAN [46], Cirfix [1]
[47]	repair	security	code-cushman-001, code-davinci-001/002, j1-jumbo/large, polycoder,gpt2-csrc	custom-created from opensource
[13]	repair	Syntax errors	gpt-3.5-turbo	derived from VerilogEval [26]
[51]	identification, repair	functional	not clearly mentioned	OpenTITAN [46]
[52]	localization, repair	functional, security	Falcon 7b, stableLM 7b Llama2 v2 7b	custom-created from opensource
[28]	debugging, repair	compilation errors	codeLlama-13b	custom-created from industry code
Ours	implicit localization, repair	functional	gpt-4o-2024-05-13	Cirfix [1]

We summarize the major features of prior work and compare them to our proposed work in Table 2.1. Unlike prior work, we focus on bug fixing and employ an implicit bug localization method proposed by [1]. While prior work has used relatively outdated releases of LLMs, we propose using the most recent LLM released by OpenAI at the time of writing this thesis. Additionally, we concentrate on fixing functional bugs. However, the insights from our study can be extended to address other types of bugs.

A comparison of assumptions necessary for successful bug fixing between prior work and our approach is shown in Table 2.2. As highlighted in Table 2.2, prior work relies on one or more assumptions that are often unrealistic in real-world scenarios. To address this, our research avoids relying on such assumptions. The only assumption we make is the presence of an oracle (in this setting, a bug-free reference design used to verify the correctness of a system under test), which is realistic, as the oracle is available during the verification process (e.g., as a reference model implemented in a higher-level of abstraction). Specifically, verification engineers know the expected simulation output for each test case while designing the verification plan.

To identify the research gap, we explored various techniques by reviewing related work. A comprehensive survey in [53] outlined how LLMs are utilized for automatic program repair and identified the most commonly used techniques in software and hardware repair: zero-shot, few-shot, and fine-tuning. In this thesis, we decided to focus on studying the impact of prompt variation on generative AI-based repair rather than training deep neural networks from scratch or fine-tuning a model, as done in prior software-related research [40]. This decision was influenced by the limited availability of open-source hardware data and hardware-specific datasets. After reviewing related work and surveys, we selected three primary techniques for investigation: zero-shot, few-shot, and feedback techniques.

To design the prompts used in this work, we reviewed prior literature. Table 2.3 summarizes how prompts for LLM-based hardware bug fixes are designed in previous studies. The data in Table 2.3 highlights that none of the prior works performed a comprehensive study of the effect of different prompt variations on bug fixing or compared these variations using the same dataset. To fill this gap, this thesis explores the effect of using zero-shot, few-shot prompts, and prompts incorporating information already available from the verification process. We also analyze the impact of structured versus unstructured prompts on bug-fixing performance. Furthermore, we investigate iterative bug fixing, where feedback from the verification process is incorporated into the prompt for each new iteration until the bug is resolved. Our study avoids making unrealistic assumptions or using data unavailable in practical scenarios. We aim to determine whether prompt variations can achieve bug-fixing performance comparable to prior work without relying on unrealistic assumptions.

Table 2.2: Comparison of our proposed work with prior work in terms of assumptions needed for the success of the method: **Bug type known?** Indicates that the bug type is known and used in the prompt to fix the bug. **Bug location known?** Indicates that the bug location is known and used in the prompt to fix the bug. **Historic bugs available?** Indicates the presence of huge dataset of historic bugs that can be used for fine-tuning LLMs or used in RAG-based technique to extract the most relevant examples. Note that we do not consider the use of few bug examples in few-shot learning as a need for historic bug data. **Historic human fixes available?** Indicates the assumption of the availability of a database that summarizes human experts fixes for similar bugs in the past. **High-cost computation available?** indicates the assumption of the availability of high-cost computation resources needed for fine-tuning LLMs. **Oracle needed?** Indicates the assumption of the presence of an oracle which is a bug-free reference design used to verify the correctness of a system under test. **Detailed specs available?** indicates the presence of detailed specification documents that describes the properties of the design-under-test in details to the extent of having information to fix issues in the RTL code.

Paper	Bug type known?	Bug location known?	Historic bugs available?	Historic human fix available?	High-cost computation available?	Oracle?	Detailed specs
[4]	✓	✓	✗	✗	✗	✗	✗
[47]	✓	✓	✗	✗	✗	✗	✗
[13]	✗	✗	✗	✓	✗	✓	✗
[51]	✗	✗	✗	✗	✗	✓	✓
[52]	✗	✗	✓	✗	✓	✗	✗
[28]	✗	✗	✓	✗	✓	✓	✓
Ours	✗	✗	✗	✗	✗	✓	✗

Table 2.3: Comparison of our proposed work with prior work in terms of techniques and prompting strategies. **Zero-shot?**: indicates whether zero-shot prompting technique has been used; **Few-shot?**: indicates whether few-shot prompting technique has been used; **Prompt variants tested?**: indicates whether the effect of different prompt variants bug fix performance is investigated; **verification information used?**: indicates whether information that is already available from the verification phase is used in the prompt variants, for example, the expected simulation output and the simulation output of the design with the bugs; **Iterative?**: indicates whether feedback from the simulation is used in multiple iterations to improve the bug fix

Paper	Zero-shot	Few-shot or one-shot	Prompt variants tested?	Verification Info used?	Iterative ?
[4]	✓	✓	✓	✗	✗
[47]	✓	✗	✓	✗	✗
[13]	✗	✓	✗	✓	✓
[51]	✗	✓	✗	✗	✓
[52]	✓	✗	✗	✗	✗
[28]	✗	✓	✗	✓	✗
Ours	✓	✓	✓	✓	✓

Chapter 3

Experimental Method

In this chapter, we present the experimental framework developed for investigating the performance of LLMs in the context of automatic bug repair for digital hardware design. The framework, along with its key components, is explained to provide an understanding of how it supports the experimental process.

The benchmark suite used in this study is presented, providing a diverse set of test cases to evaluate the effectiveness of the proposed techniques. Metrics employed for assessing performance are detailed, offering a comprehensive view of how repair accuracy and efficiency are measured. Finally, the experimental setup, including the computational environment and relevant configurations, is explained to provide clarity and help with reproducibility.

This chapter establishes the foundation for understanding the results and analyses discussed in the subsequent chapters.

3.1 Framework

To investigate the effect of different prompting strategies for LLMs in generating bug fixes using feedback from an RTL simulation tool with testbenches (instrumented as in prior work [1]), we designed the experimental framework shown in Fig. 3.1. At the core of this framework is the LLM used to generate bug fixes. The other elements of the framework are described next. Broadly, the framework takes as input the *buggy RTL code*. We assume that a *testbench* and *oracle* (golden reference model) are available for validation purposes. We do not assume the availability of a clear

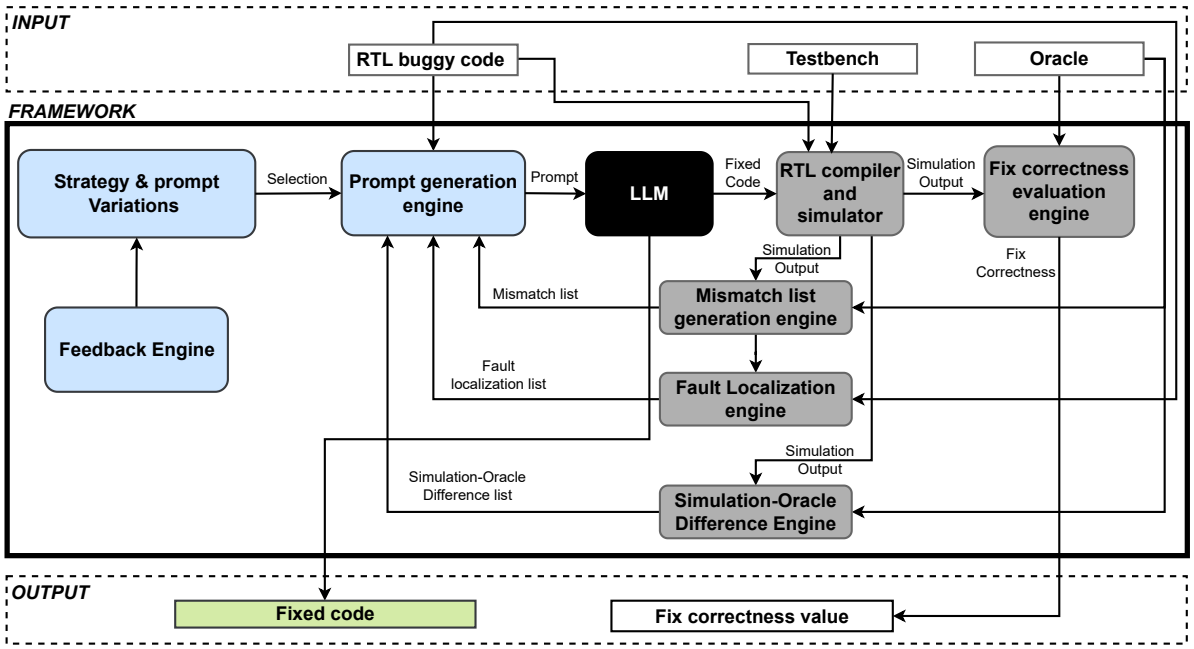

Figure 3.1: The experimental framework is illustrated with the main components we developed, shown in blue. The LLM model, used as a black box, is depicted in black. The "verification framework" used in this work is modified from [1] and represented in grey.

bug description, making the scenario more aligned with real-world situations where the exact nature of the bug may not be immediately evident. This design choice offers a more flexible and realistic setting compared to prior work [1, 4].

3.1.1 Strategy and Prompt Variations

We perform prompt engineering to identify and compare the prompts that would result in the most accurate bug fixes, varying the type of information that we provide to the LLM as context. We use the term **"strategies"** to refer to the specific types of information included in the prompts, with each strategy incorporating distinct input data, as explained in Section 4.1. For every strategy, we provide two variations: unstructured prompts, formatted as paragraphs, and structured prompts, where the inputs are presented in an organized and systematic manner. These variations are further detailed in Section 4.2.

3.1.2 Prompt Generation Engine

This engine is responsible for generating prompts based on the provided inputs. It works by integrating the inputs gathered from various components of the framework with the predefined templates corresponding to the selected prompt strategy and variant. The engine collects all necessary information, such as the input data, strategy choice, and prompt variation, to construct the prompt in accordance with the specified requirements. Once the prompt is generated, it is sent to the LLM model used within our framework (as described in Section 3.3), which processes the prompt and produces the corresponding output. This engine ensures that the prompts are tailored to the specific needs of the experiment.

3.1.3 Feedback Engine

This engine specifically designed to support the feedback technique, which involves multiple iterations. This engine can be triggered automatically, allowing for seamless execution of the feedback process within the framework. Detailed information regarding the logic behind the engine, as well as the experiments where it was applied, will be discussed in Chapter 6.

3.1.4 Verification Framework

RTL Compiler and Simulator

We use the Synopsys VCS RTL compiler (`VCS Compiler version: U-2023.03-SP1_Full64`) and simulator (`VCS Script version: U-2023.03`) to compile and simulate the RTL design and determine the correctness of the implementation. As in prior work [1], we use instrumented testbenches, an example of which is shown in Listing 3.1. The simulation output at verification time reveals the presence of bugs (functional mismatches), and this information is passed to the *mismatch list generation engine* to identify which signals are affected by the bug. This simulator, running on a `linux64` environment with `Linux 4.18.0-553.16.1.el8_10.x86_64`, is also used after the bug is fixed to evaluate the correctness of the fix.

Listing 3.1: Instrumented Testbench for the Flip-Flop Project

```
module tb;
  reg clk;
  reg instrumented_clk;
  reg rstn;
  reg t;
  wire q;

  tff u0 (.clk(clk),.rstn(rstn),.t(t),.q(q));
  always #5 clk = ~clk;
  always #20 instrumented_clk = ~instrumented_clk;
  integer f;
  integer f_in;
  initial begin
    f = $fopen("output_tff_tb.txt");
    f_in = $fopen("input2_tff_tb.txt");
    //$fwrite(f, "time,q\n");
    $fwrite(f, "time,rstn,t,q\n");
    $fwrite(f_in, "time,rstn,t\n");
    forever begin
      @(posedge clk);
      //$fwrite(f, "%g,%b\n", $time,q);
      $fwrite(f, "%g,%b,%b,%b\n", $time,rstn,t,q);
      $fwrite(f_in, "%g,%b,%b\n", $time,rstn,t);
      $display("time=%0t,rstn=%b,t=%b,q=%b", $time, rstn, t, q);
    end
  end

  initial begin
    {rstn, clk, t, instrumented_clk} <= 0;
    $monitor ("T=%0t␣rstn=%0b␣t=%0d␣q=%0d", $time, rstn, t, q);
    repeat(2) @(posedge clk);
    rstn <= 1;
    for (integer i = 0; i < 20; i = i+1) begin
      reg [4:0] dly = $random;
      #(dly) t <= $random;
    end

    // #20 $fclose(f);
    #20 $finish;
  end
endmodule
```

Mismatch List Generation Engine

Based on the method outlined in [1], this engine identifies discrepancies between expected and actual signal values in a circuit during simulation.

A "mismatch" occurs when there is a difference between simulated and expected circuit outputs, as produced by an *oracle* (bug-free reference design used to verify the correctness of a system under test), and thus indicates where current behavior diverges from intended behavior. A "mismatch list" comprises the set of signals that have mismatched in a simulation run. Fig. 3.2 illustrates an example of how a mismatch list is formed. The comparison occurs at each time step, and at time step 35, the variable x in the actual output differs from the expected output. Consequently, x is

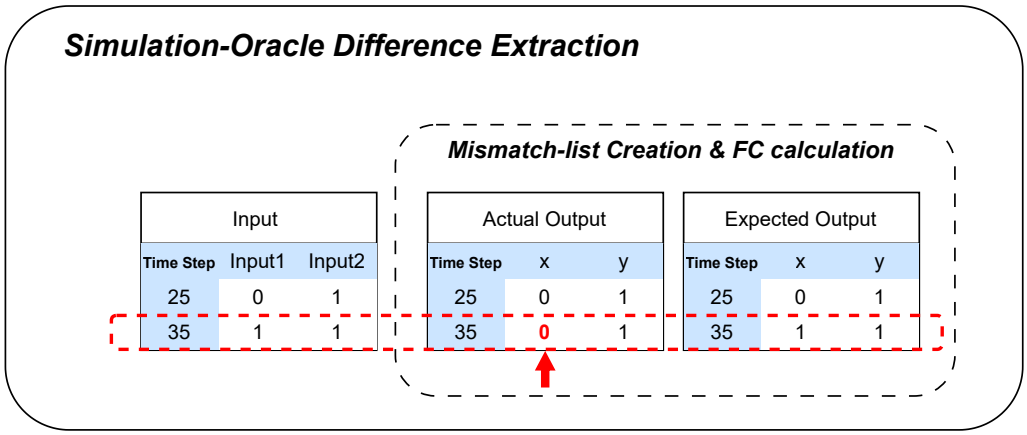

Figure 3.2: Example for the Mismatch-list Creation, Fix Correctness (FC) Calculation, and Simulation-Oracle Difference Extraction. The dotted box shows the creation of the mismatch list and the calculation of FC, where the mismatch list is $\{x\}$ and the FC is 0.5. The larger solid box shows the Simulation-Oracle Difference Extraction, where the simulation output is compared to the expected output, identifying discrepancies between the actual and expected values at each time step, while also showing the input values at the same time steps.

added to the mismatch list. Note, however, that this does not necessarily indicate that x is the root cause of the issue; rather, it signifies the set of signals where discrepancies exist.

Fix Correctness Evaluation Engine (FC)

This engine evaluates the correctness of the RTL bug fixes based on the approach introduced in [1]. The engine compares the outputs obtained from the oracle and the compiler and simulation tool at different time steps, yielding a score between 0 and 1, with 1 indicating a correct repair. For each different time step, we compare each bit of the actual output to the expected output. If a bit matches the expected value, we add a value of 1 to the score; if it doesn't match, we subtract 1. We then normalize the score. If the total score after comparisons is < 0, the FC value is set to 0. If it is > 0, the FC value is the total score divided by the maximum possible score. For example, in Fig. 3.2, we compare the actual and expected outputs at each time step. At time step 25, both x and y match the expected values, so we add 1 for each match (+2). At timestep 35, x does not match the expected value, so we subtract 1 (−1), and y matches, so we add 1 (+1). The total score is 2 out of a maximum possible score of 4, resulting in $FC = 0.5$.

Simulation-Oracle Difference Engine

The *Simulation-Oracle Difference Engine*, developed as part of our approach, aims to explore whether providing more detailed information can enhance the LLM's performance by offering deeper insights into the simulation process. This engine automatically extracts relevant data from the simulation output and compares it with the expected output (oracle) at each time step.

To enable this, we modify the test benches from prior work [1], instrumenting them to record the input signal values alongside the simulation output. This modification allows us to obtain both the input values and corresponding output values at each simulation step. The combination of this input-output information provides a clearer understanding of how specific inputs influence the outputs, which is essential for debugging.

At each time step during simulation, the engine compares the actual simulation output (from the buggy code) with the expected output (oracle) at that time. For example, as shown in Fig. 3.2, at time step 25, the input values are "input1 = 1" and "input2 = 1." At time step 35, the actual output is $x = 0$ and $y = 1$, while the expected output (from the oracle) is $x = 1$ and $y = 1$. Here, we observe that the value of x at time step 35 differs between the simulation output and the oracle output.

This difference provides valuable information. By using the *Simulation-Oracle Difference Engine*, we extract the input values, the actual output, and the expected output at the relevant time step (35) where the discrepancy occurred. This extracted information helps the LLM better understand how specific input values affect the faulty behavior in the simulation and identify the likely source of the error.

Fault Localization Engine

The *Fault Localization Engine* is based on the methodology proposed in [1] and has been adapted and extended to identify problematic sections within Verilog HDL code by analyzing mismatches between the simulated outputs of a faulty design and the expected outputs from an oracle design. The engine takes as input the abstract syntax tree (AST) of the Verilog code, the simulation outputs, and the expected outputs. The process begins by initializing two empty sets: one for tracking mismatched signals ($mismatch$) and another for the fault localization set (FL). Initially,

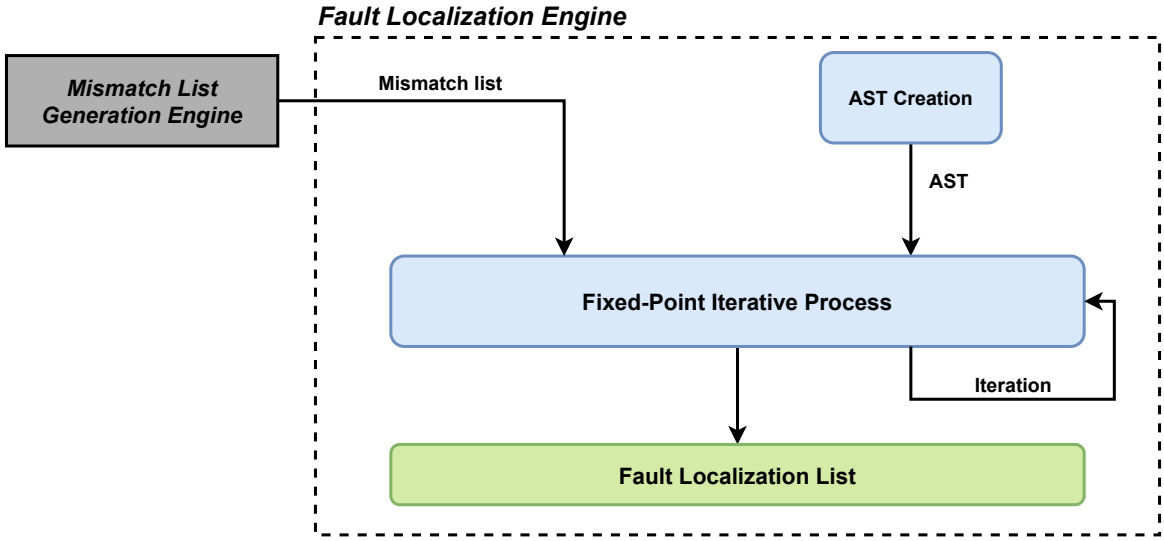

Figure 3.3: Overview of the Fault Localization Engine process.

the *Mismatch List Generation Engine* is used to compute the discrepancies between the simulation and expected outputs, which are stored in a temporary set ($mismatch'$).

The core operation of the *Fault Localization Engine* involves a fixed-point iteration process, where the mismatch set is iteratively updated until it stabilizes (i.e., no new mismatches are added). For each node in the AST, the engine checks whether the node is implicated in the current mismatch set. If a node is associated with any of the mismatches, its identifier is added to the fault localization set (FL). Additionally, the engine analyzes the node's child elements: each child's identifier is added to FL, and if the child represents a signal not already in the mismatch set, it is added to $mismatch'$ for further evaluation.

This iterative process ensures that the engine traces mismatched outputs backward through the Verilog code structure, identifying all components potentially contributing to the fault. The fixed-point iteration terminates when the mismatch set no longer changes between iterations, ensuring convergence. At this point, the fault localization set (FL) contains the identifiers of all implicated nodes and signals. By systematically propagating mismatches and analyzing the AST, the *Fault Localization Engine* effectively identifies the potential root causes of discrepancies between the simulation and expected outputs in the Verilog design.

