HDLdebugger: Streamlining HDL debugging with Large Language Models

Xufeng Yao*
CUHK & Huawei
Hong Kong SAR, China
xfyao@cse.cuhk.edu.hk

Haoyang Li*
Huawei
Hong Kong SAR, China
li.haoyang1@huawei.com

Tsz Ho Chan*
Huawei
Hong Kong SAR, China
chantszho1@huawei.com

Wenyi Xiao
Huawei
Hong Kong SAR, China
wxiaoae@cse.ust.hk

Mingxuan Yuan
Huawei
Hong Kong SAR, China
Yuan.Mingxuan@huawei.com

Yu Huang
HiSilicon
ShenZhen, China
huangyu61@hisilicon.com

Lei Chen†
HKUST & HKSUT (GZ)
Hong Kong SAR, China
leichen@cse.ust.hk

Bei Yu†
CUHK
Hong Kong SAR, China
byu@cse.cuhk.edu.hk

ABSTRACT

In the domain of chip design, Hardware Description Languages (HDLs) play a pivotal role. However, due to the complex syntax of HDLs and the limited availability of online resources, debugging HDL codes remains a difficult and time-intensive task, even for seasoned engineers. Consequently, there is a pressing need to develop automated HDL code debugging models, which can alleviate the burden on hardware engineers. Despite the strong capabilities of Large Language Models (LLMs) in generating, completing, and debugging software code, their utilization in the specialized field of HDL debugging has been limited and, to date, has not yielded satisfactory results. In this paper, we propose an LLM-assisted HDL debugging framework, namely HDLdebugger, which consists of HDL debugging data generation via a reverse engineering approach, a search engine for retrieval-augmented generation, and a retrieval-augmented LLM fine-tuning approach. Through the integration of these components, HDLdebugger can automate and streamline HDL debugging for chip design. Our comprehensive experiments, conducted on an HDL code dataset sourced from Huawei, reveal that HDLdebugger outperforms 13 cutting-edge LLM baselines, displaying exceptional effectiveness in HDL code debugging.

CCS CONCEPTS

• **Theory of computation** → **Program semantics**; • **Computing methodologies** → **Natural language processing**; • **Hardware** → **Hardware description languages and compilation**.

KEYWORDS

Code Debugging, Large Language Model, Retrieval Augmented Generation

1 INTRODUCTION

Hardware Description Languages (HDLs) are crucial in the realm of chip design, serving as the cornerstone for creating, testing, and implementing digital systems [9, 13, 48]. Due to their critical role, the domain of HDL debugging has received comparatively scant attention. Traditional debugging approaches primarily involve manual code correction based on syntactic guidelines, followed by iterative testing through compilers. This process, while straightforward for languages such as Python and Java, becomes markedly more complex for HDLs due to their sophisticated syntax and the scarcity of accessible resources online. Furthermore, compiler-based testing of HDL code, especially in the context of chip development, is exceptionally time-consuming and resource-intensive.

Despite the high demand in the industry for effective HDL debugging techniques and the promising directions they offer, existing methodologies often fall short in addressing the complexities of the problem. For example, the template-based method [15, 17, 20, 25], a traditional strategy in code debugging, utilizes expert-defined code patterns or heuristics to identify and correct errors. However, this approach is inherently limited, capable of rectifying only those errors with predefined patterns. Consequently, it lacks the flexibility and adaptability necessary to tackle a diverse array of bugs.

Recently, researchers have delved into the direct application of large language models (LLMs) to rectify buggy code. The underlying hypothesis is that LLMs, pre-trained on extensive repositories of open-source code snippets and text, such as Python, can effectively discern bug patterns and automatically repair buggy code. To assess the efficacy of current LLM-based approaches in addressing industry-level HDL debugging challenges, we conduct a pilot experiment on three typical methods as shown in Table 1. Among the methods evaluated, GPT4 [1] is the current state-of-art LLM, RTLFixer [35] leverages retrieval augmented generation (RAG) [12] and advanced prompt engineering [46], specifically tailored for HDL debugging tasks. VeriGen [34] is a hardware large language model trained by a self-contained hardware dataset. Despite these advanced approaches, our observations indicate that none of the

Table 1: Pilot Debugging Experiments

Method	GPT4 [1]	RTLFixer [35]	VeriGen [34]
Pass-rate@1	6.35%	28.35%	1.34%

Table 2: Important notations.

Notation	Description
b_i	Buggy code
e_j	Error id
m_i	Error messages for b_i
c_i	Correct code for b_i
(d_j, r_j, s_j)	Descriptions d_j, reasons r_j, and potential solutions s_j for e_j
D_e, D_c	error database, code database
$\mathbf{z}_i^w$	Keyword vector for buggy code b_i with error message m_i
$sim(I_i, I_j)$	Similarity between buggy codes I_i and I_j
$q = (b, e)$	Code query consisting of buggy code b and error message m
rag_i^d	The document RAG based on error message m_i and b_i
rag_i^c	The code RAG based on buggy code b_i
p_t	Prompt of thought generation
p_c	Prompt of buggy code correction
t_i	The thought for solving buggy code b_i

methods delivered results that met our criteria for satisfaction in the context of industry-level HDL debugging scenarios. A primary contributing factor to this shortfall is the insufficiency of HDL code resources for training. Consequently, these pre-trained LLMs struggle to accurately comprehend the syntax and functionality inherent to HDL codes.

