Fixbench-RTL: A Comprehensive Benchmark for Evaluating LLMs on RTL Debugging

Shijie Li*, Weimin Fu†, Yifang Zhao*, Xiaolong Guo†, Yier Jin*,
*University of Science and Technology of China, {shijie_li, zhaoyifang}@mail.ustc.edu.cn, jinyier@ustc.edu.cn
†Kansas State University, {weiminf, guoxiaolong}@ksu.edu

***Abstract*—The rapid advancement of large language models (LLMs) has presented new avenues for automating complex tasks in hardware design and verification. Due to the time-consuming and labor-intensive nature of hardware code debugging, there has been a growing interest in leveraging LLMs for automating this process. However, benchmarks used in existing studies tend to be simplistic, limited in bug type, and proposed by the authors themselves. In this paper, we introduce a novel benchmark named Fixbench-RTL and its construction framework, specifically designed to assess the ability of LLMs to identify and correct bugs in HDL-based hardware designs. The benchmark covers a broad spectrum of error types, including syntactic errors, functional bugs, and security vulnerabilities. Experimental evaluations show that current popular LLMs still fall short of meeting the practical requirements for hardware debugging. The benchmark aims to provide a foundation for future research in LLM-assisted hardware debugging.**

***Index Terms*—Large Language Model, Hardware Debugging, Hardware Benchmarking**

I. Introduction

The increasing scale and complexity of modern hardware systems have significantly amplified the challenges associated with design verification and debugging. Register Transfer Level (RTL) design, commonly implemented using Hardware Description Languages (HDLs) such as Verilog and VHDL, forms the foundation of digital circuit development. However, the process of debugging RTL code remains notoriously difficult. It often requires deep domain expertise and extensive manual effort, especially as designs grow in size and incorporate intricate timing and concurrency behaviors. Unlike software programs, HDL modules execute concurrently, are highly sensitive to signal timing and ordering, and must adhere to synthesis and simulation constraints. These characteristics make error localization and correction a time-consuming, iterative task that is difficult to automate.

In parallel, the advent of large language models (LLMs) has opened up promising opportunities for automation across a wide range of code-related tasks. Pretrained LLMs have shown strong performance in code generation, completion, and even bug fixing in high-level programming languages. Their success in learning complex patterns from massive code corpora raises the possibility of extending these capabilities to hardware domains. However, directly applying LLMs to RTL debugging presents unique challenges due to the syntactic and semantic gap between HDLs and general-purpose languages.

979-8-3315-8924-0/25/$31.00 ©2025 IEEE

Recent efforts [1]–[4] have begun to explore the use of LLMs for HDL understanding, generation, and debugging. Nevertheless, progress in this area is hindered by a major limitation: the absence of a standardized and comprehensive benchmark for evaluating model performance on RTL debugging tasks. Existing benchmarks used in prior work [3], [5]–[7] often suffer from limited test case coverage, a narrow range of bug types, or the absence of simulation-based functional verification, making it difficult to reliably evaluate real-world debugging performance.

To address this gap, we present a novel benchmark specifically designed to evaluate LLMs on RTL debugging tasks. Each test case in our benchmark consists of four components: a buggy RTL code, its corresponding corrected version, a natural language description of the bug, and a simulation-based testbench for functional verification. The benchmark combines both synthetic and real-world errors, covering a wide spectrum of RTL errors, including syntax violations, functional bugs, and design vulnerabilities. Our benchmark enables more accurate and meaningful assessment of LLM-based debugging systems.

The main contributions of this paper are:

- We construct the Fixbench-RTL, an open-source benchmark for evaluating RTL code debugging capabilities of LLMs. It contains a diverse set of error types and is challenging to identify and fix correctly.
- We propose a framework for constructing benchmarks, which includes methods for bug injection and testbench generation. This approach enables easy expansion of the benchmark.
- Through experiments with state-of-the-art LLMs, we assess their current strengths and limitations, offering insights into areas where further research is needed.

