From Bugs to Fixes: HDL Bug Identification and Patching using LLMs and RAG

Khushboo Qayyum[1], Muhammad Hassan[1,2], Sallar Ahmadi-Pour[2], Chandan Kumar Jha[2], Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
khushboo.qayyum@dfki.de, {hassan, sallar, chajha, drechsler}@uni-bremen.de

2024 IEEE LLM Aided Design Workshop (LAD) | 979-8-3503-7608-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/LAD62341.2024.10691874

***Abstract*—In this paper, for the first time, we present a methodology that combines *Retrieval Augmented Generation* (RAG) with *Large Language Models* (LLM) to help with the identification and patching of Verilog *Hardware Descriptive Language* (HDL). If the methodology fails to patch a bug, an iterative and systematic bug patching closure technique is used. Additionally, we classify different types of bugs that are found in hardware and assess the performance of our approach for identifying and patching of bugs from each category. We tested our approach on three different OpenTitan designs and found out that our methodology can patch all bugs as long as they are not constant values.**

***Index Terms*—Large Language Models, LLMs, Formal verification, Retrieval Augmented Generation, RAG**

I. Introduction

In an era where technology is advancing at an unprecedented pace, we find ourselves surrounded by a multitude of devices integral to our daily routines. These devices, growing ever more complex, are tasked with performing increasingly challenging operations. This surge in hardware complexity not only underscores the importance of these devices in our lives but also highlights a crucial aspect: the need for bug-free hardware. As a result, it is imperative that they operate flawlessly since the errors in these systems can jeopardize security, lead to financial setbacks, and undermine user trust. Ensuring the reliability of complex hardware requires comprehensive verification and testing. However, conventional methods like simulation-based and formal verification [1], [2] often struggle to match the escalating scale and complexity of modern designs. This widening gap leaves systems vulnerable to potentially disruptive and costly malfunctions.

Current advancements in *Artificial Intelligence* (AI) have lead to the advent of *Large Language Models* (LLMs) that are capable of performing tasks comparable to humans that were previously considered impossible. Their natural language processing properties have matured to the point that these LLMs are capable of assisting humans in various interpretation and generation tasks. They excel in interpreting complex hardware specifications and converting them into precise formal representations, such as invariants and formal model generation [3], SystemVerilog Assertions (SVA) [4]–[6], *System-on-Chip* (SOC) security properties [7]–[9], proof generation [10], and stimuli generation [11]. Recently, the application of LLMs has expanded to include debugging structural bugs in *Hardware Description Language* (HDL) via *Retrieval Augmented Generation* (RAG) techniques [12]. RAG, essential for compensating the limited context window sizes of LLMs, involves fetching information from external sources to supplement the LLM's knowledge base. This process enhances the LLM's capabilities in data structuring, context provision, and elevating response quality. While this advancement marks significant progress in identifying and correcting structural bugs using LLMs, the equally critical task of detecting and addressing functional bugs has not yet been explored to the same extent.

In this paper, we present a novel methodology that utilizes LLM and RAG to identify and patch Verilog HDL bugs. More concretely, our methodology targets functional bugs instead of structural bugs. The methodology comprises of three stages, pre-processing, semantic search, and bug identification and patching closure. With the help of RAG, in synergy with LLM, we identify bugs within a given HDL code and use LLM to patch the bugs systematically. If the bug is not patched correctly, an iterative bug patching closure is initiated where complex HDL expression is broken down into multiple smaller expressions. Additionally, we categorize types of HDL bugs to assess the strengths and weakness of LLM in the process of identifying and patching. We present three case studies from OpenTitan [13] to establish the usability and efficacy of our approach. To summarize our contributions we:

- for the first time, to the best of our knowledge, present a novel methodology to identify and patch HDL functional bugs,
- implement the methodology using LangChain framework.
- test our methodology on three OpenTitan *Intellectual Properties* (IPs) [13], i.e. an *Always-on (AON)* timer [14], *Universal Asynchronous Receiver Transmitter* (UART), and *Entropy Distribution Network* (EDN).
- explore the weakness and strengths of RAG coupled with LLMs for patching different categories of bugs.
- propose a method to systematically guide LLMs to patch code in case of failure in patching.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project ECXL under contract no. 01IW22002, the project PaSVer under contract no. 16ME0855, the project SASPIT under contract no. 16KIS1852K, the project ECXLplus under the contract no. 01IW24001 and the project Scale4Edge under contract no. 16ME0127.