To tackle the problem, we propose an HDL debugging framework, namely HDLdebugger, which consists of three components, i.e., data generation, search engine, and retrieval-augmented LLM fine-tuning. Firstly, the data generation procedure targets overcoming the obstacle of the limited availability of HDL bugs. Specifically, we employ reverse engineering to insert specific modifications into the original error-free code. Therefore, we can produce corresponding buggy versions and error messages via compilers, which are used to construct a code database and further fine-tune LLMs. Secondly, we propose an effective and efficient search engine, which is supported by the code database constructed by the data generation approach and the document database with various internal HDL documents which contain relevant information for buggy codes. Given a buggy code and its error message, the search engine retrieves relevant information (i.e., document RAG) and buggy codes (i.e., code RAG) with similar patterns from the document database and code database, respectively. The document RAG and code RAG are crucial for both the fine-tuning and inference stages of our retrieval-augmented LLM, enhancing the ability of LLMs to comprehensively understand the HDL buggy code and repair it effectively. Thirdly, to enhance the ability of LLMs to generate accurate code solutions, we propose a novel fine-tuning approach for LLMs. This approach incorporates a self-guided thought generation mechanism and a retrieval-augmented fine-tuning process, significantly improving the LLM's performance in debugging HDL code.

Our contributions are summarized as follows:

- We introduce an advanced LLM-based HDL debugging framework supporting chip designs in the industry, namely HDLdebugger, which consists of buggy data generation, search engine, and retrieval-augmented LLM fine-tuning.
- To address the scarcity of high-quality HDL debugging training data, we propose a data generation approach based on reverse engineeering to comprehensively generate diverse and realistic HDL buggy codes with the correct version.
- We propose a search engine to create code RAG (resp. doc RAG) for HDL buggy code (resp. relevant information) effectively and efficiently, enhancing the fine-tuning and inference of LLMs.
- We present a novel retrieval-augmented fine-tuning approach for HDL debugging, which integrates self-guided thought generation with RAG-based fine-tuning strategies.
- Extensive experiments on the HDL code dataset from Huawei demonstrate superior performance against 13 state-of-the-art baselines, including GPT4 and various HDL debugging LLMs.
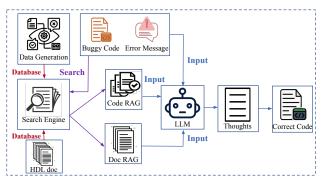

Figure 1: Framework overview of HDLdebugger.

2 METHODOLOGY

This section provides a comprehensive overview of the proposed HDLdebugger. Initially, we delve into the buggy code generation pipeline, as detailed in Section 2.1. Subsequently, the search engine mechanism tailored for Retrieval-Augmented Generation (RAG) is presented in Section 2.2. The Retrieval-Augmented LLM fine-tuning is elaborated upon in Section 2.3. The important notations in our paper are shown in Table 2.

Framework. As shown in Fig. 1, given a buggy code and its associated error message, our proposed HDLdebugger targets to repair this buggy code into the correct one. Specifically, we first propose a data generation approach in Sec. 2.1 to generate a set of HDL code instances, where each instance consists of buggy code, error messages, and its correct version. These generated HDL code instances will be used to provide context for buggy code queries and fine-tune the LLMs. Second, we propose a search engine in Sec. 2.2, which targets to retrieve relevant text information(i.e., document RAG) for error messages and retrieve buggy codes with similar buggy patterns (i.e., code RAG). Then, HDLdebugger takes the buggy code, its error message, task prompt, document RAG, and code RAG to the LLMs, and enables LLMs to predict the correct code. Specifically, we introduce a retrieval-augmented fine-tune approach to fine-tune the LLMs for HDL code debugging in Sec. 2.3.
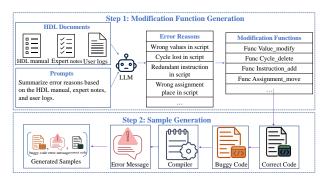
Figure 2: Reverse Engineering Pipeline for inducing error to correct code by diverse modification functions summarized from HDL documents.

2.1 HDL Buggy Data Generation

Unlike software languages like Python or C++, where code can often be crawled from open websites and platforms like GitHub, HDL codes, particularly those used for chip testing, are rarely made public due to privacy and commercial concerns. This limitation presents a significant obstacle on fine-tuning LLMs in the domain of HDL. In this section, we will introduce a reverse engineering pipeline for generating high-quality HDL code pairs that consist of both buggy and corrected versions. As shown in Fig. 2, The HDL data generation consists of two steps, i.e., modification function generation and sample generation.

2.1.1 Modification Function Generation. The first step targets to generate a set of high-quality modification functions. These functions are then employed to modify HDL codes provided by industry engineers, thereby generating a diverse collection of buggy code examples. To ensure that the modified codes exhibit a range of realistic and diverse error patterns, we construct the modification functions by leveraging the capabilities of LLMs and a comprehensive collection of industrial HDL documents. Specifically, as shown in Fig. 2, we first collect a set of HDL documents, including the HDL manual, expert notes, and user logs. Subsequently, we then carefully design a prompt that guides the LLMs to extract and summarize prevalent and critical error patterns in HDL, such as syntax misuse or logical errors. With these insights, we proceed to develop the modification functions. The distilled functions focus on simple operations such as adding, deleting, modifying, and adjusting segments with HDL scripts, while the unique set of rules governing HDL ensures that similar operations can result in vastly different errors recorded in HDL documents. These functions explicitly introduce errors into the original correct HDL codes that mirror those commonly encountered errors in industry, thereby creating an invaluable dataset to fine-tune LLMs.