II. Background and Related Work

A. LLMs for RTL Code

LLMs like Codex [8] and StarCoder [9] have revealed the tremendous potential of LLMs in software tasks. Recent work has explored the application of LLMs in various RTL code-related tasks, such as code generation [2], [10], bug detection [11], and automated repair [5], [7]. To enhance LLM performance on hardware tasks, [12], [13] have adopted prompt engineering techniques to better guide LLM behavior and improve output quality. [2], [14] have focused on building

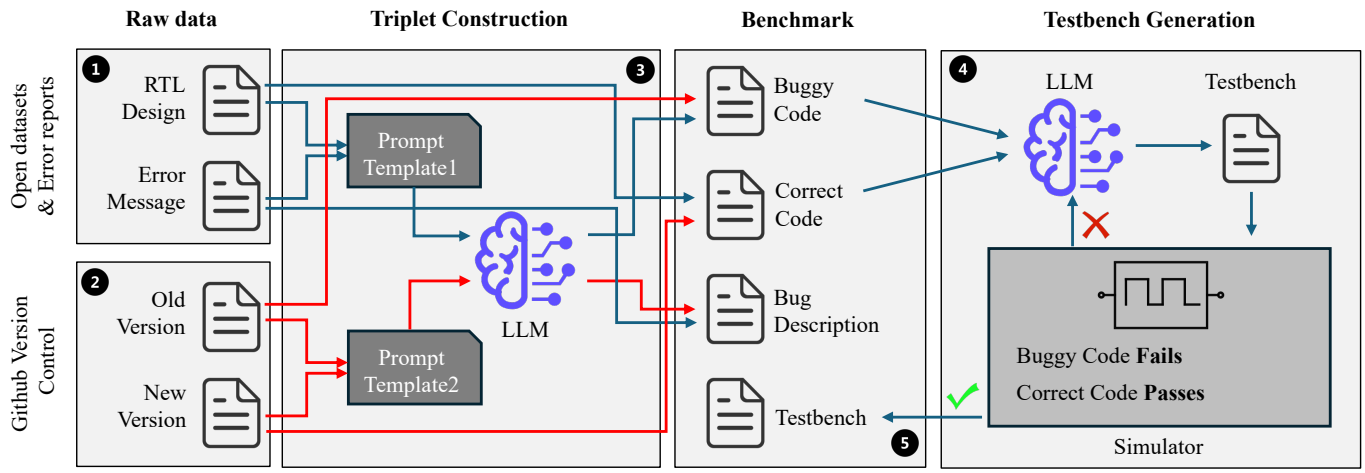

Fig. 1: Fixbench-RTL overview: the raw data of the benchmark is sourced from open-source RTL datasets and error reports (step ❶), or the GitHub version control dataset (step ❷). The framework utilizes LLM to generate **Buggy Code-Correct Code-Bug Description triples** from the raw data (step ❸). Then the testbench is automatically generated through the cooperation of LLM and the simulator (step ❹). Finally, combine the four parts to construct Fixbench-RTL Benchmark (step ❺).

domain-specific hardware datasets and training open-source models, which have been shown to further boost performance.

B. Challenges of Applying LLMs to Hardware Debugging

Using LLMs for debugging HDL code introduces several domain-specific difficulties. First, the semantics of HDL extend beyond syntax and logical correctness; they encompass concurrency, timing behavior, and hardware synthesizability. As such, a repair that appears correct syntactically may still fail under simulation or synthesis due to subtle timing or structural issues.

Second, evaluating LLM-generated repairs in the hardware domain is inherently challenging. It requires the construction of a sophisticated testbench to verify whether the repaired code can pass the simulation. However, designing such a testbench is both time-consuming and labor-intensive, often demanding substantial domain expertise. To mitigate the cost of manual testbench creation, AutoBench [15] utilizes LLM to automate the generation of a testbench for HDL design.

Finally, there is currently no widely accepted benchmark for evaluating LLMs on hardware debugging tasks. Existing studies often rely on benchmarks that are proprietary [6] or exhibit limited diversity in design types and bug patterns [7]. This makes it difficult to compare approaches or track progress in this emerging research area.

C. Related Work

Recent efforts have explored leveraging LLMs to assist in hardware debugging. SBF [16] investigates the application of prompt engineering techniques to guide LLMs in repairing hardware security vulnerabilities. Their evaluation is based on ten security bugs collected from CWE [17], OpenTitan [18], and the Hack@DAC 2021 SoC [19], offering insight into the capabilities of LLMs in this domain.

LLM4SecHW [3] constructs a dataset by mining version control histories of open-source hardware projects, capturing both design bugs and their corresponding fixes. Using this dataset, they fine-tune LLMs to better detect and correct hardware design flaws. However, their evaluation methodology primarily relies on text-based metrics such as ROUGE-1, which assess similarity rather than syntax and functional correctness.

