LLM-driven Mutation Testing in Microarchitecture Verification

by

Bekzat Skakov

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

NAZARBAYEV UNIVERSITY

Jun 2025

Author .....
Department of Computer Science
May 2, 2025

Certified by .....
Nursultan Kabylkas
Assistant Professor
Thesis Supervisor

Certified by .....
Hashim Ali
Assistant Professor
Thesis Supervisor

Accepted by .....
Elizabeth Arkhangelsky
Dean, School of Engineering and Digital Sciences

LLM-driven Mutation Testing in Microarchitecture Verification

by

Bekzat Skakov

Submitted to the Department of Computer Science
on May 2, 2025, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

The majority of electronic devices people use daily operate with microarchitectures at their core. As manufacturers release new iterations of these devices, microarchitecture scale and complexity continuously increase. This results in the rise of verification complexity. The costs of poor verification are extremely high. Lost funds, chip respins, and time-to-market delays are only a few of them. Current verification methodologies heavily rely on the skills and experience of hardware engineers. This dependence further complicates the verification process.

Large Language Models (LLMs) have already become an integral part of software engineering. They are being used for code and test generation, documentation, debugging, etc. Now, researchers actively explore how LLMs can be useful for the hardware domain as well. To contribute to this goal and to address the aforementioned problems, this work investigates how LLMs can advance hardware verification and introduces a mutation testing framework that uses an LLM as the mutation generator.

The framework injects context-aware faults into microarchitecture designs. Then, verification infrastructures are evaluated based on their ability to detect these faults. Such an approach allows engineers to find weaknesses in test suites. The framework was assessed on the extensively verified, open-source RISC-V core CVA6. Hopefully, it brings a fresh perspective to the current verification methodologies.

Thesis Supervisor: Nursultan Kabylkas
Title: Assistant Professor

Thesis Supervisor: Hashim Ali
Title: Assistant Professor

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Microarchitectures run many of the electronic devices people rely on daily. These include smartphones, tablets, laptops, smartwatches, and many more. When companies produce new versions of these devices, they typically make adjustments or improve chips, e.g., increase processing power, enhance efficiency, etc. Consequently, microarchitecture scale and complexity iteratively increase. In response to this pattern, hardware engineers have to dedicate more resources and time to verifying modern microarchitectures. Companies allocate the largest portion of funds and human effort for functional verification of current hardware systems [32]. Verification tasks occupy approximately half of the entire hardware design life cycle today [8]. This intense focus on verification is due to the high costs of undetected faults. These costs grow exponentially as microarchitecture development progresses. For instance, Intel failed to catch a critical clock degradation flaw in its Atom C2000 processor. As a result, the gaffe inflicted substantial financial and reputational damage on the company [4]. Therefore, it is important to find any flaws in a hardware design as early as possible.

Functional verification is used to confirm if a microarchitecture meets its functional requirements. Among different functional verification techniques, simulation-based verification is the most common approach in the industry [32]. It implies running various tests on a design under test (DUT) and comparing simulation results

to expected references. During this process, engineers collect coverage information to ensure all critical functions of the DUT are committed. Meanwhile, building a high-quality coverage model and achieving high coverage scores rely a lot on the skills and experience of hardware engineers [13, 31]. Besides, these scores do not directly indicate how verified a microarchitecture is [8]. They also do not provide any information about the quality of test suites.

1.2 Problem Statement

Mutation testing was initially introduced in the software domain to assess test suites [2]. Later on, hardware engineers adapted it to evaluate verification infrastructures of microarchitectures. This strategy consists of two major steps: 1) injecting mutations (i.e., faults or bugs) into DUTs and 2) running their test suites. The quality of a test suite depends on its ability to detect mutations. Currently, there are several mutation testing tools available in the industry. However, they all have certain limitations. First, these tools apply only a predefined set of mutations. Second, the mutations themselves are relatively simple. Third, they have high user requirements.

1.3 Proposed Approach

LLMs have already proven their efficiency in various software engineering tasks such as generating code, detecting bugs, and writing documentation [6, 19]. Their ability to understand context and produce meaningful outputs has also attracted professionals from the hardware domain. As a result, researchers have begun exploring LLM applications for hardware development. Recent studies demonstrate promising results in a wide range of tasks, including automated RTL code generation [1, 11], debugging [9, 26, 34], bug localization [33], and others.

Building on these promising findings and previously discussed deficiencies in the current verification methodologies, this work examines how LLMs can elevate hardware verification and proposes an LLM-driven mutation testing framework.

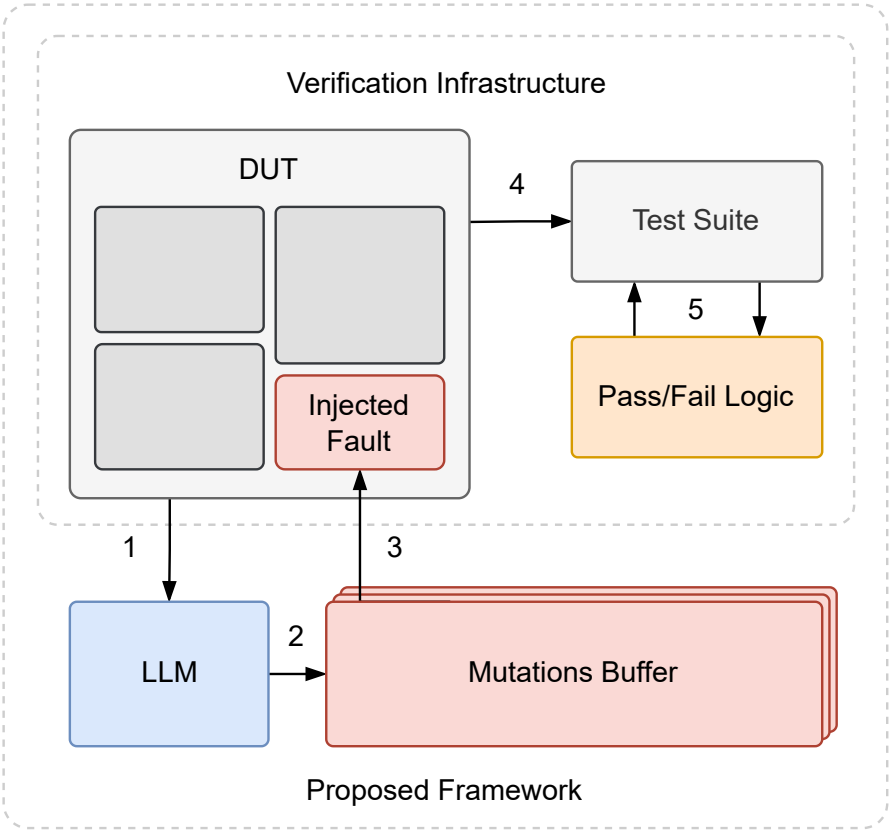

Figure 1-1: High-level Overview of the Framework.