2.1.2 Sample Generation. In this step, we gather a broad range of accurate and high-quality HDL codes $C = \{c_i\}_{i=1}^{|C|}$ from experienced chip engineers. These codes, which have been utilized across various chip designs, are comprehensive to encompass a wide range of functional testing scenarios for chips. These various HDL codes serve as the seed code for error case construction. Specifically, by applying the previous modification functions to these correct HDL
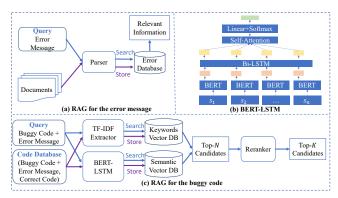

Figure 3: The search engine to retrieve relevant information given error messages and retrieve similar buggy codes based on a buggy code query.

codes, we systematically introduce errors, thereby producing a set of buggy codes. Particularly, we can apply various modification functions to one HDL code, which allows us to generate multiple instances of buggy code. Next, we employ an HDL compiler to compile these intentionally buggy codes on their respective design, which inevitably results in compilation errors. Then, for each correct HDL code $c_i \in C$, we can collect one of its buggy code b_i with associated error message m_i as an instance $I_i = (b_i, m_i, c_i)$ of training data. Normally, diagnosing and rectifying HDL errors require the expertise of chip engineers. Given that our instructions systematically brought the errors, we can employ reverse engineering to identify the faults and produce solutions directly. This bypasses the need for manual error diagnosis, streaming the process of creating a vast array of comprehensive datasets of error scripts, error messages, and corresponding solutions. This method ensures a rich diversity in the types of errors produced, which is critical for creating an extensive and effective training dataset $D_c = \{I_j\}_{j=1}^{|D_c|}$.

2.2 Search Engine for RAG

In this subsection, we propose a search engine to optimize retrieval-augmented generations (RAG) for retrieving relevant information in the HDL documents and codes instances. The retrieved RAG content will serve as contextual information for queries, thereby enhancing the capability of LLMs to understand buggy context information and identify issues within buggy codes. We illustrate the overview of search engine framework in Fig. 3.

2.2.1 Document RAG. As shown in Fig. 3 (a), we first collect a comprehensive collection of instructional documents for this HDL, encompassing language specifications, error diagnostics, and troubleshooting techniques. Then, we meticulously curate the content of documents, distilling a dedicated error database D_e tailored to this HDL. This error database contains detailed error descriptions d_j, underlying reasons r_j, and suggested remedial strategies s_j for each error id e_j. We illustrate examples of error information in error database in Tab. 7 in the Appendix. Given an error message query m_i, we first parse m_i to extract $n_{e,i}$ constituent error codes $\{e_j\}_{j=1}^{n_{e,i}}$. Subsequently, we retrieve the descriptions, reasons, and potential solutions for each identified error code from our error database.

Then, the document RAG of the query m_i can be assembled as $rag_i^d = \{(e_j, d_j, r_j, s_j)\}_{j=1}^{n_{e,i}}$, thereby helping LLMs to understand the error message m_i.

2.2.2 Buggy Code RAG. As shown in Fig. 3 (c), in the code retrieval component, we maintain a code database $D_c = \{I_i\}_{i=1}^{|D_c|}$, where each code instance $I_i = (b_i, m_i, c_i)$ consists of a buggy code b_i, its associated error messages m_i, and the correct code c_i. Given a query $q = (b, m)$ that includes a snippet of buggy code b with its associated error message e, the aim of the code RAG is to identify a subset $D_c^q \subseteq D_c$ of the top-k code instances that have the most similar to the buggy code b in the query. In such a way, the buggy code b has similar buggy patterns with the buggy codes in each instance $I_i = (b_i, m_i, c_i) \in D_c^q$. Thus, the correct code c_i for each buggy code b_i can be provided to LLMs. As a result, LLMs can use the learned patterns from the top-k similar instances to fix for the buggy code b in the query. Specifically, we first introduce how to learn low-dimensional vectors for buggy codes and then then introduce a two-stage ranker that retrieves the top-k buggy code instances for code query q.

Vector Database Construction. Given the complexity and length of HDL code and associated error messages, as well as similar-looking code snippets containing distinct buggy patterns, computing the similarity between buggy codes is challenging. To address this challenge, we propose to measure the similarity between buggy from two aspects, i.e., keyword similarity and semantic similarity. First, we extract words from all buggy codes and their error messages and use the TF-IDF [2] technique to compute the weight of each word. Then, we can generate the keyword vector z_i^w for each buggy code b_i with its error message m_i. Second, we design a BERT-LSTM model that combines BERT, for its powerful language understanding capabilities, with an LSTM, for its sequential data processing strengths. The BERT-LSTM model encodes a buggy code b_i with its error message m_i into a low-dimensional embedding z_i^s. Specifically, as introduced in Sec. 2.1, the buggy codes are generated by different modification functions, and here we take these modification functions as the labels for each buggy and optimize the BERT-LSTM model. Then, we use the final representation of BERT-LSTM as the semantic embedding for each buggy code and its error messages. The architecture details of BERT-LSTM models are introduced in Appx. B.1. Based on the TF-IDF and BERT-LSTM models, we build a keyword vector database and a semantic vector database, which together facilitate a robust framework for analyzing the similarity between instances of buggy HDL code.