RTLFixer [5] combines prompt engineering with retrieval-augmented generation (RAG), incorporating feedback from hardware compilers to refine LLM outputs iteratively. This approach focuses exclusively on resolving syntax errors in RTL code and lacks focus on functional bugs.

HDLdebugger [6] adopts a reverse engineering approach to build a debugging dataset, and enhances LLM capabilities through supervised fine-tuning and RAG. Despite its technical promise, the dataset used in this work is sourced from proprietary Huawei data and is not publicly accessible, limiting reproducibility and adoption.

MEIC [7] introduces a multi-step debugging framework involving two LLMs: a debug LLM that generates candidate fixes, and a score LLM that evaluates and ranks them. Debug LLM continues to generate new candidates until the highest-scoring candidate passes syntax and function checks. Although their benchmark consists of 178 samples, it is constructed from only 29 RTL designs and includes 18 error types, limiting its coverage for comprehensive evaluation.

III. Methodology

A. An Overview of the Fixbench-RTL

To systematically evaluate the capabilities of LLMs in debugging RTL code, we construct a comprehensive benchmark named **Fixbench-RTL**. Each benchmark entry is composed of four key components:

TABLE I: Examples of RTL code errors contained in Fixbench-RTL.

Types	Bug	Detailed Description	Buggy Code	Correct Code
Syntax Errors	Module Definition Errors	Port types not declared	module Tff (data, clk, rst, q);	module Tff(input wire data, clk, rst, output reg q);
	Signal Type Error	'reg' use continuous assignment ('assign'), 'wire' be assigned in 'always' blocks.	reg q; assign q = temp;	reg q; always@(*)begin q <= temp; end
	Missing Punctuation	Missing semi-colons, commas or brackets	assign a = b	assign a = b;
	Undeclared Variable	Use undeclared variable	output q; assign q = temp;	output q; reg temp; assign q = temp;
	Module Instantiation Error	Instantiating an unknown module	full_adder1 F0(.a(a[0]), ... ,.cout(c[0])); ... module full_adder (input a, ... , cout);	full_adder F0 (.a(a[0]), ... ,.cout(c[0])); ... module full_adder (input a, ... , cout);
	Function Block Error	Use timing control statements inside a function	function integer calculate; input a; begin #10; calculate = a + 1; ...	function integer calculate; input a; begin calculate = a + 1; ...
Function Errors	Incomplete Sensitivity Lists	Missing signals in 'always @()' blocks	always @(a) begin q <= a + b; end	always @(a,b) begin q <= a + b; end

	Initialization Missing	Registers are not initialized when reset	always @(posedge clk) begin
			if (reset) begin
			// counter is not initialized
				always @(posedge clk) begin
				if (reset) begin
				counter <= 0;

	Combinational Loop	The output of the combinational logic is fed back to its input directly or indirectly, forming a delay-free loop.	assign A = B & C; assign B = A | D;	assign A = B & C; assign B = E | D;
	Functions Call Error	Input mismatch errors when calling functions	assign c = data_rev(a);	assign c = data_rev(a, b);
	Incorrect Operator Usage	Misusing logical (&& , ||) vs. bitwise (&, |) operators.	assign result = a & b; // 4'b1000	assign result = a && b; // 0
	Incorrect Assignments	Use blocking assignment (=) for sequential logic	always @(posedge clk) begin a = b + c; end	always @(posedge clk) begin a <= b + c; end

Design Vulnerabilities	Deadlock	There is a state of FSM that has no valid outgoing transition. This can cause the system to halt or become stuck, unable to proceed to any other state.	case (current_state)
			IDLE: next_state = start ? S1:IDLE;
			S1: next_state = S2;
			S2: next_state = S2; // stuck here
			default: next_state = IDLE;
				case (current_state)
				IDLE: next_state = start ? S1:IDLE;
				S1: next_state = S2;
				S2: next_state = done ? IDLE:S2;
				default: next_state = IDLE;

	Dynamic Deadlock	FSM gets stuck in a set of states that continuously loop among themselves without reaching a stable or terminal state. While the FSM is transitioning between states, it essentially fails to make any progress, causing the system to be stuck in an undefined or undesired behavior.	case (current_state)
			IDLE: next_state = start ? S1:IDLE;
			S1: next_state = S2;
			S2: next_state = S1; // S1 S2 loop
			default: next_state = IDLE;
				case (current_state)
				IDLE: next_state = start ? S1:IDLE;
				S1: next_state = S2;
				S2: next_state = done ? IDLE:S1;
				default: next_state = IDLE;