The *Fault Localization Engine* analyzes the code and identifies specific lines most likely associated with erroneous behavior. These lines are highlighted as potential sources of the bug. However, the

Table 3.1: Summary of features of the used benchmarks in this thesis, as described in [1]

Project	Design Functionality	Defect Description	LOC	Category
decoder_3_to_8	3-to-8 decoder	Two separate numeric errors	25	1
		Incorrect assignment		2
counter	4-bit counter with overflow	Incorrect sensitivity list	56	1
		Incorrect incremental of counter		1
		Incorrect reset		1
flip_flop	T-flip flop	Incorrect conditional	16	1
		Branches of if-statement swapped		1
fsm_full	Finite state machine	Incorrect case statement	115	1
		Assignment to next state and default in case statement omitted		2
		Assignment to next state omitted, incorrect sensitivity list		2
		Incorrectly blocking assignments		1
lshift_reg	8-bit left shift register	Incorrect blocking assignment	30	1
		Incorrect conditional		1
		Incorrect sensitivity list		1
mux_4_1	4-to-1 multiplexer	Three separate numeric errors	19	2
		Hex instead of binary constants		1
		1 bit instead of 4 bit output		1
i2c	Two-wire, bidirectional serial bus for data exchange between devices	Incorrect sensitivity list	2018	2
		Incorrect address assignment		2
		No command acknowledgment		2
sha3	Cryptographic hash function	Off-by-one error in loop	499	1
		Incorrect assignment to wires		2
		Skipped buffer overflow check		2
		Incorrect bitwise negation		1
tate_pairing	Core for the Tate bilinear pairing algorithm for elliptic curves	Incorrect logic for bitshifting	2206	1
		Incorrect operator for bitshifting		1
		Incorrect instantiation of modules		2
reed_solomon_decoder	Core for Reed-Solomon error correction	Insufficient register size for decimal values	4366	1
		Incorrect sensitivity list for reset		2
sdram_controller	Synchronous DRAM memory controller	Incorrect assignments to registers during synchronous reset	420	2
		Numeric error in definitions		1
		Incorrect case statement		2

accuracy of the fault localization depends on the complexity of the bug and whether the *mismatch generation engine* successfully detects the discrepancies related to the bug. If the mismatch generation engine identifies the fault, the engine is more likely to pinpoint the source of the issue. However, for more complex or subtle bugs, further investigation of other parts of the code may still be necessary.

An abstract overview of how the fault localization engine process works can be found in Fig. 3.3.

3.2 Benchmarks

In this section, we describe the benchmarks that are used in this thesis. We use the benchmark designs that have been used in [1]. We use these designs to ensure fair comparison with prior work as the work in [1] was the first work to propose automatic RTL bug repair and research work that follows uses the same benchmarks (e.g., [4]). Moreover, the work in [1] is open source with all the benchmark design code for buggy and unbuggy examples, and test benches are available. Thus, this facilitates the direct comparison with prior work.

The benchmark designs used in [1] comprise 11 Verilog RTL modules curated from diverse

Table 3.2: Simplifying the Buggy File Name for Consistent Reference in Subsequent Chapters

Project	**Bug Description**	**Simplified Buggy File Name**
decoder_3_to_8	Two separate numeric errors	decoder_numeric_error
	Incorrect assignment	decoder_assignment_error
counter	Incorrect sensitivity list	counter_sensitivity_issue
	Incorrect increment of counter	counter_increment_error
	Incorrect reset	counter_reset_issue
flip_flop	Incorrect conditional	flipflop_conditional_error
	Branches of if-statement swapped	flipflop_branch_swap
fsm_full	Incorrect case statement	fsm_case_statement_issue
	Assignment to next state and default in case statement omitted	fsm_state_assignment_missing
	Assignment to next state omitted, incorrect sensitivity list	fsm_state_sensitivity_issue
	Incorrectly blocking assignments	fsm_blocking_assignment_error
lshift_reg	Incorrect blocking assignment	lshift_blocking_issue
	Incorrect conditional	lshift_conditional_error
	Incorrect sensitivity list	lshift_sensitivity_issue
mux_4_1	Three separate numeric errors	mux_numeric_issues
	Hex instead of binary constants	mux_hex_error
	1 bit instead of 4 bit output	mux_bitwidth_mismatch
i2c	Incorrect sensitivity list	i2c_sensitivity_issue
	Incorrect address assignment	i2c_address_assignment_error
	No command acknowledgement	i2c_acknowledge_missing
sha3	Off-by-one error in loop	sha3_loop_boundary_issue
	Incorrect assignment to wires	sha3_wire_assignment_error
	Skipped buffer overflow check	sha3_buffer_overflow_skipped
	Incorrect bitwise negation	sha3_bitwise_negation_issue
tate_pairing	Incorrect logic for bitshifting	tatepairing_bitshift_logic_error
	Incorrect operator for bitshifting	tatepairing_operator_mismatch
	Incorrect instantiation of modules	tatepairing_module_instantiation
reed_solomon_decoder	Insufficient register size for decimal values	reedsolomon_register_size_error
	Incorrect sensitivity list for reset	reedsolomon_reset_sensitivity
sdram_controller	Incorrect assignments to registers during synchronous reset	sdram_sync_reset_issue
	Numeric error in definitions	sdram_numeric_definition_error
	Incorrect case statement	sdram_case_statement_issue

sources, including 6 projects from undergraduate courses and 5 projects obtained from the OpenCores website [54]. These designs encompass various functionalities, spanning arithmetic operations, cryptography, and memory control. Several designs have bug/defect variants, producing 32 benchmark designs. The designs vary in size, with line counts ranging from 16 to a maximum of 4366 lines. The defect types include issues in conditional statements, sensitivity lists, assignments, numeric errors, and logic issues. Table 3.1 summarizes the benchmark features as it appears in [1]. The features highlighted in Table 3.1 are (1) the different designs included in the benchmark; (2) the functionality implemented by each design; (3) the lines of code that are used to implement each design (LOC); (4) the bug description of bugs present in each design; (5) the number of bugs in each design; and (6) the category of the defect as categorized by [1] which indicates the complexity of the bug. Category '1' indicates that the bug is relatively easy to detect and fix and related to simple aspects of the design. For example, when the output of a 4-bit multiplexer (mux) is assigned

to a 1-bit output wire. This bug requires basic knowledge of logic design to detect and fix. Category '2' indicates that a bug requires more effort to diagnose, comprehend, and fix. For example, an incorrect finite state machine transition requires deep analysis of the design specifications along with the simulation output to understand and fix the bug.

In Table 3.2, the first column contains the project names, the second column includes the bug descriptions, and the third column lists the corresponding simplified buggy file names. This table was added to provide simplified references for the buggy file names, making it easier to refer to and discuss them in the following chapters. The project names and their corresponding bug descriptions are consistent with the information presented in Table 3.1. Additionally, the benchmark and the code used in the CirFix paper [1] are available on their GitHub repository [55]. The benchmark used in this thesis, as presented in Table 3.1 and Table 3.2, are derived from this repository

3.3 Experimental Setup

The framework described in Section 3.1 was implemented using Python (version 3.9.19), PyVerilog (version 1.2.1), and Synopsys VCS for simulation. The VCS environment used for the experiments was configured with the following specifications: script version U-2023.03, compiler version VCS U-2023.03-SP1_Full64, and executed on a Linux 4.18.0-553.16.1.el8_10.x86_64 operating system on a linux64 machine type. For the language model component, we utilized OpenAI's `gpt-4o-2024-05-13` model [32] with its default out-of-the-box parameters.

The most important parameters influencing the behavior of the language model are the temperature and top-p, both of which have default values of 1, according to OpenAI's documentation. We chose to use the default parameters for the language model based on prior research showing that parameter tuning does not consistently improve performance across different scenarios. In [47], experiments with various models and parameter settings found that no single configuration, such as temperature or top-p, performs best universally. Similarly, in [4], experiments conducted on a range of models revealed that for GPT-4 and GPT-3.5-turbo, variations in the temperature parameter seem to have no impact on success rates on the hardware-related problems, which remained stable regardless of the settings. These findings led us to adopt the default parameters in our experiments. The experiments were conducted on a server equipped with an Intel(R) Xeon(R) CPU E5-2690 v4

running at 2.60GHz and 256 GiB of RAM.

3.3.1 Terminology

A comprehensive list of terminology used throughout the thesis is provided in Table 3.3. This table serves as a reference for understanding the key terms and concepts that will be discussed in the following chapters and their associated experiments.

Table 3.3: Key Terms Used in the Thesis

Key Terms	Zero-shot Technique	Few-shot Technique	Feedback Technique
Benchmark	Refers to the entire set of examples used for evaluation (as described in Table 3.1).		
Example	An individual instance from the benchmark. In our study, there are 32 examples in total.		
Run	Refers to processing the entire benchmark through the LLM. A single run involves attempting to fix each of the 32 examples in the benchmark.		
Iteration	N/A		Number of trials conducted for each run. In this study, 5 iterations are performed per run to ensure consistency.

3.3.2 Number of Runs and Experiment Design

In this thesis, we evaluate two main techniques: **zero-shot** and **few-shot**, which are discussed in Chapter 4 and Chapter 5, respectively. Each technique consists of 6 strategies, and each strategy includes two prompt variations (structured and unstructured).

For each prompt variation, we ran the experiments 5 times. Therefore, for each technique, the total number of runs is computed as follows:

$$\text{Total runs per technique} = 6 \times 2 \times 5 = 60 \,\text{runs per technique.}$$

Runs Per Example vs. Whole Benchmark

Our benchmark consists of 32 examples. In each run, the model is executed on all 32 examples, meaning that each individual example in the benchmark is considered during every run. Hence, for each technique, we calculate the total number of LLM invocations as follows:

$$\text{Total LLM invocations per technique (for all examples)}$$

$$32 \times 60 = 1920 \,\text{LLM calls for the entire benchmark.}$$

This means that for each technique, the model is invoked 1920 times across all examples in the benchmark.

Feedback Technique Runs

For the feedback technique, we conducted 5 runs, with each run containing a maximum of 5 iterations. Thus, the total number of iterations for the feedback technique is:

$$\text{Total iterations for the feedback technique} = 5 \times 5 = 25 \,\text{iterations for the entire benchmark.}$$

Since each iteration applies feedback to all 32 examples in the benchmark, the maximum number of LLM invocations per iteration is 32. Therefore, the total maximum number of LLM calls for the entire benchmark across all iterations is:

$$\text{Total LLM invocations for the feedback technique}$$

$$32 \times 25 = 800 \,\text{LLM invocations for the entire benchmark.}$$

This means that for the feedback technique, the model is invoked a maximum of 800 times across all 32 examples in the benchmark over 25 iterations.

3.4 Evaluation Metrics

In this section, we present the evaluation metrics used to assess the performance of the various techniques and strategies employed in our experiments. These metrics are designed to quantify various aspects of the bug-fixing process, including correctness and efficiency. The use of these metrics allows for a systematic comparison of the results, facilitating an objective assessment of the proposed approach in relation to existing methods in the field.

3.4.1 Pass@k

This work utilizes the *pass@k* metric to evaluate model performance, as introduced in [56]. This metric quantifies the probability of a correct solution appearing within the top-k model attempts. The metric is defined as:

$$\text{pass}\,@k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{3.1}$$

In this formula, n represents the total number of attempts, c is the number of correct solutions, and k is the number of top attempts considered. The expectation $\mathbb{E}$ is calculated across all problems to provide an overall measure of performance.

The value of Pass@k ranges from 0 to 1:

- **Maximum Pass@k**: When all k top attempts are correct solutions, Pass@k equals 1, indicating perfect performance.

- **Minimum Pass@k**: When none of the top-k attempts are correct solutions, Pass@k equals 0, indicating the worst performance.

Thus, a higher Pass@k value indicates better performance, as it reflects a higher probability of a correct solution appearing within the top k attempts. Lower values indicate poorer performance.

To ensure consistency and reproducibility, we employed the methodology and code from Chen et al. [56] to automate the calculation of this metric.

3.4.2 Percentage Pass

The **Percentage Pass** metric represents the fraction of all runs using a given strategy where the generated design compiles, but is not necessarily fixed. This metric provides insight into the syntactic correctness of the generated designs, independent of functional correctness.

3.4.3 Total Average FC Change

The **Total Average FC Change** is calculated by subtracting the FC value of the buggy code without any fixes from the mean FC of the designs produced in each of the five runs, and then averaging these calculations across all benchmarks. This metric provides a measure of the overall improvement in functional correctness achieved by the model.

3.4.4 Average Minimum FC Change

The **Average Minimum FC Change** metric is calculated by subtracting the FC value of the buggy code without any fixes from the minimum FC of the designs produced in each of the five runs, and then averaging these calculations across all benchmarks. This metric provides an understanding of the worst-case improvements (least FC) in functional correctness, highlighting the minimal improvements achieved by the model across all benchmarks.

3.4.5 Average Maximum FC Change

The **Average Maximum FC Change** metric is determined by subtracting the FC value of the buggy code without any fixes from the maximum FC of the designs produced in each of the five runs, and then averaging these calculations across all benchmarks. This metric provides an understanding of the best-case improvements (highest FC) in functional correctness, highlighting the maximum improvements achieved by the model across all benchmarks.

3.4.6 Number of Benchmarks Fixed

The **Number of Benchmarks Fixed** metric indicates the number of designs where at least one of the five runs produces a design that achieves FC = 1. This metric assesses the model's ability to generate fully correct solutions at least once within the given attempts.

3.4.7 Average Number of Benchmarks Fixed

The **Average Number of Benchmarks Fixed** metric represents the mean number of times a generated design achieves FC = 1 across all runs. This provides a measure of the consistency of the model in producing fully correct fixes.

3.4.8 Total Cost ($)

The **Total Cost ($)** metric indicates the total cost of running five trials for each benchmark design. The cost is calculated based on the number of tokens used by the model (gpt-4o-2024-05-13). For every million input tokens, the cost is $5.00, and for every million output tokens, the cost is $15.00.

This metric provides an understanding of the computational resources required for the repair process and the associated costs of running the model across all benchmarks.

Chapter 4

Zero-Shot Technique: LLM Automatic Repair Performance

This chapter investigates the effectiveness of zero-shot learning techniques for automatic bug repair using LLMs, which attempt to identify and fix RTL functional bugs without prior knowledge of the bug type or location.

We focus on answering the research question:

RQ1: How effective are LLMs at performing automatic hardware repairs without prior examples (zero-shot)?"

To address this question, we use the framework outlined in Chapter 3, which is designed to investigate LLM performance on a benchmark derived from the state-of-the-art Cirfix paper [1]. This framework enables us to evaluate how effectively the model can detect and resolve functional bugs based solely on its general knowledge.

Our experiments involve six prompting strategies to assess the automatic repair capabilities of LLMs in a zero-shot context. These strategies incorporate varying levels of context and instructions, categorized into two formats: **unstructured** prompts, presented as free-flowing narratives, and **structured** prompts, organized into distinct sections. Each approach is designed to guide the LLM in generating effective bug fixes without relying on assumptions about the nature of the bug descriptions.

Insights from this chapter clarify the capabilities and limitations of LLMs in performing automatic

RTL bug repairs under zero-shot conditions, ultimately contributing to the hardware design process. Further details on the strategies and prompt variations are elaborated in Section 4.1 and Section 4.2, respectively.

4.1 Strategies

4.1.1 Strategy 0 (S0): Baseline Strategy

Strategy 0 serves as a baseline to evaluate the model's performance with minimal context. In this approach, only the buggy code and a general instruction to fix any detected errors are provided to the LLM. This strategy aims to determine how effectively the model can identify and correct the bug with only the raw code as input, relying solely on its pretrained knowledge and pattern recognition abilities, as similarly explored in prior work [4, 10]. By using Strategy 0, we establish a comparison point for evaluating the benefits of additional contextual information introduced in the subsequent strategies.

4.1.2 Strategy 1 (S1): Incorporating Simulation Tool Feedback

In Strategy 1, the LLM receives the buggy code along with additional context in the form of a *mismatch list*, which is generated by the *Mismatch List Generation Engine*, as explained in Section 3.1.4. This engine runs the buggy Verilog code through a simulation tool, such as VCS, and compares its output with the output of an oracle (the correct, bug-free code). The mismatch list identifies discrepancies in key elements like input/output ports, registers, wires, and data types, highlighting areas where the buggy code diverges from expected behavior. Although this list does not diagnose the root cause of the bug, it provides the LLM with clues about the areas in need of attention, potentially helping it focus its corrective efforts more effectively. As shown in Listing 4.1 and Listing 4.2, the mismatch list section is highlighted in red within the prompts.

4.1.3 Strategy 2 (S2): Incorporating Differences between Buggy Simulation output and Oracle output

In Strategy 2, the LLM receives the buggy code along with additional information about the discrepancies between the output of the buggy simulation and the oracle's expected output at each time step. These discrepancies are generated by the *Simulation-Oracle Difference Engine*, as explained in Section 3.1.4. After simulating the buggy code, the output signals are compared against the oracle file, which represents the correct output from the verified, bug-free version. The differences are formatted as shown in Fig. 4.1

```
At time step T with inputs:
(<input1> = <actual_input_value>, <input2> = actual_input_value>, ...):

The expected output for <output1> from the correct code should be <expected_output_value
    >, but the actual value is <actual_output_value> in the buggy code.
```

Figure 4.1: Example of formatted differences for Strategy 2

This input format, developed as part of our approach, aims to explore whether providing more detailed information can enhance the LLM's performance. By extracting relevant data automatically from the simulation output, we provide the model with a step-by-step understanding of how the buggy code's behavior deviates from the correct output under various conditions, potentially guiding it toward a more precise fix. As shown in Listing 4.1 and Listing 4.2, the differences between buggy simulation output and Oracle output section is highlighted in blue within the prompts.

4.1.4 Strategy 3 (S3): Utilizing Fault Localization

Strategy 3 provides the LLM with the buggy code, along with fault localization information generated by the automated *Fault Localization Engine*, as explained in Section 3.1.4. This engine analyzes the code and identifies specific lines that are most likely to be associated with erroneous behavior. These lines are highlighted as potential sources of the bug, although the actual error may also lie in other parts of the code. By including this targeted information, the strategy aims to guide the LLM in focusing on areas of the code that are most likely contributing to the bug, potentially increasing the efficiency of the bug-correction process. As shown in Listing 4.1 and Listing 4.2, the fault localization section is highlighted in green within the prompts.

4.1.5 Strategy 4 (S4): Combined Information from Simulation Differences and Fault Localization

Strategy 4 merges the input elements from Strategy 2 and Strategy 3, providing the LLM with both simulation output differences and fault localization insights. By combining specific signal discrepancies (as in Strategy 2) with a list of potentially faulty lines (from Strategy 3), this approach supplies the model with a more comprehensive view of the bug's impact. The goal is to improve the model's debugging performance by equipping it with multiple contextual clues to aid in identifying and correcting the error.

4.1.6 Strategy 5 (S5): Comprehensive Information

In Strategy 5, the LLM is given all available contextual information from Strategies 1 through 3. This includes the buggy code, mismatch list (from Strategy 1), simulation output differences (from Strategy 2), and fault localization information (from Strategy 3). By supplying the model with the full scope of diagnostic information, we seek to provide it with every possible advantage to correctly identify and fix the bug. This comprehensive approach is designed to assess whether the combination of all input types enhances the model's ability to effectively resolve the issues in the code. As shown in Listing 4.1 and Listing 4.2, these listings present examples of Strategy 5 for structured and unstructured prompts.

4.2 Prompt Variations

As there are not yet any *universal* best practices for prompt engineering, we adopted the OpenAI documentation guidelines [32] and those from a deeplearning.ai course [57]. We also drew inspiration from prior work by Ahmad et al. [4]. Our primary principles were to write clear and specific instructions and to allow the LLM sufficient time to process the task at hand. In our approach, we ensure that every piece of input information is clearly explained. After providing an explanation, the input information extracted from the simulation and code is enclosed within clear delimiters. This allows the LLM to focus on and better understand the input information given to it, aiding in generating accurate bug fixes. We found that placing instructions at the beginning of

Listing 4.1: Structured Prompt for Strategy 5

```
### Your Task: ###

Fix the buggy Verilog code using the input information provided below to help you
    identify and resolve the bug. Be sure to follow the instructions and the response
    format mentioned below to ensure accurate corrections.

### Instructions: ###

1. Follow the input information provided carefully to identify and fix the bug.
2. Focus on making one or more simple changes in the lines where the bug might originate
    .
3. Ensure that the overall logic and structure of the code remain unchanged.

### Response Format: ###

Provide the complete, functional Verilog code without any extra words, comments, or
    explanations.
```

```
 ### Input Information: ###

1. After comparing the output of the correct and buggy code, a mismatch was identified
    between the two outputs. The elements in the code responsible for this mismatch have
     been identified as potential sources of the bug. These elements, which may include
    input ports, output ports, registers, wires, or data types, are: <<<{
    tmp_mismatch_set}>>>
```

```
2. After simulating the buggy Verilog code with the testbench, the output signal at each
     time step was compared to the output signal values from the oracle file.
The oracle file represents the correct expected output from the bug-free version of the
    code.
The differences are formatted as follows:
For example:
At time step T with inputs: <input1> = <actual_input_value>, <input2> = <
    actual_input_value>, ...
The expected output for <output1> from the correct code should be <expected_output_value
    >, but the actual value is <actual_output_value> in the buggy code.
The comparison revealed the following differences:
<<< \n {prompt_text} \n >>>
```

```
3. Fault localization has been performed to identify lines that may be contributing to
    the bug. While the following lines are suspected as potential sources, the bug may
    exist in other parts of the code as well. Consider all possible sources of the error
     when providing your corrected version of the Verilog code.
Lines identified by fault localization that might contain the bug are:
<<< \n {implicated_lines_summary} \n >>>
```

```
4. The buggy Verilog code that needs to be fixed:
<<< {buggy_verilog_code} >>>
```

the prompt yielded the best results, making it easier for the LLM to follow. We implemented two main variations in our prompts: the **structured** format and the **unstructured** format.

As shown in Listing 4.1, the **structured** format prompt is methodically divided into specific sections to guide the LLM clearly:

- **Your task**: Here, we clearly state the main objective for the LLM to focus on.
- **Instructions**: In this section, we provide a detailed list of instructions that the model must follow.
- **Output format**: We specify and explain the desired format for the output, ensuring clarity in the response we expect.
- **Input information**: We include any necessary input data that will assist in locating and fixing the bug. The input information varies depending on the chosen strategy and is structured to guide the LLM effectively.

As shown in Listing 4.2, the **unstructured** format follows the same organizational logic but presents the prompt in a paragraph-like format without explicitly separating sections with titles like "Your Task" or "Instructions." By testing both structured and unstructured approaches, we aim to investigate which method yields better results for bug fixing, with the structured format providing clear guidance and the unstructured format offering a more narrative-driven context.