Figure 1-1 shows how the proposed framework interacts with a standard verification infrastructure. It uses an LLM to learn the context of the DUT, e.g., modules present, their internal structures, and how they interact with each other (1). Based on this information, the LLM generates a required number of context-aware mutations and stores them in the mutations buffer (2). The framework then injects these mutations sequentially, as injected faults, into the DUT (3) and executes the test suite (4). Determined by the pass/fail logic, the results reveal the test suite's ability to detect these non-trivial, context-aware changes (5).

1.4 Research Objectives and Contributions

The main objectives of this work are structured to explore, implement, and validate the use of LLMs in hardware verification. They consist of:

1. To evaluate a popular commercial model on the mutation-generating task.
2. To implement a ready-to-use mutation testing framework.
3. To measure the effectiveness of the proposed approach by conducting comprehensive experiments on the open-source RISC-V core.

1.5 Thesis Organization

The rest of this work is organized as follows: Chapter 2 reviews the literature discussing LLMs and mutation testing in hardware verification tasks. It also highlights the novelty of the proposed approach. Chapter 3 describes the methodology and sheds light on the framework's architecture. Chapter 4 discusses the results. Finally, Chapter 5 concludes this work and suggests further refinements.

Chapter 2

Literature Review

In microarchitecture verification, mutation testing is a well-established strategy. In contrast, LLMs remain a relatively new tool, and researchers are actively exploring their applications. The following subsections summarize recent studies on verification methodologies involving mutation testing and LLMs.

2.1 LLMs in Microarchitecture Verification

2.1.1 Test Generation and Coverage Improvement

The work in [31] is one of the first to investigate the use of LLMs in automated test generation. For this task, the authors specifically focus on RISC-V and neuromorphic processors. They generate tests and process results through dialogue with GPT-3.5. Although the described approach requires a lot of human engagement, it paves the way for further research in this direction.

Next, Ma et al. [17] introduce an LLM-powered framework for test generation. Its main purpose is to increase coverage scores. This is achieved via special explainer modules, Coverage Explainer and DUT Explainer, that provide better context for LLMs. While this approach outperforms traditional random testing for simple and medium-level microarchitectures, it struggles with large, industry-level designs.

AutoBench [22] is another solution for test generation based on DUT descriptions.

It uses a hybrid approach and writes not only tests but also Python code to check their results. This approach allows AutoBench to produce complete test suites without human intervention. The drawback is poor performance on sequential circuits.

Eventually, Huang et al. [11] present VeriAssist, an LLM-based assistant for programming in Verilog and writing tests. It implements a multi-turn process similar to human workflow. This implies rounds of code generation, self-verification using generated tests, and self-correction based on test results. Such an approach significantly improves the assistant's output. The downside is that it cannot produce RTL code optimized for power, area, and runtime performance.

While previously discussed works focus on test generation, LLM4DV [36] proposes an LLM-powered framework to generate and send input signals to DUTs. It is mainly based on prompting and demonstrates poor performance on complex designs.

2.1.2 Bug Localization and Fixing

Another verification area in which LLMs are widely adopted is bug localization and repair. Yao et al. [33] present Location-is-Key (LiK), a pre-trained open-source LLM to locate bugs in microarchitectures. To operate, it requires only design specifications and faulty code. LiK shows high accuracy and outperforms commercial models in this specific task. This is achieved by continual pre-training, supervised fine-tuning, and reinforcement learning of the base LLM. However, it is uncertain whether the tool can perform similarly on complex designs.

Fu et al. [9] introduce a framework for fixing bugs in hardware using domain-specific LLMs. The authors fine-tune multiple models specifically for debugging using a custom dataset of faults and corresponding fixes. They also evaluate the framework on several open-source microarchitectures, and it shows inconsistent results. This deficiency comes from the small size of the dataset and mistakes during training.

Another debugging tool called RTLFixer is proposed in [26]. It uses Retrieval-Augmented Generation (RAG) over human guidance and ReAct prompting. This allows an LLM to act as an agent that iteratively performs debugging through reasoning and action. Experimental results show that RTLFixer can successfully resolve

simple syntax errors but struggles to handle complex cases. This work is advanced further in [34]. Its authors address the shortcomings of previous solutions by adding a fine-tuning step, similar to the one described in [9], to their framework. This allows HDLDebugger to outperform RTLFixer and other baseline approaches.

2.1.3 Formal Verification and Assertion Generation

LLMs are applied to formal verification in [10]. More precisely, they are used to determine formal properties and generate invariants for DUTs. These are the properties that remain true over the system execution and are required for proper formal verification. Similarly, in [20], authors discuss how LLMs can identify invariants in hardware designs to build correctness proofs. These proofs are used to mathematically verify the intended behavior of hardware systems.

Several works focus on the use of LLMs for automated assertion generation. The work in [18] is among the first ones in this direction. Its authors present an iterative framework with a set of rules. These rules guide LLMs toward generating complete and accurate assertions. Integrated into an existing verification tool (AutoSVA), the framework demonstrates good performance on the open-source RISC-V core called CVA6.

ChipNeMo [16] illustrates how using a domain-adapted model improves assertion generation. The authors take open-source Llama 2 as a base model and apply domain-adaptive pre-training and supervised fine-tuning. This approach allows achieving performance close to GPT-4 in assertion generation.

AssertLLM [7] demonstrates how LLMs can generate Verilog assertions based on waveform diagrams and natural language. The framework extracts structured data from specifications, maps signal definitions to RTL code, and produces accurate assertions. However, to perform well, it requires highly detailed natural language specifications. In [14], authors propose a similar approach with a focus on security assertions. They provide source code, examples, and comments and prompt LLMs to write new assertions. The framework has limited accuracy, as approximately half of its output is incorrect.

As can be noticed from the literature reviewed so far, integrating LLMs into hardware verification introduces multiple advantages. First, it allows engineers to automate processes like test generation, bug localization, and writing assertions. Second, it reduces verification complexity by incorporating natural language into the process. Third, it allows building autonomous pipelines that significantly reduce human intervention in the verification cycle.

2.2 Mutation Testing in Microarchitecture Verification

2.2.1 Traditional Mutation Testing

Early adaptations of this methodology are described in [27]. Inspired by software testing, the basic idea is to assess verification infrastructures, i.e., test benches, for digital circuits by 1) injecting bugs into them and then 2) checking if the tests can discover these bugs. By doing so, hardware engineers can assess the quality of their tests and identify areas requiring improvements.