Two-stage Ranker. In general, we design a two-stage ranking approach to identify the top-k most relevant buggy code instances to a given query of buggy code. In the first-ranking stage, for any given buggy code query $q = (b, e)$, we first use the TF-IDF encoder and BERT-LSTM encoder to generate the keyword vector z_i^w and semantic vector z_i^s. Then, we define the similarity between the query code q and each instance $I_i \in D_c$ in the code database D_c as:

$$sim(q, I_i) = \lambda \cdot cosine(z^w, z_i^w) + (1 - \lambda) \cdot cosine(z^s, z_i^s) + 1, \quad (1)$$

where $\lambda \in [0, 1]$ is a hyper-parameter between semantic similarity and keyword similarity and $sim(q, I_i) \in [0, 2]$. We select the top-N similar instances $\hat{D}_c^q$ from the code database D_c based on the

similarity score. In the second-ranking stage, our goal is to pinpoint the top-k relevant yet diverse buggy codes. These selections are aimed at providing a broader range of buggy patterns, which can help LLM repair the bugs in the query. Formally, given N buggy instances $\hat{D}_c^q$ and the query buggy code $q = (b, e)$, we select the top-k relevant and diverse code D_c^q by maximizing the following objective:

$$\max_{D_c^q \subseteq \hat{D}_c^q} \sum_{I_i \in D_c^q} sim(q, I_i) + \frac{1}{k} \cdot \sum_{I_i \in D_c^q} dis(I_i, D_c^q), \quad (2)$$

where distance $dis(I_i, D_c^q) = \min_{I_j \in D_c^q \setminus I_i} (2 - sim(I_i, I_j))$ denotes the diversity value between each instance I_i and the other instances in D_c^q and $dis(I_i, D_c^q) \in [0, 2]$. Eq. (2) is an NP-hard problem, which can be reduced from the well-known k-clique problem [36] by setting $sim(q, I_i) = 1$ for all $I_i \in D_c$. Therefore, we propose a greedy algorithm with an approximation ratio $1 - 1/e$ to identify the top-k relevant buggy codes. The details of the greedy algorithm and approximation ratio are introduced in Appx. B.2. Thus, given a buggy code b_i and error message m_i, we can generate the code retrieval-augmented generation $rag_i^c = \{(b_j, m_j, c_j)\}_{j=1}^{k}$.

2.3 Retrieval-augmented LLM Fine-tuning

In this subsection, we introduce how to fine-tune the LLMs based on the training dataset constructed in Sec. 2.1 and the search engine proposed in Sec. 2.2. Specifically, we first propose to generate a thought to help LLM repair each buggy code in the training dataset. Second, based on the generated thought and the retrieved buggy codes, we propose a retrieval-augmented supervised fine-tuning technique for LLMs.

2.3.1 Self-guided Thought Generation. One straightforward way is to feed buggy code and error messages to LLMs and let LLMs directly predict the correct codes. However, this approach is insufficient for LLMs to deeply comprehend the problem and provide accurate solutions. Recent research [40, 45] suggests that when LLMs are prompted to produce a series of intermediate and explanatory thought before finally outputting the solution to the given task, the performance of LLMs can be significantly improved. It is because these reasoning thoughts improve the understanding of LLMs on the input tasks and thus generate more relevant and accurate outputs. Therefore, before fine-tuning the LLMs, we propose to generate high-quality thought for each training code instance.

Specifically, as illustrated in Fig. 4, we first design a precise and explicit prompt p_t to clarify the thought generation task for LLMs. Following this, we input the thought generation prompt p_t, the buggy code b_i, its associated error message m_i, and the document RAG rag_i^d, and its correct version c_i into the LLM with its inference mode LLM_1. This enables the LLM to generate a thought t_i on how to repair the buggy code b_i into the correct code c_i as follows:

$$t_i = LLM_1(p_t \circ b_i \circ m_i \circ c_i \circ rag_i^d), \quad (3)$$

where $\circ$ means concatenate operator. We omit the general task requirements prompt for simplification. Empirically, we find that when LLMs generate the thought t_i only once, the output might be irrelevant to the buggy code b_i or incorrect due to the hallucination phenomenon and randomness of LLMs. Therefore, to guarantee the quality of the generated thought, we iteratively generate L different

thoughts $T_i = \{t_{i,j}\}_{j=1}^{L}$ for each buggy code b_i by running LLM_1 L times following Eq. (3). This is achieved by modifying temperature parameters and the details are described in appendix.

To assess the quality of each thought $t_{i,j} \in T_i$, we adopt a self-guidance strategy to select the highest quality thought from the thought set T_i for the buggy code b_i. Specifically, we feed the buggy code b_i, its associated error message m_i, and the document RAG rag_i^d to the LLM. A prompt p_c, "Based on the analysis, the correct script is," is appended to guide the LLM towards generating the predicted correct script, represented as:

$$\hat{c}_{i,j} = LLM_1(p_c \circ b_i \circ m_i \circ rag_i^d \circ t_{i,j}). \quad (4)$$

After we obtain the output $\hat{c}_{i,j}$ for each thought $t_{i,j}$, we employ the edit distance metric [26] to evaluate the similarity between the predicted correct code $\hat{c}_{i,j}$ and the ground-truth of corrected code c_i as

$$d_{i,j} = EditDistance(c_i, \hat{c}_{i,j}). \quad (5)$$

Intuitively, if the distance $d_{i,j}$ between the predicted correct code $\hat{c}_{i,j}$ and the ground truth of corrected code c_i is smaller, the thought $t_{i,j}$ is more helpful to LLMs to repair the buggy code b_i. Therefore, we select the thought with the smallest edit distances for each b_i, i.e., $t_i = \min_{t_{i,j} \in T_i} d_{i,j}$. Finally, we can obtain the training dataset $\mathcal{D} = \{(b_i, m_i, rag_i^d, rag_i^c, t_i, c_i)\}_{i=1}^{|D_c|}$. The details of the thought generation are illustrated in Alg. 2 in the Appendix.