4.3 Experimental Results

After implementing the experiments using our framework (Section 3.1), we present the results obtained from applying the zero-shot technique. The performance of the LLMs is evaluated based on various metrics detailed in the Evaluation Metrics section (Section 3.4). These include pass@k, Percentage Pass, Number of Benchmarks Fixed, Total Average FC Change, Average Minimum and Maximum FC Change, Average Number of Benchmarks Fixed, and Total Cost.

We compare the results obtained with minimal context against those with additional instructions, allowing us to draw insights into how these factors influence the repair capabilities of LLMs. The

Listing 4.2: Unstructured Prompt for Strategy 5

```
Your task is to fix the given buggy Verilog code and provide the complete, functioning
    version without adding any extra words, comments, or explanations in your response.
 Your goal is to make one or more simple changes, focusing on the lines where the bug
     might originate, while ensuring that the overall logic and structure of the code
     remain unchanged.
```

```
After comparing the output of the correct and buggy code, a mismatch was identified
    between the two outputs. The elements in the code responsible for this mismatch have
     been identified as potential sources of the bug. These elements, which may include
    input ports, output ports, registers, wires, or data types, are: <<<{
    tmp_mismatch_set}>>>
```

```
After simulating the buggy Verilog code with the testbench, the output signal at each
    time step was compared to the output signal values from the oracle file.
The oracle file represents the correct expected output from the bug-free version of the
    code.
The differences are formatted as follows:
For example:
At time step T with inputs: <input1> = <actual_input_value>, <input2> = <
    actual_input_value>, ...
The expected output for <output1> from the correct code should be <expected_output_value
    >, but the actual value is <actual_output_value> in the buggy code.
The comparison revealed the following differences:
<<< \n {prompt_text} \n >>>
```

```
Fault localization has been performed to identify lines that may be contributing to the
    bug. While the following lines are suspected as potential sources, the bug may exist
     in other parts of the code as well. Consider all possible sources of the error when
     providing your corrected version of the Verilog code.
Lines identified by fault localization that might contain the bug are:
<<< \n {implicated_lines_summary} \n >>>
```

```
The buggy Verilog code that needs to be fixed is as follows:
<<< \n{buggy_verilog_code}\n >>>.
```

findings from this analysis are critical for understanding the strengths and weaknesses of the zero-shot approach in automatic RTL bug repairs.

4.3.1 Unstructured Approach Results

The performance of the different strategies is presented in Table 4.1, which displays metrics such as Pass@1, Percentage of Pass, and Total Cost. Additional metrics, including Total Average FC Change, Average Minimum FC Change, and Average Maximum FC Change, are visualized in Fig. 4.2. Fig. 4.3 further illustrates the Total Number of Benchmarks Fixed and the Average Number of Complete Fixes. Together, these metrics provide a comprehensive basis for comparing the effectiveness and reliability of each strategy in automatic RTL bug repair using LLMs.

Starting with the **Pass@1** results, calculated using the equation shown in Equation 3.1, Strategy

Table 4.1: Unstructured Zero-Shot Results by Strategy

Metric	**Strategy 0**	**Strategy 1**	**Strategy 2**	**Strategy 3**	**Strategy 4**	**Strategy 5**
Pass@1	0.55625	0.39375	0.46875	0.45	0.5125	0.45625
Percentage of pass	96.25%	98.75%	97.50%	98.13%	96.25%	95.63%
Total Cost ($)	$4.39	$4.53	$5.72	$4.73	$6.05	$6.14

0 achieves the highest pass@1 score of 0.55625, highlighting its effectiveness compared to other strategies. Strategy 4 follows closely with a pass@1 of 0.5125, demonstrating strong performance. Strategies 2 (0.46875), 5 (0.45625), and 3 (0.45) show moderate scores, indicating reasonable performance but not as high as Strategies 0 and 4. Strategy 1 has the lowest pass@1 score at 0.39375, suggesting it is the least effective among the strategies assessed.

In terms of the **Percentage of Pass**, Strategy 1 achieves the highest pass percentage at 98.75%. This indicates a strong level of syntactic correctness, suggesting that this strategy is highly effective in generating compilable designs, even though its overall pass@1 score is lower than some others. Strategies 3 and 2 also demonstrate solid performance with pass percentages of 98.13% and 97.50%, respectively, reflecting their reliability in producing compilable outputs. Strategies 0 and 4 both attain a pass percentage of 96.25%, indicating that they also maintain a high level of syntactic correctness. Conversely, Strategy 5 ranks the lowest in this metric with a pass percentage of 95.63%, suggesting it is slightly less reliable in generating compilable designs compared to the other strategies. Overall, while the pass percentage highlights the syntactic correctness of the generated designs, it is essential to consider it alongside other metrics to evaluate the strategies' effectiveness fully.

When considering **Total Cost**, Strategy 0 has the lowest cost at $4.39, closely followed by Strategy 1 at $4.53 and Strategy 3 at $4.73. Strategies 2, 4, and 5 incur higher costs, at $5.72, $6.05, and $6.14, respectively. While higher costs may reflect additional benefits in performance, they also suggest a trade-off that needs to be considered.

As shown in Fig. 4.2, the **Total Average FC Change** reveals that Strategy 0 achieves the highest value of 0.28, indicating the most significant overall change across all benchmarks. This suggests Strategy 0 may lead to more substantial functional corrections on average. Strategies 1 and 4 follow closely with average changes of 0.25, demonstrating moderate levels of adjustment. Strategies 2, 3, and 5 have a slightly lower average change of 0.24, suggesting that these strategies make smaller overall adjustments by comparison.

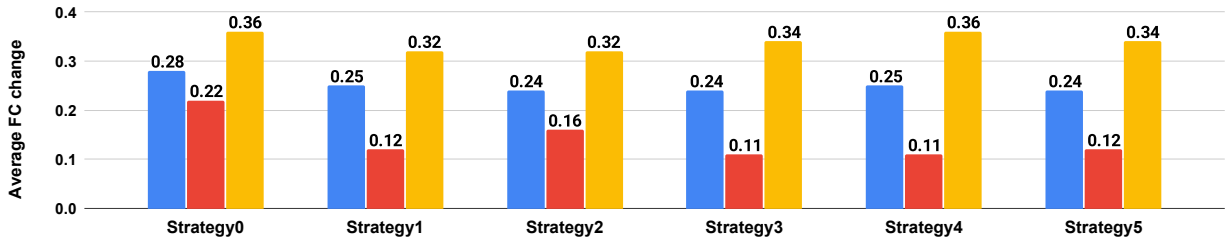

Figure 4.2: Comparison of total average FC change (blue), average minimum FC change (red), and average maximum FC change (yellow) across all benchmarks. The x-axis represents different strategies, and the y-axis shows the average FC change.

The **Average Minimum FC Change** metric highlights the range of lower-bound adjustments achieved by each strategy. Strategy 0 again leads with an average minimum change of 0.22, indicating that even the least effective adjustments from this strategy tend to be more impactful than those of the others. Strategies 1 and 5 both yield a lower average minimum change of 0.12, while Strategy 2 performs slightly better at 0.16. Strategies 3 and 4 show the lowest average minimum changes at 0.11, indicating that their least effective fixes result in minimal functional correction.

For the **Average Maximum FC Change**, which represents the most substantial adjustments made by each strategy, Strategies 0 and 4 are tied at the highest value of 0.36. This indicates that these two strategies can achieve more significant functional improvements in specific cases. Strategies 3 and 5 closely follow with maximum changes of 0.34, demonstrating their potential for strong adjustments when necessary. Strategies 1 and 2 both have slightly lower maximum values of 0.32, which may reflect a somewhat limited capacity for extreme corrections.

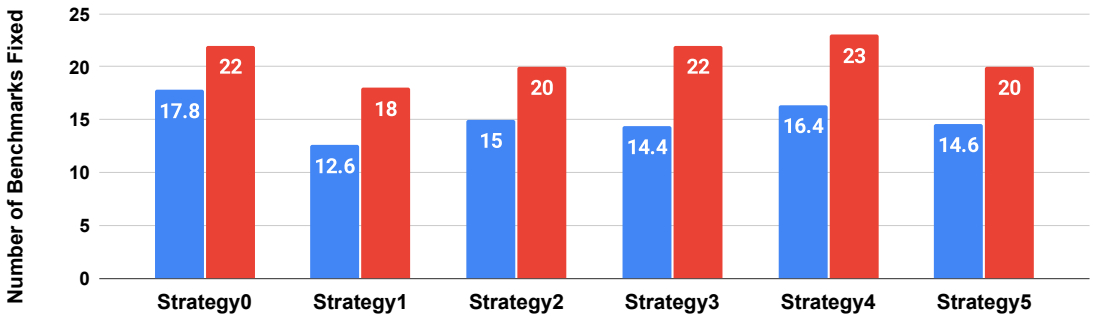

Figure 4.3: Number of benchmarks fixed (red) and the average number of complete fixes (blue).(Number of Benchmarks Fixed vs. Strategy)

As shown in Fig. 4.3, in terms of the **Average Number of Complete Fixes**, Strategy 0 performs best with an average of 17.8 complete fixes, reflecting its effectiveness in addressing issues.

Strategy 4 also performs well with 16.4 fixes, while Strategies 3, 2, and 5 range between 12.6 and 15 complete fixes. Strategy 1 is the least effective here, with only 12.6 complete fixes on average.

Finally, for the **Number of Benchmarks Fixed**, Strategy 4 again stands out with 23 successful fixes, closely followed by Strategies 0 and 3, each with 22. Strategy 2 and Strategy 5 demonstrate moderate performance with 20 fixed bugs, while Strategy 1 has the lowest count at 18.

Overall, while each strategy exhibits unique strengths, **Strategy 4** emerges as the best-performing choice in this experiment. By combining fault localization and the differences between the buggy simulation output and oracle output, Strategy 4 achieves a high pass@1 score and corrects a large number of bugs, indicating effective execution. However, this strategy also comes at a higher cost.

In contrast, **Strategy 0**, which utilized only the buggy code in the prompt, stands out as the second-best option. Despite its simplicity, it produced strong results on pass@1, demonstrating that effective bug repair can be achieved with minimal additional information.

Strategy 1, which integrated the buggy code with a mismatch list, was the least effective, showing that additional information does not always lead to better outcomes. Similarly, while **Strategies 2 and 3**—using differences between buggy simulation output and oracle output, and fault localization, respectively—could enhance results, they also introduced higher costs.

Lastly, **Strategy 5**, which combined all information from Strategies 1, 2, and 3, showed that while comprehensive data may provide some advantages, it did not significantly outperform Strategy 0. Overall, these findings suggest that while extra information can be useful, it must be balanced against the associated costs and the diminishing returns on performance improvements.

4.3.2 Structured Approach Results

This subsection, similar to Section 4.3.1, details the performance of the different strategies in Table 4.2, which showcases key performance indicators such as Pass@1, Percentage of Pass, and Total Cost. Additional metrics, including Total Average FC Change, Average Minimum FC Change, and Average Maximum FC Change, are shown in Fig. 4.4. Furthermore, Fig. 4.5 illustrates the Total Number of Correctly Fixed Bugs and the Average Number of Complete Fixes. Together, these metrics provide a solid foundation for evaluating the effectiveness and efficiency of each strategy in

the context of automatic RTL bug repair.

Table 4.2: Structured Zero-Shot Results by Strategy

Metric	Strategy 0	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
Pass@1	0.50625	0.3875	0.475	0.45	0.45625	0.425
Percentage of pass	98.75%	98.75%	98.13%	98.13%	96.88%	98.13%
Total Cost ($)	$4.28	$4.48	$5.53	$4.67	$5.98	$6.15

Starting with the **Pass@1** results, calculated using the equation shown in Equation 3.1, Strategy 0 has the highest pass@1 score of 0.50625, indicating it is the most effective overall. Strategy 2 follows with a pass@1 of 0.475, showing strong performance. Strategy 4 achieves a pass@1 of 0.45625, placing it third, while Strategy 3 has a pass@1 of 0.45, reflecting moderate performance. Strategies 5 (0.425) and 1 (0.3875) have lower pass@1 scores, with Strategy 1 being the least effective.

The **Percentage of Pass** indicates that both Strategies 0 and 1 achieve the highest percentage at 98.75%, signifying a strong level of syntactic correctness. However, despite this high percentage, Strategy 1's lower pass@1 suggests it does not perform as effectively overall. Strategies 2 and 3 also demonstrate reliable pass percentages of 98.13%, while Strategy 4 has a pass percentage of 96.88%. Strategy 5 ranks lowest in this metric with a pass percentage of 98.13%, indicating slightly lower reliability.

Regarding **Total Cost**, Strategy 0 has the lowest cost at $4.28, followed by Strategy 1 at $4.48. Strategies 2 and 3 incur moderate costs of $5.53 and $4.67, respectively. Strategy 4's cost of $5.98 reflects its enhanced performance metrics, while Strategy 5 is the most expensive at $6.15. This indicates a trade-off between cost and performance that should be considered in strategy selection.

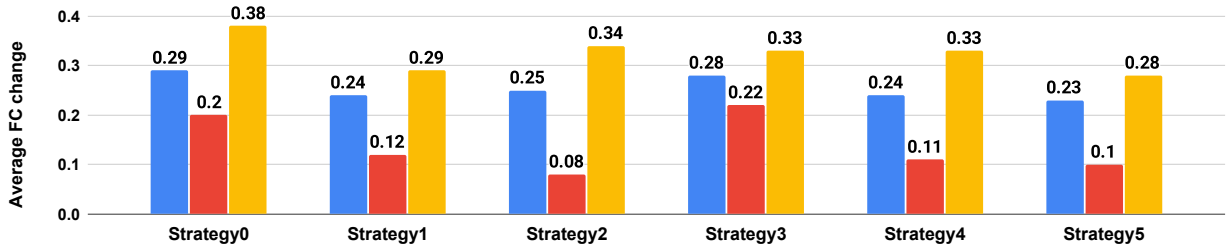

Figure 4.4: Comparison of total average FC change (blue), average minimum FC change (red), and average maximum FC change (yellow) across all benchmarks. The x-axis represents different strategies, and the y-axis shows the average FC change.

As shown in Fig. 4.4, the **Total Average FC Change** reveals that Strategy 0 leads with a Total Average FC change of 0.29, indicating its effectiveness in improving functional correctness.

Strategy 3 follows closely with 0.28, showcasing its strong performance. Strategies 2 (0.25), 1 (0.24), and 4 (0.24) also demonstrate substantial improvements. Strategy 5 has the lowest Total Average FC change at 0.23.

The **Average Minimum FC Change** metric highlights the range of lower-bound adjustments achieved by each strategy. Strategy 3 stands out with the highest minimum FC change of 0.22, indicating its consistent performance even at lower improvements. Strategy 0 is next with 0.2, while Strategy 1 shows 0.12. Strategies 4 and 5 follow closely with 0.11 and 0.1 respectively. Strategy 2 records the lowest minimum FC change at 0.08

For the **Average Maximum FC Change**, which represents the most substantial adjustments made by each strategy, Strategy 0 again leads with a maximum FC change of 0.38, showcasing its potential for significant improvements. Strategy 2 follows with 0.34, while Strategies 3 and 4 both show 0.33. Strategy 1 and Strategy 5 have lower values at 0.29 and 0.28.

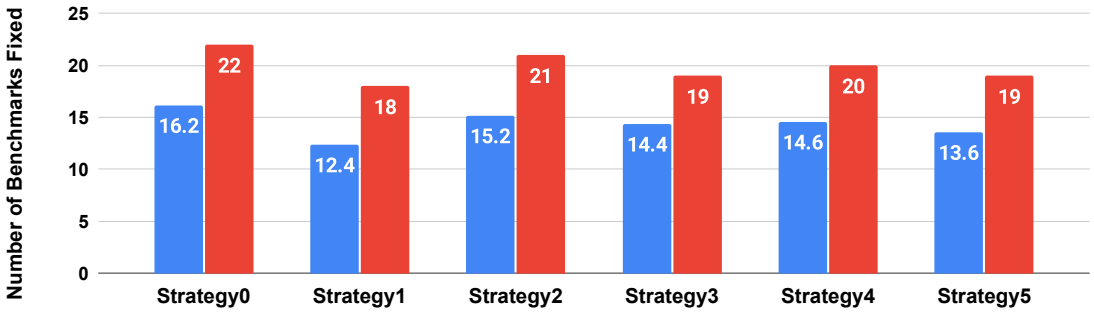

Figure 4.5: Number of benchmarks fixed (red) and the average number of complete fixes (blue).(Number of Benchmarks Fixed vs. Strategy)

As shown in Fig. 4.5, in terms of the **Average Number of Complete Fixes**, Strategy 0 excels with an average of 16.2 complete fixes, demonstrating its capability to effectively solve problems. Strategy 2 performs well with 15.2 complete fixes, followed by Strategy 4 with 14.6 and Strategy 3 with 14.4. Strategies 5 and 1 have the lowest number of complete fixes, with 13.6 and 12.4 respectively, indicating they are less successful in fully resolving issues.

Finally, for the **Number of Benchmarks Fixed**, Strategy 0 tops the list with 22 correctly fixed bugs, highlighting its overall effectiveness. Strategy 2 follows with 21, while Strategy 4 has 20. Strategy 3 shows 19 fixed bugs, tying with Strategy 5. Strategy 1 has the lowest number at 18, reaffirming its struggles in bug fixing.

In conclusion, After evaluating all the metrics, **Strategy 0**, which only has the buggy code, consistently outperforms others, making it the most effective strategy. **Strategy 2**, which includes buggy code plus differences between simulation and oracle output, follows closely in performance. **Strategy 3**, using fault localization alongside the buggy code, ranks third. **Strategy 1** (buggy code plus mismatch list) underperforms and is the least effective. **Strategies 4 and 5**, combining multiple information inputs, show balanced performances but do not surpass **Strategy 0**.

4.3.3 Discussion and Observations

Table 4.3 presents five primary bug categories we identified after examining the unsolved outputs from the LLM model. By analyzing the differences between the original buggy code and the model's output, we found that the model's changes—while failing to fix the bugs—could be classified into these categories. **Change in Logic (L)** involves modifications that alter the overall implementation or logic of the code, such as switching algorithms or restructuring control flow. **Change in If/Else If Condition (C)** includes adjustments to conditions within if or else if statements, such as adding, removing, or modifying conditions. **Add, Remove, or Change in Assignment (A)** pertains to changes in value assignments to variables, covering both new assignments and modifications to existing ones. **Add or Change in Sensitivity List (S)** applies to adjustments in the always block sensitivity list, impacting triggers without removing signals. Finally, **No Change (N)** indicates cases where the model left the buggy code unaltered. These codes (L, C, A, S, N) will be used throughout the following discussion to refer to each category.

The unstructured and structured Zero-shot results tables, shown in Table 4.4 and Table 4.5, respectively, display results for examples that remained unfixed across all strategies, meaning each example had an FC below 1 in all five runs for every strategy. In these tables, we use the simplified bug names introduced in Table 3.2, which will also be used throughout this section for consistency and clarity. These tables document the specific modifications the model applied to these examples, leading to an FC value below 1. We categorized these modifications into five types, as outlined in Table 4.3.

This section analyzes the performance of the LLM model in resolving Verilog code bugs across various strategies, comparing the results to the initial bug classifications outlined in the benchmark

Table 4.3: Bug Categories and Changes

Bug Category	**Description**	**Examples**
Change in Logic (L)	Modifications that change the overall implementation or logic of the code, such as using a different algorithm, restructuring the logic flow, or replacing "don't cares" (e.g., `X` or `-`) with specific values like `0` or `1`.	- Changing from a sequential implementation to a combinational one - Replacing a case statement with a series of if-else statements - Converting "don't cares" to defined values
Change in If/Else If Condition (C)	Adding, removing, or changing conditions in if or else if statements, including adjustments that alter the logic of conditions.	- Changing `if (a == 1)` to `if (a == 2)` - Adding a new condition like `else if (b == 3)` - Removing an `else if` condition entirely, e.g., removing `else if (c == 4)` - Modifying an `else if` condition, e.g., changing `else if (x > y)` to `else if (x >= y)`
Add, Remove, or Change in Assignment (A)	Changes in how values are assigned to variables, which include modifying, adding, or removing assignment statements.	- Changing `assign i <= 1` to `assign i <= 2` - Adding a new assignment like `assign j <= a & b`
Add or Change in Sensitivity List (S)	Adding new signals or modifying existing ones in the sensitivity list of an always block, affecting what triggers it without removing signals.	- Changing `always @(a)` to `always @(a or b)` - Adding new signals like `always @(posedge clk or posedge rst)`
No Change	Indicates the buggy code remained unaltered by the model, with no changes made to the original code.	- Buggy code retained as-is with no modifications

study by [1]. The benchmark divides the files into two main categories: category 1 (19 files) and category 2 (13 files). However, the LLM model also faced difficulty with category 1 files, suggesting that these bugs are not straightforward for the model to understand and resolve, potentially indicating a difference in how the model approaches functional bugs compared to the benchmark classifications.

Based on the analysis of the **unstructured zero-shot** results in Table 4.4, we observe that the model's modification attempts—categorized as S, L, C, A, or No Change, as explained in

Table 4.4: Unstructured Zero-Shot Results. Changes are categorized as Add, Remove, or Change in Assignment, labeled as (A); Change in If/Else If Condition, labeled as (C); Change in Logic, labeled as (L); Change in Sensitivity List, labeled as (S); and No Change indicates the buggy code remains unchanged.