	Unreachable State	There is a state that cannot be entered from the initial state, no matter what sequence of inputs is applied. This could occur due to errors in the state transition logic or improper design.	case (current_state)
			IDLE: next_state = start ? S1:IDLE;
			S1: next_state = IDLE;
			S2: next_state = S1; //S2:unreachable
			default: next_state = IDLE;
				case (current_state)
				IDLE: next_state = start ? S1:IDLE;
				S1: next_state = IDLE;
				default: next_state = IDLE;

	Unhandled transitions	The FSM is not designed to account for all possible inputs or states, resulting in undefined behavior when certain inputs are received.	case (state)
			IDLE: begin
			if (control == 2'b01)
			next_state = RUN;
			end
				case (state)
				IDLE: begin
				if (control == 2'b01)
				next_state = RUN;
				else next_state = IDLE;
				end

	Hamming Distance	If two consecutive (reachable) unprotected states have a Hamming Distance (HD) greater than 1, a single-bit fault or glitch could cause a transition to an unintended third state	parameter IDLE = 3'b000;
			parameter RUN = 3'b011;
			parameter DONE = 3'b110;
				parameter IDLE = 3'b000;
				parameter RUN = 3'b001;
				parameter DONE = 3'b011;

	Expression Always True	An "Expression Always True" typically indicates a logical flaw or unintended behavior in the code.	if (data_in || 1'b1) begin //always true
			data_out <= data_in;
			end
				if (data_in) begin
				data_out <= data_in;
				end

- **Buggy RTL Code**: the RTL code with bug.
- **Correct Code**: the corresponding fixed code.
- **Bug Description**: natural language description of the bug.
- **Testbench**: the testbench for verifying that the fix is correct.

The benchmark has 100 cases, and it is built from two complementary sources: LLM-synthesized data derived from open RTL datasets and error types (❶, 60%), and real-world bug fixes extracted from GitHub version control histories (❷, 40%). Figure 1 illustrates the overall framework for constructing the Fixbench-RTL. We construct the Buggy Code-Correct Code-Bug Description triplets through error injection (Section III-B) and the GitHub version control dataset (Section III-C) as shown in process ❸. Then we generate the testbench using LLMs and feedback from a simulator (Section III-D) as shown in process ❹ in Figure 1.

The Fixbench-RTL benchmark is available at https://huggingface.co/datasets/KSU-HW-SEC/Fixbench-RTL.

B. Error Injection

To enrich the diversity of bugs, we perform targeted bug injection on correct RTL designs. We categorize bugs based on their types into three groups: syntax errors, functional errors, and design vulnerabilities. We collect correct RTL code from open-source hardware datasets such as RTLLM [13], and obtain bug descriptions from prior work [7], [20] or websites such as CWE. Table I shows the examples of bugs we injected. We list 6 bugs in each type.

Prompt template1

You are an expert in hardware design. I will give you a piece of correct RTL code and a description of a bug to inject. Please modify the code to introduce the specified bug.

Requirements:
1. Only introduce the bug described. Do not make unrelated changes.
2. Maintain valid Verilog syntax unless the bug is a syntax error.
3. Return only the modified Verilog code.

Correct RTL Code:
<INSERT_CORRECT_CODE_HERE>

Bug Description:
<INSERT_BUG_DESCRIPTION_HERE>

Fig. 2: Prompt template for bug injection in RTL code

Leveraging LLMs to inject bugs into RTL code has proven to be an effective strategy [20]. To complete the task, we design a prompt template—illustrated in Figure 2—to guide Deepseek-V3 [21] in generating buggy code. This process allows us to construct clean pairs of buggy and fixed code, along with a bug description.

C. Github Version Control Dataset

Prompt template2

You are an expert in hardware design and verification. I will provide you with two versions of RTL code: a buggy version and a corrected version. Your task is to analyze the difference and summarize the bug that was fixed.

Instructions:
1. Carefully compare the two versions.
2. Identify the cause of the bug in the buggy version.
3. Write a concise and clear natural language description of the bug.
4. Do not reference line numbers or variable names unless necessary.

Buggy Version:
<INSERT_OLD_CODE_HERE>

Corrected Version:
<INSERT_NEW_CODE_HERE>

Output Format:
Bug Description: <A short and precise explanation of the bug that was fixed.>

Fig. 3: Prompt template for analyzing and summarizing RTL bug fixes.