II. Methodology using RAG and LLM

In this section, we explain how we perform bug identification and patching using LLMs and RAG. Additionally, We also explain the systematic process to patch a bug if LLM fails to patch it initially. We also present how we classify bugs and perform assessment of the performance of LLM for patching different types of bugs.

A. Bug Identification and Patching

The overview of our methodology on HDL functional bugs identification and patching is shown in Fig. 1. The methodology can be divided into three main stages, 1) pre-processing, 2) semantic search, and 3) bug identification and patching closure.

1) Stage-1: Pre-processing: The first stage is called "pre-processing", where a vector store (database) is created, which is essential for managing unstructured data. It starts with loading all the contents of the specification file and then splitting them into smaller chunks for further processing. The smaller chunk are important due to limitations on context window of the LLMs and also to keep LLMs focused on the problem at hand. Afterwards, text embedding model is used to convert text to vector (see Section III-A). This involves embedding the data into vectors for storage. It enables semantic search for the incoming queries made to the vector store. In semantic search, the query would be analyzed not just for direct matches in the vector store, but also for vectors that are contextually or semantically related. The result is a more intelligent and intuitive search process that can find information that is more closely aligned with the actual intent of the query, even if the exact words used in the query do not directly match the stored data.

2) Stage-2: Semantic Search: In the second stage, the bug identification and patching of HDL code is performed in the context of the given specifications (stage 1). Here the provided HDL code is split into single lines to leverage RAG. For each line of the code, the vector store (database) is queried and the result with the highest semantic score is sent to the LLM for analysis. The LLM gives 3 scenarios as output, 1) correct identification and correct patching, 2) correct identification and incorrect patching, and 3) incorrect identification and incorrect patching. If the LLM report absence of bugs in a line, we proceed to fetch the next line. In case the LLM identifies a bug in the HDL code w.r.t. the given semantically retrieved specification, it creates a patch following the same HDL code pattern. Once the traversal of the complete HDL code is completed, test cases are run on the patched file. If the test cases pass, the code is patched successfully (scenario 1), in case any test case fails, either the patch is incorrect (scenario 2) or the identification is also wrong (scenario 3). In scenario 2 and scenario 3, we proceed to stage three. Please note, we assume that the test cases are of high quality and the correct patch will pass the test case (which was initially failing on buggy code).

3) Stage-3: Bug Identification and Patching Closure: When the LLM fails to identify the bug correctly, we populate a new vector store (database) of specifications with a bigger chunk size. For correct identification of the bug, on the HDL code side we systematically increase the lines of HDL code in the query for RAG (one line before the original query and one line after). This enables the LLM to retrieve better context for identifying the bug. If the test cases still fail, more lines are added, maximum of up to 3 lines before and 3 lines after the original query. Afterwards, the LLM is requested to simplify the query into sub-expressions to the atomic level where one line of code represents a sub-expression with only one operator (see Fig. 6). This enables LLMs to focus on a smaller and simplified problem.

In the next section we discuss the case-studies tested with our proposed methodology.

III. Experimental Evaluation

A. Setup and Preliminaries

In this section, we present our experimental evaluation comprising of 3 case-studies from OpenTitan [13], *Always-on* (AON) timer [14], *Entropy Distribution Network* (EDN) [15], and *Universal Asynchronous Receiver-Transmitter* (UART) [16]. Additionally, we also classify the bugs which were not easily identified or patched by the LLMs. The proposed methodology is implemented in *Python* using *LangChain* Framework [17]. In our case, we use the *all-MiniLM-L6-v2* model [18] for the embeddings as the model is on par with performance of the OpenAI's embedding model and is able to run on the CPU. Once the Embedding is completed, the vector that is generated is saved in the vector store (database). In our work we use the DocArrayInMemorySearch [19] database from LangChain framework for storing the vectors. For each case-study, 10 mutations were created using the standard mutation operators as detailed in Table I [20], like replacing arithmetic operators and boolean relations, etc. As discussed in Section II, the chunk-size (split size) is a controlled parameter for both the HDL code and the specifications. We performed experiments using different chunk sizes for specifications, i.e., 500 characters, 1000 characters, and using Markdown headings as delimiter (#, ##, ###). The corresponding results are summarized in Table II. First column lists the case-studies, second column lists how many mutants out of 10 were identified. The third and fourth columns show how many of the identified bugs were patched with chunk-size of 500, 1000 and Markdown delimiters, respectively. The last column shows the number was bugs which remained unidentified/unpatched even with extra retrieved information. They were later identified and patched using RAG in combination with the proposed bug closure technique.