2.3.2 Retrieval-augmented Fine-tuning. After obtaining the final training dataset $\mathcal{D} = \{(b_i, m_i, rag_i^d, rag_i^c, t_i, c_i)\}_{i=1}^{|D_c|}$, we supervise fine-tune the LLMs based on buggy codes and retrieval-augmented generation in Sec. 2.2. Specifically, given each training instance $D_i = (b_i, m_i, rag_i^d, rag_i^c, t_i, c_i) \in \mathcal{D}$, we first feed thought generation prompt p_t, buggy code b_i, its error message m_i, document RAG rag_i^d, code RAG rag_i^c, to a target LLM, LLM_2 to generate the predicted thought $\hat{t}_i$ as follows:

$$\hat{t}_i = LLM_2(p_t \circ p_c \circ b_i \circ m_i \circ rag_i^d \circ rag_i^c). \quad (6)$$

Then, based on the predicted thought $\hat{p}_t$ and code correction prompt p_c, LLM LLM_2 predicts the correct code $\hat{c}_i$ as follows:

$$\hat{c}_i = LLM_2(p_t \circ p_c \circ b_i \circ m_i \circ rag_i^d \circ rag_i^c \circ \hat{t}_i). \quad (7)$$

Following [33, 39], we fine-tune the target LLM with its training mode LLM_2 by using the conventional next-token prediction objective and minimize the cross-entropy loss $\mathcal{L}_\mathcal{D}$:

$$\mathcal{L}_{t_i} = \sum_{w_{i,j} \in t_i} \log p_{LLM_2}(w_{i,j}|C \circ w_{i,<j}),$$

$$\mathcal{L}_{c_i} = \sum_{w_{i,k} \in c_i} \log p_{LLM_2}(w_{i,k}|C \circ \hat{t}_i \circ w_{i,<k}),$$

$$\mathcal{L}_\mathcal{D} = \frac{1}{2|\mathcal{D}|} \sum_{D_i \in \mathcal{D}} (\mathcal{L}_{t_i} + \mathcal{L}_{c_i}), \quad (8)$$

where the context $C = p_t \circ p_c \circ b_i \circ m_i \circ rag_i^d \circ rag_i^c$ denotes the concatenate of the inputs for clarification $w_{i,j} \in t_i$ denotes the j-th word in t_i and $w_{i,<j}$ denotes a set of words in t_i before $w_{i,j}$, and $p_{LLM_2}(w_{i,j}|w_{i,<j})$ denote the probability of $w_{i,j}$. $\mathcal{L}_{t_i}$ and $\mathcal{L}_{c_i}$ denote the prediction cross-entropy loss on thought ground truth t_i and correct code ground truth c_i regarding the buggy code b_i, respectively.

#1: Task Requirement Prompt
Now you are expert in HDL(Hardware Description Language). Your task involves debugging in HDL. You are given snippets of HDL script that contain errors.
Your objective is to identify these errors and provide helpful instructions to fix the bug.

#2: Buggy Code
```
cycle {
    assign clk_s_top 0
    assign clk_top_core_3 0
}
cycle {
    assign clk_s_top 1
}
…
```

#3: Correct Code
```
cycle {
    assign clk_s_top 0
}
cycle {
    assign clk_s_top 1
}
…
```

#4: Error Messages
…
Error: Can't find core when set prt cat top hierarchy.
Error: RULE(PRT-1)() test failed"
…

#5: Retrieved Information
…
The port defined at the core level is not consistent with the actual port. Reconnect the port that is not connected to the top-level.
…

#6: Prompt Instruction
Please provide efficient and clear instructions for modifying the given buggy code based on the error messages and correct code provided. Your instructions should help improve the buggy code without directly including any information from the correct code. The helpful instruction is: Based on the error message

#7: Thought Generation
Based on the error message, it appears that there is an inconsistency between the ports defined at the core level and the actual ports. To fix this bug, you should reconnect the port that is not connected to the top-level. In this case, the port "clk_top_core_3" is not connected to the top-level, so you should reconnect it

Figure 4: Thought generation example.

3 EXPERIMENTS
3.1 Experiment Setting

3.1.1 HDL Datasets in Huawei. We gather a diverse collection of HDL code files from Huawei. These files are meticulously curated, with each file being specifically utilized for distinct chip design scenarios, reflecting the varied and specialized demands of circuit design. Each HDL code file contains an extensive array of data, including variable assignments, detailed circuit designs, clocking information, function testing protocols, and various test items that are critical for the chip design process. Based on these HDL code files, we use the data generation in Sec. 2.1 to generate 92,143 distinct HDL training code instances. Each HDL training code instance consists of the buggy HDL code, error messages, and the correct HDL code. Specifically, in the experiments, we split the data into training and testing sets at a ratio of 8:2, respectively.

3.1.2 Baselines. We compare our proposed HDLdebugger with 13 baselines in three types of code debugging approaches as follows:

- **Five large language models**: We compare HDLdebugger against two profound and state-of-the-art LLMs available through API services, including ChatGPT [4] and GPT-4 [1]. Additionally,

we compare HDLdebugger with three open-source LLMs: Open-Chat [37], Orca2 [24], and Mistral [16]. These open-source models have shown performance on par with ChatGPT across various open LLM benchmarks.