To capture real-world bug patterns, we extract version control histories from open-source hardware projects hosted on GitHub [11]. We identify commit pairs that involve changes to RTL files and apply filtering rules to isolate those likely corresponding to bug fixes. Candidate commit pairs are then evaluated using an LLM-based classifier to confirm whether the changes represent actual bug fixes.

Using this version control dataset, we construct a debug dataset. Figure 3 illustrates the prompt template employed with Deepseek-V3 to extract bug information by comparing the old and new code versions. Specifically, the old version serves as the buggy code, the new version as the corrected code, and the LLM's output provides the bug description, forming a triplet. This LLM-based extraction is necessary because GitHub commit messages are often brief or lack sufficient detail to describe the bug fully.

This methodology allows the benchmark to incorporate authentic, developer-written bug-fix examples that reflect realistic coding styles, thereby enhancing its representativeness and practical relevance.

D. Testbench Generation

Testbenches are essential for verifying whether the repaired RTL code successfully resolves the original bug. To serve as a reliable functional oracle, each testbench must satisfy two key conditions: (1) the *buggy* code should fail to pass verification, and (2) the *correct* code should pass. These criteria ensure that the testbench can accurately reflect the presence or absence of errors.

Prompt template3

You are an expert in hardware verification. I will provide you with a buggy RTL module and its corrected version. Your task is to generate a Verilog testbench that satisfies the following conditions:
1. The testbench should **fail** when run with the buggy code.
2. The testbench should **pass** when run with the correct code.
3. Include meaningful stimulus to trigger the bug behavior.
4. Ensure the testbench includes all necessary components (e.g., clock, reset, input stimulus, and output checks).
5. Do not include explanations—only output the testbench code.

Buggy Code:
<INSERT_BUGGY_CODE_HERE>

Correct Code:
<INSERT_CORRECT_CODE_HERE>

Output Format:
<Verilog testbench code>

Fig. 4: Prompt template for automatic testbench generation.

We employ DeepSeek-V3 to generate candidate testbenches from both the buggy and corrected code, leveraging feedback from simulation results and failure diagnostics. The prompt template is shown in Figure 4. If a generated testbench does not satisfy the required criteria, we iteratively refine the prompt

TABLE II: Performance of LLMs on the Fixbench-RTL benchmark under the 5-shot setting. Each model generates five candidate solutions per case, and a task is considered successful if any candidate passes the corresponding testbench.

Affiliation	Model	Syntax Error		Functional Error		Vulnerabilities		Github Issue	
		Detect	Fix	Detect	Fix	Detect	Fix	Detect	Fix
OpenAI	GPT-3.5-turbo	70.0%	30.0%	25.0%	25.0%	26.7%	10.0%	12.5%	0%
	GPT-4-turbo [22]	70.0%	50.0%	40.0%	25.0%	20.0%	10.0%	15.0%	0%
	GPT-4o	70.0%	50.0%	35.0%	25.0%	33.3%	20.0%	15.0%	0%
	GPT-4o-mini	40.0%	30.0%	40.0%	15.0%	6.67%	6.67%	2.50%	0%
	GPT-o1-mini	90.0%	60.0%	55.0%	30.0%	33.3%	20.0%	20.0%	2.50%
	GPT-o3-mini	80.0%	40.0%	55.0%	25.0%	36.7%	16.7%	10.0%	0%
	GPT-4.1	90.0%	80.0%	50.0%	30.0%	33.3%	26.7%	12.5%	0%
	GPT-4.1-mini	90.0%	60.0%	50.0%	25.0%	30.0%	13.3%	15.0%	0%
Anthropic	Claude-3.5-sonnet	70.0%	30.0%	45.0%	25.0%	23.3%	13.3%	12.5%	0%
	Claude-3.7-sonnet	70.0%	40.0%	35.0%	25.0%	30.0%	16.7%	10.0%	0%
Google	gemini-2.5-pro-preview	50.0%	20.0%	35.0%	20.0%	23.3%	10.0%	15.0%	0%
DeepSeek	DeepSeek-V3 [21]	80.0%	60.0%	30.0%	15.0%	43.3%	20.0%	15.0%	0%
	DeepSeek-R1 [23]	90.0%	80.0%	25.0%	20.0%	26.7%	20.0%	12.5%	0%
	DeepSeek-R1-Distill-Qwen-32B	50.0%	30.0%	20.0%	10.0%	16.7%	10.0%	22.5%	7.50%
	DeepSeek-R1-Distill-Qwen-14B	50.0%	30.0%	15.0%	10.0%	13.3%	6.67%	22.5%	7.50%
	DeepSeek-R1-Distill-Qwen-7B	30.0%	10.0%	20.0%	10.0%	10.0%	6.67%	10.0%	0%
	DeepSeek-R1-Distill-Qwen-1.5B	30.0%	10.0%	10.0%	5.00%	10.0%	0%	12.5%	0%
Qwen	Qwen2.5-72B-Instruct [24]	70.0%	40.0%	15.0%	10.0%	16.7%	6.67%	15.0%	7.50%
	Qwen2.5-Coder-32B-Instruct	80.0%	50.0%	20.0%	10.0%	13.3%	6.67%	17.5%	10.0%
	Qwen2.5-Coder-7B-Instruct	50.0%	30.0%	15.0%	5.0%	6.67%	6.67%	12.5%	7.50%
MetaAI	Llama3.3-70B-Instruct	40.0%	20.0%	30.0%	10.0%	23.3%	6.67%	17.5%	0%
	Llama-3.1-405B-Instruct [25]	70.0%	40.0%	40.0%	15.0%	20.0%	10.0%	20.0%	10.0%