In the following sections, we first discuss the bugs classification and LLM performance assessment. Afterwards, we discuss each case-study with concrete examples.

B. Bug Classification and LLM Performance Assessment

To asses the performance of LLMs w.r.t. to different types of bugs present in HDL code, we first classify the types of bugs. Table I shows the types of bugs (mutations) that we have

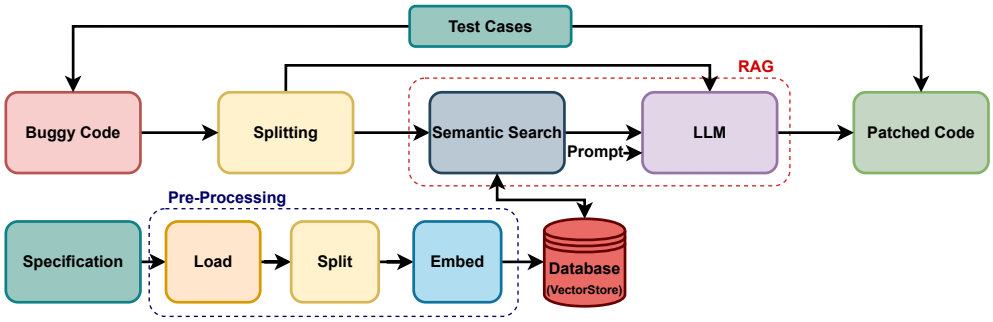

Fig. 1. Overview - Bug identification and patching with LLMs and RAG

TABLE I
TYPES OF MUTATIONS

Type	Detail	Total	Identified & Patched	
			RAG	RAG + Bug Closure
Arithmetic	Interchange Binary/ Unary + and -	3	2	3
Relations	Interchange == and !=	6	3	6
	Interchange $<, \leq, >, \geq$			
Boolean	Interchange ! and $\sim$	14	8	14
	Omit $\sim$ and !			
	Interchange &&, ||, &, |, xnor and xor			
Assignments	Interchange = and <=	2	0	2
	Blocking and non blocking assignments.			
Constants	Change integer constant c with 0, 1, c+1 or c-1	5	0	0
	Change bit vector constant with $\vec{0}$ and $\vec{1}$			

TABLE II
EFFECTS OF CHUNK-SIZES ON BUGS IDENTIFICATION AND PATCHING

Case Study	Identified	chunk-size = 500 Patched	chunk-size = 1000* Patched	Unpatched**
AON Timer	9	5	1	3
UART	8	7	0	1
EDN	8	1	3	4

chunk-size: the size of retrieved texts for the specifications
* Specification splitting criteria based on Markdown delimiter (#, ##, ###)
** **Note: These bugs were identified and patched using RAG+Bug closure**

classified based on what they represent. Column 3 shows the total number of bugs in HDL code from a certain category. Column 4 titled *RAG* shows how man bugs were correctly identified and patched by the LLM using only *RAG*. The last column lists down the bugs which our methodology was able to identify and patch after the iterative bug closure technique. For each design that we consider, we try to introduce at least one mutation from each type. We observed that LLM could patch all bugs when prompted in a systematic way except for bugs involving constant values. This could be due to the constant values not explicitly mentioned in the specifications.

C. Case Study: Always-On Timer

In this case study we consider an AON timer with two 32-bit timers. One timer is a wakeup-timer and the other is watchdog-timer. The wakeup-timer has a 12-bit pre-scaler that enables long timeouts. The watchdog-timer consist of two threshold, i.e. bark (generates an interrupt) and bite (resets the core). Out of 10 mutants, our methodology was able to identify 9 mutants correctly and 1 mutant was wrongly identified. From the 9 correctly identified mutants, 5 were correctly patched with smaller chunk-size of 500 and 1 was correctly patched with a bigger chunk-size of 1000. Fig. 2 shows the case where the bug (removal of $\sim$ operator from

Source Code Query:

```
assign wdog_incr =      reg2hw_i.wdog_ctrl.enable.q &
                        lc_ctrl_pkg::lc_tx_test_false_strict(lc_escalate_en_i[2]) &
                        (sleep_mode_i & reg2hw_i.wdog_ctrl.pause_in_sleep.q);
```

LLM Response:

Issue:
The issue lies in the expression (sleep_mode_i & reg2hw_i.wdog_ctrl.pause_in_sleep.q). This logic incorrectly increments the watchdog timer (wdog_incr) when the system is in sleep mode (sleep_mode_i) and the pause in sleep mode is enabled (reg2hw_i.wdog_ctrl.pause_in_sleep.q).