Simplified Buggy File Name	**Cat**	**Strategy 0**	**Strategy 1**	**Strategy 2**	**Strategy 3**	**Strategy 4**	**Strategy 5**
counter_sensitivity_issue	1						L,S
counter_increment_error	1			L	L,S	L	
flipflop_conditional_error	1	C,S	S	S	C,S		
flipflop_branch_swap	1	S	C,S	S			
fsm_state_assignment_missing	2	S	A,S		S		S
lshift_conditional_error	1		L,S				
i2c_sensitivity_issue	2		No change			L,A	L,A
i2c_address_assignment_error	2		A	A			A
sha3_loop_boundary_issue	1	A,C	A,C	A,C	A,C,S	L,A	L,A,C
sha3_wire_assignment_error	2	L,A	L,A,C	L,A,C	L,A,S	L,A,C	L,A,C
sha3_buffer_overflow_skipped	2	A,C	C	A	A,C	A	C
sha3_bitwise_negation_issue	1	L,C	A,C	L,A	A,C		
tatepairing_bitshift_logic_error	1	A,C	A,C	A,C			A,C
tatepairing_operator_mismatch	1			A	A,C	A,C	A,C
tatepairing_module_instantiation	2	L,A	L,A	A	A,C	A,C	A,C
reedsolomon_register_size_error	1		L,A			A	A
sdram_case_statement_issue	2	L,A	L,A	L,A	L,A	L,A	L,A

Table 4.3—frequently fell into predictable patterns. Analyzing these changes over five runs per strategy, it is clear that the model often made similar types of changes across all strategies, without yielding successful fixes. Among these categories, the most frequent type of unsuccessful modification was **Add, Remove, or Change in Assignment (A)**, appearing 48 times. This was followed by **Change in If/Else If Condition (C)**, which appeared 29 times without achieving a solution. **Change in Logic (L)** was attempted 26 times but did not resolve the issues, while **Add or Change in Sensitivity List (S)** was applied unsuccessfully 16 times. The **No Change** category appeared only once, indicating that the model typically attempted some form of modification rather than leaving the code unchanged.

After reviewing the model's attempts to resolve bugs across all strategies, we observed consistent behavior for certain files. These files can be grouped into three main categories: unresolved bugs, fully resolved bugs, and resolved bugs with new issues introduced. The following outlines each category, with observations based on the model's performance across different strategies:

1. **Unresolved Bugs Across All Strategies**: The following files remained unresolved across all strategies, despite multiple attempts at modification. These issues persisted across different strategies and iterations, indicating challenges in detecting and fixing these bugs:

   - **SHA3 Project**: sha3_loop_boundary_issue, sha3_wire_assignment_error, sha3_buffer_overflow_skipped
   - **Tate Pairing Project**: tatepairing_module_instantiation
   - **SDRAM Controller Project**: sdram_case_statement_issue

2. **Resolved Bugs Across All Strategies**: The following files were successfully fixed in at least one run within each strategy, demonstrating the model's ability to address these issues across different configurations:

   - **Decoder Project**: decoder_numeric_error, decoder_assignment_error
   - **Counter Project**: counter_reset_issue
   - **FSM Project**: fsm_case_statement_issue, fsm_state_sensitivity_issue, fsm_blocking_assignment_error
   - **LShift Project**: lshift_blocking_issue, lshift_sensitivity_issue
   - **Mux Project**: mux_numeric_issues, mux_hex_error, mux_bitwidth_mismatch
   - **I2C Project**: i2c_acknowledge_missing
   - **Reed Solomon Decoder Project**: reedsolomon_reset_sensitivity
   - **SDRAM Controller Project**: sdram_sync_reset_issue, sdram_numeric_definition_error

3. **Resolved Bugs with New Issues**: In some cases, the model successfully fixed the initial bugs but inadvertently introduced new issues due to excessive or unnecessary changes. The following projects had resolved issues with additional introduced bugs:

   - **Counter Project**: counter_sensitivity_issue
   - **Flip-Flop Project**: flipflop_conditional_error, flipflop_branch_swap
   - **LShift Project**: lshift_conditional_error

Table 4.5: Structured Zero-Shot Results. Changes are categorized as Add, Remove, or Change in Assignment, labeled as (A); Change in If/Else If Condition, labeled as (C); Change in Logic, labeled as (L); Change in Sensitivity List, labeled as (S); and No Change indicates the buggy code remains unchanged.

Simplified Buggy File Name	**Cat**	**Strategy 0**	**Strategy 1**	**Strategy 2**	**Strategy 3**	**Strategy 4**	**Strategy 5**
counter_sensitivity_issue	1		L	L		L	L
counter_increment_error	1	L				L	L
flipflop_conditional_error	1	S	C,S	C,S	C,S		
flipflop_branch_swap	1		C,S	C,S	C,S		
fsm_state_assignment_missing	2	No change	A,S		L,S	A,S	A,S
lshift_blocking_issue	1			L			
lshift_sensitivity_issue	1				L,S		
i2c_sensitivity_issue	2	A	No change	L,A		A,C,S	A
sha3_loop_boundary_issue	1	A,C	A,C	A,C	A,C	A,C	A,C
sha3_wire_assignment_error	2	L,A,C	L,A,C	L,A,C	L,A,S	L,A,S	L,A
sha3_buffer_overflow_skipped	2	L,A,C	L,A,C		C	C	No change
sha3_bitwise_negation_issue	1	L,A,C	L,A,C		L,A,C		L,A,C
tatepairing_bitshift_logic_error	1	C	A,C	A	A,C	A,C	A,C
tatepairing_operator_mismatch	1		A,C		A,C	A,C	A
tatepairing_module_instantiation	2		L,A,C	A	A,C	A,C	A,S
reedsolomon_register_size_error	1		A	A	L,A,C	A	No change
sdram_case_statement_issue	2	L	L	L	L,S	L,C	L,A

For **structured zero-shot** results in Table 4.5, a similar analysis was conducted, revealing that the most frequent type of unsuccessful modification was **Add, Remove, or Change in Assignment (A)**, appearing 44 times. Following this, **Change in If/Else If Condition (C)** occurred 37 times without achieving a solution. **Change in Logic (L)** was attempted 31 times but also failed to resolve the issues, while **Add or Change in Sensitivity List (S)** was applied unsuccessfully 17 times. The **No Change** category appeared only 4 times, suggesting that the model generally attempted some form of modification rather than leaving the code unchanged.

Similar to the unstructured zero-shot results, we analyzed the model's attempts to resolve bugs across all strategies and categorized our observations into distinct groups. Below, we outline each category, highlighting insights based on the model's performance across various strategies:

1. **Unresolved Bugs Across All Strategies**: The following files remained unresolved across all strategies, highlighting bugs that were particularly challenging for the model to detect and resolve:

- **SHA3 Project**: sha3_loop_boundary_issue, sha3_wire_assignment_error
- **Tate Pairing Project**: tatepairing_bitshift_logic_error
- **SDRAM Controller Project**: sdram_case_statement_issue

2. **Resolved Bugs Across All Strategies**: The following files were successfully fixed in at least one run within each strategy, demonstrating the model's capability to resolve these bugs across different configurations:
   - **Decoder Project**: decoder_numeric_error, decoder_assignment_error
   - **Counter Project**: counter_reset_issue
   - **FSM Project**: fsm_case_statement_issue, fsm_state_sensitivity_issue, fsm_blocking_assignment_error
   - **LShift Project**: lshift_conditional_error
   - **Mux Project**: mux_numeric_issues, mux_hex_error, mux_bitwidth_mismatch
   - **I2C Project**: i2c_address_assignment_error, i2c_acknowledge_missing
   - **Reed Solomon Decoder Project**: reedsolomon_reset_sensitivity
   - **SDRAM Controller Project**: sdram_sync_reset_issue, sdram_numeric_definition_error
3. **Resolved Bugs with New Issues**: In these cases, the model successfully fixed the initial bugs but introduced new issues due to unnecessary or excessive modifications:
   - **Counter Project**: counter_sensitivity_issue
   - **Flip-Flop Project**: flipflop_conditional_error, flipflop_branch_swap
   - **LShift Project**: lshift_conditional_error
   - **Reed Solomon Decoder Project**: reedsolomon_register_size_error (in some cases)
   - **SDRAM Controller Project**: sdram_case_statement_issue (fixed in one run under strategy 3)

Finally, the model faced significant challenges in addressing bugs that required deleting lines without explanatory comments or relied on specific external requirements not explicitly embedded

within the code. In such scenarios, the model's responses were primarily informed by its pre-existing training data and general knowledge base. This approach proved effective for simpler, well-documented RTL bugs commonly found in typical RTL examples. However, it fell short when dealing with complex issues or context-specific bugs that deviated from standard RTL design patterns or required a deeper understanding of the intended design requirements.

As a result, simpler files and examples were consistently resolved across all strategies, both structured and unstructured. In contrast, the more complex or requirement-driven bugs posed a greater challenge for the model. Additionally, the analysis indicates that the LLM model exhibited consistent behavior throughout the experiments, further emphasizing its limitations and strengths.

Chapter 5

Few-Shot Technique: LLM Automatic Repair Performance

In this chapter, following our study of zero-shot learning, we continue our investigation of LLMs automatic repair performance by addressing the research question: **RQ2: How does providing LLMs with example repairs and additional information influence their ability to perform automatic hardware repairs (few-shot)?**

Here, we apply the few-shot technique by providing LLMs with example repairs for different bug scenarios, inspired by the repair templates used in [1]. These templates were adapted to create examples that accompany the prompt, allowing the LLMs to reference specific repair patterns before being tasked with fixing new bugs. We evaluate LLMs performance in this few-shot setting, examining how they respond to minimal context and the effect of incorporating detailed instructions and targeted strategies.

The experiments in this chapter will help us assess the influence of a few-shot approach on LLMs's ability to perform automatic RTL bug repairs under different conditions.

5.1 Strategies and Prompt variations

In this chapter, we apply the same strategies and prompt variations defined in Section 4.1 (for strategies) and Section 4.2 (for prompt variations) as used in the zero-shot experiments, but within a few-shot setup. This approach includes example-based learning, which aims to potentially enhance

Listing 5.1: initial context

```
### Repair Guidelines and Examples: ###
Follow the guidelines and examples provided below to learn how to spot and correct
    common issues.
Use these as inspiration to identify bugs and apply the necessary fixes. For detailed
instructions and additional examples on bug repair, refer to the following guidelines:
```

Listing 5.2: Repair Guidelines: Conditional Statements

```
1. **Conditional Statements**: If there's a defect in conditional statements, invert the
condition of the code block or correct the logical structure.
   **Examples:**
   - Example 1:
     ```verilog
     // Buggy line in conditional statement: Incorrect condition causes failure in
         signal check.
     if (signal_ready == 1'b1) begin // buggy line
     // Fixed line: Inverted the condition to correct signal check.
     if (signal_ready == 1'b0) begin // fixed line
     ```
```

the model's capability to repair RTL code automatically.

To implement the few-shot approach, we adapted repair templates from the CirFix paper [1] and created multiple examples based on these templates. Since we do not know the specific bug in each piece of code, all available guidelines and examples are provided in the prompt for each strategy and with both structured and unstructured prompt variations.

These repair templates are integrated within the prompt as "Repair Guidelines and Examples:" and offer instructions for correcting issues in common RTL code areas, including conditional statements, sensitivity lists, assignment blocks, numeric values, default cases, and bitshifting logic.

The listings below include the initial context provided in the "Repair Guidelines and Examples" part of the prompt, followed by small snippets of the repair guidelines and examples for each category:

- **initial context provided in the "Repair Guidelines and Examples" part of the prompt** (Listing 5.1)
- **Conditional Statements**: Adjusts conditional logic issues by modifying conditions, adding missing cases, or correcting reset behavior (Listing 5.2).
- **Sensitivity List**: Ensures appropriate sensitivity list triggers in always blocks (Listing 5.3).

Listing 5.3: Repair Guidelines: Sensitivity List

```
2. **Sensitivity List**: If the issue lies in the sensitivity list, verify that
the always block triggers appropriately. This includes triggering the always block
on signal's falling edge, rising edge, or any change to variables.
   **Examples:**
   - Example 1:
     ```verilog
     // Buggy line in sensitivity list: Incorrect edge trigger causes missed timing
         events.
     always @(posedge clk) begin // buggy line
     // Fixed line: Changed posedge to negedge for correct event trigger.
     always @(negedge clk) begin // fixed line
     ```
```

Listing 5.4: Repair Guidelines: Assignment Block

```
3. **Assignment Block**: When dealing with assignment block defects, convert
between blocking and non-blocking assignments as needed.
   **Examples:**
   - Example 1:
     ```verilog
     // Buggy line in assignment block: Incorrect blocking assignment causes race
         conditions.
     out_signal = 1'b1; // buggy line
     // Fixed line: Converted blocking to non-blocking to avoid race conditions.
     out_signal <= 1'b1; // fixed line
     ```
```

Listing 5.5: Repair Guidelines: Numeric Value

```
4. **Numeric Value**: For numeric value discrepancies, adjust the identifier by
    incrementing
or decrementing it, or correct width mismatches.
   **Examples:**
   - Example 1:
     ```verilog
     // Buggy line in numeric value: Off-by-one error in parameter causes incorrect
         count.
     parameter COUNT_MAX = 7; // buggy line
     // Fixed line: Corrected the parameter value by incrementing.
     parameter COUNT_MAX = 8; // fixed line
     ```
```

- **Assignment Block**: Provides guidance on switching between blocking and non-blocking assignments as needed to prevent race conditions or timing errors (Listing 5.4).
- **Numeric Value**: Addresses off-by-one errors, width mismatches, and other value discrepancies affecting data accuracy (Listing 5.5).
- **Default Case in Case Statements**: Emphasizes adding default cases to ensure proper code execution (Listing 5.6).
- **Bitshifting Logic**: Instructs on applying the correct bitshifting operators for data manipula-

Listing 5.6: Repair Guidelines: Default Case in Case Statements

```
5. **Default Case in Case Statements**: If the default case in a case statement is
    missing,
add the default case to ensure proper operation.
   **Examples:**
   - Example 1:
     ```verilog
     // Buggy code in case statement: Missing default case leads to unexpected behavior.
     case (current_state)
         START: next_state = IDLE;
         IDLE: next_state = EXEC;
     endcase // buggy code
     // Fixed code: Added default case for proper state handling.
     case (current_state)
         START: next_state = IDLE;
         IDLE: next_state = EXEC;
         default: next_state = ERROR;
     endcase // fixed code
     ```
```

Listing 5.7: Repair Guidelines: Bitshifting Logic

```
6. **Bitshifting Logic**: If the bug is in bitshifting logic,verify the shift operator
used and correct it.
   **Examples:**
   - Example 1:
     ```verilog
     // Buggy line in bitshifting logic: Incorrect operator used for bitshifting.
     data_shifted = value > 2; // buggy line
     // Fixed line: Corrected operator for bitshift operation.
     data_shifted = value >> 2; // fixed line
     ```
```

tion (Listing 5.7).

Incorporating these guidelines and examples into all strategies and prompt variations is intended to potentially aid in RTL bug repair.

5.2 Experimental Results

Using the framework outlined in Chapter 3, we conduct and analyze experiments applying the few-shot technique. In this approach, the performance of the LLMs is evaluated across several metrics detailed in the Evaluation Metrics section(Section 3.4). These metrics include pass@k, Percentage Pass, Number of Benchmarks Fixed, Total Average FC Change, Average Minimum and Maximum FC Change, Average Number of Benchmarks Fixed, and Total Cost.

Our results compare performance when examples and contextual information are provided to guide the model versus when guidance is limited. This comparison allows us to examine how adding

context influences LLMs repair capabilities. The insights gained from this analysis shed light on the effectiveness of the few-shot approach for enhancing automatic RTL bug repairs.

5.2.1 Unstructured Approach Results

The performance of the different strategies is presented in Table 5.1, which displays metrics such as Pass@1, Percentage of Pass, and Total Cost. Additional metrics, including Total Average FC Change, Average Minimum FC Change, and Average Maximum FC Change, are visualized in Fig. 5.1. Fig. 5.2 further illustrates the Total Number of Benchmarks Fixed and the Average Number of Complete Fixes. Together, these metrics provide a comprehensive basis for comparing the effectiveness and reliability of each strategy in automatic RTL bug repair using LLMs.

Table 5.1: Unstructured Few-Shot Results by Strategy

Metric	**Strategy 0**	**Strategy 1**	**Strategy 2**	**Strategy 3**	**Strategy 4**	**Strategy 5**
Pass@1	0.475	0.33125	0.425	0.40625	0.45625	0.45625
Percentage of pass	98.13%	97.50%	97.50%	98.13%	99.38%	98.13%
Total Cost ($)	$5.51	$5.77	$6.98	$5.9	$7.26	$7.4

To begin with the **Pass@1** results, calculated using the equation outlined in (Equation 3.1), Strategy 0 stands out with the highest Pass@1 score of (0.475), making it the most effective strategy for accurately resolving bugs on the first attempt. Following closely is Strategy 4, which records a Pass@1 score of (0.45625), indicating commendable performance. Strategy 5 also shows a Pass@1 score of (0.45625), placing it in a tie with Strategy 4. Strategy 2 has a score of (0.425), reflecting a moderate level of effectiveness, while Strategy 3 has a lower score of (0.40625), suggesting it is less capable in this regard. Strategy 1 has the lowest Pass@1 score of (0.33125), making it the least effective.

Turning to the **Percentage of Pass**, which measures the compilability of the generated responses from the LLMs, Strategy 4 stands out with the highest pass percentage of (99.38%), indicating its exceptional ability to produce compilable code. Strategies 0, 3, and 5 each achieve a strong pass percentage of (98.13%), reflecting reliable performance as well, though they fall slightly short of Strategy 4. Meanwhile, Strategies 1 and 2 show a lower pass percentage of (97.50%), suggesting that they may have some limitations in producing compilable outputs.

Regarding the **Total Cost**, the expenses associated with each strategy are relatively similar,

primarily influenced by the token sizes and the length of the prompts used. Strategy 0, which utilizes only the buggy code as input, incurs the lowest cost at ($5.51). Strategy 1 follows closely with a cost of ($5.77), while Strategy 2 has a slightly higher cost of ($6.98). Strategy 3 also has a moderate cost of ($5.90). Strategies 4 and 5, which include additional input information, incur higher costs of ($7.26) and ($7.40), respectively. This variation in cost underscores the importance of considering both the input complexity and the overall effectiveness when selecting the best strategy for automatic hardware bug repairs.

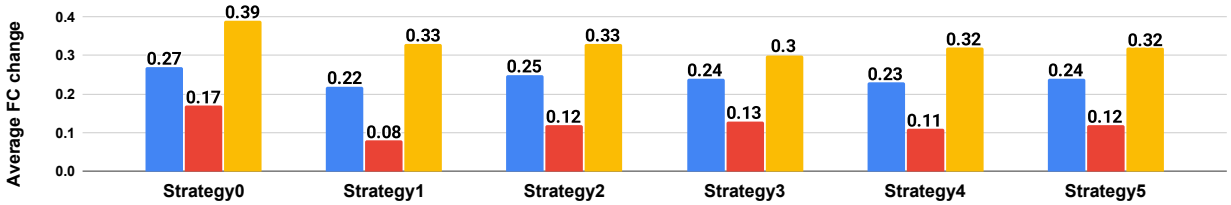

Figure 5.1: Comparison of total average FC change (blue), average minimum FC change (red), and average maximum FC change (yellow) across all benchmarks. The x-axis represents different strategies, and the y-axis shows the average FC change.

As shown in Fig. 5.1, for the **Total Average FC Change**, **Strategy 0** achieves the highest value with an average of (0.27), indicating that it produces the most significant overall adjustment in the FC metric. **Strategy 2** follows closely with an average change of (0.25), demonstrating similarly strong adjustments. **Strategies 3** and **5** each have a moderate average change of (0.24), suggesting that they also offer substantial FC change impact. By contrast, **Strategy 4** and **Strategy 1** display the lowest average changes, with values of (0.23) and (0.22) respectively, indicating more conservative adjustments on average.

In terms of the **Average Minimum FC Change**, **Strategy 0** again shows the highest performance, achieving an average minimum change of (0.17), highlighting its consistent effectiveness even at the lower range. **Strategy 3** ranks next with an average minimum change of (0.13), showing moderate minimum effectiveness. Meanwhile, **Strategies 2** and **5** both have average minimum FC changes of (0.12), followed by **Strategy 4** and **Strategy 1**, with (0.11) and (0.08) respectively, indicating that they may produce less impactful adjustments in some cases.

For the **Average Maximum FC Change**, **Strategy 0** leads with the highest average maximum change at (0.39), emphasizing its capability to create substantial FC modifications when necessary. **Strategies 1** and **2** exhibit strong maximum FC changes as well, each achieving (0.33). **Strategies**

4 and **5** are close behind with maximum changes averaging (0.32), indicating comparable effectiveness at the higher end. **Strategy 3** has the lowest maximum FC change at (0.3), suggesting a more moderate impact on FC change at the upper limit.

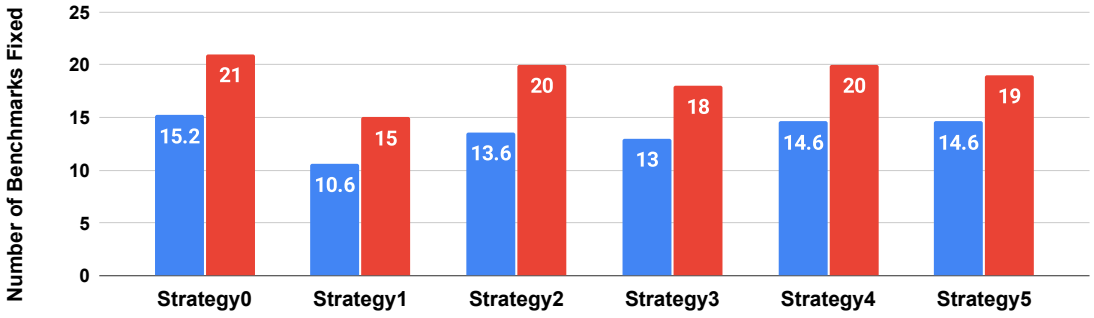

Figure 5.2: Number of benchmarks fixed (red) and the average number of complete fixes (blue).(Number of Benchmarks Fixed vs. Strategy)

As shown in Fig. 5.2, for the **Number of Benchmarks Fixed**, Strategy 0 demonstrates the highest number of benchmarks fixed (**21**), closely followed by Strategies 2 and 4, each with (**20**). Strategy 5 fixed (**19**) benchmarks, while Strategy 3 fixed (**18**). Strategy 1 has the lowest count at (**15**).