- **Four Code Debugging and HDL Models**: We compare HDLdebugger with two code debugging and hardware code generation models, i.e., Self-debug [7] and RTLfixer [35]. Self-debug [7] is one of the most classical methods of code debugging. RTLfixer [35] is proposed to solve HDL debugging problems. VeriGen [34] and RTLCoder [22] are two LLMs targeting hardware language.
- **Four Code Language Models**: We compare four SOTA pre-trained code LLMs, i.e., Deepseek [3], Starcoder [18], Stablecode [31], and WizardCoder [23].

For baselines except for ChatGPT [4] and GPT-4 [1]., we adopt three strategies, i.e., the raw model, the raw model with RAG, and the raw model with supervised fine-tuning (SFT).

- **Raw model**: We only feed buggy code and error messages to the model and enable raw models to infer the correct code directly.
- **Raw Model with RAG**: For buggy code, we feed the buggy code, its error message, and the document RAG and code RAG obtained in Sec. 2.2 to the raw model and enable the raw models to infer the correct code directly.
- **Raw Model with SFT**: We take the buggy code and error message as inputs of LLMs and use the correct code as ground-truth to fine-tune the raw models.

Specifically, we selected CodeLlama-13b as our base model from the available code LLMs. The choice of 13b was driven by its optimal model size, which strikes a balance between training and deployment, taking into account both performance and cost factors.

3.1.3 Evaluation Metrics. For the overall debug system, we mainly evaluate its pass rate for correcting codes, relative code runtimes, and edit distance between correct code and buggy code. The calculation of these metrics is listed below.

- **Pass-Rate**: Pass rate for executing code file corrected by each method is defined as $P = \frac{\sum_{i=1}^{n_c} \mathbb{S}(y_i)}{n_c}$, where y_i stands for the corrected code, $\mathbb{S}$ denotes for executing code successfully.
- **Run-Time**: The relatively average compilation time for Huawei's internal HDL compiler to execute all test code files. The Run-Time for results from our HDLdebugger is set as the base unit.
- **Edit-Distance**: Edit-Distance calculates the minimum number of operations (insertion, deletion, substitution) required to transform one code snippet into the other.

  Also, we evaluate our code search engine in Sec. 2.2 by the hit ratio, mean average precision, and mean reciprocal rank metrics, which are formulated as follows.

- **H@K**: Hit ratio for top-K recommendation results on n_t code queries is formulated as $H@K = \frac{1}{n_t}\sum_{i=1}^{n_t}\frac{1}{K}\sum_{k=1}^{K}\mathbb{I}(y_{i,k}, y_i)$, where $y_{i,k}$ denotes the error type of the retrieved k-th buggy code for query code b_i, and the indicator function $\mathbb{I}(y_{i,k}, y_i) = 1$ if $y_{i,k} = y_i$
- **MAP@K**: Mean average precision (MAP) for top-K results on n_t queries is defined as $MAP@K = \frac{1}{n_t}\sum_{i=1}^{n_t}\frac{1}{K}\sum_{k=1}^{K}\frac{\mathbb{I}(y_{i,k}, y_i)\cdot n(b_{i,\leq k})}{k}$, where $n(b_{i,\leq k})$ denotes the number of recommendations in the first top-k that has the same error label with query b_i.

Table 3: Main Results. Pass-Rate describes the absolute value, which is the higher the better. $< 1\%$ describes the Pass-Rate less than 1%. Run-Time and Edit-Distance are relative values compared with ours, which are both the lower the better.

Method	Pass-Rate	Run-Time	Edit-Distance
ChatGPT*	3.01%	2.25	31.28
GPT4*	6.35%	1.94	18.17
OpenChat	<1%	2.51	34.47
OpenChat w/ RAG	3.01%	2.21	10.37
Orca2	2.68%	2.08	2.94
Orca2 w/ RAG	9.03%	2.10	4.74
Mistral	7.36%	2.39	6.68
Mistral w/ RAG	24.75%	2.01	9.88
Self-debug	5.02%	2.49	10.24
RTLfixer	28.35%	2.11	11.05
VeriGen	<1%	2.46	49.16
VeriGen w/ RAG	1.34%	2.56	37.12
VeriGen w/ SFT	67.55%	1.35	1.38
RTLCoder	<1%	2.49	43.08
RTLCoder w/ RAG	<1%	2.65	34.85
RTLCoder w/ SFT	64.21%	1.53	3.83
Deepseek	2.34%	2.58	32.17
Deepseek w/ RAG	3.34%	2.51	34.38
Deepseek w/ SFT	51.63%	1.64	3.35
Starcoder	<1%	2.58	28.85
Starcoder w/ RAG	<1%	2.56	34.85
Starcoder w/ SFT	68.27%	1.21	1.57
Stablecode	6.69%	2.46	4.56
Stablecode w/ RAG	8.02%	2.50	6.25
Stablecode w/ SFT	41.47%	2.38	6.10
WizardCoder	3.01%	2.41	7.19
WizardCoder w/ RAG	4.68%	2.58	8.67
WizardCoder w/ SFT	71.57%	1.04	1.06
HDLdebugger(ours)	**81.93%**	**1.00**	**1.00**

- **MRR@K**: Mean reciprocal rank (MRR) for top-K results on n_t queries is formulated as $MRR@K = \frac{1}{n_t}\sum_{i=1}^{n_t}\frac{1}{K}\sum_{k=1}^{K}\frac{\mathbb{I}(y_{i,k}, y_i)}{k}$.

3.2 Main Results

Table 3 demonstrates the main performance of our results and other methods. It's clear that our method outperforms other methods by all means by a large margin including direct approach, RAG and SFT, which demonstrates the effectiveness of our framework. For both runtime and edit distance metrics, we normalize all results and present only the relative values in comparison with ours to enhance the clarity and effectiveness of the comparison.