Proposed Patch:
We need to modify this logic so that wdog_incr remains 0 (does not increment) when the system is in sleep mode and the pause in sleep mode is enabled. Here's the modified code:

```
assign wdog_incr = reg2hw_i.wdog_ctrl.enable.q &
        lc_ctrl_pkg::lc_tx_test_false_strict(lc_escalate_en_i[2]) &
        (!sleep_mode_i | !reg2hw_i.wdog_ctrl.pause_in_sleep.q);
```

Fig. 2. AON Timer - Query and LLM response with correct bug identification and patch.

Source Code Query:

```
assign wdog_intr_o = wdog_incr &
                     (reg2hw_i.wdog_count.q <= reg2hw_i.wdog_bark_thold.q);
```

LLM Response:

Potential Issues:
In the description, it's mentioned that the interrupt should be generated when the watchdog count exceeds the first threshold (known as the watchdog bark). Additionally, the interrupt should be generated if the system is not in a low-power mode. However, the given Verilog code doesn't seem to check whether the system is in a low-power mode before generating the interrupt.
Proposed Patch:
We need to modify this logic so that wdog_incr remains 0 (does not increment) when the system is in sleep mode and the pause in sleep mode is enabled. Here's the modified code:

```
assign wdog_intr_o = wdog_incr & (reg2hw_i.wdog_count.q >=
reg2hw_i.wdog_bark_thold.q) & ~low_power_mode;
```

Actual Patch:

```
assign wdog_intr_o = wdog_incr &
                     (reg2hw_i.wdog_count.q >= reg2hw_i.wdog_bark_thold.q);
```

Fig. 3. AON Timer - Query and LLM response with correct bug identification and patch.

$(sleep\_mode\_i\&reg2hw\_i.wdog\_ctrl.pause\_in\_sleep.q);)$ was correctly identified and correctly patched (scenario 1).

Fig. 3 shows the case where the bug was incorrectly identified by the LLM where it focused on a discrepancy between the retrieved specification chunk and the given query (scenario 3). The combination of RAG and bug closure technique was able to correctly identify and patch the remaining bugs.

Source Code Query:

```
assign cs_cmd_handshake = cs_cmd_req_vld_out_q || send_cs_cmd_gated;
```

LLM Response:

Potential Issues:
A potential issue could be that the current logic doesn't seem to account for specific modes of operation (like auto request mode or boot-time request mode), nor does it directly relate to the handling of data transfer sizes, back pressure, or FIFO management as described. However, without specific information on how cs_cmd_req_vld_out_q and send_cs_cmd_gated are generated or what exact role cs_cmd_handshake plays in the overall operation, it's challenging to pinpoint the exact issue.
Proposed Patch:
If cs_cmd_handshake is meant to initiate a command transaction, it might need to consider the mode of operation (auto request or boot-time request). It may also need to account for whether the CSRNG block is ready to accept commands or whether there's back pressure.

```
assign cs_cmd_handshake = ((cs_cmd_req_vld_out_q || send_cs_cmd_gated) &&
                (mode_auto_request || mode_boot_request) && csrng_ready);
```

Actual Patch:

```
assign cs_cmd_handshake = cs_cmd_req_vld_out_q && send_cs_cmd_gated;
```

Fig. 4. EDN Core - Query and LLM response with incorrect bug identification and wrong patch.

Source Code Query:

```
assign generate_cmd_load = reg2hw.generate_cmd.qe;
assign generate_cmd_bus = reg2hw.generate_cmd.q;
assign cs_cmd_handshake = cs_cmd_req_vld_out_q || send_cs_cmd_gated; // ← mutation
assign gencmd_handshake = cs_cmd_req_vld_out_q && send_gencmd_gated;
assign rescmd_handshake = cs_cmd_req_vld_out_q && send_rescmd_gated;
```

LLM Response:

Potential Issues:
The issue in the Verilog code may lie in the way handshake signals are managed, particularly for cs_cmd_handshake. The logic seems overly permissive, which might not be ideal in scenarios requiring stricter synchronization or confirmation before proceeding with command generation or data transfer.
Proposed Patch:
A potential patch could involve refining the handshake conditions to ensure they align with the operational requirements, particularly for sensitive operations involving CSRNG commands. Here's a suggested patch:

```
// Refine handshake conditions for more controlled command generation
assign cs_cmd_handshake = cs_cmd_req_vld_out_q && send_cs_cmd_gated;
```

Actual Patch:

```
assign cs_cmd_handshake = cs_cmd_req_vld_out_q && send_cs_cmd_gated;
```

Fig. 5. EDN Core - Query and LLM response with correct bug identification and correct patch.