In terms of the **Average Number of Complete Fixes**, Strategy 0 leads with an average of (**15.2**), indicating a high quality of fixes. Strategies 4 and 5 show comparable performance, both achieving an average of (**14.6**). Strategies 2 and 3 follow with averages of (**13.6**) and (**13**), respectively. Strategy 1, with an average of (**10.6**), ranks lowest in both metrics

In conclusion, based on the metrics shown above, **Strategy 0** emerges as the most effective approach, achieving the highest **Pass@1** score (0.475), the largest **Total Average FC Change** (0.27), and the highest count of benchmarks fixed (21) along with a high average of complete fixes (15.2). This suggests it provides both accuracy and quality in bug repair.

Strategy 4 ranks second, with a strong **Pass@1** score (0.45625), the highest **Percentage of Pass** (99.38%), and solid performance in complete fixes (14.6) and benchmarks fixed (20), indicating consistent reliability. **Strategy 5** follows closely in third, tying with Strategy 4 on **Pass@1** and showing comparable values in complete fixes and benchmarks fixed.

Strategy 2 shows moderate effectiveness with mid-range values across most metrics. **Strategy 3** demonstrates a more limited impact, with low values in **Average Maximum FC Change** (0.3)

and a lower **Pass@1** score, indicating reduced effectiveness. **Strategy 1** ranks lowest overall, with room for improvement in accuracy and completeness.

5.2.2 Structured Approach Results

This subsection, similar to Section 5.2.1, details the performance of the different strategies in Table 5.2, which showcases key performance indicators such as Pass@1, Percentage of Pass, and Total Cost. Additional metrics, including Total Average FC Change, Average Minimum FC Change, and Average Maximum FC Change, are shown in Fig. 5.3. Furthermore, Fig. 5.4 illustrates the Total Number of Correctly Fixed Bugs and the Average Number of Complete Fixes. Together, these metrics provide a solid foundation for evaluating the effectiveness and efficiency of each strategy in the context of automatic RTL bug repair.

Table 5.2: Structured Few-Shot Results by Strategy

Metric	**Strategy 0**	**Strategy 1**	**Strategy 2**	**Strategy 3**	**Strategy 4**	**Strategy 5**
Pass@1	0.41875	0.35625	0.4125	0.4	0.4	0.4625
Percentage of pass	98.75%	97.50%	96.88%	98.75%	96.88%	99.38%
Total Cost ($)	$5.73	$5.84	$6.97	$6.01	$7.36	$7.42

Analyzing the **Pass@1** results, Strategy 5 achieves the highest score at (0.4625), indicating it performs best in generating effective automatic hardware repairs under structured prompting conditions. Strategy 0 follows closely with a pass@1 score of (0.41875), showcasing strong performance. Strategy 2 closely trails Strategy 0 with a score of (0.4125), suggesting both strategies are effective, with Strategy 0 being slightly better. Strategies 3 and 4 both share a pass@1 score of (0.4), reflecting moderate performance, while Strategy 1 records the lowest score at (0.35625), indicating it is the least effective overall.

In terms of the **Percentage of Pass**, Strategy 5 leads with an impressive rate of (99.38%), demonstrating its high reliability in producing compilable code. Strategy 0 also shows strong performance with a pass percentage of (98.75%). Strategies 1 and 3 both achieve pass percentages of (97.50%) and (98.75%), respectively, indicating solid performance but with slightly diminished reliability compared to Strategy 5. Strategies 2 and 4 have lower pass percentages of (96.88%), suggesting some limitations in their ability to generate compilable code.

Considering the **Total Cost**, Strategy 0 has the lowest cost at ($5.73), closely followed by

Strategy 1 at ($5.84). Strategy 3 incurs a moderate cost of ($6.01), while Strategy 2's cost rises to ($6.97). Strategies 4 and 5 have the highest costs at ($7.36) and ($7.42), respectively. Despite the higher costs, Strategy 5's performance metrics, particularly in pass@1 and percentage of pass, position it as a valuable option. This analysis highlights the trade-offs between cost and performance across different strategies for automatic hardware bug repairs, emphasizing the need for careful evaluation based on both effectiveness and expense.

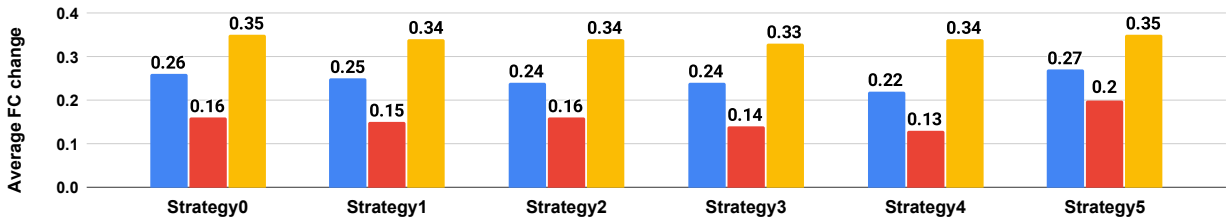

Figure 5.3: Comparison of total average FC change (blue), average minimum FC change (red), and average maximum FC change (yellow) across all benchmarks. The x-axis represents different strategies, and the y-axis shows the average FC change.

As shown in Fig. 5.3, for the **Total Average FC Change**, **Strategy 5** achieves the highest value with an average of (0.27), indicating a substantial overall adjustment. **Strategy 0** follows closely with an average change of (0.26), showing a similar strength in adjustments. **Strategy 1** has a slightly higher average change of (0.25), reflecting a consistent impact across benchmarks. **Strategies 2** and **3** each have a moderate total average change of (0.24), suggesting similar performance in adjustments. On the other hand, **Strategy 4** has the lowest total average FC change at (0.22), which may suggest a more conservative adjustment approach compared to the other strategies.

In terms of the **Average Minimum FC Change**, **Strategy 5** again stands out, with a notably high minimum change of (0.2), reflecting effective performance even at the lower bounds. **Strategies 0** and **2** both have a minimum average change of (0.16), showing steady impact. **Strategy 1** follows with (0.15), while **Strategy 3** and **4** have slightly lower minimum changes at (0.14) and (0.13), respectively, suggesting that these strategies may be less effective at the lower end of adjustments.

For the **Average Maximum FC Change**, both **Strategy 0** and **Strategy 5** reach the highest values at (0.35), indicating a strong capacity for impactful maximum adjustments. **Strategies 1**, **2**, and **4** show comparable maximum change averages, each around (0.34), reflecting reliable

performance at the higher end of adjustments. **Strategy 3** shows a slightly lower maximum FC change at (0.33), suggesting a more moderate approach to significant changes.

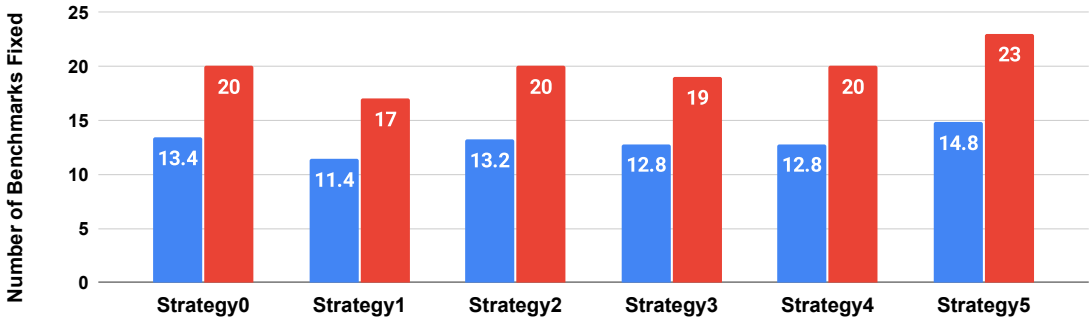

Figure 5.4: Number of benchmarks fixed (red) and the average number of complete fixes (blue).(Number of Benchmarks Fixed vs. Strategy)

As shown in Fig. 5.4, the **Number of Benchmarks Fixed** metric highlights each strategy's success in identifying and resolving bugs. **Strategy 5** leads with the highest count at **23**, making it the most effective in terms of bug correction accuracy. Following closely, **Strategies 0, 2, and 4** each achieve **20** correctly fixed bugs, indicating a high degree of reliability and effectiveness. **Strategy 3** follows with a total of **19** fixed bugs, showing moderate success. **Strategy 1** has the lowest count at **17** fixed bugs, suggesting it may require further refinement to match the effectiveness of the other strategies.

In terms of the **Average Number of Benchmarks Fixed**, **Strategy 5** again stands out with the highest average of **14.8**, indicating that it not only fixes bugs accurately but also does so more comprehensively than other strategies. **Strategy 0** follows with an average of **13.4**, making it another strong performer in delivering complete fixes.**Strategy 2** has an average of **13.2**, while **Strategies 3 and 4** each have similar averages around **12.8**, reflecting moderate success in completeness. **Strategy 1** has the lowest average of **11.4**, suggesting that it may leave room for improvement in the thoroughness of its fixes.

In conclusion, based on the metrics, **Strategy 5** is the most effective, excelling in **Pass@1** (0.4625), **Percentage of Pass** (99.38%), **Total Average FC Change** (0.27), and benchmarks fixed, despite higher cost.

Strategy 0 ranks second, with strong results across metrics, balancing effectiveness and cost.

Strategy 2 and **Strategy 3** perform moderately, while **Strategy 4** is conservative in adjust-

ments but shows high compilability.

Strategy 1 is the least effective, with the lowest **Pass@1** and fewer benchmarks fixed.

5.2.3 Discussion and Observations

The unstructured and structured Few-shot results tables, shown in Table 5.3 and Table 5.4, respectively, display the results for examples that remained unsolved across all strategies.In these tables, we use the simplified bug names introduced in Table 3.2, and these names will be referenced throughout this section for consistency and clarity. Each example achieved an FC below 1 across all five runs for every strategy. These tables document the specific modifications the model applied to these examples, resulting in an FC value below 1. We classified these modifications into five categories, as outlined in Table 4.3.

This section presents an analysis of the LLM model's performance in resolving Verilog code bugs across different strategies, comparing the outcomes with the initial bug classifications from the benchmark study by [1]. The benchmark categorizes the files into two main groups: category 1 (19 files) and category 2 (13 files). However, it is important to note that the bug categories from prior work do not align with the model's behavior. While the benchmark categorized the files into these two categories, the LLM model struggled with category 1 files, which suggests a different approach in its understanding and resolution of functional bugs.

In analyzing the **unstructured few-shot** results from Table 5.3, we observe that the model's modification attempts—categorized as S, L, C, A, or No Change, as explained in Table 4.3 —followed a consistent pattern across all strategies. The model most frequently adjusted assignments, with **Add, Remove, or Change in Assignment (A)** appearing 47 times, suggesting a focus on modifying assignments as a primary approach to fixing bugs. This was followed by **Add or Change in Sensitivity List (S)**, appearing 42 times, likely as attempts to address timing or reactivity issues. Modifications to conditional statements, or **Change in If/Else If Condition (C)**, occurred 39 times, while the least frequent modification was **Change in Logic (L)**, at 25 appearances. Interestingly, the **No Change** category, where the model leaves code unchanged, did not appear in this table, indicating that the model consistently attempted modifications. However, none of these modifications successfully resolved the bugs, underscoring the model's limitations in effectively

Table 5.3: Unstructured Few-Shot Results. Changes are categorized as Add, Remove, or Change in Assignment, labeled as (A); Change in If/Else If Condition, labeled as (C); Change in Logic, labeled as (L); Change in Sensitivity List, labeled as (S); and No Change indicates the buggy code remains unchanged.

Simplified Buggy File Name	**Cat**	**Strategy 0**	**Strategy 1**	**Strategy 2**	**Strategy 3**	**Strategy 4**	**Strategy 5**
counter_sensitivity_issue	1		S		L,S		L,S
counter_increment_error	1	L	L,S		L,S		L
flipflop_conditional_error	1	C,S	C,S	S	C,S		
flipflop_branch_swap	1	C,S	S	S			
fsm_state_assignment_missing	2	L,S	S		L,S	S	L,S
fsm_blocking_assignment_error	2			C,S			
lshift_blocking_issue	1		L,S	L			
lshift_conditional_error	1	S	L,S				
lshift_sensitivity_issue	1	L,S	L,S				
i2c_sensitivity_issue	2		A			A	A
i2c_address_assignment_error	2		A		L,C,S		A
sha3_loop_boundary_issue	1	C,S	A,C,S	A,C	S	A,C,S	C,S
sha3_wire_assignment_error	2	L,A,C,S	L,A,C,S	L,A,C,S	L,A,S	L,A,C	L,A,C,S
sha3_buffer_overflow_skipped	2	A,S	A,C,S	A,S	A,C	A,C	A,C
sha3_bitwise_negation_issue	1		A,C	L,A,C	L,A,C	A,C	A,C
tatepairing_bitshift_logic_error	1	A,S	A,C	A,S	A,C	A,C	A,C
tatepairing_operator_mismatch	1				A,C	A,C	A,C
tatepairing_module_instantiation	2		A,C	A,S	A,C	A,C	A,C
reedsolomon_register_size_error	1	L		L	A,C	A	A
sdram_case_statement_issue	2		A	A,C	A,S	A,C	A,S

addressing Verilog code errors

After reviewing the model's attempts to resolve bugs across all strategies, we observed consistent behavior for certain files. These files can be grouped into three main categories: unresolved bugs, fully resolved bugs, and resolved bugs with new issues introduced. The following outlines each category, with observations based on the model's performance across different strategies:

1. **Unresolved Bugs Across All Strategies**: Four files remained unresolved across all strategies, despite multiple attempts at modification. These files consistently resisted correction, even with various strategies and runs. The unresolved issues were found in the following projects:

    - **SHA3 Project**: sha3_loop_boundary_issue, sha3_wire_assignment_error,

sha3_buffer_overflow_skipped

- **Tatepairing Project**: tatepairing_bitshift_logic_error

2. **Resolved Bugs Across All Strategies**: Twelve files were successfully fixed in at least one run within each strategy, indicating that the model was able to resolve the issues across different configurations. The files that were consistently fixed in all strategies belong to the following projects:

   - **Decoder Project**: decoder_numeric_error, decoder_assignment_error
   - **Counter Project**: counter_reset_issue
   - **FSM Project**: fsm_case_statement_issue, fsm_blocking_assignment_error
   - **Mux Project**: mux_numeric_issues, mux_hex_error, mux_bitwidth_mismatch
   - **I2C Project**: i2c_acknowledge_missing
   - **Reedsolomon Project**: reedsolomon_reset_sensitivity
   - **SDRAM Project**: sdram_sync_reset_issue, sdram_numeric_definition_error

3. **Resolved Bugs with New Issues**: In some instances, the model successfully fixed the initial bugs but inadvertently introduced new issues due to excessive or unnecessary changes. The model corrected the original problems in the following projects, but new errors were introduced:

   - **Counter Project**: counter_sensitivity_issue, counter_increment_error
   - **Flipflop Project**: flipflop_conditional_error, flipflop_branch_swap
   - **LShift Project**: lshift_conditional_error

For **structured few-shot** results in Table 5.4, a similar analysis was conducted, revealing consistent modification patterns across all strategies. The most frequent modification type was **Add or Change in Sensitivity List (S)**, appearing 44 times, which suggests that the model often targeted timing or signal responsiveness in these structured examples. Following this, **Add, Remove, or Change in Assignment (A)** appeared 37 times, indicating that assignment modifications remained a primary approach. Adjustments in conditional logic, or **Change in**

Table 5.4: Structured Few-Shot Results. Changes are categorized as Add, Remove, or Change in Assignment, labeled as (A); Change in If/Else If Condition, labeled as (C); Change in Logic, labeled as (L); Change in Sensitivity List, labeled as (S); and No Change indicates the buggy code remains unchanged.

Simplified Buggy File Name	**Cat**	**Strategy 0**	**Strategy 1**	**Strategy 2**	**Strategy 3**	**Strategy 4**	**Strategy 5**
counter_sensitivity_issue	1			S	L,S		
counter_increment_error	1	L	L,S	L			
counter_reset_issue	1		L,S	L,S	L,S		
flipflop_conditional_error	1	C,S	C,S	C,S	C,S		
flipflop_branch_swap	1	C,S		C,S	C,S		
fsm_state_assignment_missing	2	S	S		S	S	
lshift_blocking_issue	1	S					
lshift_conditional_error	1	L,S	L,S				
lshift_sensitivity_issue	1		L,S			L,S	
i2c_sensitivity_issue	2		A	A	A,S	A	A
i2c_address_assignment_error	2		A				A
sha3_loop_boundary_issue	1	C,S	A,C,S	C,S	S	A,C,S	S
sha3_wire_assignment_error	2	L,A,C,S	L,A,C,S	L,A,C,S	L,A,S	L,A,S	L,A,S
sha3_buffer_overflow_skipped	2	S	A,S	A	C,S	C	C,S
sha3_bitwise_negation_issue	1	C	C	L,A	A,C		L,A,C
tatepairing_bitshift_logic_error	1	A,C,S	A,C	A,S		A,C	
tatepairing_operator_mismatch	1	A			A,C	A,C	A,C
tatepairing_module_instantiation	2		A,C	A,S	A,C	A,C	A,C,S
reedsolomon_register_size_error	1					A	
sdram_case_statement_issue	2		L	C	L,A	L,A,C	L,A,C,S

If/Else If Condition (C), occurred 34 times, while the least frequent type was **Change in Logic (L)**, with 23 instances. The **No Change** category was again absent, underscoring the model's consistent attempts at modifications rather than leaving the code as-is.

Similar to the unstructured few-shot results, after reviewing the model's attempts to resolve bugs across all strategies, we classified our observations into distinct categories. The following outlines each category, with observations based on the model's performance across different strategies:

1. **Unresolved Bugs Across All Strategies**: The following files remained unresolved across all strategies, despite multiple attempts at modification. These issues were consistent across different strategies and iterations:

    - **SHA3 Project**: sha3_loop_boundary_issue, sha3_wire_assignment_error,

sha3_buffer_overflow_skipped

2. **Resolved Bugs Across All Strategies**: The following files were successfully fixed in at least one run within each strategy, indicating that the model was able to resolve these issues across different configurations:

   - **Decoder Project**: decoder_numeric_error, decoder_assignment_error
   - **FSM Project**: fsm_case_statement_issue, fsm_state_sensitivity_issue, fsm_blocking_assignment_error
   - **Mux Project**: mux_numeric_issues, mux_hex_error, mux_bitwidth_mismatch
   - **I2C Project**: i2c_acknowledge_missing
   - **Reedsolomon Project**: reedsolomon_reset_sensitivity
   - **SDRAM Project**: sdram_sync_reset_issue, sdram_numeric_definition_error

3. **Resolved Bugs with New Issues**: In some cases, the model successfully fixed the initial bugs but inadvertently introduced new issues due to excessive or unnecessary changes. The following projects had resolved issues along with new introduced bugs:

   - **Counter Project**: counter_sensitivity_issue, counter_increment_error, counter_reset_issue
   - **LShift Project**: lshift_blocking_issue, lshift_conditional_error, lshift_sensitivity_issue
   - **Flipflop Project**: flipflop_conditional_error, flipflop_branch_swap

Finally, upon reviewing the results, we found that introducing the few-shot technique, which provided different examples and guidelines for various bug types, did not lead to a significant improvement in the model's performance compared to the zero-shot approach discussed in Chapter 4. One consistent observation was that the model tended to modify the sensitivity list more frequently when it was unsure of the root cause of the issue. The model also continued to face challenges in resolving errors that required knowledge of external requirements or in situations where the issue involved missing lines of code. In these cases, the model often relied on its pre-existing training data and general knowledge, which proved more effective for resolving simpler, well-documented RTL bugs. However, this reliance limited its ability to handle more complex, context-specific issues that

deviated from standard RTL design patterns or required a deeper understanding of the intended design requirements. Consequently, while simpler bugs and examples were consistently resolved across all strategies, more complex or requirement-driven bugs posed greater difficulties for the model.

Chapter 6

Feedback-Based Technique

We explored zero-shot and few-shot techniques in the previous chapters to understand their effectiveness in facilitating code repairs. In this chapter, we extend this investigation by examining a feedback-based technique.

Inspired by approaches like those used by Thakur et al. [29], where iterative feedback was integrated to refine model outputs, we seek to determine the potential of feedback as a tool for enhancing automatic repair accuracy in LLMs. The goal is to understand if providing feedback or issuing multiple queries in iterative steps can help the LLM converge on a solution more effectively than single-query methods. To guide this exploration, we address the following research question:

RQ3: How well do LLMs perform when feedback and multiple queries are involved?

The following sections outline our feedback-based approach, the strategies employed, and the detailed logic implemented to optimize automatic repairs of Verilog code through iterative feedback.

6.1 Feedback Experimental Setup

The experimental setup for applying feedback involves the framework outlined in Section 3.1. For the feedback technique, we conduct up to 5 runs, with each run allowing a maximum of 5 iterations. The process will stop if we achieve a Fix Correctness (FC) score of 1, indicating the issue has been fixed, or if we complete all 5 iterations without reaching this score.

6.2 Strategies and Prompt Variations

In this experiment, we select strategies based on the results from the zero-shot technique, as the few-shot technique did not yield significantly higher results. We will focus on the strategies that performed best according to the results shown in Section 4.3. Specifically, the strategies we employ, as described in Section 4.1, include:

- **Strategy 0**: This strategy provides only the basic information to the LLM, which includes the buggy code.
- **Strategy 2**: This strategy includes both the buggy code and the differences between the Buggy Simulation output and the Oracle output.
- **Strategy 4**: This comprehensive strategy provides the buggy code, the differences between the Buggy Simulation and Oracle outputs, and fault localization information.

