MEIC: Re-thinking RTL Debug Automation using LLMs

Ke Xu[1], Jialin Sun[1], Yuchen Hu[1], Xinwei Fang[2], Weiwei Shan[1], Xi Wang[1] and Zhe Jiang[1]
[1]National Center of Technology Innovation for EDA, School of Integrated Circuits, Southeast University, China
[2]Department of Computer Science, University of York, UK

Abstract

The deployment of Large Language Models (LLMs) for code debugging (e.g., C and Python) is widespread, benefiting from their ability to understand and interpret intricate concepts. However, in the semiconductor industry, utilising LLMs to debug Register Transfer Level (RTL) code is still insufficient, largely due to the underrepresentation of RTL-specific data in training sets. This work introduces a novel framework, **Make Each Iteration Count (MEIC)**, which contrasts with traditional one-shot LLM-based debugging methods that heavily rely on prompt engineering, model tuning, and model training. MEIC utilises LLMs in an iterative process to overcome the limitation of LLMs in RTL code debugging, which is suitable for identifying and correcting both syntax and function errors, while effectively managing the uncertainties inherent in LLM operations. To evaluate our framework, we provide an open-source dataset comprising 178 common RTL programming errors. The experimental results demonstrate that the proposed debugging framework achieves fix rate of 93% for syntax errors and 78% for function errors, with up to 48x speedup in debugging processes when compared with experienced engineers. The Repo. of dataset and code: **https://anonymous.4open.science/r/Verilog-Auto-Debug-6E7F/**.

1 Introduction

In hardware development, the verification and debugging processes are notably laborious and time-consuming, requiring twice the duration of the design phase itself [21]. This significant investment in time and resources shows the need for more efficient methods.

Large Language Models (LLMs) have the potential to revolutionise this process, which have demonstrated a remarkable capability to interpret hardware specifications using natural language and generate corresponding Register Transfer Level (RTL) code, such as Verilog and VHDL. Existing efforts [3, 9, 23, 36] have showcased the potential of LLMs in automating hardware design, but have also revealed significant limitations. The primary issues are related to the unstable performance of LLMs and the intrinsic complexities of RTL code itself [48], which often result in error-prone outputs. In response to these limitations, a growing body of research, including RTLFixer [38], SBF [1], LLM4SecHW [13], HDLdebugger [46], and AssertLLM [11], has been undertaken to enhance LLM-based RTL debugging. These studies employed techniques such as prompt engineering [30, 43], model tuning [4, 23], and model training [15, 22] to address these challenges. However, despite these efforts, the performance of these approaches is still far from practical, as evidenced by persistently low *pass@k* rates[1].

In contrast to previous works, our approach is inspired by established human debugging practices, recognising that *"there is never one-shot debugging"*. As depicted in Figure 1, the debugging process

[1]The pass@k metric measures the probability that at least one of the top k outputs generated by a model correctly solves a given problem, used to evaluate the effectiveness of solutions in tasks like code generation and debugging.

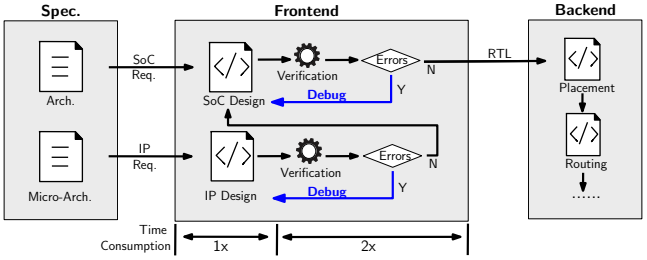


Figure 1: Hardware development flow in the human world *(Spec: Specification; Arch: Architecture; Req: requirement)*: the flow involves the specification definition, frontend development, and backend implementation. After the design requirement is defined, the RTL is coded at both IP and SoC levels. To ensure the design's correctness, multiple iterations of the verification and debugging must proceed, usually consuming twice the duration compared to the design phase.

in human-led environments is not only collaborative but also iterative [19, 20, 24, 41, 42]. Each stage of the debugging process, from the initial RTL design phase to final verification, involves multiple iterations where different individuals with diverse capabilities engage in verifying and debugging the code. This method continues until the code achieves an error-free state or meets stringent coverage criteria. Acknowledging that uncertainties in LLM outputs are similar to the variabilities in human performance, this human-led model provides a solid foundation for developing LLM-based RTL debugging methods. Particularly, employing an iterative approach addresses the inherent uncertainties associated with LLM models.

Contributions. We present **Make Each Iteration Count (MEIC)**, a novel framework that utilises multiple LLMs to enable automated and iterative debugging of RTL code. MEIC is designed to address the above limitations commonly associated with applying LLMs in hardware debugging. The main contributions of this paper are:

- **An iterative framework:** MEIC integrates RTL toolchain (e.g., compilers and simulators), with two LLM agents, and a code repository. This allows continuous evaluation, testing and debugging of RTL code, mitigating uncertainties caused by the fluctuations in the performance of LLM outputs.
- **Dual LLM deployment:** MEIC employs a fine-tuned debug agent that identifies and attempts to correct syntax and function errors, followed by an LLM scoring agent that assesses the quality of RTL candidates derived from the debug agent. This deployment provides quantified and traceable feedback that informs further iterations.
- **An open-sourced tooling:** To ease the adoption of MEIC, we developed a tooling to enable wide and early utilisation of MEIC which is publicly available on **https://anonymous.4open.science/r/Verilog-Auto-Debug-6E7F/**.

- **Extensive empirical validation:** We present an open-source error dateset derived from RTLLM [25]. This evaluation dataset contains 178 code instances generated from our random error generator which include common syntax and function errors across various modules of combinational and timing logic circuits.
- **Demonstrated performance improvement:** MEIC, incorporating GPT-4, significantly enhances debugging automation and performance, achieving a syntax error fix rate of 93% and a function error fix rate of 78% using our test dataset. Also, it delivers up to 48x speedup in debugging processes when compared with experienced engineers.

Organisation. The rest of the paper is organised as follows: Section 2 presents the top-level concepts of MEIC, Section 3 introduces the details of MEIC and the rationales. Section 4 evaluates our framework against five research questions, followed by the related work given in Section 5. Section 6 concludes and offers the insights.

2 MEIC: An Overview

Intended for use in the design and verification stages, MEIC aims to help hardware developers identify and correct both syntax and function errors in RTL code. This systematic framework (see Figure 2), wraps a RTL toolchain (e.g., compilers and simulators), two LLM agents (fine-tuned[2] for code debugging and assessment), and a code repository. To ensure the framework's applicability across different scenarios, we standardised its inputs as:

- **Design specification**: outlining the intended and expected behaviour of the hardware component;
- **RTL code**: containing the untested RTL code of the initial hardware design, i.e., Design Under Test (DUT).
- **Testbench**: acting as the reference for verifying the functional correctness of the RTL code.