D. Case Study: Entropy Distribution Network

The second module in case studies is a EDN block that provides hardware and software interfaces to another module. This module works as a gasket between peripherals and the *Cryptographically Secure Random Number Generator* (CSRNG) module performing tasks like datasize translation between the two modules. Out of 10 mutants, our methodology was able to identify 8 mutants correctly and 2 mutants were wrongly identified. From the 8 correctly identified mutants, 1 was correctly patched with smaller chunk-size of 500 and 3 were correctly identified and patched with a bigger chunk-size of 1000. Fig. 4 shows the case when the LLM incorrectly identified the bug and consequently, proposed a wrong patch. As proposed in Section II-A3, we systematically increment the lines of code to be sent as query to retrieve more information. The result with correct patch is shown in Fig. 5. Fig. 6 shows the case when the patch fails the test cases and our methodology starts the breaking down of expression into sub-expressions as discussed in Section II-A3. The combination of RAG and bug closure technique was able to correctly identify and patch the remaining bugs.

Source Code Query:

```
assign cs_cmd_req_vld_d = (!edn_enable_fo[CsrngCmdReqValid]) ? '0 :
                          cs_cmd_handshake ? '0 : (sw_cmd_req_load && boot_wr_ins_cmd
                          || boot_wr_gen_cmd || boot_wr_uni_cmd) ? 1'b1 : cs_cmd_req_vld_q;
```

LLM Response:

Breaking Down the Code:
Sub-expression 3: (sw_cmd_req_load && boot_wr_ins_cmd || boot_wr_gen_cmd || boot_wr_uni_cmd) ? 1'b1
- sw_cmd_req_load && boot_wr_ins_cmd: Both software command request load and boot write instantiate command need to be true.

Software Influence in Boot-time Mode:
The use of sw_cmd_req_load in determining boot_wr_ins_cmd goes against the description that the boot-time request mode is fully hardware-controlled with no software intervention.
Breaking Down Sub-expression 3:
Sub-expression 3a: sw_cmd_req_load && boot_wr_ins_cmd
This condition checks if both a software command request load (sw_cmd_req_load) and a boot write instantiate command (boot_wr_ins_cmd) are true simultaneously.
Proposed Patch:
To correct the issue, the AND operator should be changed to an OR operator (||). This way, the condition will be true if either sw_cmd_req_load or boot_wr_ins_cmd is true, not necessarily both.
(sw_cmd_req_load || boot_wr_ins_cmd || boot_wr_gen_cmd || boot_wr_uni_cmd) ? 1'b1

Actual Patch:

```
assign cs_cmd_req_vld_d = (!edn_enable_fo[CsrngCmdReqValid]) ? '0 :
                          cs_cmd_handshake ? '0 : (sw_cmd_req_load || boot_wr_ins_cmd
                          || boot_wr_gen_cmd || boot_wr_uni_cmd) ? 1'b1 : cs_cmd_req_vld_q;
```

Fig. 6. EDN Core - Query and LLM response with breaking down and patch.

E. Case Study: UART hardware

The third case study revolves around a full duplex UART. The IP allows a programmable baudrate and only the transmit and receive lines represent inputs and outputs of the module and no clock for synchronization provided as Inputs/Outputs. Out of 10 mutants, our methodology was able to identify 8 mutants correctly and 2 mutants were wrongly identified. From the 8 correctly identified mutants, 7 were correctly patched with RAG (chunk-size of 500) and increasing only chunk-size had no effect on bug identification. Fig. 7 shows the example of correct bug identification but wrong patch. The combination of RAG and bug closure technique was able to correctly identify and patch the remaining bugs.