For prompt variations, we use unstructured prompting as detailed in Section 4.2, as this approach yielded higher results in both the zero-shot and few-shot settings. Each iteration in the experiment uses a distinct prompt incorporating one of the selected strategies. Below is an outline of the prompts for each iteration:

- **Initial Prompt (Prompt 0)**:

Listing 6.1: Feedback Initial Prompt (Prompt 0)

```
This is an iterative prompt. You have up to {max_trials} attempts to make the best
     possible corrections based on feedback I will provide after each response. I
    would like you to learn from the feedback and make the best changes based on
    the task given to you.
Your task is to fix the given buggy Verilog code with the smallest possible
    changes needed to correct any existing issues, ensuring the rest of the code
    remains unaltered and fully functional.
Focus on making only one essential change at a time.
In future iterations, avoid repeating changes that have not resolved the issue. If
     a change did not work, do not apply it again. Keep track of the changes you
    made in each iteration to avoid repeating unsuccessful fixes.
Always think about the logic of the code and use any input information provided to
     help identify the bug and make the appropriate changes. Double-check that the
```

```
     changes you make do not interfere with the overall functionality or introduce
     new issues.
Provide the complete, functioning version without adding any extra words, comments
    , or explanations in your response.
After generating the code, append a distinct section at the end, labeled 'Code
    Changes'. In this section, outline and explain the variances between the
    corrected and original buggy code, specifying the exact changes made,
    including line numbers. Ensure this section remains commented to avoid
    compiler interference.

The buggy Verilog code requiring correction is as follows:
 <<< \n{buggy_verilog_code}\n >>>.
```

- **Prompt 1**:

Listing 6.2: Feedback Prompt 1

```
Feedback for iteration "+str(current_trial)+": The changes you have made have
    improved the fix correctness, but it is still not fully correct. Please make
    only one change at a time. Here are the code changes you made before:" +
    previous_iteration_LLM_changes
```

- **Prompt 2**:

Listing 6.3: Feedback Prompt 2

```
Feedback for iteration "+str(current_trial)+": The previous code changes you made
    did not resolve the functional issue and have worsened the code. Please review
     the bug and apply other changes that directly address the problem. Here are
    the code changes you made before: + previous_iteration_LLM_changes
```

- **Prompt 3**:

Listing 6.4: Feedback Prompt 3

```
Feedback for iteration {current_trial}: The changes you made have improved the
    correctness of the fix, but it is still not fully correct.
Please make only one change at a time to refine the solution further. Use the
    extra information below, along with previous details, to identify and correct
    the bug effectively.
{Input_information_Strategy2}
Here are the code changes you in the previous iteration:
{previous_iteration_LLM_changes}
```

- **Prompt 4**:

Listing 6.5: Feedback Prompt 4

```
Feedback for iteration {current_trial}: The previous code changes you made did not
     resolve the functional issue and have worsened the code. Please review the
    bug carefully and apply only changes that directly address the problem. Use
    the simulated output differences provided below, along with previous
    information, to detect and fix the bug effectively.
{Input_information_Strategy2}
Here are the code changes you in the previous iteration:
{previous_iteration_LLM_changes}
```

- **Prompt 5**:

Listing 6.6: Feedback Prompt 5

```
Feedback for iteration {current_trial}: The changes you made have improved the
    correctness of the fix, but it is still not fully correct. Please make only
    one change at a time to refine the solution further. Use the extra information
     below, along with previous information, to identify and correct the bug
    effectively.
The extra information includes:
{Input_information_Strategy4}
Here are the code changes you made in the previous iteration:
{previous_iteration_LLM_changes}
```

- **Prompt 6**:

Listing 6.7: Feedback Prompt 6

```
Feedback for iteration {current_trial}: The previous code changes you made did not
     resolve the functional issue and have worsened the code. Please review the
    bug carefully and apply only changes that directly address the problem. If you
     made a change to a line in the previous iteration and it didn't improve the
    code, do not repeat it in subsequent iterations. Use the simulated output
    differences and fault localization results provided below, along with previous
     information, to detect and fix the bug effectively.
{Input_information_Strategy4}
Here are the code changes you made in the previous iteration:
{previous_iteration_LLM_changes}
```

- **Final Prompt**:

Listing 6.8: Feedback Final Prompt

```
Feedback for iteration {current_trial}: This is your final iteration. All previous
     changes have not resolved the bug, so it is essential to think creatively and
     apply a fresh approach. Analyze your past responses carefully and consider
    the logic of the code along with the input information. The bug may lie in the
     lines identified through fault localization, so focus on making the correct
    change based on this insight.
Avoid repeating changes that did not work and ensure that the fix fully addresses
    the issue. Provide the corrected Verilog code without extra words, comments,
    or explanations.
```

6.2.1 Iterative Feedback Logic for Verilog Bug Fixing using LLM

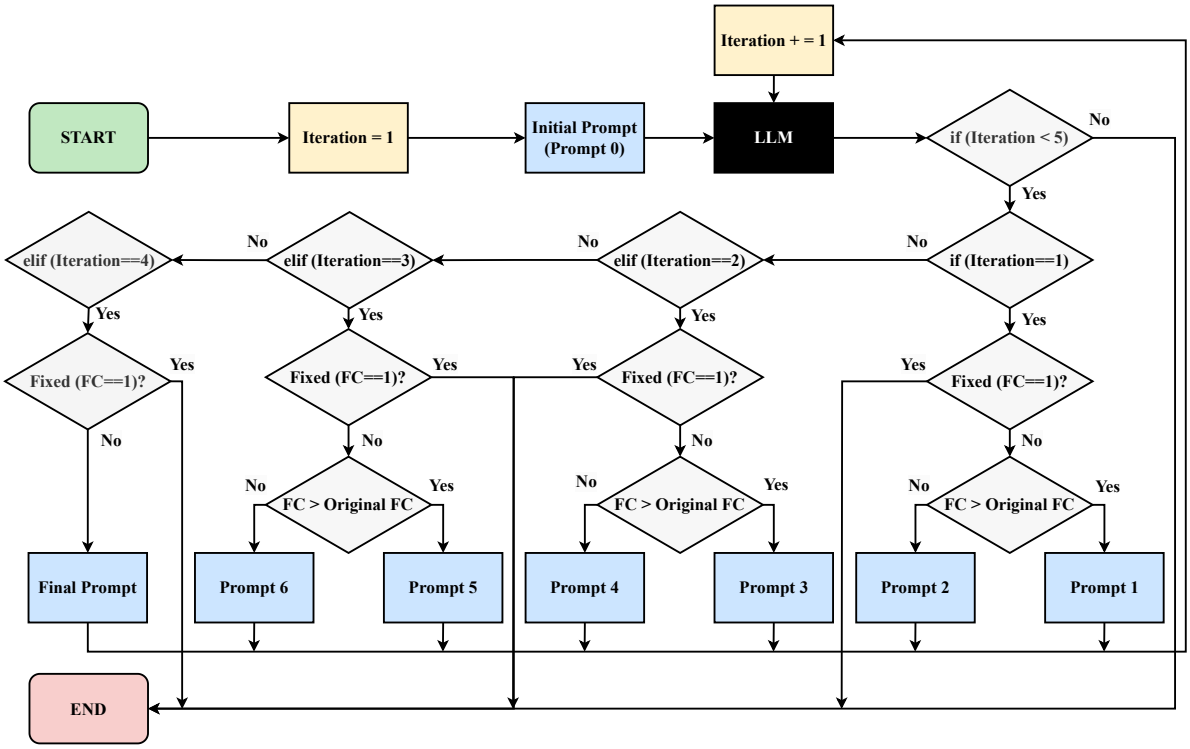

Figure 6.1: Feedback Logic

Fig. 6.1 illustrates the feedback-driven process employed to correct buggy Verilog code using an LLM. This process involves a maximum of five iterations for each file in the benchmark set. Each iteration includes prompt adjustments based on feedback and simulation results, with the goal of achieving a correctness score of 1.

The rationale behind this iterative feedback logic is to systematically guide the LLM toward identifying and fixing bugs more effectively. Starting from the second iteration, the model has access to the changes made during the previous iteration. This allows us to evaluate the effect of those changes on the buggy code, particularly by calculating the FC value. If the changes improve the FC value, a tailored prompt is sent in the subsequent iteration to build on the progress. Conversely, if the changes degrade the FC value or introduce new issues, a different prompt is provided to steer the model in a better direction. This approach is designed to provide the model with more information and context in each iteration, along with specific feedback on its performance. By iteratively refining the model's understanding of the bug and guiding it with feedback-driven prompts, the process aims to enhance the model's ability to detect and repair the underlying issues in the Verilog code. This feedback mechanism is an integral part of the framework described in Section 3.1. The detailed steps are as follows:

1. **Iteration 1: Initial Prompting** The process begins by setting the maximum iteration limit to 5 and constructing the initial prompt. In this step, the LLM is provided with *Prompt 0* (Listing 6.1), which contains the buggy Verilog code along with tailored instructions emphasizing minimal, iterative corrections. This prompt is based on *Strategy 0*, as explained in Section 4.1.

2. **Iteration 2: Initial Feedback** Upon receiving the LLM's response, a series of checks is performed:

    - If the iteration count is less than 5 and the current iteration is the feedback from Iteration 1, a simulation is run to evaluate the fix correctness (FC).
    - If the FC equals 1 (indicating the bug is resolved), the process terminates for the file.
    - Otherwise, the process evaluates whether the FC has improved compared to the original buggy code:
        - If FC has improved: *Prompt 1* (Listing 6.2) is sent, providing positive feedback to the LLM, acknowledging the improvement and encouraging further corrections along the same lines.

- If FC is unchanged or worsened: *Prompt 2* (Listing 6.3) is sent, suggesting alternative approaches to address the issue.

The iteration counter is then incremented, and the selected prompt is sent to the LLM.

3. **Iteration 3: Strategy Adjustments** Similar checks are performed for feedback from Iteration 2:

   - If FC equals 1, the process ends for the file.
   - If FC has improved: *Prompt 3* (Listing 6.4) is sent. This prompt builds on the improvement by incorporating additional information from *Strategy 2* (Section 4.1), offering more detailed guidance.
   - If FC is unchanged or worsened: *Prompt 4* (Listing 6.5) is sent, providing alternative suggestions and additional information based on *Strategy 2*.

4. **Iteration 4: Strategy Refinement** Feedback from Iteration 3 is processed using the same logic:

   - If FC has improved: *Prompt 5* (Listing 6.6) is sent, leveraging additional information from *Strategy 4* (Section 4.1) to further refine the corrections.
   - If FC is unchanged or worsened: *Prompt 6* (Listing 6.7) is sent, suggesting new approaches and including information from *Strategy 4*.

5. **Iteration 5: Final Attempt** If no satisfactory fix has been achieved by this point:

   - *Prompt Final* (Listing 6.8) is sent, urging the LLM to apply a creative and fresh approach.
   - This prompt advises avoiding ineffective prior changes and focuses on fault-localized lines for a last attempt.

6. **End Condition** The process terminates for a file when either a correctness score of 1 is achieved or the iteration limit of 5 is reached.

6.3 Experimental Results

Using the framework described in Chapter 3 and the feedback logic outlined in Section 6.2.1, we perform and analyze experiments employing the feedback technique. This approach evaluates the performance of LLMs based on several key metrics, as detailed in the Evaluation Metrics section (Section 3.4). These metrics include Pass@k, calculated across all runs, where k represents the total number of runs in this experiment (5 runs). Specifically, we calculate Pass@1, which evaluates the proportion of benchmarks successfully fixed in the first run. This metric provides a basis for direct comparison with strategies results in the zero-shot and few-shot techniques discussed in Section 4.3 and Section 5.2. Other metrics include the Percentage Pass, which is determined by tracking the number of attempts required for each run to process all benchmarks (with each run consisting of 5 iterations) and calculating the number of passes and failures, the Number of Benchmarks Fixed (per iteration), the Average Number of Benchmarks Fixed, Total Average FC Change and the Total Cost. Additionally, we measure the Cumulative Total of Fixed Benchmarks in this experiment, as the iterative nature of the process allows us to assess performance across all iterations.

Our results demonstrate how the model responds when prompts are provided iteratively, with feedback from each iteration informing the subsequent prompt. This iterative feedback process allows us to assess how multiple iterations influence LLMs's repair capabilities. The insights gained from this analysis provide valuable perspectives on the effectiveness of the feedback approach for improving automatic RTL bug repairs.

As shown in Fig. 6.2, the cumulative total of fixed benchmarks is presented across all iterations for each run. The runs are represented by different colors: Run 1 (blue), Run 2 (red), Run 3 (yellow), Run 4 (green), and Run 5 (black).

In the first iteration, all runs started with comparable results: Run 1 and Run 2 fixed (18) benchmarks each, Run 3 achieved (19), Run 4 fixed (16), and Run 5 also reached (19). By the second iteration, all runs improved. Run 1 increased by 2 fixes to (20), while Run 2 showed the largest improvement, adding 4 fixes to reach (22). Run 3 improved by 2 fixes to (21), Run 4 increased by 3 fixes to (19), and Run 5 added 2 fixes, reaching (21).

Progress slowed in the third iteration, with only Run 4 showing an increase of 2 fixes to reach (21). Runs 1, 2, 3, and 5 remained stable at (20), (22), (21), and (21), respectively. During the

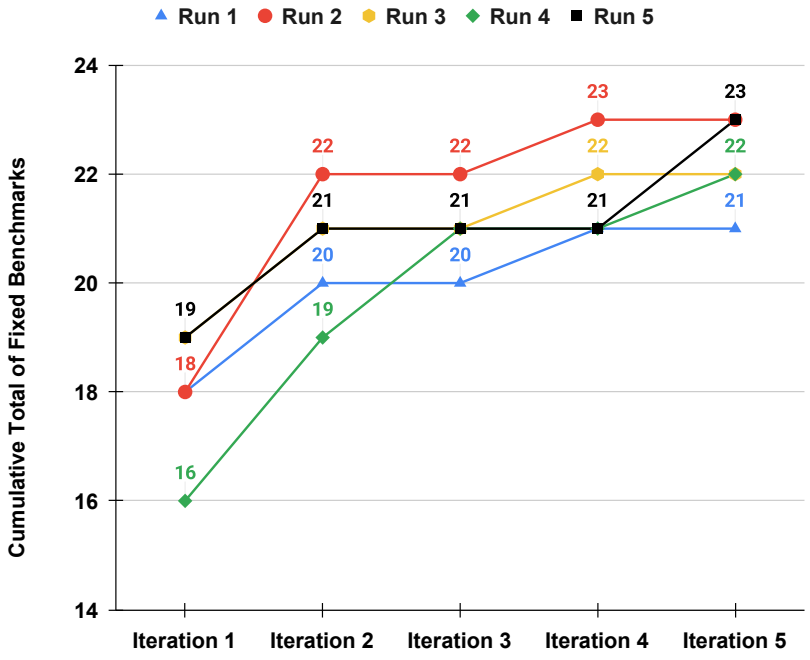

Figure 6.2: Cumulative total of fixed benchmarks across iterations for each run. The y-axis represents the cumulative number of fixed benchmarks, while the x-axis corresponds to the iteration number. Each line represents a different run, with different colors indicating separate runs. Run 1 is shown in blue, Run 2 in red, Run 3 in yellow, Run 4 in green, and Run 5 in black. The chart illustrates how the number of fixed benchmarks progresses across iterations for each run.

fourth iteration, incremental improvements were observed. Run 1 added 1 fix, reaching (21), while Runs 2 and 3 also added 1 fix each, reaching (23) and (22), respectively. Runs 4 and 5 remained at (21).

In the final iteration, changes were limited. Run 4 improved by 1 fix, reaching (22), while Run 5 added 2 fixes to also reach (23). Runs 1, 2, and 3 stayed at (21), (23), and (22), respectively.

Overall, the majority of fixes occurred during the first and second iterations, with incremental progress in subsequent iterations. This trend suggests that the feedback technique was most effective in the early iterations, with diminishing returns as the process continued.

To better understand the model's overall performance across iterations, we analyze the Average Number of Benchmarks Fixed per Iteration as shown in Fig. 6.2. This metric provides insight into how the cumulative fixes progress on average throughout the feedback process.

During Iteration 1, the average number of benchmarks fixed was 18, reflecting the initial fixes achieved by the LLM across all runs. By Iteration 2, the average increased to 20.6 benchmarks,

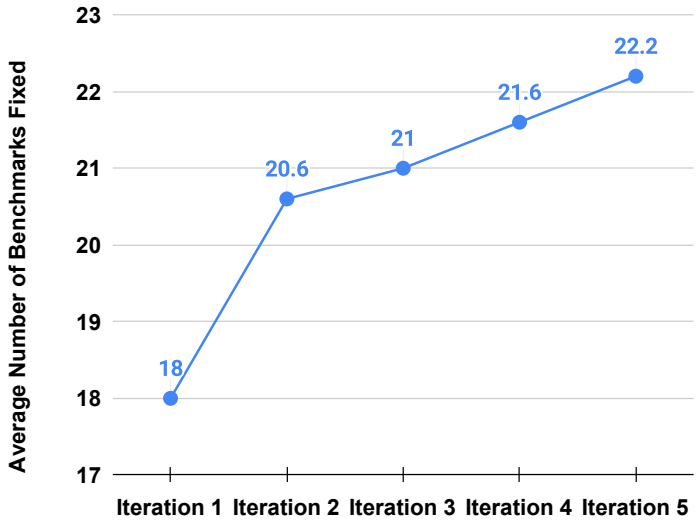

Figure 6.3: Average Number of Benchmarks Fixed per Iteration across 5 Runs. The x-axis shows iteration numbers, while the y-axis represents the average fixes.

indicating a significant improvement as the feedback technique began to take effect.

In Iteration 3, the average rose slightly to 21 benchmarks, showing a slower rate of improvement compared to earlier iterations. This trend continued into Iteration 4, with the average reaching 21.6 benchmarks. By Iteration 5, the cumulative average reached 22.2 benchmarks, reflecting minimal but consistent progress in the later iterations.

Overall, the data suggests that the most substantial improvements occurred during the first two iterations, with diminishing returns in subsequent iterations. This aligns with the observation that the feedback mechanism was most effective early in the iterative process.

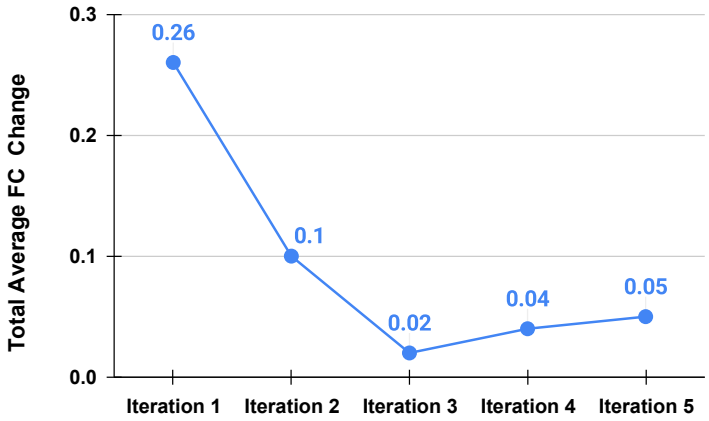

Figure 6.4: Total Average FC Change for every iteration across all 5 runs

As shown in Fig. 6.4, the total average FC values were calculated by first averaging the FC changes for each benchmark within each iteration and run. For each run, we computed the average FC for all benchmarks in that iteration. Then, for each iteration, we averaged the results across all five runs to get the total average FC value for that iteration. This process was repeated for each iteration, from Iteration 1 to Iteration 5, providing an overall measure of the LLM's performance across all benchmarks and runs.

The results reveal a clear trend: Iteration 1 achieved the highest average FC value of 0.26, demonstrating significant improvement over the baseline and highlighting the effectiveness of the LLM in its initial attempt to fix the benchmarks. However, the FC values dropped substantially in subsequent iterations, as the remaining benchmarks were likely more challenging. Iteration 2 achieved a positive but reduced FC value of 0.10, while Iteration 3 showed the lowest FC value of 0.02, indicating minimal progress. Iterations 4 and 5 showed slight recoveries with average FC values of 0.04 and 0.05, respectively, but these gains remained modest compared to Iteration 1.

This decline in FC values across iterations reflects the increasing difficulty of fixing the remaining benchmarks, as those successfully fixed in earlier iterations were no longer included. Nonetheless, the consistently positive FC values across all iterations confirm the LLMs ability to outperform the baseline, with the most significant improvements occurring in the first iteration and diminishing returns in subsequent attempts as the focus shifted to the most challenging examples.

Table 6.1: Feedback Metrics: Number of Attempts (Total attempts to process all benchmarks per run), Pass Attempts, Failed Attempts, Pass Percentage, Total Cost ($), and Average across all runs.

Metric	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Average**
Number of Attempts	81	75	77	83	78	78.8
Pass Attempts	80	70	73	78	74	75
Failed Attempts	1	5	4	5	4	3.8
Percentage of Pass	98.77%	93.33%	94.81%	93.98%	94.87%	95.15%
Total Cost ($)	$3.2	$3.44	$3.32	$3.38	$3.07	$3.28
Pass@1	0.69375					N/A

The results, shown in Table 6.1, provide an overview of the performance across five runs, focusing on key metrics such as the number of attempts, pass attempts, failed attempts, percentage of pass, and total cost.

Number of Attempts: The number of attempts varied slightly across the runs. Run 4 required the most attempts (83), while Run 2 required the fewest (75). On average, each run required (78.8) attempts to complete all benchmarks. The maximum number of attempts for the entire benchmark per run is 160. A lower number of attempts indicates better performance, as it suggests the model was able to find fixes quicker. This slight variation indicates that the number of attempts was relatively consistent across the different runs.

Pass Attempts: The number of pass attempts ranged from 70 in Run 2 to 80 in Run 1. On average, 75 attempts were successful across all runs. This suggests that the majority of attempts resulted in passing, with a relatively high number of successful fixes in each run.

Failed Attempts: The number of failed attempts varied from a minimum of 1 in Run 1 to a maximum of 5 in Runs 2 and 4. On average, there were 3.8 failed attempts per run. Despite this, the low number of failures suggests that the technique was generally effective in fixing the benchmarks.

Percentage of Pass: The percentage of pass remained consistently high across all runs, ranging from 93.33% in Run 2 to 98.77% in Run 1. The average pass percentage across all runs was 95.15%, indicating that the model was highly successful in addressing the benchmarks in most iterations.

Total Cost: The total cost per run varied from $3.07 in Run 5 to $3.44 in Run 2, with an average cost of $3.28 per run. These minor fluctuations in cost reflect the relatively small differences in resource usage across the runs.