MEIC assumes that the LLMs employed can perform better through proper fine-tuning and prompt engineering. Our approach for this processing (e.g., fine-tuning and prompt engineering) of LLMs is discussed in Section 3. Under these assumptions, MEIC attempts to correct the RTL code as necessary across a number of iterations from four pipeline stages (Step ❶– ❹) in Figure 2, highlighting our principle that debugging is an iterative, not an one-shot process. The iterative MEIC pipeline involves the following steps:

- **Step ❶**: the debug agent is expected to correct the erroneous RTL code (both syntax and function errors) by producing a code candidate based on its inputs;
- **Step ❷**: the RTL code candidate is analysed and evaluated by the scorer agent, which assesses the quality of the generated code candidate and assigns a numerical score;
- **Step ❸**: the RTL code candidate and its score are stored in the repository to enable a rollback mechanism, preventing the current RTL code from being overwritten by potentially incorrect LLM outputs, e.g., skipping lines of the code;
- **Step ❹**: the repository selects the RTL code with the highest score for the new interaction round, continuing until the framework meets the termination condition.

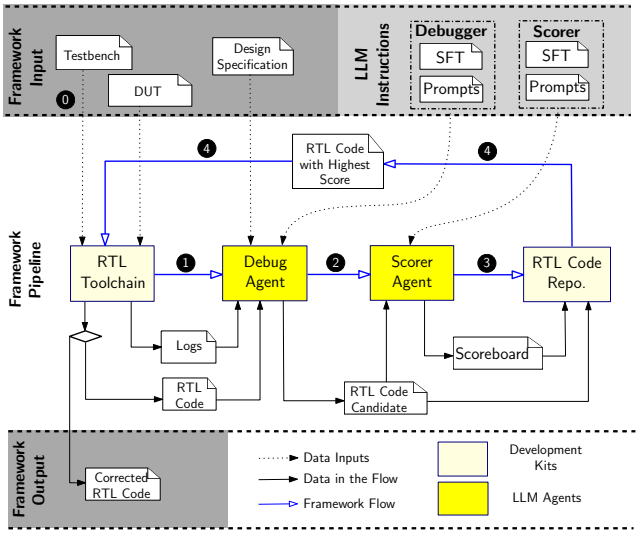


Figure 2: MEIC overview: the framework initialises with the DUT, which is compiled and simulated by the RTL toolchain (step ❶). The resultant logs and code are forwarded to the debug agent for error resolution (step ❶). The revised RTL code is examined by the scorer agent (step ❷) and stored in the repository (step ❸), from which the highest-scored code is selected for the following debugging iteration (step ❹).

The framework terminates and outputs the latest design code analysed in the toolchain if **i)** no error is found or **ii)** it reaches the threshold of the maximum number of iterations.

Modularisation and flexibility. It is worth noting that standard interfaces between different stages of the pipeline are used. This ensures easy upgrades and extensions. For instance, replacing the debug agent with a domain-specific model or altering the RTL toolchain can be achieved by simply modifying the API or maintaining consistent log formats. This flexibility allows MEIC to adapt to various debugging scenarios and technological advancements.

Contexts. As a framework, MEIC is agnostic to the RTL and LLMs. For illustrative purposes, we used Verilog, the most widely adopted RTL language in the industry, along with its associated toolchain, *ModelSim* [31], as an example throughout the paper. For LLMs, we used different versions of GPT models from OpenAI (see Section 4).

3 MEIC: The Framework Pipeline

We first discuss our preparations before the operation of the pipeline, including the error classifications (Section 3.1) and the tuning method (Section 3.2), both of which aim to enhance the LLM's performance. For the operation of the pipeline, we introduce the simulation process that uses the RTL toolchain combined with feedback engineering (Section 3.3), followed by the discussion on the microsystems integrated with the LLM agents for debugging

[2]We acknowledged that fine-tuning's definitions are various in different LLMs. Here we use the GPT4 as an example: https://platform.openai.com/docs/guides/fine-tuning.

Table 1: Common Verilog error categories and examples.

Types	Error	Description	Expected Form	Unexpected Form
Syntax Errors	Premature Termination	Missing or redundant punctuation (e.g., semi-colons or commas) causing premature end of the execution.	module A(input a, **output b);**	module A(input a, **output b)**
	Undefined Variable	Using variables that have not been previously declared.	assign **result** = temp;	assign **resutl** = temp;
	Operator Misuse	Operator misuse (e.g., incorrectly using the assignment operator '=' instead of the comparison operator '==' for evaluating conditions) resulting in unacceptable expression format.	if (**a == 2'b10**) begin b <= 1'b1; end	if (**a = 2'b10**) begin b <= 1'b1; end
	Redundant Variable Declaration	Declaring the same variable multiple times e.g. in port definitions.	module A(input a, output b); reg **a_temp**;	module A(input a, output b); reg **a**;
	Incorrect Encoding	Presence of characters not aligning to ANSI encoding standard.	module A(input **a**, output b);	module A(input **â**, output b);
	Incorrect Data Type Assignment	Failing to comply with assignment rule for reg- and wire-type data.	reg a; always @(*) begin **a = b**; end	reg a; **assign a = b**;
	Port Mode Declaration Error	Failing to declare module port according to the rules.	module A(a, b); input a; **output b;**	module A(a, b); input a; **//Declaration for b is missing.**
	Data Index Out-of-Bounds Error	Exceeding allowable data bounds during array or vector operations.	reg [32:1]a; assign b = **a[16:1]**;	reg [32:1]a; assign b = **a[15:0]**;
	Improper Use of Keywords	Using reserved keywords incorrectly or as identifiers.	reg **alway**;	reg **always**;
Function Errors	Insufficient Bit Width	Defining registers with inadequate bitwidth.	wire **[3:0]** a; assign a = 4'b1000;	wire **[3:1]** a; assign a = 4'b1000;
	Incomplete Port Connection	Failing to connect all ports during module instantiation in Verilog.	mod md(.a(a), **.b(b)**);	mod md(.a(a), **.b()**);
	Flawed Sensitivity List	Omitting or mis-specifying signal data in Verilog's sensitivity list.	always @(posedge clk or **negedge** rst_n) begin a <= b + c; end	always @(posedge clk or **posedge** rst_n) begin a <= b + c; end
	Misuse of Assignments	Misusing blocking (=) and non-blocking (<=) in sequential design.	always @(posedge clk or negedge rst_n) begin **a <= b + c**; end	always @(posedge clk or negedge rst_n) begin **a = b + c**; end
	Logical Errors in Expressions	Complex and incorrect module logic during code formulation.	assign a = **b + c**;	assign a = **b & c**;
	Concurrent Variable Use	Assigning the same variable in multiple processes.	always @(*) begin a=**1'b1**; end	always @(*) begin a=**1'b1**; end always @(*) begin a=**1'b0**; end
	Mismatched Assignment Values	Omitting base indication in values that leads to unexpected assignments.	if (a == **2'b10**) begin b <= 1'b1; end	if (a == **10**) begin b <= 1'b1; end
	Incorrect Module Instantiation	Instantiating a non-existent module, but only fails in functionality.	**mod** md(.a(a), .b(b));	**mdo** md(.a(a), .b(b));
	Infinite Loop Constructs	Loops using *forever*, *while*, or *for* without a clear termination condition will not end.	next_stage <= **next_stage_temp**;	next_stage <= **current_stage**;

(Section 3.4) and for the best-version code selection (Section 3.5). Finally, we briefly introduce the proposed open-source Verilog error dataset (Section 3.6) that is used in the evaluation.