using the simulation feedback and regenerate the testbench until a valid one is obtained, ensuring it accurately detects the bug in the faulty code and passes with the corrected version. This automated, feedback-driven process facilitates the efficient construction of high-quality testbenches while minimizing manual effort.

IV. Experiment

A. Experimental Setup

In this section, we evaluate the hardware debugging capabilities of various LLMs on the Fixbench-RTL benchmark. The performance of the models is assessed on two primary tasks:

(1) **Bug Detection**: accurately identifying bugs in the given buggy code;
(2) **Bug Fixing**: repairing the buggy code and generating the correct version.

We evaluate both mainstream commercial and open-source models, including: (1)OpenAI: GPT series models; (2)Anthropic: Claude 3.5 Sonnet and Claude 3.7 Sonnet; (3)Google: Gemini-2.5-pro-preview; (4)DeepSeek: DeepSeek series and Distill models; (5)Qwen: Qwen2.5 series models.

Only the buggy code—stripped of all comments—is provided as input to the LLMs, without any additional contextual information related to the bugs. This setting evaluates each model's ability to identify and fix issues solely from the code itself, simulating a scenario in which a developer is unaware of the potential errors.

We compute the pass rates for both tasks across the four types described in Sections III-B and III-C, offering a comprehensive assessment of the LLMs' debugging performance. Each model generates five candidate solutions for every test case in Fixbench-RTL. For the bug detection task, another LLM is used to determine whether the detected bugs match the "Bug Description" in the benchmark. For bug fixing, a model is considered successful if at least one of its generated solutions passes the corresponding testbench.

To further analyze the complexity of buggy code across different bug categories, we report the average number of lines in buggy code for each type in Table III. Cases derived from GitHub issues generally involve substantially longer code fragments, suggesting that they often correspond to broader or more complex modifications.

TABLE III: The number of cases and average number of lines in buggy code of the benchmark across different bug categories

Bug Type	Syntax Errors	Functional Errors	Design Vulnerabilities	GitHub Issues
Numbers of cases	10	20	30	40
Avg. Lines	72.30	92.55	100.87	210.70

B. Experimental Results

Table II summarizes the performance of all evaluated LLMs across the two tasks. The pass rate denotes the fraction of test cases successfully solved for each error type.

For **syntax errors**, LLMs demonstrate strong capabilities in both detection and repair. GPT-4.1 and DeepSeek-R1 achieve detection rates of 90% and repair rates of 80%, indicating that most syntax issues can be readily recognized and corrected.

For **functional errors**, the highest repair rate observed is only 30%. This limitation arises from the removal of all code comments and the absence of functional descriptions, requiring models to infer functionality purely from code semantics. Without explicit functional guidance, LLMs often struggle to determine the correct repair direction.

Design vulnerabilities are notably more challenging to detect than functional or syntax errors. Since these flaws do not disrupt normal functionality, they tend to be subtle and harder to identify. The benchmark includes many FSM-related vulnerabilities, where accurately extracting state transition logic and identifying latent flaws poses an additional challenge. Among all models, DeepSeek-V3 achieves the best performance, with a detection rate of 43.3% and a repair rate of 20%.