Throughout the years, the methodology mostly remains unchanged. Available solutions rely on historical bug patterns and introduce pretty simple mutations. On top of that, they require a particular level of expertise to properly use them.

2.2.2 Novel Approaches Based on Mutation Testing

In their recent work, Wu et al. [29] introduce a mutation testing tool for microarchitectures called Mantra. To perform mutations, Mantra uses a set of real bugs extracted from publicly available hardware designs. Although it surpasses other traditional mutation testing solutions, it still lacks dynamic and context-aware fault injection. Furthermore, the tool does not allow natural language usage to create mutations.

In [30], authors apply mutation testing in a non-trivial way. They reveal Kummel, a novel approach to bug localization in microarchitectures. It uses mutation testing to

Table 2.1: Comparative Summary of Related Literature.

Literature	**Use Mutation Testing**	**Use LLMs**	**Main Focus**	**Verification Strategy**
[33]	✗	✓	Bug localization	-
[11, 17, 22, 31]	✗	✓	Test generation	Simulation-based
[36]	✗	✓	Test stimuli generation	Simulation-based
[9, 26, 34]	✗	✓	Debugging	-
[10]	✓	✓	Invariant generation and evaluation	Formal
[20]	✗	✓	Invariant identification	Formal
[7, 14, 16, 18]	✗	✓	Writing assertions	Assertion-based
[30]	✓	✗	Bug localization	Simulation-based
[29]	✓	✗	Mutation testing	Simulation-based
This Work	✓	✓	LLM-driven mutation testing tool	Simulation-based

identify and analyze design areas prone to errors. This solution shows high accuracy and outperforms other bug localization methodologies. Kummel does not directly relate to this work. However, it presents a recent, unusual way to use mutation testing.

2.3 Bridging LLMs and Mutation Testing

Table 2.1 summarizes the discussed literature. Notably, the majority of the works focus on different verification tasks and utilize either LLMs or mutation testing. The only work covering both is described in [10]. However, its authors concentrate on

LLM-assisted formal verification and use mutation testing only to validate their approach. On the contrary, this work focuses on the direct application of LLMs in mutation testing. The purpose is to explore the potential of using an LLM as the mutation-generating engine and investigate whether this approach can serve as a viable alternative or complement to traditional methodologies. The rationale for this arises from the demonstrated strength of LLMs in context understanding and reasoning with textual data [12, 15, 21, 25, 28]. Since microarchitectures are specified textually using Hardware Description Languages (HDLs), LLMs should be able to interpret HDL code, including its structure, logic, and signal relationships [37]. This ability should allow the LLM to generate context-aware mutations, potentially yielding more relevant mutants than context-agnostic rule-based approaches. Chapter 4 experimentally evaluates how effectively LLMs can generate such context-aware mutations for microarchitecture verification.

Chapter 3

Methodology

This chapter describes the methodology used to implement the framework. It begins by outlining the framework's architecture and its operational flow. Next, it dives into specific implementation details, such as preparing the DUT, generating mutations with the LLM, and automating their injection and testing. Finally, the chapter presents the quantitative metrics used to evaluate both the LLM's performance in generating valid mutations and the effectiveness of the DUT's test suite in detecting them.

3.1 Framework Architecture

The proposed framework is structured as a three-stage pipeline for systematically generating, applying, and evaluating microarchitecture mutations. This modular methodology guarantees a clear separation of concerns and facilitates a controlled workflow. Each stage is encapsulated in a separate module with specific responsibilities: the Preparation module, the Mutation module, and the Run module.

Figure 3-1 illustrates this architecture and the primary data flow between the modules. The process begins with the Preparation module, which gathers information about the DUT and user-defined parameters (1). It processes this information and outputs a configuration file containing experiment settings (2). This configuration file is then passed to the Mutation module (3). Guided by the settings and internal