3.1 Error Classifications

Understanding error classifications in Verilog is essential for effective debugging. For human engineers, knowing whether an issue is a syntax or function error allows for the selection of appropriate tools and techniques (compilers and linters for syntax errors, and simulators, waveform analysers, and timing analysis tools for function errors). Similarly, for LLMs, this classification would aid in executing more accurate debugging processes as it would allow for a more structured reasoning [8, 26, 40, 45].

Syntax errors. Syntax errors are errors that occur when the code violates the formal structure of the Verilog language. Such errors are typically identified by compilers during the parsing stage, preventing further simulation or synthesis. The compiler, e.g., ModelSim and DC, produces logs detailing the location and nature of these errors, thus helping with quick error detection and correction.

Function errors. Function errors encompass all other errors that affect the operation code and include semantic errors, logical errors, timing errors, etc. Unlike syntax errors, function errors are concerned with the behaviour and outcome of the code rather than its grammatical correctness. These errors can be more challenging to detect and often require extensive testing, simulation, formal verification, and detailed examination of timing and synthesis reports. For practical identification of such errors, assertions and testbench are commonly employed to identify unexpected outcomes for subsequent correction (see Section 3.3).

```
// Preset the role of the LLM, determine the input and output information.
1  #You are an expert in IC design, specialising in Verilog language. Your primary role is
   to analyse Verilog source code, identify errors, and provide updated, error-free designs.
2  #You will be offered the specification, design code, compilation logs and simulation logs.
   // Give inputs in format.
3     Here is the design specification:
      {Spec.} // ... Spec. details omitted ...
4     Here is the design code:
      {Code} // ... Module details omitted ...
5     Here are the logs:
      {Logs} // ... Log details omitted ...
   // Emphasize the process of Self-planning.
6  #Please offer necessary steps to debug the design module code. Then try to achieve
   the debugging.
7  #Your goal is to assist in achieving a fully functional and error-free Verilog design.
```

Figure 3: Part of the input patterns for the self-planning. In addition to debugging based on provided files (lines 1-4), the agent is also required to plan the debugging process (line 5).

Drawing on the practical errors identified in our past hardware designs and insights from previous studies [7, 34], we categorised the Verilog errors and gave corresponding examples in Table 1.

3.2 Tuning Method

Domain-specific knowledge. It is well-known that LLMs can achieve better performance in a specific domain by utilising system-level instructions and extra domain knowledge [13, 22, 28]. Based on this property, we fine-tuned the LLM by supplying relevant information as system-level instructions and incorporating domain knowledge through the prompts [12, 27, 32, 39, 47].

To facilitate the interpretation and correction of Verilog code, we incorporated the Verilog-2001 standard [33] into the knowledge, along with a number of RTL code examples (e.g., HDLBits [44], Verilog-G [35], RTL Coding Guidelines [29], etc.) and the error knowledge (in Table 1). The previously gathered debugging-related data, subjected to cleansing and extraction, given through prompts.

```
1  After reviewing the documents, I will now proceed to analyse and update the Verilog
   code to meet the design specification and resolve any issues indicated in the
   compilation and simulation logs.
2  Firstly, I have examined the design specification:
   // ... Spec. analysis details omitted ...
3  Next, I will scrutinise the design code for structural and logical correctness:
   // ... Module code analysis details omitted ...
4  The compilation logs are crucial to identify any syntax errors or warnings:
   // ... Compilation log analysis details omitted ...
5  The simulation logs provide insights into the behavior of the design:
   // ... Simulation log analysis details omitted ...
6  Based on the analysis of the design documents, here are my suggested modifications and
   the updated design file.
   // ... Updated module details omitted ...
```

Figure 4: Reply of the LLM. With self-planning technique, the LLM provides a set of steps for RTL analysis and debugging.

Prompting techniques. To further improve the performance of the LLMs in code generation and debugging, we utilised advanced prompt engineering techniques, e.g., *self-planning* and *role prompting* as described in [18, 25]. The use of self-planning helps the LLMs to break down a complex task into several planned steps and the use of *role prompting* enables more relevant and contextually appropriate responses, allowing structured and relevant LLM responses.

We captured snapshots of our prompts when fine-tuning the LLM, as shown in Figure 3. The response of the LLM, as presented in Figure 4, provides a clear outline of the five structured steps (i.e., from line 2-6). It suggests that the specification is analysed first, followed by the analysis of the design code, the compliance logs and the simulation logs. Finally, it suggests the RTL modifications.

3.3 Toolchain, Compilation, and Simulation

We employed a toolchain to verify RTL code compliance with design requirements by evaluating compilation and simulation results.

As mentioned in Section 3.1, syntax errors can be directly detected during the compilation and detailed in the compilation logs, while function errors often go undetected during compilation and result in unexpected outputs during simulation. Hence, building test cases is crucial for automatically identifying and correcting function errors, ensuring that the output meets expectations.

For Verilog, function errors are typically identified using two common methods: testbenches and assertions. Because the testbenches and assertions provide different granularity of error information, we integrated both methods in MEIC.

Testbench-based detection. The testbench serves as a reference model, continuously providing stimulus to the DUT and verifying its outputs against the expected results. To mitigate common-cause errors between the DUT and the testbench, we developed the test cases in the testbench using Python. Specifically, we employed `Random` library to generate input data and wrote the corresponding functionality given in the specification. With that, we developed an automated script shown in Fig 5 that translates the reference model into Verilog syntax, maintaining alignment with the standards. The `$display()` function is used to report the results (Figure 6(a)).

Assertion-based detection. To provide more traceable error feedback, assertions are employed in the conventional debugging approach. The original Verilog code is transformed into SystemVerilog code, and assertions are incorporated into the code for error-prone areas. If the function at the assertion is erroneous at simulation, the simulation will be terminated directly, and logs will be displayed (Figure 6(b)). Different from the testbench-based detection, which only records comparisons between inputs and outputs, the assertion-based approach offers more precise about the errors.

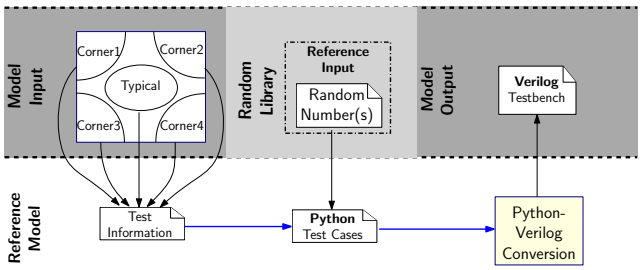


Figure 5: Testbench from reference model to Verilog.