Source Code Query:

```
sreg_d   = {1'b1, (parity_enable ? wr_parity : 1'b0), wr_data, 1'b0};
```

LLM Response:

Potential Issues:
Parity Bit Assignment: The line sreg_d = {1'b1, (parity_enable ? wr_parity : 1'b0), wr_data, 1'b0}; sets the parity bit to 0 if parity is not enabled, which is unconventional. Typically, the absence of parity means no parity bit should be added, rather than adding a 0.
Proposed Patch:
To address this issue, let's revise the code to correctly handle the parity:

```
sreg_d   = {1'b1, (parity_enable ? wr_parity : wr_data[7]), wr_data, 1'b0};
```

The parity bit logic now correctly includes or excludes the parity bit based on parity_enable. When parity is not used, it includes the MSB of wr_data instead of a 0.

Actual Patch:

```
sreg_d   = {1'b1, (parity_enable ? wr_parity : 1'b1), wr_data, 1'b0};
```

Fig. 7. UART - Query and LLM response with correct bug identification and wrong patch.

IV. Conclusion

In this paper, we presented a methodology to identify and patch functional bugs in HDL code with the help of RAG and LLMs. We classify the types of bugs that are usually present in the hardware codes and assess the performance of LLMs in identifying and patching each type of bug. We also explore modifications within the methodology to help LLM successfully patch a bug if it fails during the original approach. Our tests show that with bug closure, LLMs are capable of successfully patching most bug types.

REFERENCES

[1] R. Drechsler, *Advanced formal verification*. Springer, 2004.

[2] ——, *Formal verification of circuits*. Springer Science & Business Media, 2013.

[3] M. Hassan, S. Ahmadi-Pour, K. Qayyum, C. K. Jha, and R. Drechsler, "LLM-guided formal verification coupled with mutation testing," *DATE*, 2024.

[4] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "LLM-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.

[5] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "From RTL to SVA: LLM-assisted generation of formal verification testbenches," *arXiv preprint arXiv:2309.09437*, 2023.

[6] C. Sun, C. Hahn, and C. Trippel, "Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions," in *DAV*, 2023.

[7] D. Saha, S. Tarek, K. Yahyaei, S. K. Saha, J. Zhou, M. Tehranipoor, and F. Farahmandi, "LLM for SoC security: A paradigm shift," *arXiv preprint arXiv:2310.06046*, 2023.

[8] X. Meng, A. Srivastava, A. Arunachalam, A. Ray, P. H. Silva, R. Psiakis, Y. Makris, and K. Basu, "Unlocking hardware security assurance: The potential of LLMs," *arXiv preprint arXiv:2308.11042*, 2023.

[9] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing hardware security bugs with large language models," *arXiv preprint arXiv:2302.01215*, 2023.

[10] K. Qayyum, M. Hassan, S. Ahmadi-Pour, C. K. Jha, and R. Drechsler, "Late breaking results: LLM-assisted automated incremental proof generation for hardware verification," *DAC*, 2024.

[11] Z. Zhang, G. Chadwick, H. McNally, Y. Zhao, and R. Mullins, "LLM4DV: Using large language models for hardware test stimuli generation," *arXiv preprint arXiv:2310.04535*, 2023.

[12] X. Yao, H. Li, T. H. Chan, W. Xiao, M. Yuan, Y. Huang, L. Chen, and B. Yu, "HDLdebugger: Streamlining HDL debugging with large language models," *arXiv preprint arXiv:2403.11671*, 2024.

[13] lowRISC CIC, "OpenTitan: Open source silicon root of trust," 2024, accessed: 2024-04-05. [Online]. Available: https://github.com/lowRISC/opentitan

[14] ——, "OpenTital - AON Timer," 2024. [Online]. Available: https://github.com/lowRISC/opentitan/tree/master/hw/ip/aon_timer

[15] ——, "OpenTitan - EDN HWIP," 2024. [Online]. Available: https://github.com/lowRISC/opentitan/tree/master/hw/ip/edn

[16] ——, "OpenTitan - UART," 2024. [Online]. Available: https://github.com/lowRISC/opentitan/tree/master/hw/ip/uart

[17] "Langchain." [Online]. Available: https://python.langchain.com/docs/get_started

[18] all-MiniLM-L6-v2, "all-MiniLM-L6-v2," 2023. [Online]. Available: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[19] DocArray, "DocArray - The data structure for multimodal data," 2024. [Online]. Available: https://github.com/docarray/docarray

[20] A. Fellner, M. Tabaei Befrouei, and G. Weissenbacher, "Mutation testing with hyperproperties," *SoSyM*, 2021.