Pass@1: The **Pass@1** metric for the feedback experiment is 0.69375, calculated across the 5 runs conducted in this experiment, where each run is treated as a single trial. This metric reflects the proportion of benchmarks successfully fixed within the first run itself. Achieving a Pass@1 value of 0.69375 indicates a commendable performance of the feedback approach across all runs.

The results suggest that the feedback technique applied to large language models (LLMs) yielded consistent performance across all runs. The majority of successful fixes occurred in the first and second iterations, with some incremental progress observed in the subsequent iterations. This pattern indicates that the feedback method showed strong initial results, with diminishing progress

in later iterations. The high pass percentage across all runs and the low number of failed attempts suggest that the feedback approach was reliable in addressing the benchmarks. The Pass@1 result is approximately 0.7, which is considered a good outcome, especially when compared to other strategies in different techniques, where it outperforms them.. While the results show promise, there is potential for further improvement, particularly in the later iterations. Overall, the feedback technique demonstrates the feasibility of utilizing LLMs in this context, providing a solid foundation for further refinement and exploration

6.4 Discussion and Observations

In this experiment, we utilized a probability metric to measure the likelihood of successful bug fixes across multiple runs of the feedback technique. This metric provides a quantifiable way to evaluate the model's performance, considering the nondeterministic nature of large language models (LLMs).

The probability metric P_{fixes} is a general measure that calculates the likelihood of an event occurring over several trials. It is defined as the ratio of the number of successful fixes to the total number of trials:

$$P_{\text{fixes}} = \frac{\text{Number of fixes}}{\text{Total number of trials}} \tag{6.1}$$

In the context of our experiment, each example was evaluated over 5 runs to account for the LLM's inherent variability. For every example, each run consisted of up to 5 iterations using the feedback technique. The result of each run after completing the iterations was binary: either the model successfully fixed the issue (1) or failed to fix it (0). The probability of an example being fixed was then calculated as the proportion of successful fixes across the 5 runs. Specifically, the metric is represented as:

$$P_{\text{fixes}} = \frac{\text{Number of successful fixes (1s) across 5 runs}}{5} \tag{6.2}$$

This approach ensures that the metric reflects the model's average performance and accounts for the randomness in its output. By using this probability metric, we can better understand the effectiveness of the feedback technique in resolving issues under nondeterministic conditions.

6.4.1 Analysis of Probability of Fixes Metric

Table 6.2: Probability of fix for each buggy file across 5 runs

Project	Simplified Buggy File Name	Probability of fixes
decoder_3_to_8	decoder_numeric_error	1
	decoder_assignment_error	1
counter	counter_sensitivity_issue	1
	counter_increment_error	0.6
	counter_reset_issue	1
flip_flop	flipflop_conditional_error	1
	flipflop_branch_swap	1
fsm_full	fsm_case_statement_issue	1
	fsm_state_assignment_missing	0
	fsm_state_sensitivity_issue	1
	fsm_blocking_assignment_error	1
lshift_reg	lshift_blocking_issue	0.8
	lshift_conditional_error	1
	lshift_sensitivity_issue	1
mux_4_1	mux_numeric_issues	1
	mux_hex_error	1
	mux_bitwidth_mismatch	1
i2c	i2c_sensitivity_issue	0.8
	i2c_address_assignment_error	1
	i2c_acknowledge_missing	0.8
sha3	sha3_loop_boundary_issue	0
	sha3_wire_assignment_error	0
	sha3_buffer_overflow_skipped	0
	sha3_bitwise_negation_issue	0.2
tate_pairing	tatepairing_bitshift_logic_error	0
	tatepairing_operator_mismatch	0.6
	tatepairing_module_instantiation	0.2
reed_solomon_decoder	reedsolomon_register_size_error	0.2
	reedsolomon_reset_sensitivity	1
sdram_controller	sdram_sync_reset_issue	1
	sdram_numeric_definition_error	1
	sdram_case_statement_issue	0

The data in Table 6.2 provides the probability-of-fix metric for various buggy files across different projects, calculated using Equation 6.2. The probability values represent the likelihood that the LLM model successfully identifies and fixes the issues within the files after running the feedback technique for five runs. These five runs serve as the maximum number of trials for the model to attempt and find a fix. Below, we group the buggy files by their respective projects and analyze the

results.

High Probability of Fixes (1)

The files with a probability of 1 indicate that the LLM model was able to consistently fix the issues across all five runs. These files represent cases where the LLM successfully understood the functionality of the file and applied the necessary fixes. The projects with 100% fix probabilities include:

- **decoder_3_to_8:** decoder_numeric_error, decoder_assignment_error
- **counter:** counter_sensitivity_issue, counter_reset_issue
- **flip_flop:** flipflop_conditional_error, flipflop_branch_swap
- **fsm_full:** fsm_case_statement_issue, fsm_state_sensitivity_issue, fsm_blocking_assignment_error
- **lshift_reg:** lshift_conditional_error, lshift_sensitivity_issue
- **mux_4_1:** mux_numeric_issues, mux_hex_error, mux_bitwidth_mismatch
- **i2c:** i2c_address_assignment_error
- **reed_solomon_decoder:** reedsolomon_reset_sensitivity
- **sdram_controller:** sdram_sync_reset_issue, sdram_numeric_definition_error

These results suggest that the LLM was able to consistently identify and resolve the issues in these files. The files were likely straightforward in terms of functionality, and the LLM was able to accurately diagnose and apply the fixes. This demonstrates the LLMs effectiveness in handling certain types of design bugs in these projects.

Moderate Probability of Fixes (0.6 - 0.8)

Files with probabilities between 0.6 and 0.8 show that the LLM was able to fix the issues in most runs, but not consistently. These files represent more complex or ambiguous bugs that might require more specific context or a better understanding of the design. These cases include:

- **counter:** counter_increment_error (0.6)
- **lshift_reg:** lshift_blocking_issue (0.8)
- **i2c:** i2c_sensitivity_issue (0.8), i2c_acknowledge_missing (0.8)
- **tate_pairing:** tatepairing_operator_mismatch (0.6), tatepairing_module_instantiation (0.2)

These bugs were not resolved in every instance, but the LLM demonstrated a reasonable level of success in identifying fixes in a majority of the attempts. The fact that these files had moderate fix probabilities suggests that while the LLM could sometimes fix them, it might have struggled with the complexity of the bugs or lacked sufficient information to apply the fixes in all instances.

Low Probability of Fixes (0 - 0.2)

Files with probabilities between 0 and 0.2 represent cases where the LLM was unable to find a fix in most or all runs. These bugs were likely challenging for the model to understand or required a deeper level of insight into the design, which the LLM may not have had. These include:

- **sha3:** sha3_loop_boundary_issue (0), sha3_wire_assignment_error (0), sha3_buffer_overflow_skipped (0)
- **tate_pairing:** tatepairing_bitshift_logic_error (0)
- **reed_solomon_decoder:** reedsolomon_register_size_error (0.2)
- **sdram_controller:** sdram_case_statement_issue (0)

These results suggest that, even after multiple iterations, the LLM model struggled to resolve these issues. The bugs may be particularly complex or may require domain-specific knowledge that the LLM model does not possess. Additionally, the model may not have been able to process the necessary context to make an accurate fix. These cases highlight areas where the LLM might need further refinement or more targeted feedback to successfully identify solutions.

In conclusion, The data indicates that the LLM model demonstrated promising performance in resolving bugs in certain projects, with high fix probabilities for many files. The files with a 100% fix probability show that the LLM can effectively understand and address simple to moderately

complex design issues. However, there were also cases with moderate and low fix probabilities, suggesting that the LLM may need more information or refinement to handle more complex bugs. This highlights the potential of the feedback technique in automating hardware design bug fixes, but also points to areas for future improvement, especially when dealing with more intricate or context-dependent issues.

Chapter 7

Discussion

This chapter synthesizes the findings presented in Chapters 4, 5, and 6, offering a comprehensive analysis of the results and evaluating the effectiveness of the various prompting techniques explored. It discusses the implications of these findings for automatic bug repair in hardware design and their potential broader impact on the field. Additionally, the chapter includes a comparison with prior studies that utilized the same benchmark, emphasizing similarities, differences, and improvements.

7.1 Comparison of All Techniques

In this section, we compare the results of all the experiments presented in previous chapters, as summarized in Table 7.1. For each technique, we analyze the performance of structured and unstructured prompt variations, identify the best-performing strategies, and evaluate results based on Pass@1, Total Average FC Change, Number of Benchmarks Fixed, and Average Number of Input Prompt Tokens.

7.1.1 Zero-Shot Technique

For the zero-shot technique, the highest Pass@1 value is achieved by unstructured strategy 0, with a score of 0.55625. This indicates that the probability of obtaining a correct fix on the first attempt is higher for this strategy. In contrast, strategy 1 in both prompt variations results in the lowest Pass@1 values, suggesting that the inclusion of the mismatch list may have confused the model.

Table 7.1: Total comparison of all the techniques, prompt variations, and strategies presented in Chapters 4, 5, and 6, evaluated based on Pass@1, Total Average FC Change, Number of Benchmarks Fixed, and Average Number of Input Prompt Tokens.

Techniques	Prompt variation	Strategies	Pass@1	Total Average FC change	Number of Benchmarks Fixed	Average number of input prompt tokens
Zero-shot	Unstructured	Strategy0	0.55625	0.28	22	1684
		Strategy1	0.39375	0.25	18	1882
		Strategy2	0.46875	0.24	20	3340
		Strategy3	0.45	0.24	22	2065
		Strategy4	0.5125	0.25	23	3722
		Strategy5	0.45625	0.24	20	3841
	Structured	Strategy0	0.50625	0.29	22	1703
		Strategy1	0.3875	0.24	18	1882
		Strategy2	0.475	0.25	21	3347
		Strategy3	0.45	0.28	19	2149
		Strategy4	0.45625	0.24	20	3809
		Strategy5	0.425	0.23	19	3927
Few-shot	Unstructured	Strategy0	0.475	0.27	21	3183
		Strategy1	0.33125	0.22	15	3372
		Strategy2	0.425	0.25	20	4911
		Strategy3	0.40625	0.24	18	3563
		Strategy4	0.45625	0.23	20	5291
		Strategy5	0.45625	0.24	19	5409
	Structured	Strategy0	0.41875	0.26	20	3414
		Strategy1	0.35625	0.25	17	3531
		Strategy2	0.4125	0.24	20	5071
		Strategy3	0.4	0.24	19	3794
		Strategy4	0.4	0.22	20	5457
		Strategy5	0.4625	0.27	23	5575
Feedback	Feedback prompts (5 Runs)	Feedback Logic	0.69375	0.094	26	3708

In terms of Total Average FC Change, all strategies exhibit similar values. However, strategy 0 in both variations achieves the highest improvement, indicating that, on average, the model's fixes are more effective at improving the buggy code.

When considering the Number of Benchmarks Fixed, most strategies yield comparable results. However, unstructured strategy 4 achieves the highest number of fixes, successfully addressing 23 buggy examples. Strategy 1, on the other hand, performs the worst in both variations, fixing only 18 examples.

7.1.2 Few-Shot Technique

In the few-shot technique, the Pass@1 values are generally lower than in zero-shot, with unstructured strategy 0 achieving the best performance at 0.475. This decline highlights the increased complexity of this approach.

For Total Average FC Change, there is a tie between unstructured strategy 0 and structured strategy 5, both achieving a value of 0.27. Regarding the Number of Benchmarks Fixed, structured strategy 5 performs the best, successfully fixing 23 examples.

7.1.3 Feedback Technique

The feedback technique builds on the results of the zero-shot and few-shot approaches by iteratively refining the prompts with varying strategies. Across five runs, the feedback technique achieves the highest Pass@1 value at 0.69375. However, it records a low Total Average FC Change of 0.094. Despite this, it fixes the largest number of benchmarks, with 26 examples successfully repaired. Additionally, the Average Number of Input Prompt Tokens for the feedback technique is competitive, balancing effectiveness with cost.

7.1.4 Cost Analysis Based on Token Usage

For the Average Number of Input Prompt Tokens (a key factor in determining computational cost), a similar trend is observed across both techniques. The token count increases as more information is added to the prompt, such as additional examples and guidelines. Strategy 2, in particular, shows a noticeable spike in token usage due to the inclusion of substantial information, such as the difference between simulation output and the oracle. This demonstrates how the level of detail in the input can significantly impact the computational cost.

Using these token counts, the total cost of running these strategies was calculated based on OpenAI's pricing [58]. At the time of the experiments, the pricing for our model, gpt-4o-2024-05-13, was $5.00 per 1M input tokens and $15.00 per 1M output tokens. These costs may vary as models and pricing evolve.

7.1.5 General Observations

Overall, the zero-shot technique outperformed the few-shot technique in several metrics, with unstructured prompt variations demonstrating better performance in zero-shot, while structured prompts slightly outperformed unstructured ones in the few-shot approach. The feedback technique showed significant potential, achieving the highest number of fixed examples among all techniques.

These findings indicate that even providing the model with minimal information, such as the buggy code alone, can yield strong results with a lower token count. However, adding extra details does not always lead to better performance. For instance, in Strategy 1, the additional information sometimes misled the model, resulting in less effective outputs. This suggests that functional bugs are inherently challenging to identify and fix without precise details regarding the bug type, location, and project requirements.

The difficulty of repairing specific examples further illustrates these observations. While most designs in the benchmark were successfully fixed, certain bugs proved particularly challenging. For example, in the SDRAM Controller Project, the bug in the example *sdram case statement issue* proved particularly challenging for the zero-shot and feedback techniques. In this case, the bug involved the erroneous removal of the default case from the code, which the model struggled to fix. This bug was only resolved in one of the few-shot strategies, where the model was provided with guidelines and examples. Similarly, in the Tate Pairing Design, bugs like the *tatepairing bitshift logic error* and *tatepairing module instantiation* were hard to resolve across most techniques and strategies. The SHA3 design posed the greatest challenges, as two examples—*sha3 loop boundary issue* and *sha3 wire assignment error*—remained unfixed across all techniques and strategies presented in this thesis. These persistent failures highlight the limitations of the model when dealing with complex bugs.

These examples underscore the importance of selecting the most effective type of information to include in the prompt. While specific, targeted details can enhance the model's ability to generate accurate fixes, overly generic or irrelevant information can reduce performance, emphasizing the need for careful prompt engineering.

Table 7.2: Comparison of results with prior work and the best strategies from zero-shot, few-shot, and feedback techniques. The table compares our approach with prior work [1] and [4], where ✓ indicates a design that achieves $FC = 1$, and ✗ indicates no fix. We include the best strategies from each of the three techniques (zero-shot, few-shot, feedback) and their combined results. Our method fixed 28 out of 32 designs, surpassing the 16 designs fixed by Cirfix and the 22 fixed by Ahmad et al

Project	Defect Description	Fixed? from [1]	Fixed? from [4] [var a]	Fixed? from [4] [var b]	Unstrucutred zero-shot Strategy4	Structured Few-shot Strategy5	Feed-back	Com-bined
decoder_3_to_8	Two separate numeric errors	✓	✓	✓	✓	✓	✓	✓
	Incorrect assignment	✗	✗	✗	✓	✓	✓	✓
counter	Incorrect sensitivity list	✓	✗	✗	✓	✓	✓	✓
	Incorrect incremental of counter	✓	✓	✓	✗	✓	✓	✓
	Incorrect reset	✓	✓	✓	✓	✓	✓	✓
flip_flop	Incorrect conditional	✓	✗	✗	✓	✓	✓	✓
	Branches of if-statement swapped	✓	✓	✓	✓	✓	✓	✓
fsm_full	Incorrect case statement	✗	✗	✗	✓	✓	✓	✓
	Assignment to next state and default in case statement omitted	✗	✓	✓	✓	✓	✗	✓
	Assignment to next state omitted, incorrect sensitivity list	✓	✗	✓	✓	✓	✓	✓
	Incorrectly blocking assignments	✗	✗	✗	✓	✓	✓	✓
lshift_reg	Incorrect blocking assignment	✓	✗	✓	✓	✓	✓	✓
	Incorrect conditional	✓	✓	✓	✓	✓	✓	✓
	Incorrect sensitivity list	✓	✓	✗	✓	✓	✓	✓
mux_4_1	Three separate numeric errors	✗	✓	✓	✓	✓	✓	✓
	Hex instead of binary constants	✗	✓	✓	✓	✓	✓	✓
	1 bit instead of 4 bit output	✗	✓	✓	✓	✓	✓	✓
i2c	Incorrect sensitivity list	✓	✓	✓	✗	✗	✓	✓
	Incorrect address assignment	✗	✗	✗	✓	✗	✓	✓
	No command acknowledgment	✓	✗	✗	✓	✓	✓	✓
sha3	Off-by-one error in loop	✓	✗	✓	✗	✗	✗	✗
	Incorrect assignment to wires	✗	✓	✓	✗	✗	✗	✗
	Skipped buffer overflow check	✓	✗	✗	✗	✗	✗	✗
	Incorrect bitwise negation	✗	✓	✓	✓	✗	✓	✓
tate_pairing	Incorrect logic for bitshifting	✗	✓	✓	✓	✓	✗	✓
	Incorrect operator for bitshifting	✗	✗	✓	✗	✗	✓	✓
	Incorrect instantiation of modules	✗	✓	✓	✗	✗	✓	✓
reed_solomon_decoder	Insufficient register size for decimal values	✗	✓	✓	✗	✓	✓	✓
	Incorrect sensitivity list for reset	✓	✓	✓	✓	✓	✓	✓
sdram_controller	Incorrect assignments to registers during synchronous reset	✓	✓	✓	✓	✓	✓	✓
	Numeric error in definitions	✗	✓	✓	✓	✓	✓	✓
	Incorrect case statement	✗	✗	✗	✗	✗	✗	✗
Total Values		16	19	22	23	23	26	28

7.2 Comparisons with Prior work

In comparison with prior work [1, 4], our results demonstrate that the best-performing strategies from zero-shot, few-shot, and feedback techniques achieve superior performance in terms of the Number of Benchmarks Fixed metric. This is particularly notable as our approach does not assume prior knowledge of the bug location or type, whereas previous methods relied on such assumptions.

Table 7.2 highlights that our unstructured zero-shot Strategy 4 fixed 23 examples, while the structured few-shot Strategy 5 also fixed 23 examples. Furthermore, the feedback technique, run in five iterations, successfully fixed 26 examples from the benchmark. These results outperform Ahmad et al. [4], where 19 examples were fixed using variation A and 22 using variation B, and Cirfix [1], which fixed 16 examples. This demonstrates that our approach, free of bug-specific assumptions, offers a more robust and generalizable solution.

If all the best-performing strategies from each technique were applied in parallel, our approach would produce 28 fixed examples. This combined result further underscores the strength of leveraging multiple prompting strategies. When compared to Cirfix [1], our findings suggest that LLMs are a viable alternative to genetic algorithms for generating potential bug fixes.

Regarding computational efficiency, our approach also demonstrates significant advantages. Running the structured few-shot Strategy 5 for five parallel runs achieved an average repair time of 14.9 seconds, while the unstructured zero-shot Strategy 4 required 14.8 seconds for the same configuration. The feedback technique, when run for five iterations in parallel, achieved an average repair time of 20.5 seconds. Although direct comparison is challenging due to differences in computational resources, these times are substantially faster than the hours reported in [1], suggesting a potential speedup using our methodology.

An in-depth analysis of the benchmark examples reveals that our approach achieves comparable or better results than prior work in most cases.

However, there are notable exceptions. For three examples in the `sha3` project, our approach was unable to provide fixes across any of the strategies or techniques. These examples involve the following bugs: *Off-by-one error in loop*, *Incorrect assignment to wires*, and *Skipped buffer overflow check*. These examples were resolved in [4], where the model was given specific information about the bug type and location. This suggests that such examples may require different contextual

information to facilitate repair.

Similarly, for the `sdram_controller` project, the *incorrect case statement* bug was not resolved by any of our best-performing strategies or by prior work, indicating the inherent difficulty of addressing this particular issue.

7.3 Limitations

While this thesis provides valuable insights into the application of LLMs for RTL functional bug repair, several limitations need to be acknowledged.

Benchmark Choice: The benchmark used in this study was selected primarily due to its availability for use and modification and its use in the literature. It was created to evaluate CirFix [1], a state-of-the-art framework in hardware bug repair, and was also adopted in another recent study [4]. This choice facilitated a straightforward comparison of our results with prior works, enabling us to measure progress effectively. However, the benchmark is relatively small, comprising only 32 examples. This limited size means it might not cover the full range of functional bug categories or exhibit significant diversity in projects and examples. As a result, the generalizability of the findings to broader and more complex scenarios may be limited. Nevertheless, this work remains comparable to other studies in the field.

Model Selection: gpt-4o-2024-05-13 was chosen as the LLM [32] for this research because it was the newest model available at the time of the experiments. Its large input token limit (128,000 tokens) and output token capacity (4,096 tokens) allowed us to send and process larger files, making it well-suited for handling Verilog code and associated contextual information. While gpt-4o-2024-05-13 provided distinct advantages, it is important to note that newer models are introduced every year, each trained on larger datasets and often exhibiting improved performance over previous versions. As the field of LLMs continues to evolve, newer models may yield better outcomes or improved performance, which is a limitation of relying on gpt-4o-2024-05-13 as the sole model in this research. However, our framework can be used to evaluate future models.

Model Parameters: We used the default out-of-the-box model parameters, as detailed in Section 3.3 and guided by observations from previous work, where modifying the parameters from the provided defaults did not change the LLMs' success. However, future research could explore

alternative parameter settings that could be more effective for fixing hardware bugs, which were not investigated in this work.

Addressing the Nondeterministic Behavior of LLMs: LLMs often exhibit non-deterministic behavior, meaning that their outputs can vary even when the same prompt is provided multiple times. To account for this, each strategy in every technique was run five times. While there is no established standard for the number of runs required to address this variability, we found that five runs produced consistent results, balancing reliability and practical constraints. Conducting more runs, such as 20 or 30, might yield slightly better insights into the model's variability, but time and cost limitations made this infeasible for this thesis.