Following the acquisition of the RTL code and different detection techniques, ModelSim is used for local simulation. To ensure the smooth operation of the entire framework, the simulation process is automated to reduce the needs for manual intervention. This is achieved through `CMD` commands that automate the simulation while generating compilation and simulation logs.

Other detection techniques. We acknowledged that alternative detection techniques are being explored within the community. For instance, references [3, 16, 25] demonstrate the use of LLMs to generate testbenches. As outlined in Section 2, MEIC is designed to be versatile, supporting a broad ranges of error detection approaches, including those based on LLMs. This compatibility only requires that these techniques supply formalised simulation logs, which are then used as input for the framework.

3.4 Debug Agent

The LLM agent functions as an expert in RTL debugging, expecting four inputs: specification, Verilog code, compilation logs, and simulation logs. The specification includes a design description that outlines the code's functionality as well as its inputs and outputs. Following the RTL simulation, both compilation and simulation logs are generated (Section 3.3). Using these logs, it is expected that the debug agent can locate the error and leverage its expansive knowledge base (Section 3.1 and 3.2), giving suggestions for modifications and providing an updated version of the code.

Debug iteration(s). Based on the knowledge outlined in Section 3.2, the debug agent then returns the modified design file. However, in most cases, neither human engineers nor LLM agents can get the code right on the first attempt. Therefore, we followed the debugging process of human engineers and introduced an iterative process. If the agent does not generate the correct code in the current iteration, the next iteration will be performed using the highest-scored code from the previous iterations.

Furthermore, the type of error encountered dictates the error messages generated. For example, syntax errors only result in the generation of compilation logs, as simulation logs are generated when the syntax is correct. Therefore we need to perform format control based on these two situations. The key prompts for the debug agent are shown in Figure 7.

Formalising agent's outputs. It is often observed that LLM's response often containing irrelevant information to code debugging e.g., the LLM's debugging reasoning process or explanations of the code. Based on our observation, the LLM's code outputs typically follow a specific format, as shown in Figure 8. To prevent noisy inputs from being used in subsequent iterations, we systematically clean and extract the LLM's responses to ensure that only the

```
'timescale 1ns/1ps
module tb ();
//... Module Instantiation ...
initial begin
   ...
   if(error!=0) begin
      $display ("The testbench inputs are: var1 = m'H% h,
      var2 = n'H% h,... . But the actual results are: rslt1 = a'H% h,
      rslt2 = b'H% h,...", var1, var2, ..., rslt1, rslt2...);
   end
   ...
   if(error==0) begin
      $ display("=====Your Design Passed=====");
   end
   $finish
 end
//... Test cases ...
endmodule
```

(a) Testbench.

```
module div_16bit (
   input wire [15:0] A,
   input wire [7:0] B,
   output wire [15:0] result,
   output wire [15:0] odd ) ;
...
   always@(*) begin
      for (i = 0; i <16; i = i + 1) begin
         tmp_a_prev = tmp_a;
         tmp_a = tmp_a >>1; //Logic error here
         tmp_result = tmp_result <<1;

         assert (tmp_a[31:16] == {tmp_a_prev[30:16], tmp_a_prev[15]})
         else $error ("Shift left operation failed"); //Assertion here
      end
   end
...
endmodule
```

(b) Assertion.

Figure 6: Methods for function errors' detection, which provide different granularity of information.

```
// Prompts in runtime, emphasising not changing the code structure to avoid new problems.
#You are an expert in the field of Verilog design. I will provide you with
the design Spec., design code, as well as the compilation logs and simulation logs.
Please modify the design code based on this information. Do not drastically change
the structure of code.
// Modify code according to comments.
#If there are comments in the code, modify the code with reference to the comments
and retain these comments after modification.
// Processed on a case-by-case basis depending on whether compilation passes.
#If the Compile fails, I will provide you with the compilation logs, please modify the
corresponding lines of design code based on the information in the logs.
#If the Compile passes, I will only provide you with the simulation logs.
The simulation logs contain error information during simulation.
Based on the information, please modify the design code to make it functionally right.
// Determine output format.
#Offer corrected Verilog design code omitting testbench. Please fix the error(s)
according to the design specification and logs.
```

Figure 7: System prompts in runtime for the debug agent.

essential parts (i.e., the code as shaded in Figure 8) are carried forward for use in the next iteration.

```
... LLM Description ...
"'{language} #In our application, the language is Verilog.
module {module name}
   ... Code Details ...
endmodule
"'
... LLM Description ...
```

Figure 8: Code output format from the LLM.

3.5 Scorer Agent and Exception Handling

While employing the LLMs, unexpected situations may arise, also known as "hallucination" [2, 6, 14, 17]. For example, the LLM may return incomplete code. If subsequent iterations are based on these flawed outputs, the quality of the results may be compromised. Also, the LLM can inadvertently modify both erroneous and correct portions of the code, leading to a situation where most of the iterations are spent addressing new errors introduced by the LLM itself. Two common "exceptions" are shown in Figure 9. To avoid these exce[illegible]s the e[illegible]

```
module traffic_light (
   input wire rst_n,
   input wire clk,
   input wire pass_request,
   output wire [7:0] clock,
   output wire red,
   output wire yellow,
   output wire green ) ;

// Code is missing.

endmodule
```

(a) Code missing.

```
module traffic_light (
   input wire rst_n,
   input wire clk,
   input wire pass_request,
   output wire [7:0] clock,
   output wire red,
   output wire yellow,
   output wire green ) ;
...
// State transition logic
   always @( posedge clk or negedge rst_n) begin
      if (!rst_n) state <= idle;
      else case (state)
         ...
         s1_red: if (cnt == 0) state <= s3_green; else state <= s1_red;
   // Here should be 'cnt==d3', which is correct in the original code.
         ...
      endcase
   end
...
endmodule
```

(b) New error generated.

Figure 9: Exceptions should be handled by the scorer agent.

Scorer agent. We introduced a scorer agent to detect unexpected cases. During the debugging process, modifications should aim to minimise the change to the original code. After the debug agent proposes corrections, the modified code is compared to the version from the previous iteration. The scorer agent then evaluates the code based on its completeness and overall quality. If the assessed score falls below a predefined threshold, a rollback mechanism is activated to revert the design code to the most recent comparable and simulatable version. To improve the reliability of the scoring, the scorer agent is prompted with a variety of metrics:

- **Readability:** The clarity of the code that can be understood.
- **Maintainability:** The ease with which the code can be updated or altered.
- **Robustness:** The capacity to handle errors and anomalies.
- **Standards Compliance:** The alignment of the code with established Verilog coding standards.