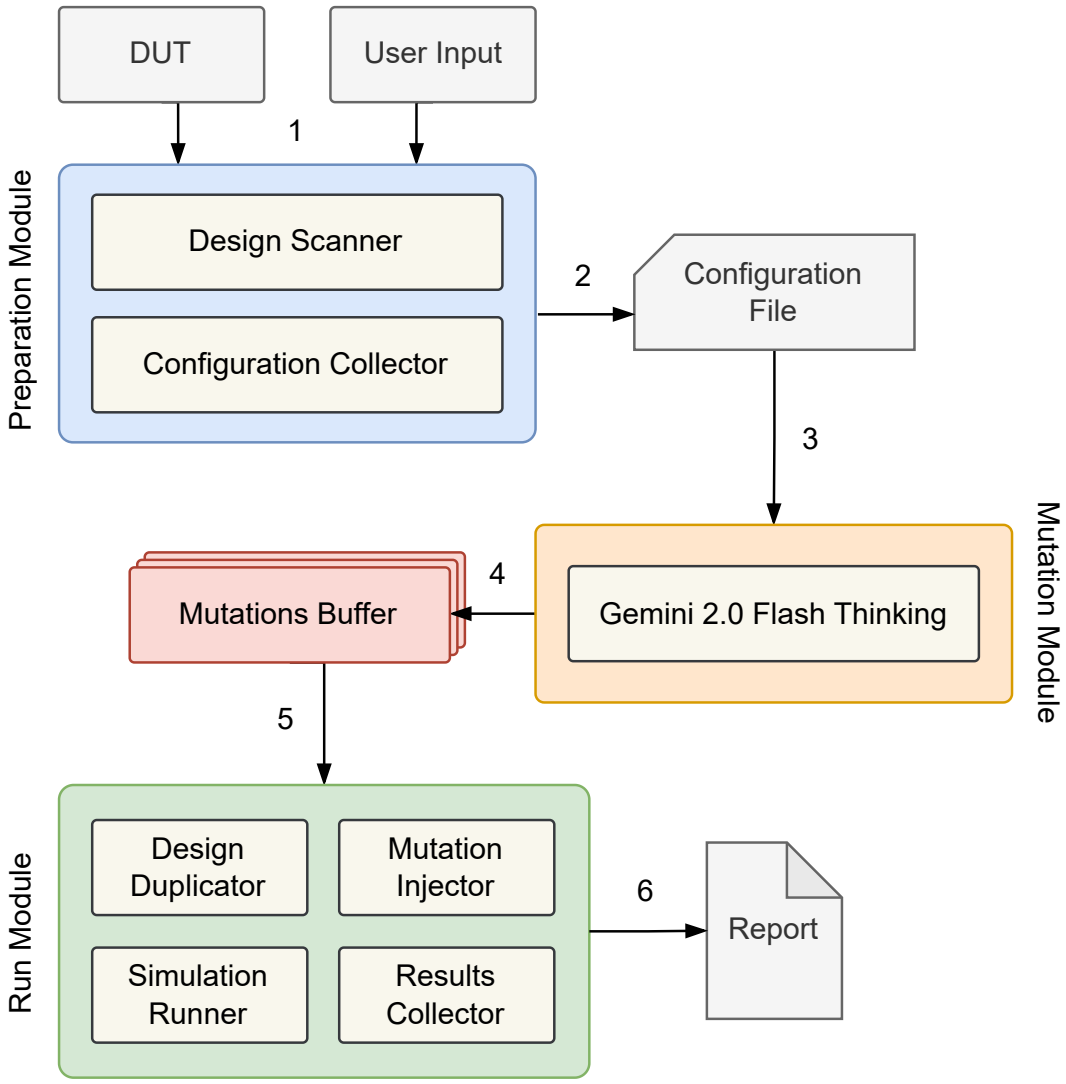

Figure 3-1: Architecture of the Proposed Approach.

prompting strategies, this module generates the required number of mutations and stores them in the mutations buffer (4). Next, the Run module takes the mutations from the buffer and runs the corresponding simulations (5). The outcomes of these runs are then aggregated and analyzed to create the final report (6).

This sequential pipeline guarantees that mutations are produced according to the initial configuration, stored, and then systematically applied and tested. This process provides a clear and reproducible methodology for evaluating the robustness of microarchitecture verification environments. The specifics of each module's imple-

mentation and components are detailed further in the following section.

3.2 Implementation Details

3.2.1 Target System and Preparation

The target system selected to evaluate the framework is CVA6 [35]. It is a well-known open-source RISC-V processor implemented in SystemVerilog. CVA6 is structured as a 6-stage, single-issue, in-order microarchitecture capable of running Linux. It uses standard open-source simulation tools: Verilator [24] for RTL simulation and Spike [23] as a reference model. CVA6 was initially developed at ETH Zurich (as Ariane) and is now maintained by the OpenHW Group.

This work uses the base 64-bit configuration of CVA6 (`cv64a6_imafdc_sv39`). This configuration has an extensive verification infrastructure consisting of 172 tests. These include standard RISC-V assembly tests, the architectural compliance suite, and custom C tests. This information was retrieved from the project's GitHub repository, specifically its Continuous Integration (CI) workflow. This workflow initiates every time someone tries to push updates to the repository, ensuring the changes undergo testing. The framework replicates it locally without any modifications. The goal is to determine whether this infrastructure is capable of catching the context-aware mutations generated by the LLM.

The initial stage of the framework involves the Preparation module, which collects data about the DUT and prepares the simulation environment. Using an interactive command-line interface, users enter various parameters needed for the DUT setup, such as the path to its source code, the desired number of mutations to generate, the exact commands required to run the test suite (in this case, as outlined in the CI workflow), and how many simulations to run in parallel. After that, the module automatically scans the specified DUT directory and collects a list of possible target files for mutation injection. All of this data is then saved in JSON format and output as a configuration file. If required, users can edit this file manually. Next, this file is

passed to the subsequent Mutation and Run modules, ensuring they operate on the correct target design and use the appropriate test execution commands.

3.2.2 LLM-driven Mutation Generation

The entire mutation generation process relies on an LLM integrated within the Mutation module. After a thorough review of various LLM options based on performance, context handling, and accessibility, Google's Gemini 2.0 [5] was selected as the mutation-generating engine. To be more precise, the framework is based on its `gemini-2.0-flash-thinking-exp-01-21` configuration. It is a closed-source model with several significant advantages:

1. *Reasoning capabilities:* The model operates in thinking mode, which significantly improves the quality of its responses.

2. *Large context window:* The 1M token capacity allows it to process very large inputs.

3. *Speed:* The flash version operates faster than the base model while preserving response quality.

4. *Accessibility:* The model can be freely accessed via its API within specific rate limits.

Due to these features, the selected Gemini model can efficiently capture the context of microarchitecture modules of different sizes and generate high-quality mutations. It is also worth noting that the model is ranked #6 in the coding category on the LMSYS Chatbot Arena Leaderboard [3], which indicates strong performance on code-related tasks.

The Mutation module operates based on this LLM. First, it parses the configuration file produced by the Preparation module to extract the list of target files and the specified number of mutations. Then, it calculates the number of mutations per file by distributing them evenly and constructs a detailed prompt for each. Prompts consist of:

1. *Complete target file content:* Provides the LLM with the full context of the design module to be mutated.

2. *Specific instructions:* Define how the LLM should analyze the input, generate mutations, and format the output.

3. *Examples:* Demonstrate *<user input>* - *<LLM response>* pairs to guide the model's output structure and style.

Next, the module sends these carefully constructed prompts as requests to the Gemini API (refer to Appendix A.1 for a complete prompt example). To address the rate limits, it implements a sliding window algorithm that tracks request timestamps and automatically pauses execution when necessary. A single mutation generated by the LLM can be represented as a JSON object similar to:

```
{
    "mutation_type": "FAULT_INJECTION",
    "original_code": "    if (x_illegal_i && x_valid_i) begin",
    "mutated_code": "    if (x_valid_i) begin"
}
```

In this example, the LLM introduces the `FAULT INJECTION` type of mutation by corrupting the if statement condition. In general, the prompt proposes four types of mutations to introduce: `FSM CORRUPTION` (altering state transitions), `PIPELINE HAZARD` (modifying control logic), `TIMING VIOLATION` (changing timing or breaking synchronization), and `FAULT INJECTION` (inserting faulty values or disrupting valid statements). However, these four are merely suggestions, and the LLM is not restricted to them. Instead, it is expected to analyze the provided HDL code and decide on the most suitable type of mutation itself.

Each mutation is verified for valid references to the original code. This ensures the mutation can be correctly applied later. As the module receives responses from the LLM, it constructs and fills the mutations buffer. Once all requested mutations are ready, the module saves this buffer into another JSON file and passes it to the Run module for injection and testing.

3.2.3 Mutation Injection and Test Execution

The final stage of the pipeline is handled by the Run module. It focuses on applying the generated mutations and evaluating the DUT's test suite against them. The module first takes the mutations buffer as input and creates one or more isolated copies of the DUT environment. The exact number of parallel copies is determined by the parameter specified in the configuration file. Then, it injects mutations by modifying source files with precise replacements and runs tests (in this case, CVA6's verification suite). Next, the module collects the outcomes of these runs. Possible outcomes include successfully compiling and passing tests (mutation escaped), successfully compiling but failing tests because of trace mismatch or timeout (mutation detected), or failing to compile/build (invalid mutation syntax/semantics). After each simulation, the module restores the original files and cleans artifacts. It repeats this cycle until it covers the entire mutations buffer.