GitHub issues typically involve large-scale and complex hardware designs, making it difficult for LLMs to pinpoint potential defects. Most models fail to produce valid repairs. Although some models, such as Qwen2.5-Coder-32B, can identify multiple potential bug locations—some of which may indeed be correct—the overall repair rate remains low, with Qwen2.5-Coder-32B achieving only 10%.

In summary, current state-of-the-art LLMs perform well on syntax-related debugging but face significant challenges in addressing functional errors, design vulnerabilities, and complex real-world code. Providing additional contextual or semantic information appears essential for improving their debugging performance.

V. Conclusion

In this paper, we introduce Fixbench-RTL, a novel benchmark for evaluating LLMs on hardware debugging tasks, focusing on RTL code. Fixbench-RTL contains 100 test cases that cover a wide range of error types, including syntax errors, functional bugs, and design flaws, as well as real-world examples from GitHub issues. The proposed framework allows for easy extension of the benchmark. Our experimental results indicate that the hardware debugging capabilities of current popular LLMs are insufficient for practical use. These results underscore the need for further improvements in model capabilities, particularly in understanding hardware-specific patterns and handling complex designs. Our work lays the groundwork for future research that aims to enhance the hardware debugging abilities of LLMs.

Acknowledgment

Portions of this work were supported by the National Science Foundation (2340949 and 2419880).

References

[1] S. Thakur, B. Ahmad *et al.*, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[2] S. Liu, W. Fang *et al.*, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," *arXiv preprint arXiv:2312.08617*, 2023.

[3] W. Fu, K. Yang *et al.*, "Llm4sechw: Leveraging domain-specific large language model for hardware debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2023, pp. 1–6.

[4] S. Li, W. Fu *et al.*, "Intelligence in the fence: Construct a privacy and reliable hardware design assistant llm," in *Proceedings of the Great Lakes Symposium on VLSI 2025 (GLSVLSI '25)*. ACM, 2025, p. 659–666.

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

[6] X. Yao, H. Li *et al.*, "Hdldebugger: Streamlining hdl debugging with large language models," *arXiv preprint arXiv:2403.11671*, 2024.

[7] K. Xu, J. Sun *et al.*, "Meic: Re-thinking rtl debug automation using llms," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[8] M. Chen, J. Tworek *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[9] R. Li, L. B. Allal *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[10] M. Liu, T.-D. Ene *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[11] W. Fu, S. Li *et al.*, "A generalize hardware debugging approach for large language models semi-synthetic, datasets," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 72, no. 2, pp. 623–636, 2025.

[12] S. Thakur, J. Blocklove *et al.*, "Autochip: Automating hdl generation using llm feedback," *arXiv preprint arXiv:2311.04887*, 2023.

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

[14] W. Fu, S. Li *et al.*, "Hardware phi-1.5b: A large language model encodes hardware domain specific knowledge," in *Proceedings of the 29th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '24. Incheon, Republic of Korea: IEEE Press, 2024, p. 349–354. [Online]. Available: https://doi.org/10.1109/ASP-DAC58780.2024.10473927

[15] R. Qiu, G. L. Zhang *et al.*, "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*, ser. MLCAD '24. New York, NY, USA: ACM, 2024. [Online]. Available: https://doi.org/10.1145/3670474.3685956

[16] B. Ahmad, S. Thakur *et al.*, "Fixing hardware security bugs with large language models," *arXiv preprint arXiv:2302.01215*, 2023.

[17] T. M. Corporation, "CWE - CWE-1194: Hardware Design (4.1)," 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/1194.html

[18] "Hardware | OpenTitan Documentation," 2019. [Online]. Available: https://docs.opentitan.org/hw/

[19] HACK@EVENT, "HACK@DAC21 – HacK@EVENT," 2022. [Online]. Available: https://hackatevent.org/hackdac21/

[20] D. Saha, K. Yahyaei *et al.*, "Empowering hardware security with llm: The development of a vulnerable hardware database," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2024, pp. 233–243.

[21] DeepSeek-AI, "Deepseek-v3 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2412.19437

[22] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.08774

[23] DeepSeek-AI, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[24] A. Yang, B. Yang *et al.*, "Qwen2.5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

[25] A. Grattafiori, A. Dubey *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.