Although these metrics are qualitative, the scoring process remains effective because all RTL codes are evaluated using the same scorer, ensuring consistency between assessments. In addition, the MEIC only interests the relative scores between the iterations. To further mitigate the uncertainty of the qualitative scoring, a low temperature[3] is configured to the scorer agent.

[3]A hyper-parameter in GPT models that controls the randomness of GPT's responses. A lower value is associated with less randomness in their responses

Rollback mechanism. Within the scorer agent, we introduced a mechanism to support possible rollback. This is achieved by saving each version of the Verilog code, along with its corresponding compilation and simulation logs from each iteration, in a designated location. This mechanism not only enables rollback but also enhances traceability as the iteration evolves.

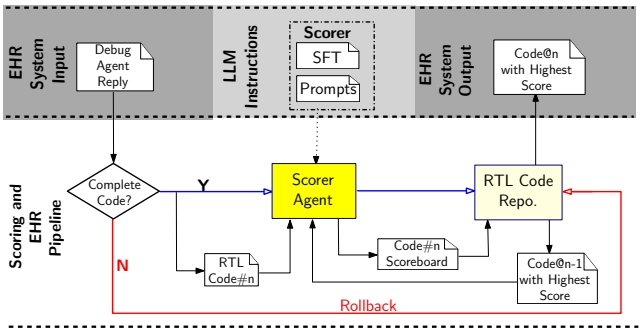


Figure 10: Flow of Exception Handling and Rollback (EHR).

The rollback mechanism is primarily triggered based on the completeness of the code produced by the LLM and the score provided by the scorer agent. We assume that the completeness is indicated by the number of lines in the code. If a significant reduction in lines or missing code is observed, we trigger the rollback mechanism regardless of the scorer agent. Otherwise, rollback occurs when the code scored below a predetermined threshold in the scorer agent. Once triggered, the highest-scored code from the previous iteration is used for the next iteration, rather than the most recently produced code. This handling of exception is illustrated in Figure 10.

3.6 Error Dataset

Different modules may exhibit diverse error distributions. To evaluate the debugging capabilities of MEIC, we used an open-source dataset [25] and intentionally introduced a variety of common errors to construct a specialised dataset tailored for debugging. Our goal is to comprehensively represent the spectrum of prevalent Verilog errors by selecting a representative sample of designs.

Because our dataset is developed based on RTLLM [25], some error types might not be generated in certain modules, such as the infinite loop construct in the loop-free modules. Therefore, this error dataset primarily includes the errors that are prevalent across most modules (summarised in Table 2), such as premature termination, undefined variable, etc.

Table 2: Verilog error dataset.

Module type \ Error type	Syntax	Function	Total
Arithmetic	57	38	95
Logic	50	33	83
Total	107	71	178

Random error generation. Given that the majority of the errors introduced are common errors, we also implemented an automatic error generation method, which could be used for more compressive evaluation. For simpler error types (e.g., misuse of assignments), we devised a set of bug-pattern lists, using regular expressions to identify segments suitable for random error insertion. For more complex function errors (e.g., infinite loop), error generation is facilitated through GPT-4 or manual insertion, followed by a data cleansing process. The final dataset size could potentially expand to 200 times the original data size.

4 Evaluation

This section presents our experimental setup, research questions, evaluation metrics and results to answer the questions.

Setup. In our experiment, the LLM agents powered by the OpenAI website interface were utilised. Unless otherwise stated, we used the GPT-4 Turbo model as our default LLM agent for both debugging and scoring. We set the temperature of the agent, which controls the randomness of the LLM's output, to 0.7 for the debug agent as deemed optimal in our evaluation of **RQ1**, and 0.1 for the scorer agent to minimise the randomness of each scoring process as discussed in Section 3.5. We then developed test cases containing various Verilog design scenarios using ModelSim SE 10.7 simulation environment. We set the threshold of iterations to 10, as based on our experiences, the improvement is hardly observed after that.

4.1 Research Questions

We carried out the experiments to evaluate our framework against five key Research Questions (RQs):

RQ1 (Sensitivity): How does *temperature* in GPT-4 setting impact the performance of MEIC in terms of FR? This research question explores the effect of LLM's temperature which controls randomness and lowering the temperature results in less random completions. It seeks to understand how the randomness of LLM's output impact the dubugging performance[4].

RQ2 (Effectiveness): Can MEIC correct different types of errors across various modules? This research question investigates the performance of the MEIC system in terms of its ability to fix errors across a variety of code modules, focusing on both syntax and function errors. It seeks to understand how the system's debugging effectiveness varies depending on the complexity of the code and the type of error encountered.

RQ3 (Impactability): How do various LLM-based configurations and integration impact debugging performance in terms of fix rate? This research question explores the potential improvements in error correction capabilities through both fine-tuning the models and integrating them with the MEIC framework. It aims to evaluate whether the proposed MEIC framework can significantly enhance RTL debugging performance.

RQ4 (Usability): How does MEIC work with different LLM agents? This research question aims to compare the effectiveness of integrating different LLM models in MEIC, specifically GPT-3.5 and GPT-4, in debugging code. It seeks to understand how different LLM models influence the performance of MEIC, quantifying their usability for RTL debugging.

RQ5 (Performability): How does MEIC compare with human experts in debugging performance? This research question evaluates how the MEIC debugging performance compares to that of human experts. It seeks to determine whether the framework, when

[4]We focused on the debug agent because we argued that the debug agent plays a more important role in debugging workflow than the scorer agent.

Table 3: The syntax and function error debugging with different LLMs. The highest FR of each module is marked.

Types	GPT-3.5		GPT-4		GPT-3.5+Knowledge		GPT-4+Knowledge		GPT-3.5+MEIC		GPT-4+MEIC	
	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.	Syntax	Func.
accu	57.14%	36.67%	28.57%	60.00%	47.61%	33.33%	66.67%	40.00%	42.86%	33.33%	74.29%	50.00%
adder_8bit	62.50%	58.33%	91.67%	91.67%	50.00%	58.33%	100.00%	100.00%	66.67%	91.67%	100.00%	100.00%
adder_32bit	62.96%	46.67%	85.19%	66.67%	51.85%	33.33%	88.89%	73.33%	77.78%	46.67%	97.14%	94.00%
adder_pipe_64bit	23.81%	40.00%	33.33%	26.67%	23.81%	40.00%	95.24%	86.67%	42.86%	60.00%	94.29%	90.00%
div_16bit	20.00%	0.00%	27.78%	40.00%	16.67%	20.00%	72.22%	26.67%	16.67%	20.00%	81.67%	62.00%
multi_booth_8bit	100.00%	26.67%	75.00%	33.33%	100.00%	46.67%	100.00%	80.00%	100.00%	60.00%	100.00%	77.50%
multi_pipe_8bit	9.52%	33.33%	80.95%	73.33%	28.57%	46.67%	100.00%	60.00%	28.57%	42.86%	100.00%	80.00%
radix2_div	74.07%	61.11%	11.11%	50.00%	66.67%	55.56%	70.83%	55.56%	75.00%	61.11%	66.25%	46.00%
alu	28.57%	60.00%	61.90%	86.67%	66.67%	66.67%	90.48%	93.33%	85.71%	73.33%	97.14%	97.50%
asyn_fifo	37.50%	37.50%	83.33%	54.16%	25.00%	33.33%	91.67%	62.50%	41.67%	33.33%	89.29%	78.57%
freq_div	100.00%	83.33%	100.00%	100.00%	95.24%	83.33%	100.00%	83.33%	100.00%	94.44%	100.00%	100.00%
parallel2serial	45.83%	50.00%	75.00%	75.00%	66.67%	91.67%	91.67%	100.00%	66.67%	100.00%	95.00%	100.00%
serial2parallel	75.00%	26.67%	79.17%	40.00%	75.00%	20.00%	100.00%	26.67%	79.17%	20.00%	98.75%	58.75%
traffic_light	19.04%	20.00%	0.00%	40.00%	9.52%	46.67%	90.48%	40.00%	28.57%	40.00%	95.71%	44.00%
width_8to16	73.33%	74.44%	100.00%	100.00%	100.00%	80.00%	100.00%	80.00%	100.00%	100.00%	100.00%	97.50%
FR	54.28%	45.73%	62.83%	61.96%	55.26%	49.57%	90.69%	66.24%	64.26%	56.90%	**92.68%**	**78.39%**