Possibility of Benchmarks Appearing in the Training Data: Since all the benchmarks and datasets used in our experiments are open source, it is possible that some of these benchmarks were included in the training data of the foundational LLM. Despite this potential inclusion, our experiments reveal that certain bugs remain unresolved, and the outcomes vary significantly depending on the prompting strategies employed. This suggests that even if the model encountered similar examples during training, there is no guarantee that the model will inherently possess the ability to fix these bugs without effective prompt engineering and additional information about the bugs. This observation opens up an interesting avenue for future research to further investigate the impact of training data composition on model performance in bug repair tasks.

Techniques and Prompts Used: This work focuses on three techniques—zero-shot, few-shot, and feedback; these were among the most commonly used in related research, as discussed in Section 2.2.3. However, future research can further explore constantly emerging techniques and prompt variations that could further enhance the effectiveness of automatic hardware program repair. The work in this thesis provides a potential baseline for comparison.

Despite the aforementioned limitations, this thesis provides meaningful insights into the field of automatic hardware bug repair using LLMs. The selection of tools, benchmarks, and models ensures that the findings are comparable to prior works.

Chapter 8

Conclusion and Future Work

This thesis investigated the effectiveness of large language models (LLMs) in repairing RTL functional bugs using three distinct techniques: zero-shot, few-shot, and feedback. These techniques varied in their use of contextual information and the way prompts were structured to guide the LLM in performing repairs.

The results demonstrate that LLMs can understand Verilog code and detect functional bugs even without prior knowledge of bug types or locations. While the model's ability to generate successful repairs slightly improved in some cases as more contextual information was added to the prompt, not all bugs were resolved successfully. This highlights the importance of prompt engineering and underscores the need for further exploration of how input information impacts repair performance.

The proposed methodology achieves comparable results to previous approaches that relied on precise bug location information, suggesting the potential of LLM-driven automation to streamline RTL functional bug repair. Below, we revisit the research questions posed in this thesis and summarize our findings:

RQ1: How effective are LLMs at performing automatic hardware repairs without prior examples (zero-shot)?

Zero-shot techniques, which rely solely on the LLM's pretrained knowledge, proved effective in repairing RTL functional bugs. The best-performing zero-shot strategy achieved a success rate of 71.88% (23 out of 32 benchmarks). This indicates that LLMs can detect and fix functional bugs even without any information about the bug type or location, highlighting

their capability as a baseline approach for hardware repair.

RQ2: How does providing LLMs with example repairs and additional information influence their ability to perform automatic hardware repairs (few-shot)?

The few-shot technique, which supplements the LLM with examples and general repair guidelines, also achieved a success rate of 71.88% (23 out of 32 benchmarks). While this performance matches that of zero-shot, the results suggest that providing generic examples and guidelines may not significantly improve repair success when the examples are not tailored to specific bug categories. This finding points to the importance of aligning examples and context more closely with the type of bugs being repaired.

RQ3: How well do LLMs perform when feedback and multiple queries are involved?

The feedback technique, which uses iterative prompting and incorporates simulation outputs and prior results into subsequent prompts, achieved the highest success rate among the three techniques, with 81.25% (26 out of 32 benchmarks) of bugs repaired. This demonstrates that feedback mechanisms are highly effective, as they allow the LLM to refine its responses incrementally based on evolving context and previous iterations. The results highlight the promise of feedback-based approaches for improving repair success rates in complex scenarios.

8.1 Takeaways

When comparing the three techniques, we observed that zero-shot and few-shot methods achieved similar repair rates, suggesting that generic repair guidelines and examples in few-shot did not provide significant advantages over the pretrained knowledge of the LLM. In contrast, the feedback technique outperformed both zero-shot and few-shot approaches, achieving the highest fix rates. This success can be attributed to its iterative process, which incorporates contextual updates and leverages information from previous iterations, allowing the LLM to address complex repair tasks more effectively. These findings highlight the potential of feedback mechanisms as a promising direction for enhancing automatic hardware repair using LLMs. While zero-shot and few-shot methods provide a solid baseline, feedback-based approaches demonstrate the ability to tackle more challenging scenarios and significantly improve repair performance.

8.2 Future Work

While the research presented in this thesis demonstrates the effectiveness of various techniques, strategies, and prompt variations in addressing functional bugs, several opportunities for future work have been identified. One such direction involves experimenting with newer language models that have been trained on a broader range of HDLs. These models may possess a deeper understanding of hardware-specific syntax and semantics, and by incorporating additional information about the project requirements, they might better understand the bug types, enabling more accurate bug detection and repair in Verilog and other HDLs.

Additionally, future research could focus on improving the accuracy of bug identification by exploring alternative techniques. This could involve experimenting with new methods for automatically detecting bugs. By enhancing these techniques, the system could become more precise in locating bugs and providing more accurate fixes.

Another promising direction is fine-tuning the language model specifically for Verilog code and bug repair. This process would involve curating or creating a dataset that includes Verilog code, known bugs, and their corresponding fixes. By training the model on such data, it would gain a deeper understanding of Verilog syntax, semantics, and common bug patterns, allowing it to generate more accurate and domain-specific fixes.

In addition to these approaches, a multi-agent framework could further enhance bug detection and repair by enabling specialized agents to collaborate on distinct tasks, such as identifying bug types, proposing fixes, and validating solutions. By combining their strengths, agents could address functional bugs, security vulnerabilities, and performance issues more effectively and efficiently.

The model could be extended to address security-related bugs by incorporating data from the CWE database. CWE [15] is a community-developed list of common software and hardware security weaknesses, providing a classification of vulnerabilities to help organizations identify and address potential security risks in their code. For instance, training the model on existing CWEs and continuously updating it with newly identified CWEs would enhance its ability to identify and repair security vulnerabilities. This iterative retraining would result in a model finely tailored to Verilog, with a specialized focus on both general and security-specific bugs.

Integrating this fine-tuned model into development environments could further enhance its

utility. For example, pairing the model with tools that detect and fix bugs during code writing could streamline the debugging process, reducing both development time and costs for companies. This integration could serve as a valuable asset in ensuring higher code quality and mitigating security risks in real time.

Bibliography

[1] H. Ahmad, Y. Huang, and W. Weimer, "CirFix: automatically repairing defects in hardware design code," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 990–1003. [Online]. Available: https://dl.acm.org/doi/10.1145/3503222.3507763

[2] "Intel 2023 Product Security Report," https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2024-01/intel-2023-product-security-report.pdf, accessed: 2024-06-01.

[3] C. L. Goues *et al.*, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, p. 56–65, nov 2019.

[4] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "On Hardware Security Bug Code Fixes by Prompting Large Language Models," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4043–4057, 2024, conference Name: IEEE Transactions on Information Forensics and Security. [Online]. Available: https://ieeexplore.ieee.org/document/10462177

[5] 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 Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 3, Apr. 2024.

[6] P. Vijayaraghavan, A. Nitsure, C. Mackin, L. Shi, S. Ambrogio, A. Haran, V. Paruthi, A. Elzein, D. Coops, D. Beymer, T. Baldwin, and E. Degan, "Chain-of-descriptions: Improving code llms for vhdl code generation and summarization," in *Proceedings of the 2024 ACM/IEEE*

International Symposium on Machine Learning for CAD, ser. MLCAD '24. Association for Computing Machinery, 2024.

[7] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, Aug 2014.

[8] F. F. Xu *et al.*, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, Jun. 2022.

[9] S. Thakur *et al.*, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation &; Test in Europe Conference &; Exhibition (DATE)*. IEEE, Apr. 2023.

[10] Z. Fan *et al.*, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.

[11] C. S. Xia *et al.*, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1482–1494.

[12] AbdelrahmanElnaggar/automatic_rtl_repair_llm: Investigating automatic bug repair using large language models for digital hardware design. [Online]. Available: https://github.com/AbdelrahmanElnaggar/Automatic_RTL_Repair_LLM

[13] Y.-D. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models," May 2024, arXiv:2311.16543 [cs]. [Online]. Available: http://arxiv.org/abs/2311.16543

[14] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable Hardware Bugs," in *Usenix Security Symp.*, 2019, pp. 213–230. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

[15] "CWE - Common Weakness Enumeration." [Online]. Available: https://cwe.mitre.org/

[16] D. N. Gadde, A. Kumar, T. Nalapat, E. Rezunov, and F. Cappellini, "All artificial, less intelligence: Genai through the lens of formal verification," 2024. [Online]. Available: https://arxiv.org/abs/2403.16750

[17] B. Ahmad *et al.*, "Don't cweat it: Toward cwe analysis techniques in early stages of hardware design," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. ACM, Oct. 2022.

[18] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, "A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[19] N. S. Soliman, K. Salah, and A. H. Madian, "Automatic rtl coding correction linting tool for critical issues," in *2018 30th International Conference on Microelectronics (ICM)*, 2018, pp. 252–255.

[20] "Understanding formal verification." [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2024/09/05/understanding-formal-verification/#

[21] T.-Y. Jiang, C.-N. Liu, and J. Y. Jou, "Estimating likelihood of correctness for error candidates to assist debugging faulty hdl designs," in *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2005, pp. 5682–5685 Vol. 6.

[22] B. Ahmad, B. Tan, R. Karri, and H. Pearce, "Flag: Finding line anomalies (in code) with generative ai," 2023. [Online]. Available: https://arxiv.org/abs/2306.12643

[23] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[25] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.00515

[26] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting verilogeval: Newer llms, in-context learning, and specification-to-rtl tasks," 2024. [Online]. Available: https://arxiv.org/abs/2408.11053

[27] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639222

[28] X. Yao, H. Li, T. H. Chan, W. Xiao, M. Yuan, Y. Huang, L. Chen, and B. Yu, "HDLdebugger: Streamlining HDL debugging with Large Language Models," Mar. 2024, arXiv:2403.11671 [cs]. [Online]. Available: http://arxiv.org/abs/2403.11671

[29] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "AutoChip: Automating HDL Generation Using LLM Feedback," Jun. 2024, arXiv:2311.04887 [cs]. [Online]. Available: http://arxiv.org/abs/2311.04887

[30] A. Ayalasomayajula, R. Guo, J. Zhou, S. K. Saha, and F. Farahmandi, "Lasp: Llm assisted security property generation for soc verification," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, ser. MLCAD '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3670474.3685967

[31] 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," 2023. [Online]. Available: https://arxiv.org/abs/2401.12224

[32] OpenAI, "OpenAI Platform." [Online]. Available: https://platform.openai.com

[33] R. Corchuelo, J. A. Pérez, A. Ruiz, and M. Toro, "Repairing syntax errors in lr parsers," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, p. 698–710, Nov. 2002. [Online]. Available: https://doi.org/10.1145/586088.586092

[34] A. Griesmayer, R. Bloem, and B. Cook, "Repair of boolean programs with an application to c," in *Proceedings of the 18th International Conference on Computer Aided Verification*, ser. CAV'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 358–371. [Online]. Available: https://doi.org/10.1007/11817963_33

[35] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, no. 5, p. 109–116, May 2010. [Online]. Available: https://doi.org/10.1145/1735223.1735249

[36] M. Monperrus, "The Living Review on Automated Program Repair," HAL Archives Ouvertes, Technical Report hal-01956501, 2018. [Online]. Available: https://www.monperrus.net/martin/repair-living-review.pdf

[37] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan, "Planning with large language models for code generation," *arXiv preprint arXiv:2303.05510*, 2023.

[38] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[39] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2103.06333

[40] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, Feb. 2017. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/10742

[41] R. Gupta, A. Kanade, and S. Shevade, "Deep reinforcement learning for syntactic error repair in student programs," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 930–937, Jul. 2019. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/3882

[42] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "InferFix: End-to-End Program Repair with LLMs," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*,

ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1646–1656. [Online]. Available: https://dl.acm.org/doi/10.1145/3611643.3613892

[43] "OpenAI codex editor," https://openai.com/index/gpt-3-edit-insert/, accessed: 2024-06-01.

[44] "What is a large language model (LLM)?" [Online]. Available: https://www.cloudflare.com/learning/ai/what-is-large-language-model/

[45] "Our approach to data and AI." [Online]. Available: https://openai.com/index/approach-to-data-and-ai/

[46] "OpenTitan Documentation." [Online]. Available: https://opentitan.org/book/hw/

[47] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," Aug. 2022, arXiv:2112.02125 [cs]. [Online]. Available: http://arxiv.org/abs/2112.02125

[48] OpenAI, "OpenAI Models." [Online]. Available: https://platform.openai.com/docs/models/

[49] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[50] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[51] K. Qayyum, M. Hassan, S. Ahmadi-Pour, C. K. Jha, and R. Drechsler, "From Bugs to Fixes: HDL Bug Identification and Patching using LLMs and RAG."

[52] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "LLM4SecHW: Leveraging Domain Specific Large Language Model for Hardware Debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, Dec. 2023, pp. 1–6, arXiv:2401.16448 [cs]. [Online]. Available: http://arxiv.org/abs/2401.16448

[53] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, Y. Yang, and Z. Chen, "A Systematic Literature Review on Large Language Models for Automated Program Repair," May 2024, arXiv:2405.01466 [cs]. [Online]. Available: http://arxiv.org/abs/2405.01466

[54] "Opencores.org," https://opencores.org/, accessed: 2024-06-01.

[55] H. Ahmad, "hammad-a/verilog_repair," Sep. 2024, original-date: 2020-05-15T21:09:28Z. [Online]. Available: https://github.com/hammad-a/verilog_repair

[56] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," Jul. 2021, arXiv:2107.03374. [Online]. Available: http://arxiv.org/abs/2107.03374

[57] "DLAI - ChatGPT Prompt Engineering for Developers." [Online]. Available: https://learn.deeplearning.ai/courses/chatgpt-prompt-eng/lesson/1/introduction

[58] "Pricing." [Online]. Available: https://openai.com/api/pricing/

Appendix

In Table A1, we provide additional details and information about the bugs and the corresponding files, all of which are available in our repository [12]. The repository contains the complete code and related files. The table includes the following columns:

1. **Simplified Bug Name**: A concise identifier for each bug.
2. **Buggy Files**: The specific design files where each bug is located.
3. **Buggy Lines**: The lines of code where the bug occurs.
4. **Detailed Bug Description**: An explanation of the bug, its cause, and the correct modification needed to fix it.

This table serves as a comprehensive reference for understanding and addressing the identified bugs.

Table A1: Additional Bug Details

Simplified Buggy File Name	Buggy files	Buggy Lines	Detailed bug description
decoder_numeric_error	decoder_3_to_8_wadden_buggy1.v	17,23	The bug involves incorrect binary assignments: 4'b1000 instead of 4'b1010 and 8'b0111_1111 instead of 8'b1111_1111, caused by errors in bit values.
decoder_assignment_error	decoder_3_to_8_wadden_buggy2.v	from 15 to 23	The bug in the code is that the buggy version uses 7-bit assignments (e.g., 8'b1111110), while the correct version uses 8-bit assignments (e.g., 8'b1111_1110), leading to incorrect output behavior.
counter_sensitivity_issue	first_counter_overflow_wadden_buggy1.v	34, 49	Incorrect sensitivity list: missing the edge trigger in the buggy code, which uses always@(clk) instead of the correct always@(posedge clk), leading to improper clock edge detection.
counter_increment_error	first_counter_overflow_wadden_buggy2.v	48,50	Incorrect overflow detection due to an else blocking the overflow check, fixed by removing the else.
counter_reset_issue	first_counter_overflow_kgoliya_buggy1.v	39, 50	The reset logic is incomplete due to a missing counter reset assignment. The line counter_out <= #1 4'b0000; is commented out, which prevents counter_out from being set to zero during reset, causing improper reset behavior
flipflop_conditional_error	tff_wadden_buggy1.v	line 8 in buggy code or 7 in correct code	The bug is caused by incorrectly using if (rstn) instead of if (!rstn), treating the active-low reset signal as active-high and leading to improper reset behavior.
flipflop_branch_swap	tff_wadden_buggy2.v	7, 11, 13	The bug swaps the branches of an if statement and incorrectly treats the reset signal as active high (if (rstn)), instead of active low (if (!rstn)), leading to reversed logic and improper initialization.
fsm_case_statement_issue	fsm_full_wadden_buggy1.v	104,105,106	The issue is that commenting out the lines for GNT3 (GNT3 : begin gnt_3 <= #1 1'b1; end) results in an incorrect case statement due to missing logic for GNT3.
fsm_state_assignment_missing	fsm_full_wadden_buggy2.v	41,74, 107, 108, 109	The bug omits the assignment to next_state and the default case in the case statement, causing potential undefined state behavior. Specifically, the deletion of three key components—next_state = 0;, default : next_state = IDLE;, and default : state <= #1 IDLE;—results in incorrect state transitions. The fix requires restoring these assignments to ensure proper default behavior and state transitions.
fsm_state_sensitivity_issue	fsm_full_ssscrazy_buggy1.v	39, 41 ,74	need to add state to this always block–>always @ (req_0 or req_1 or req_2 or req_3) , add next_state = 0; to line 41 and add default : next_state = IDLE; to line 74
fsm_blocking_assignment_error	fsm_full_ssscrazy_buggy2.v	Buggy lines from 81 to 108 OR from 87 to 114 in buggy code	The bug is that every <= assignment was changed to =, which alters the behavior of the code by changing non-blocking assignments to blocking assignments.
lshift_blocking_issue	lshift_reg_wadden_buggy1.v	14,20,23,25	The bug is due to using a blocking assignment (=) instead of a non-blocking assignment (<=)
lshift_conditional_error	lshift_reg_wadden_buggy2.v	13	The bug is caused by incorrectly using if (rstn) instead of if (!rstn), treating the active-low reset signal as active-high and leading to improper reset behavior.
lshift_sensitivity_issue	lshift_reg_kgoliya_buggy1.v	12	Incorrect sensitivity list; change negedge to posedge for proper functionality.
mux_numeric_issues	mux_4_1_wadden_buggy1.v	13,14,15	The bug is that it sets all the bits to 00 instead of correctly changing the output based on the input.
mux_hex_error	mux_4_1_wadden_buggy2.v	12,13,14,15	The bug is that it uses hexadecimal (h) values instead of binary (b) values, leading to incorrect bit representation.
mux_bitwidth_mismatch	mux_4_1_kgoliya_buggy1.v	6	The bug is that the output reg out should be 4 bits, but the 4-bit width was removed, causing incorrect output size.
i2c_sensitivity_issue	i2c_slave_model_wadden_buggy1.v	157	Incorrect sensitivity list; change posedge to negedge for proper functionality.
i2c_address_assignment_error	i2c_slave_model_wadden_buggy2.v	133	Incorrect address assignment; change my_adr assignment from sr to sr[7:1] for proper address comparison.
i2c_acknowledge_missing	i2c_master_bit_ctrl_kgoliya_buggy1.v	403	The bug is that the line " cmd_ack <= #1 1'b0;" // default no command acknowledge + assert cmd_ack only 1clk cycle is commented out, causing the logic for cmd_ack to be missing.
sha3_loop_boundary_issue	keccak_wadden_buggy1.v	75	The bug is an off-by-one error in the loop, where for(w=0; w<8; w=w+1) incorrectly iterates only 8 times. The fix is to change the loop to for(w=0; w<9; w=w+1), ensuring it runs the correct number of iterations.
sha3_wire_assignment_error	keccak_wadden_buggy2.v	line 33 and Lines from 84 to 88	The bug incorrectly assigns out_ready using combinational logic, and the fix requires changing the assignment back to sequential logic with always @(posedge clk), while restoring the conditionals for reset and i[22] to ensure proper updates of the out_ready signal based on clock events.
sha3_buffer_overflow_skipped	padder_ssscrazy_buggy1.v	43	The bug does not check for buffer overflow during the assignment of update. The fix requires adding the check for buffer_full to the assignment, changing it to assign update = (accept — (state & (∼buffer_full))) & (∼done); to prevent overflow.
sha3_bitwise_negation_issue	round_ssscrazy_buggy1.v	126	The bug incorrectly uses logical negation instead of bitwise negation in the assignment of f[x][y]. The fix requires changing the logical negation ! back to bitwise negation ∼, updating the code to assign f[x][y] = e[x][y] ^((∼e[add_1(x)][y]) & e[add_2(x)][y]); to ensure proper bitwise operations.

Table A1: (Continued) Additional Bug Details

Simplified Buggy File Name	**Buggy files**	**Buggy Lines**	**Detailed bug description**
tatepairing_bitshift_logic_error	tate_pairing_wadden_buggy1.v	line 75 in correct code or 85 in buggy code	The bug incorrectly uses the left shift operator (<<) instead of the right shift operator (>>), leading to unintended multiplication of the value instead of the intended division by 2
tatepairing_operator_mismatch	tate_pairing_kgoliya_buggy1.v	line 75 in correct code or 85 in buggy code	The bug results from the incorrect use of the greater-than operator (>) instead of the right shift operator (>>) for bit shifting.
tatepairing_module_instantiation	tate_pairing_wadden_buggy2.v	line 56 in correct code or 66 in buggy code	The bug results from the incorrect instantiation of the ins6 module, swapping the positions of nmu and mu, which leads to improper signal connections and incorrect functionality.
reedsolomon_register_size_error	BM_lamda_ssscrazy_buggy1.v	315	The buggy code uses an insufficient register size for decimal values, with const_timing <= 8'd500, whereas the correct code should be const_timing <= 500
reedsolomon_reset_sensitivity	out_stage_ssscrazy_buggy1.v	74	The bug is that the posedge reset was removed, causing the reset functionality to be lost.
sdram_sync_reset_issue	sdram_controller_wadden_buggy1.v	181,182	The bug is that removing the lines wr_data_r <= 16'b0; rd_data_r <= 16'b0; and adding rd_data_r <= data; causes incorrect initialization and handling of rd_data_r.
sdram_numeric_definition_error	sdram_controller_wadden_buggy2.v	82	The bug is that 1 bit was incorrectly changed from 1 to 0 in the definition, causing incorrect behavior.
sdram_case_statement_issue	sdram_controller_kgoliya_buggy2.v	406,407,408,409	The bug is caused by omitting the default case in the case statement, which can lead to undefined behavior; the fix requires restoring the default case to ensure that next is assigned to IDLE.