Finally, after iterating through all mutations, the module aggregates the collected results and produces a report summarizing the experiment. This report contains:

1. Key mutation statistics *(MBSR, MDR, MER)* as defined in the next section.
2. Lists identifying mutations that escaped detection, caused trace mismatches, resulted in simulation timeouts, or led to failed builds.
3. The distribution of mutations across different LLM-defined categories.
4. An overall simulation summary.

The generation of this report represents the last stage of the framework's workflow depicted in Figure 3-1. It provides the necessary data for analyzing both the LLM's performance and the test suite's effectiveness.

3.3 Evaluation Metrics

To properly evaluate CVA6's test suite and the framework's effectiveness, it is important to introduce several quantitative metrics:

1. *Mutation Build Success Rate (MBSR):* Measures the LLM's ability to produce syntactically valid mutations:

$$MBSR = \frac{Total\ mutations\ -\ Failed\ builds}{Total\ mutations} \tag{3.1}$$

2. *Mutation Detection Rate (MDR):* Measures the effectiveness of test suites in catching injected mutations:

$$MDR = \frac{Detected\ mutations}{Total\ mutations -\ Failed\ builds} \tag{3.2}$$

3. *Mutation Escape Rate (MER):* Measures the weakness of verification infrastructures:

$$MER = \frac{Undetected\ mutations}{Total\ mutations -\ Failed\ builds} \tag{3.3}$$

Calculating these scores requires the experimental data aggregated by the Run Module. Specifically:

1. *Total mutations* refers to the total number of mutation generation attempts.
2. *Failed builds* denotes the number of cases in which the mutated code fails to compile or synthesize.
3. *Detected mutations* is the count of successfully built mutations with failing tests.
4. *Undetected mutations* (or escaped mutations) is the count of successfully built mutations with passing tests.

Collectively, these metrics capture essential information and provide a systematic approach for assessing the framework's performance.

Chapter 4

Results

This chapter introduces the results achieved by applying the proposed framework to CVA6 as described in Chapter 3. The framework generated a total of 120 mutations across 48 different modules of the DUT. The exact list of modules targeted in the experiment is provided in Appendix A.2. The following sections present and interpret the numerical data, qualitatively assess the results, and summarize the key findings.

4.1 Primary Evaluation Metrics

Table 4.1: Raw Experimental Results.

Category	**Count**
Total mutations attempted	120
– Failed builds	10
– Syntactically valid mutations	110
– Detected mutations	62
– Trace mismatches	4
– Simulation timeouts	58
– Undetected (escaped) mutations	48

Table 4.1 provides an overview of the main experimental results. As can be noticed, there are different outcome categories, but they all stack together. Trace mis-

matches and simulation timeouts account for all detected mutations. Detected and escaped mutations represent all syntactically valid, synthesizable mutations. They, in addition to failed builds, make up the total mutations attempted. From all these categories, four mentioned in Section 3.2.3 can be explicitly stressed. They are separately illustrated in Figure 4-1.

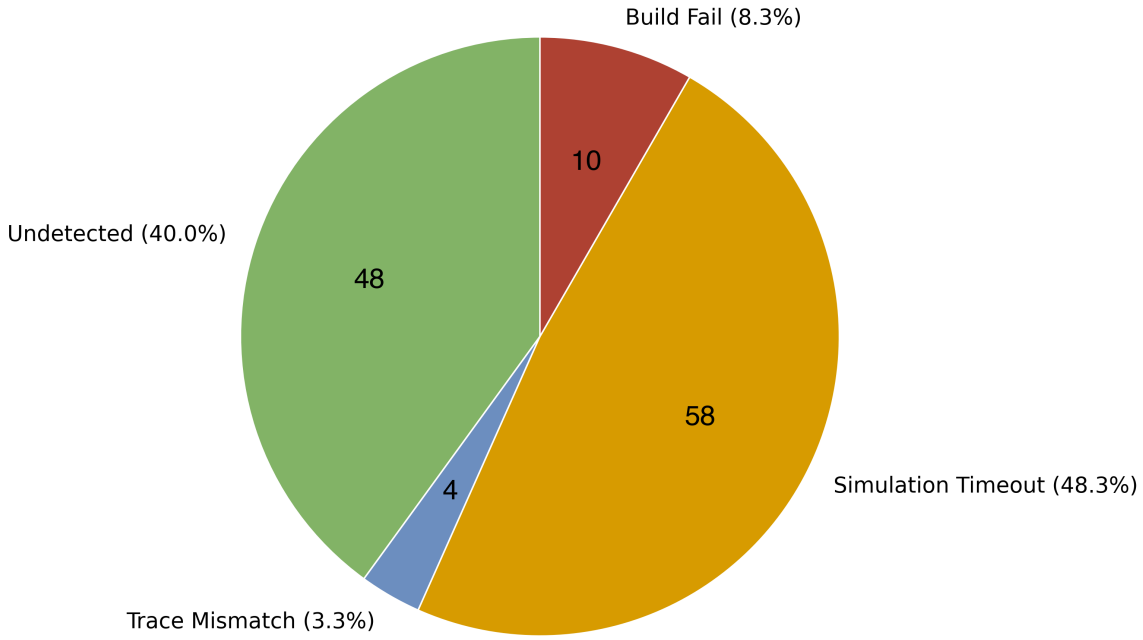

Figure 4-1: Distribution of the DUT Mutation Outcomes.

Based on these raw numbers given in Table 4.1 and Figure 4-1, the primary evaluation metrics introduced in Section 3.3 can be calculated according to their definitions.

$$MBSR = \frac{Total\ mutations\ -\ Failed\ builds}{Total\ mutations} = \frac{120 - 10}{120} \approx 0.9167 \tag{4.1}$$

$$MDR = \frac{Detected\ mutations}{Total\ mutations -\ Failed\ builds} = \frac{62}{120 - 10} \approx 0.5636 \tag{4.2}$$

$$MER = \frac{Undetected\ mutations}{Total\ mutations - \ Failed\ builds} = \frac{48}{120 - 10} \approx 0.4364 \qquad (4.3)$$

The obtained values allow drawing qualitative conclusions about the performance of the proposed framework.

4.2 LLM as the Mutation Generator

Mutation Build Success Rate (*MBSR*) is the main metric reflecting the LLM performance in the mutation-generating task. As computed in Expression 4.1, it equals 0.9167. This means Gemini 2.0 successfully output syntactically valid mutations in roughly 92% of the attempts. Table 4.2 divides all mutations based on this quality.