integrated with LLM models, can achieve comparable or superior results to human experts in identifying and fixing errors in code.

4.2 Evaluation Metrics

Fix Rate (FR). In recent work, such as [5, 10], the use of pass@k metrics to assess function correctness was mentioned. For each problem in the problem set, k code samples are generated at a time, and the problem is considered solved if any of the k samples pass the simulation test (without syntax and function errors).

Specifically, we used Fix Rate (FR) to quantify the debug ability of the debugging framework[37]. For an error code θ_i and its fixed version θ_i^*, we had a corresponding set of test cases in testbench $\left(x_i^0, y_i^0\right)$,$\left(x_i^1, y_i^1\right)$, …,$\left(x_i^m, y_i^m\right)$. For the correct version of the code, θ_i^*, it should produce the correct output y_i^j when applied to the input data x_i^j from the test cases. That is, $a_{\theta_i^*}\left(x_i^j\right) = y_i^j$, the test case $\left(x_i^j, y_i^j\right)$ can be regarded as passing. Whether the error is successfully fixed can be described as $\wedge_{j=0}^{m}\left[a_{\theta_i^*}\left(x_i^j\right) = y_i^j\right]$, an aggregate result of all test cases. The FR that represents the test result on the bug instances are defined as:

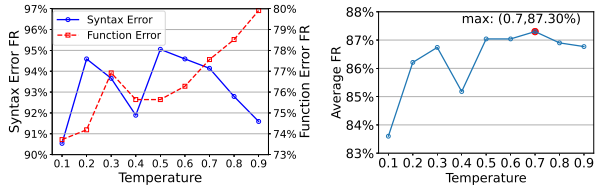
$$\mathbf{FR} = \sum_{i=0}^{n} \frac{\wedge_{j=0}^{m}\left[a_{\theta_i^*}\left(x_i^j\right) = y_i^j\right]}{n} \times 100\% \quad (1)$$

It is worth noting that all FR presented in this paper are calculated based on the average of 10 repeated experiments.

Execution time. This paper also considers the execution time of the framework as an important indicator of the performance, which is determined as the time elapsed between when the MEIC receives the initial design files and MEIC outputs the final modified code.

4.3 Results and Discussions

RQ1 (Sensitivity). Figure 11(a) illustrates the impact of temperature on the FR of syntax errors and function errors. The results



(a) FR of syntax and function errors. (b) Average FR.

Figure 11: FR against the temperature of the debug agent.

indicate that within the temperature range of the experiment, higher temperatures result in higher function error FR and lower syntax error FR. The function and syntax error FR reach their highest point of 80% and 95% respectively with temperature settings being 0.9 and 0.5 respectively. This may be attributed to the fact that syntax error is relatively straightforward, whereas function error is more complex. While attempting to rectify a syntax error, the agent with a higher temperature may inadvertently introduce alterations that result in the generation of new errors. Such errors were rarely corrected by the debug agent in the subsequent iterations according to our observation. In the case of function errors, the higher degree of randomness allowed LLM to to avoid modifying the same error all the time. Based on Figure 11(a), we calculated an average FR of all our test cases as shown in Figure 11(b). The overall FR reached the best case (87.30%) when the temperature was 0.7.

RQ2 (Effectiveness). Figure 12 shows the MEIC's FR for 8 syntax errors and 7 function errors across 15 common hardware modules. The FR was calculated based on Equation 1, and the values were colouring-coded for readability. The results suggest that the FR varied significantly depending on module complexity and error types. For instance, for modules with straightforward logic and shorter lengths such as the *adder_8bit* module, MEIC consistently achieved a high FR, indicating its effectiveness in correcting all error types. Conversely, for more intricate modules like *accu*, the FR diminished, highlighting the challenge of debugging such code. Regarding error

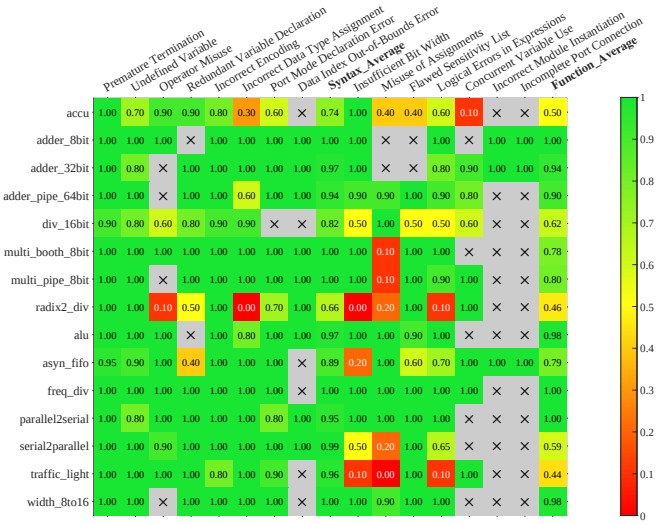


Figure 12: Heatmap result for FR. The symbol X represents an error that could not be imposed due to the limitations of the specific module structure. Syntax_Average and Function_Average represent the arithmetic mean of the FR for syntax errors and function errors, respectively.

types, while syntax errors exhibited a higher (10% higher) overall FR than function errors, the latter posed greater difficulty in correction, particularly in complex modules. On average, the MEIC achieved FR of 93% for syntax errors and 78% for function errors, demonstrating a greater effectiveness than existing practices. By contrast, the average FR achieved by RTLFixer [38] stood at 16%.