3.2.1 Comparison with different types of LLMs. In our study, we conducted a comparative evaluation of both general-purpose LLMs such as ChatGPT, GPT-4, OpenChat, Orca, Mistral, Deepseek, Starcoder, Stablecode, WizardCoder, and specialized code LMs including Self-debug, RTLfixer, VeriGen, RTLCoder, within the HDL debugging scenario. Due to privacy concerns, certain code specifications have been omitted for testing purposes when using ChatGPT or GPT4. To distinguish these versions, we will refer to them as ChatGPT* and GPT4*. It is evident that our approach exhibits

Table 4: Ablation on different strategies

Method	Pass-Rate	Run-Time	Edit-Distance
Direct(CodeLlama)	4.01%	2.26	35.51
+ RAG	15.05%	2.20	5.34
+ SFT	70.56%	1.19	1.19
+ SFT w/ th	74.91%	1.09	1.13
+ RAG & SFT w/ th	**81.93%**	**1.00**	**1.00**

superior performance against all 13 state-of-the-art benchmarks, including GPT-4 and other domain-specific hardware-based language models.

3.2.2 Analysis of Different Strategies. In this study, we implement three distinct evaluation strategies to assess the efficacy of various methodologies: with retrieval-augmented generation (RAG), with supervised fine-tuning (SFT), and via a direct approach. Within the context of HDL debugging, our analysis reveals that SFT holds greater significance and applicability across all evaluated baselines, including VeriGen, RTLCoder, Deepseek, Starcoder, Stablecode, and WizardCoder. In the majority of scenarios, we note that methods enhanced through SFT consistently outperform those augmented with RAG by a substantial margin. Besides, our proposed method integrates both RAG and SFT strategies, achieving unparalleled performance, indicating that it is better to incorporate SFT and RAG for the HDL debugging task.

3.2.3 Impact on Domain-Specific Solutions. In addition to general and code-specific Language Models (LLMs), methodologies such as Self-debug and RTLfixer are specifically devised to address code debugging scenarios. While these approaches demonstrate improvements over other general and code LLMs, their effectiveness still falls short of being fully satisfactory. Our analysis extends to evaluating our dataset with LLMs exclusively trained on hardware languages, namely VeriGen and RTLCoder. Contrary to expectations, these specialized hardware language models do not outperform their general and code LLM counterparts in our HDL debugging context, suggesting the possibility of an inherent task domain generalization issue within HDL debugging scenarios. On the other hand, our approach consistently surpasses domain-specific solutions regardless of the varied prompt engineering techniques employed or the domain-specific data used for training, which underscores the effectiveness of our methodology.

3.3 Ablation Studies

Firstly, we provide a detailed analysis of the impact of different strategies of our method. Table 4 illustrates the performance of different strategies. For baseline ,we only use direct inference strategy on base model, i.e., CodeLlama. SFT w/ th indicates supervised fine-tuning with generated thoughts. In terms of RAG & SFT w/ th, we mainly refer to retrieval augmented LLM fine-tuning where both retrieved code instances and relevant information are combined together for LLM fine-tuning. From Table 4 we can observe that SFT w/ th outperforms baseline by a large margin. Moreover, combing RAG & SFT also significantly improves the performance.

Besides, we evaluate our search engine for RAG, comparing it with four methods: F-IDF [2], BM25 [30], random forest [29], and XGBoost [6]. TF-IDF and BM25 assess relevance scores between buggy codes, while Random Forest and XGBoost generate

Table 5: Evaluation on the search engine in Sec. 2.2 on the top-k recommendation. XGB and RF denote the XFBoost and Random Forest, respectively. MAP@1 and MRR@1 are the same as H@1 mathematically. The first row *Optimal* is the optimal performance under each metric.

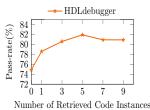|	H@1	H@3	H@10	MAP@3	MAP@10	MRR@3	MRR@10
Optimal	*1.00*	*1.00*	*1.00*	*1.00*	*1.00*	*0.61*	*0.29*
TF-IDF	0.97	0.87	0.75	0.86	0.71	0.55	0.24
BM25	0.95	0.80	0.59	0.81	0.54	0.53	0.22
XGB	1.00	0.35	0.18	0.34	0.14	0.34	0.12
RF	0.31	0.43	0.41	0.34	0.29	0.25	0.11
Ours	**1.00**	**0.98**	**0.94**	**0.98**	**0.93**	**0.60**	**0.28**

Figure 5: Pass-rate and code retrieved code instances

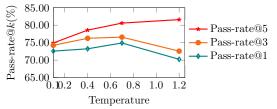**Figure 6: Inference time and retrieved code instances**

Figure 7: Pass-rate@k and Temperature

low-dimensional representations for codes and error messages, similar to our BERT-LSTM model's approach to computing relevance through representation similarity.

Table 5 indicates our engine surpasses all baselines in accurately retrieving and correcting buggy code queries, highlighting its superior ability to decode complex buggy code patterns beyond the capabilities of traditional and machine learning models. Traditional methods like TF-IDF and BM25 lack the depth to understand complex code bugs, while models like XGBoost and Random Forest fall short in semantic comprehension. Our engine effectively combines textual and semantic analysis, enhancing bug detection and correction.

3.4 Parameter Sensitivity

In the following experiments, we evaluate parameter sensitivity across different hyper-parameters including retrieved code instances samples and related inference time. We also consider temperature and pass-rate@k for various k.