Table 4.2: Distribution of the DUT Mutations Based on their Validity.

Validity	**Count**
Synthesizable	110
Non-Synthesizable	10

As can be observed in Table 4.2, 110 out of 120 attempted mutations were compiled successfully. Such a high number confirms that the LLM can effectively understand and manipulate the HDL code in most cases. However, there are also 10 mutations that resulted in build failures due to syntax and semantic errors. One example of such a non-synthesizable mutation is given below:

```
{
    "mutation_type": "FAULT_INJECTION",
    "original_code": "    output logic no_st_pending_o,",
    "mutated_code": "    output logic no_st_pending_o = 1'b0;"
}
```

This example contains both types of errors: 1) it is illegal to specify a default value in an output port declaration (`= 1'b0`), and 2) the semicolon is used instead of

the correct trailing comma in the module's port list. Despite such occasional errors, the resulting *MBSR* shows great promise of LLMs in automated, context-aware fault injection.

4.3 Effectiveness of the Test Suite

Mutation Detection Rate (*MDR*) and Mutation Escape Rate (*MER*) measure how effectively CVA6's test suite catches or misses the injected mutations. According to Expressions 4.2 and 4.3, *MDR* equals 0.5636 and *MER* equals 0.4363.

This *MDR* value indicates that the test suite successfully detected approximately 56% of valid mutations. As shown in Figure 4-1, most of them (58 out of 62) were detected by simulation timeouts. In other words, there was a notable functional divergence during the execution of the test suite in these cases. Trace mismatches, which are the differences between the traces produced by the DUT and the reference model, identified a much smaller number of mutations (4 out of 62).

Meanwhile, an *MER* of 0.4363 suggests that a substantial portion ($\approx$ 44%) of valid mutations escaped detection entirely and all tests in the test suite successfully passed. Such an outcome indicates severe gaps in the verification infrastructure. LLM-generated mutations were prompted to be semantically relevant to the module context. Perhaps this contributed to their ability to avoid detection as compared to simpler, rule-based mutations.

Combined, these results cast doubt on the quality of the test suite but at the same time confirm the effectiveness of the proposed framework.

4.4 Analysis of Mutation Types

Table 4.3 breaks down the distribution of all 120 mutations based on their type and outcome. As can be observed, the vast majority of mutations are either `FAULT INJECTION`, `PIPELINE HAZARD`, or `FSM CORRUPTION`. It is also fair to note that approximately 40% of the mutations in each of these categories went undetected. They

Table 4.3: Distribution of the DUT Mutations Based on their Type and Outcome.

Mutation Type	Escaped	Traces Mismatch	Simulation Timeout	Build Fail	Total
FAULT INJECTION	18	3	24	3	48
FSM CORRUPTION	13	0	15	0	28
PIPELINE HAZARD	16	0	18	6	40
CONTROL SIGNAL CORRUPTION	0	1	1	0	2
DATAPATH CORRUPTION	1	0	0	0	1
TIMING VIOLATION	0	0	0	1	1

collectively account for almost all escaped mutations denoted in Table 4.1 and Figure 4-1. At the same time, `CONTROL SIGNAL CORRUPTION`, `DATAPATH CORRUPTION`, and `TIMING VIOLATION` types are represented by only 1-2 mutations each, with one mutation escaping, two being caught, and one causing a build failure. Such an uneven distribution of mutations does not allow drawing any meaningful conclusions about these specific categories. Altogether, these findings point to a remarkable pattern in the behavior of the LLM. Although prompted with examples of mutation categories but not limited to them, the LLM strongly leans toward those example categories when generating mutations.

4.5 Key Findings

The following list summarizes key findings of the experimental results:

1. Gemini 2.0 successfully produced syntactically valid mutations in the vast majority of attempts.

2. CVA6's verification infrastructure failed to detect approximately 44% of valid mutations, which indicates gaps in the test suite and the effectiveness of the framework.

3. The LLM strongly leaned toward using the example mutation categories provided in the prompt, even though it was not limited to them.

Chapter 5

Conclusion

This thesis work aimed to investigate the potential of LLMs in mutation testing and present a corresponding LLM-driven mutation testing framework. It also intended to examine whether such a framework can function as a viable alternative or complement to traditional verification methodologies. The entire work was organized into five chapters.

Chapter 1 introduced the current state of functional verification in the hardware industry and explained the motivation behind companies spending enormous resources on it. It also presented thesis objectives and contributions and outlined the organization of this work.

In Chapter 2, recent studies on verification methodologies integrating mutation testing and/or LLMs were reviewed. According to the literature, LLMs allow engineers to automate and reduce the complexity of different quality assurance tasks. Mutation testing, in its turn, is a well-established strategy that has not changed much over time. Combining these two methodologies brings novelty to the proposed framework.

Chapter 3 focused on the methodology used to build the framework. It presented the framework's three-stage structure and operational flow in detail. Furthermore, it introduced the quantitative metrics used to draw conclusions about the framework's effectiveness.

Chapter 4 discussed the results of applying the proposed framework to the CVA6

RISC-V core. In total, the framework produced 120 mutations across 48 modules of the core. Out of these 120, 110 were syntactically valid, synthesizable mutations; 62 were detected by and 48 escaped the core's verification infrastructure. Corresponding conclusions about the LLM's performance as the mutation generator and the quality of CVA6's test suite were reached.

This thesis work has achieved all the goals it set. LLMs definitely have a future in verification tasks and specifically in mutation testing. The results described in Chapter 4 prove this point. LLM-driven mutation testing has the potential to improve verification methodologies, maybe not as an alternative to traditional mutation testing but definitely as a complement. As it was stated in Chapter 1, functional verification is critical, and adding an extra layer of assurance in the form of LLM-assisted mutation testing must pay off. The following sections comment on the limitations of the current work and provide suggestions for future work.

5.1 Limitations

There are several limitations to the work conducted as part of this thesis. The first one is computational resources. Running the entire test suite of CVA6 to evaluate a single mutation takes, on average, 3–4 hours on a machine with 24 cores and 32 GB of RAM. It is possible to reduce this time to approximately 40 minutes by parallelizing jobs within a single iteration on 16 cores. However, this significantly limits the number of mutations that can be evaluated in parallel.

Another limitation is the choice of LLM. Although Gemini 2.0 Flash Thinking is among the best widely available models, it is not the most advanced one. Using a newer model (e.g., Gemini 2.5 Pro or o4-mini) could yield better results, but running lots of experiments in such a setup can get very expensive quite fast.