RQ3 (Impactability). Table 3 compares the FR of both syntax and function errors achieved by two LLM models (GPT-3.5 and GPT-4) in their standard forms, after incorporating external domain-specific knowledge and with the integration of MEIC. Results indicate that, for the same core GPT models, the models integrating MEIC achieved the highest FR of both syntax and function errors, followed by the models incorporating knowledge only. The standard form models exhibited the lowest FR.

Furthermore, when comparing the best performance (colouring-coded) across all LLMs and their variants, the models with MEIC accounted for 80% of the best results, whereas the models with knowledge and standard-form models only scored 16% and 4%, respectively. It is worth noting that standard models benefited significantly from adding knowledge for identifying and correcting syntax errors, but this was less effective for function errors. With the integration of MEIC, while improvements were observed in both syntax and function error debugging, the performance gap is more favourable for function errors, as GPT-4 with MEIC improved the FR to 78.30% from 66.24% for GPT-4 with knowledge, representing an improvement of over 12%. This result suggested that the integration of MEIC could indeed enhance the performance of standard models, surpassing those with incorporating knowledge.

RQ4 (Usability). According to Table 3, GPT-4 consistently outperformed GPT-3.5 across their standard forms. After incorporating external knowledge, the performance of GPT-3.5 still failed to meet the standard form of GPT-4. This demonstrated differences in the debugging capabilities of the models themselves. However, after integrating MEIC, GPT-3.5 achieved FR of 64% for syntax errors and 56% for function errors, the performance was comparable to or even exceeds the standard form of GPT-4, highlighting the framework's effectiveness in directing LLM models' debugging capability.

Table 4: Execution time of MEIC against human (s: syntax error; f: function error; "Total" is calculated in seconds).

Types	GPT-4+MEIC					Human	Speedup
	Simu.	Debug	Score	Trans.	Total	Total	
accu s	4.4%	88.0%	5.6%	2.1%	116.0	382	3.29x
adder_8bit s	8.0%	82.6%	6.0%	3.4%	30.6	136	4.44x
adder_32bit s	2.4%	93.4%	2.4%	1.9%	130.3	402	3.09x
adder_pipe_64bit s	1.8%	94.4%	2.5%	1.4%	169.4	575	3.39x
div_16bit s	6.9%	82.9%	8.5%	1.7%	63.6	249	3.91x
multi_booth_8bit s	9.5%	77.6%	9.0%	3.9%	23.6	272	**11.53x**
multi_pipe_8bit s	4.2%	90.1%	3.9%	1.8%	53.2	775	**14.56x**
radix2_div s	3.0%	89.1%	4.1%	3.8%	219.9	620	2.82x
alu s	3.4%	90.5%	3.5%	2.6%	80.6	318	3.95x
asyn_fifo s	2.2%	93.4%	3.0%	1.4%	146.6	827	5.64x
freq_div s	8.3%	81.3%	6.9%	3.5%	25.9	197	7.60x
parallel2serial s	8.2%	80.6%	8.6%	2.6%	32.9	239	7.26x
serial2parallel s	8.8%	79.2%	8.1%	3.8%	25.5	268	**10.49x**
traffic_light s	3.9%	77.9%	3.9%	14.4%	68.3	284	4.16x
width_8to16 s	8.8%	80.3%	7.5%	3.3%	24.3	232	9.56x
accu f	4.6%	87.5%	5.9%	2.0%	223.4	1578	7.06x
adder_8bit f	9.1%	81.9%	6.3%	2.7%	28.8	293	**10.17x**
adder_32bit f	2.1%	92.8%	2.4%	2.8%	168.7	871	5.16x
adder_pipe_64bit f	1.9%	94.1%	2.3%	1.7%	205.1	1814	8.84x
div_16bit f	5.3%	83.1%	7.6%	4.0%	145.3	1482	**10.20x**
multi_booth_8bit f	5.9%	83.2%	8.4%	2.5%	97.9	816	8.33x
multi_pipe_8bit f	3.0%	92.1%	3.6%	1.4%	204.1	915	4.48x
radix2_div f	2.9%	92.0%	4.3%	0.8%	427.5	1650	3.86x
alu f	3.7%	91.5%	3.6%	1.2%	85.8	939	**10.94x**
asyn_fifo f	1.7%	92.5%	3.1%	2.7%	331.7	1746	5.26x
freq_div f	8.6%	81.9%	6.8%	2.7%	31.8	1527	**48.00x**
parallel2serial f	11.9%	75.0%	9.4%	3.7%	22.0	677	**30.80x**
serial2parallel f	5.6%	77.7%	7.6%	9.1%	183.7	993	5.41x
traffic_light f	2.8%	84.6%	4.0%	8.6%	497.3	1869	3.76x
width_8to16 f	9.1%	79.6%	8.4%	2.9%	34.1	912	**26.74x**
Average	3.6%	88.4%	4.5%	3.5%	129.9	795	**6.12x**

RQ5 (Performability). To assess the proposed framework's effectiveness compared to human experts, we compared their debugging performance across various modules and error types as shown in Table 4. While human experts are experienced in debugging, the framework demonstrated competitive performance in addressing syntax errors and modules with simple logic. For example, in the *multi_pipe_8bit* module, MEIC had a 14.56x speedup. This performance gap was further increased for more complex function errors as MEIC demonstrated up to 48x speedup of the human expert. This result illustrated the significant enhancement in the debugging capabilities with greater automation and improved efficiency.

5 Related Work

Recent advances in LLMs have significantly transformed hardware design, primarily through enhanced efficiency and automation [3, 9, 23, 36]. A key application of these models is in RTL debugging, which represents a substantial portion of total design costs. In response, various approaches, such as RTLFixer [38], SBF [1], LLM4SecHW [13], HDLdebugger [46], and AssertLLM [11], have been developed to reduce costs and increase efficiency in this area.

These existing approaches have focused mainly on refining LLM models' performance by employing techniques like prompt engineering [30, 43], model tuning [4, 23], and model training [15, 22]. Although these efforts have led to some improvements, they have not yet successfully addressed applications to correcting function errors [37] nor achieved sufficient performance as measured by the *pass@k* rates [38]. In contrast, our approach adopts a collaborative process, by utilising two LLM models iteratively, to enhance debugging effectiveness for syntax and function errors.

6 Conclusion

In this work, a systematical automated debugging framework, **MEIC**, is introduced. The framework demonstrates that it is feasible to employ the LLMs for the purpose of debugging Verilog code, encompassing both syntax and function errors. The utilisation of prompt engineering and feedback engineering leads to an improvement in the debug capability of the LLMs, achieving fix rate of 93% for syntax errors and 78% for function errors. In comparison to human engineers, debugging with our framework has the potential to save up to 48 times the time overhead. Our work not only rethinks the Verilog code debugging process with the LLMs, but also paves the way for more efficient hardware design.

Lessons we learnt. Throughout this study, we observed considerable variations in performance of the different LLMs when it comes to debugging RTL code. In line with findings from existing literature, it is clear that no single model can effectively manage all debugging scenarios. In addition, despite prompt engineering, model tuning, and model training bringing overall improvement to the model performance, decreased performance was observed in certain tests compared to the models in their standard forms. This observation highlights the need for setting up realistic expectations before LLM deployment and for understanding their operational limits, both of which remain an open challenge.

References

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215* (2023).

[2] Xavier Amatriain. 2024. MEASURING AND MITIGATING HALLUCINATIONS IN LARGE LANGUAGE MODELS: AMULTIFACETED APPROACH. (2024).

[3] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. *arXiv preprint arXiv:2305.13243* (2023).

[4] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019* (2023).

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[6] Yuyan Chen, Qiang Fu, Yichen Yuan, Zhihao Wen, Ge Fan, Dayiheng Liu, Dongmei Zhang, Zhixu Li, and Yanghua Xiao. 2023. Hallucination detection: Robustly discerning reliable answers in large language models. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 245–255.

[7] Shivakumar S Chonnad and Needamangalam B Balachander. 2007. *Verilog: Frequently Asked Questions: Language, Applications and Extensions*. Springer Science & Business Media.

[8] Clayton Cohn, Nicole Hutchins, Tuan Le, and Gautam Biswas. 2024. A Chain-of-Thought Prompting Approach with LLMs for Evaluating Students' Formative Assessment Responses in Science. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 23182–23190.

[9] Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, Shailja Thakur, Ramesh Karri, Siddharth Garg, and Jeyavijayan Rajendran. 2024. Make Every Move Count: LLM-based High-Quality RTL Code Generation Using MCTS. *arXiv preprint arXiv:2402.03289* (2024).

[10] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-based Test-driven Interactive Code Generation: User Study and Empirical Evaluation. arXiv:2404.10100 [cs.SE]

[11] Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. 2024. AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs. *arXiv preprint arXiv:2402.00386* (2024).

[12] Shangbin Feng, Weijia Shi, Yike Wang, Wenxuan Ding, Vidhisha Balachandran, and Yulia Tsvetkov. 2024. Don't Hallucinate, Abstain: Identifying LLM Knowledge Gaps via Multi-LLM Collaboration. *arXiv preprint arXiv:2402.00367* (2024).

[13] Weimin Fu, Kaichen Yang, Raj Gautam Dutta, Xiaolong Guo, and Gang Qu. 2023. LLM4SecHW: Leveraging domain-specific large language model for hardware debugging. In *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 1–6.

[14] Boris A Galitsky. 2023. Truth-o-meter: Collaborating with llm in fighting its hallucinations. (2023).

[15] Emil Goh, Maoyang Xiang, I Wey, T Hui Teo, et al. 2024. From English to ASIC: Hardware Implementation with Large Language Model. *arXiv preprint arXiv:2403.07039* (2024).

[16] Muhammad Hassan, Sallar Ahmadi-Pour, Khushboo Qayyum, Chandan Kumar Jha, and Rolf Drechsler. [n. d.]. LLM-guided Formal Verification Coupled with Mutation Testing. ([n. d.]).

[17] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards mitigating LLM hallucination via self reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 1827–1843.

[18] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2023. Self-planning Code Generation with Large Language Models. arXiv:2303.06689 [cs.SE]

[19] Zhe Jiang, Shuai Zhao, Ran Wei, Dawei Yang, Richard Paterson, Nan Guan, Yan Zhuang, and Neil C Audsley. 2021. Bridging the pragmatic gaps for mixed-criticality systems in the automotive industry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 4 (2021), 1116–1129.

[20] Kevin Laeufer, Brandon Fajardo, Abhik Ahuja, Vighnesh Iyer, Borivoje Nikolić, and Koushik Sen. 2024. RTL-Repair: Fast Symbolic Repair of Hardware Design Code. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 867–881.

[21] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 898–911.

[22] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2023. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176* (2023).

[23] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.

[24] Raoni Lourenço, Juliana Freire, Eric Simon, Gabriel Weber, and Dennis Shasha. 2023. BugDoc: Iterative debugging and explanation of pipeline. *The VLDB Journal* 32, 1 (2023), 75–101.

[25] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2023. RTLLM: An open-source benchmark for design rtl generation with large language model. *arXiv preprint arXiv:2308.05345* (2023).

[26] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[27] OpenAI. 2024. ChatGPT - Fine-Tuning. https://platform.openai.com/docs/guides/fine-tuning

[28] Soumen Pal, Manojit Bhattacharya, Sang-Soo Lee, and Chiranjib Chakraborty. 2024. A domain-specific next-generation large language model (LLM) or ChatGPT is required for biomedical engineering and research. *Annals of Biomedical Engineering* 52, 3 (2024), 451–454.

[29] S Ramachandran. 2007. RTL Coding Guidelines. *Digital VLSI Systems Design: A Design Manual for Implementation of Projects on FPGAs and ASICs Using Verilog* (2007), 187–214.

[30] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. *arXiv preprint arXiv:2402.07927* (2024).

[31] SEIMENS. 2024. ModelSim. https://eda.sw.siemens.com/en-US/ic/modelsim/

[32] Priya Srikumar. 2023. Fast and wrong: The case for formally specifying hardware with LLMS. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM. ACM Press.*

[33] Stuart Sutherland. 2000. The IEEE Verilog 1364-2001 Standard What's New, and Why You Need It. In *9th Internatioinal HDL Conference (HDLCon)*.

[34] Stuart Sutherland and Don Mills. 2010. *Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them*. Springer Science & Business Media.

[35] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[36] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2023. Verigen: A large language model for verilog code generation. *arXiv preprint arXiv:2308.00708* (2023).

[37] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621* (2024).

[38] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2023. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models. *arXiv preprint arXiv:2311.16543* (2023).

[39] Lily Jiaxin Wan, Yingbing Huang, Yuhong Li, Hanchen Ye, Jinghua Wang, Xiaofan Zhang, and Deming Chen. 2024. Software/Hardware Co-design for LLM and Its Application for Design Verification. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 435–441.

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

[41] Ran Wei, Zhe Jiang, Xiaoran Guo, Haitao Mei, Athanasios Zolotas, and Tim Kelly. 2022. Designing critical systems with iterative automated safety analysis. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 181–186.

[42] Ran Wei, Zhe Jiang, Xiaoran Guo, Ruizhe Yang, Haitao Mei, Athanasios Zolotas, and Tim Kelly. 2023. DECISIVE: Designing Critical Systems With Iterative Automated Safety Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023).

[43] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[44] Henry Wong. 2019. HDLBits - Practice FPGA Problems. https://hdlbits.01xz.net/wiki/Main_Page

[45] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI conference on human factors in computing systems*. 1–22.

[46] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024. HDLdebugger: Streamlining HDL debugging with Large Language Models. *arXiv preprint arXiv:2403.11671* (2024).

[47] JD Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny can't prompt: how non-AI experts try (and fail) to design LLM prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–21.

[48] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the ai ocean: A survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).