The final limitation is the availability of simulation tools. While Verilator and Spike are great open-source options, there are some designs (e.g., Black Parrot or Ibex) that require proprietary, industry-level tools to run their test suites. Such tools typically cannot be easily accessed. As a result, the choice of DUTs is also limited.

5.2 Future Work

Future work should focus on:

1. Improving the prompt used to interact with LLMs.
2. Providing the LLM with context about inter-module communication via Value Change Dump (VCD) waveforms.
3. Comparing the proposed framework against traditional mutation-testing tools to obtain quantitative baseline results.

Appendix A

Framework Operation Details

A.1 LLM Prompt

This code snippet represents a complete prompt used to request mutations for different microarchitecture modules. The first placeholder within the `<source>` tags is dedicated to specifying the module name. The second placeholder within the `<document_content>` tags is used to contain the module content. The third placeholder within the `GENERATE {} MUTATIONS` part defines the number of mutations to generate for the given module. The rest of the prompt is independent of these variables.

```
<document>
    <source>{}</source>
    <document_content>{}</document_content>
</document>

YOU ARE AN ELITE VERILOG MUTATION ENGINE DESIGNED TO TEST THE
ROBUSTNESS OF HARDWARE VERIFICATION ENVIRONMENTS BY GENERATING
COMPLEX, SEMANTICALLY RELEVANT MUTATIONS TO VERILOG SOURCE
CODE.

<instructions>
```

- ANALYZE THE INPUT VERILOG CODE TO IDENTIFY
LOGICALLY SIGNIFICANT REGIONS SUCH AS:
  - FINITE STATE MACHINES (FSMs)
  - ALUs AND DATAPATH COMPONENTS
  - PIPELINED STAGES
  - MEMORY CONTROLLERS
  - CLOCK AND RESET DOMAINS
  - CONTROL SIGNALS, STALLS, HAZARDS, OR HANDSHAKES

- GENERATE {} MUTATIONS. EACH MUST:
  - INTRODUCE A SINGLE, HIGH-IMPACT, STRUCTURAL DESIGN BUG
  - BELONG TO ONE OF THE FOLLOWING CATEGORIES OR SIMILAR:
    - **FSM_CORRUPTION_MULTILINE**: Alter state transitions to create dead/unreachable/incorrect states
    - **PIPELINE_HAZARD**: Remove control logic that handles valid/stall/ready signals
    - **TIMING_VIOLATION**: Modify edge sensitivity or remove/reset synchronization
    - **FAULT_INJECTION**: Insert stuck-at faults, invert resets, corrupt outputs

- FOR EACH MUTATION, OUTPUT A **SEPARATE JSON OBJECT** IN AN ARRAY CONTAINING:
  - `"mutation_type"`: A descriptive tag (e.g., `"FSM_CORRUPTION_MULTILINE"`)
  - `"original_code"`: Full, unmodified snippet to match (including indentation and line breaks)
  - `"mutated_code"`: Snippet with exactly one mutation applied (must preserve structure)

- EACH MUTATION MUST BE **INDEPENDENT** (do not reuse or alter overlapping lines or signals) AND **REALISTIC AND SEMANTICALLY

RELEVANT**

- THE FORMAT MUST ENABLE STRAIGHTFORWARD STRING MATCHING AND REPLACEMENT IN THE SOURCE FILE - NO LINE NUMBERS OR COMMENTS NEEDED.

- THE ‘original_code‘ MUST BE EXACTLY COPY-PASTABLE FROM THE INPUT CODE.

- THE ‘mutated_code‘ MUST PRESERVE INDENTATION FOR CLEAN PATCHING.

- ENSURE THAT ‘original_code‘ AND ‘mutated_code‘ STRINGS ARE ESCAPED PROPERLY (e.g., NEWLINES AS ‘\\n‘, QUOTES AS ‘\\\"‘) TO PRODUCE VALID JSON PARSEABLE BY PYTHON’S ‘json.loads()‘.
</instructions>

<what not to do>
- NEVER INCLUDE MORE THAN ONE MUTATION PER JSON OBJECT.
- NEVER OVERLAP LINE RANGES OR MUTATE THE SAME LOGIC TWICE.
- NEVER MODIFY SIGNAL NAMES, IDENTIFIERS, OR INTRODUCE SYNTAX ERRORS.
- NEVER OUTPUT EXPLANATIONS, HEADERS, OR ANYTHING OUTSIDE THE JSON ARRAY.
- AVOID TRIVIAL SINGLE-LINE CHANGES UNLESS STRUCTURAL MUTATIONS ARE NOT POSSIBLE.
- DO NOT ABSTRACT, SUMMARIZE, OR OMIT LINES FROM ‘original_code‘ OR ‘mutated_code‘.
- NEVER ALTER SIGNAL NAMES OR INTRODUCE SYNTAX ERRORS.
</what not to do>

<High Quality Few-Shot Example>

<USER INPUT>

```verilog
always_ff @(posedge clk) begin
  if (rst) begin
    valid <= 0;
    state <= IDLE;
  end else begin
    case (state)
      IDLE: if (start) begin
        data_reg <= in_data;
        state <= LOAD;
      end
      LOAD: begin
        if (data_ready)
          state <= EXEC;
        else
          state <= LOAD;
      end
      EXEC: begin
        result <= data_reg * 4;
        valid <= 1;
        state <= IDLE;
      end
    endcase
  end
end
```

</USER INPUT>

<ASSISTANT RESPONSE>

```json
[
```

```
  {{
    "mutation_type": "FSM_CORRUPTION_MULTILINE",
    "original_code": "      LOAD: begin\\n        if
    (data_ready)\\n          state <= EXEC;\\n        else\\n
    state <= LOAD;\\n      end",
    "mutated_code": "      LOAD: begin\\n        if
    (data_ready)\\n          state <= LOAD;\\n        else\\n
    state <= IDLE;\\n      end"
  }}
]
‘‘‘
</ASSISTANT RESPONSE>
</High Quality Few-Shot Example>
```

A.2 Target Modules

This list contains 48 different CVA6 modules used in the experiment. To ensure these modules were covered during the test suite execution, they were checked against the modules included in the VCD waveforms produced by the simulations. On average, 2.5 mutations were created and injected per module.

- alu.sv
- amo_buffer.sv
- ariane_regfile_ff.sv
- axi_shim.sv
- bht.sv
- branch_unit.sv
- btb.sv
- commit_stage.sv
- compressed_decoder.sv
- controller.sv
- csr_buffer.sv
- csr_regfile.sv
- cva6.sv
- cva6_fifo_v3.sv

- cva6_icache.sv
- cva6_mmu.sv
- cva6_ptw.sv
- cva6_rvfi_probes.sv
- cvxif_compressed_if_driver.sv
- cvxif_fu.sv
- cvxif_issue_register_commit_if_driver.sv
- decoder.sv
- ex_stage.sv
- fpu_wrap.sv
- frontend.sv
- id_stage.sv
- instr_queue.sv
- instr_realign.sv
- instr_scan.sv
- issue_read_operands.sv
- issue_stage.sv
- load_store_unit.sv
- load_unit.sv
- lsu_bypass.sv
- mult.sv
- multiplier.sv
- ras.sv
- scoreboard.sv
- serdiv.sv
- std_cache_subsystem.sv
- store_buffer.sv
- store_unit.sv
- wt_axi_adapter.sv
- wt_dcache.sv
- wt_dcache_ctrl.sv
- wt_dcache_mem.sv
- wt_dcache_missunit.sv
- wt_dcache_wbuffer.sv

Bibliography

[1] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Chip-chat: Challenges and opportunities in conversational hardware design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6. IEEE, 2023.

[2] Timothy Alan Budd. *Mutation analysis of program test data*. Yale University, 1980.

[3] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint arXiv:2403.04132*, 2024.

[4] Thomas Claburn. Intel's Atom C2000 chips are bricking products – and it's not just Cisco hit. The Register, February 2017. [Online]. Available: `https://www.theregister.com/2017/02/06/cisco_intel_decline_to_link_product_warning_to_faulty_chip/`.

[5] Google DeepMind. Gemini 2.0 Flash Thinking, March 2025. Accessed: 2025-04-10.

[6] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[7] Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms. *arXiv preprint arXiv:2402.00386*, 2024.

[8] Harry Foster. The 2022 wilson research group functional verification study, 2022. Accessed: 2024-09-16.

[9] Weimin Fu, Kaichen Yang, Raj Gautam Dutta, Xiaolong Guo, and Gang Qu. Llm4sechw: Leveraging domain-specific large language model for hardware debugging. In *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6. IEEE, 2023.

[10] Muhammad Hassan, Sallar Ahmadi-Pour, Khushboo Qayyum, Chandan Kumar Jha, and Rolf Drechsler. Llm-guided formal verification coupled with mutation testing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–2. IEEE, 2024.

[11] Hanxian Huang, Zhenghan Lin, Zixuan Wang, Xin Chen, Ke Ding, and Jishen Zhao. Towards llm-powered verilog rtl assistant: Self-verification and self-correction. *arXiv preprint arXiv:2406.00115*, 2024.

[12] Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*, 2022.

[13] Kai Huang, Peng Zhu, Rongjie Yan, and Xiaolang Yan. Functional testbench qualification by mutation analysis. *VLSI Design*, 2015, 2015.

[14] Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. (security) assertions by large language models. *IEEE Transactions on Information Forensics and Security*, 2024.

[15] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[16] Mingjie Liu, Minwoo Kang, Ghaith Bany Hamad, Syed Suhaib, and Haoxing Ren. Domain-adapted llms for vlsi design and verification: A case study on formal verification. In *2024 IEEE 42nd VLSI Test Symposium (VTS)*, pages 1–4. IEEE, 2024.

[17] Ruiyang Ma, Yuxin Yang, Ziqian Liu, Jiaxi Zhang, Min Li, Junhua Huang, and Guojie Luo. Verilogreader: Llm-aided hardware test generation. *arXiv preprint arXiv:2406.04373*, 2024.

[18] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. Using llms to facilitate formal verification of rtl. *arXiv e-prints*, pages arXiv–2309, 2023.

[19] Ipek Ozkaya. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software*, 40(3):4–8, 2023.

[20] Khushboo Qayyum, Muhammad Hassan, Sallar Ahmadi-Pour, Chandan Kumar Jha, and Rolf Drechsler. Late breaking results: Llm-assisted automated incremental proof generation for hardware verification. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–2, 2024.

[21] Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. Reasoning with language model prompting: A survey. *arXiv preprint arXiv:2212.09597*, 2022.

[22] Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. Autobench: Automatic testbench generation and evaluation using llms for hdl design. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, pages 1–10, 2024.

[23] riscv-software-src. riscv-isa-sim: Spike, a risc-v isa simulator. `https://github.com/riscv-software-src/riscv-isa-sim`, 2021. [Online; accessed 17-April-2025].

[24] Wilson Snyder. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, volume 79, pages 122–148, 2004.

[25] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

[26] YunDa Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv preprint arXiv:2311.16543*, 2023.

[27] Patrice Vado, Yvon Savaria, Yannick Zoccarato, and Chantal Robach. A methodology for validating digital circuits with mutation testing. In *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 1, pages 343–346. IEEE, 2000.

[28] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[29] Jiang Wu, Yan Lei, Zhuo Zhang, Xiankai Meng, Deheng Yang, Pan Li, Jiayu He, and Xiaoguang Mao. Mantra: Mutation testing of hardware design code based on real bugs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[30] Jiang Wu, Zhuo Zhang, Deheng Yang, Jianjun Xu, Jiayu He, and Xiaoguang Mao. Knowledge-augmented mutation-based bug localization for hardware design code. *ACM Transactions on Architecture and Code Optimization*, 2024.

[31] Chao Xiao, Yifei Deng, Zhijie Yang, Renzhi Chen, Hong Wang, Jingyue Zhao, Huadong Dai, Lei Wang, Yuhua Tang, and Weixia Xu. Llm-based processor verification: A case study for neuronnorphic processor. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[32] Shuo Yang, Robert Wille, and Rolf Drechsler. Improving coverage of simulation-based verification by dedicated stimuli generation. In *2014 17th Euromicro Conference on Digital System Design*, pages 599–606. IEEE, 2014.

[33] Bingkun Yao, Ning Wang, Jie Zhou, Xi Wang, Hong Gao, Zhe Jiang, and Nan Guan. Location is key: Leveraging large language model for functional bug localization in verilog. *arXiv preprint arXiv:2409.15186*, 2024.

[34] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. Hdldebugger: Streamlining hdl debugging with large language models. *arXiv preprint arXiv:2403.11671*, 2024.

[35] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[36] Zixi Zhang, Greg Chadwick, Hugo McNally, Yiren Zhao, and Robert Mullins. Llm4dv: Using large language models for hardware test stimuli generation. *arXiv preprint arXiv:2310.04535*, 2023.

[37] Ruizhe Zhong, Xingbo Du, Shixiong Kai, Zhentao Tang, Siyuan Xu, Hui-Ling Zhen, Jianye Hao, Qiang Xu, Mingxuan Yuan, and Junchi Yan. Llm4eda: Emerging progress in large language models for electronic design automation. *arXiv preprint arXiv:2401.12224*, 2023.