3.4.1 The number of retrieved code instances. Figure 5 depicts the impact of varying the number of retrieved code instances on model performance. We find that performance improves with each additional instance between 1 to 5, achieving the most significant gains. However, beyond five instances, gains plateau or even decrease, indicating diminishing returns. This suggests that while

adding retrieved code instances enhances model performance up to a point, increasing instances beyond this threshold leads to inefficiency. Moreover, we observe a tendency for the language model to generate repetitive or redundant content derived from previous input as the number of retrieved code instances increases. This phenomenon underscores a critical area for future exploration.

3.4.2 Feasibility on Inference. We assess the impact of incorporating the retrieved code instances on the additional inference budget. Figure 6 illustrates the normalized inference times to clearly highlight the incremental budget required. A value of 0 indicates the absence of retrieved code instances. Our observations reveal that as the number of retrieved code instances increases, the corresponding inference time exhibits a slow, logarithmic increase rather than a linear one. This pattern underscores the efficiency of our approach, demonstrating that integrating retrieved code instances significantly enhances performance without proportionally increasing the inference overhead.

3.4.3 Pass-rate@k and Temperature. We explore the effects of varying the temperature settings and the pass-rate@k for different values of k, where k represents the number of answers generated by the LLM, as discussed in [5]. Typically, a lower temperature setting yields more deterministic outcomes, whereas higher temperatures result in more varied outputs. Figure 7 demonstrates that as k increases, so does overall performance. Specifically, at a lower temperature, such as 0.1, outputs are more consistent, leading to a narrower performance range. Conversely, at higher temperatures, like 1.2, outputs become more varied, enhancing the likelihood of generating correct answers as k increases. Notably, for a pass-rate@5, the performance at a temperature of 1.2 surpasses that at 0.7, indicating that increased temperature settings can improve outcomes, particularly at higher values of k.

4 RELATED WORK

4.1 Automatic Code Debugging

Automatic code debugging has emerged as a promising area within software engineering [15, 25]. Given a code with bugs, the task is to automatically fix the code bugs with the correct functions, which alleviates the burden of manual debugging and fixing code faults. Classic techniques can be mainly classified as template-based [17, 20], heuristic-based [41, 47], constraint-based [43, 44], and neural network-based approaches [8, 11, 49]. Specifically, template-based approaches apply expert-defined code patterns to fix bugs. These approaches can only repair codes in specific patterns and lack generalization to other bugs. Heuristic-based approaches apply predefined heuristics and cannot cover all types of bugs. Constraint-based approaches repair buggy codes by solving a constraint problem. These methods can be accurate, but they are computationally expensive. Neural network-based approaches need numerous high-quality labeled training data pairs (i.e., pairs of buggy codes and fixed codes) to optimize parameters, which is time-consuming to collect the high-quality code pairs.

Recently, large language models (LLMs) have shed new light on automatic code debugging. The prevailing hypothesis suggests that LLMs, through training on extensive repositories of open-source code snippets, are adept at identifying bug patterns and facilitating the repair of defective code. Contemporary strategies employing LLMs predominantly utilize retrieval augmented generation (RAG) and sophisticated prompt engineering techniques to address debugging challenges. Notably, Self-debug [7] represents a pioneering effort in applying LLMs to code debugging, employing targeted prompt engineering for enhanced effectiveness. RTLfixer [35] utilizes both RAG and prompt engineering to tackle the HDL debugging problem. However, these methods do not show satisfactory results in our industry-level cases due to the lack of requisite knowledge of HDL codes. Fine-tuning with HDL code resources is one alternative to tackle the problem. Nevertheless, these approaches need abundant and high-quality labeled data, which is not suitable for HDL codes, since the related HDL codes are limited due to privacy and commercial issues.

4.2 large language models for Code Generation

Large language models (LLMs) have transformed the landscape of code generation by leveraging vast amounts of code data to predict and generate syntactically and semantically correct code snippets [38]. Notable among these models is OpenAI's Codex [5], which powers GitHub Copilot, offering context-aware code suggestions and completions to developers directly within their IDEs. Another key contribution is from DeepMind's AlphaCode [19], which excels in generating code solutions for competitive programming challenges and obtains the top percentile of participants in coding competitions. Recent advancements in LLMs tailored for the coding domain have seen significant contributions, with notable examples including DeepSeek [3], Starcoder [18], Stabelcode [31], Codellama [32], and WizardCoder [23]. These models have been trained on extensive datasets, both proprietary and open-source, to enhance capabilities in code generation, completion, and debugging. Within the realm of hardware description languages (HDLs), ChipNeMo [21] represents a pioneering effort in developing a domain-specific LLM, highlighting the challenges and potential of applying LLMs to the hardware domain. Despite being trained on vast hardware-specific datasets, ChipNeMo achieves performance that, while competitive, does not surpass that of state-of-the-art general LLMs, such as GPT-4. This underscores the inherent complexities of adapting LLMs to the nuances of HDL. Concurrently, initiatives like VeriGen [34] and RTLCoder [22], which focus on fine-tuning LLMs using specialized datasets, have demonstrated remarkable results. These developments underscore the evolving landscape of LLM applications in HDL debugging, highlighting both achievements and areas ripe for further exploration.

5 CONCLUSION

In this paper, we propose an LLM-assisted HDL debugging framework, namely HDLdebugger, which consists of HDL debugging data generation, a search engine, and a retrieval-augmented LLM fine-tuning approach. Through extensive and varied experimentation with multiple LLMs, we have unearthed pivotal findings within the domain. Our method significantly surpasses existing techniques, achieving an exceptional pass-rate of up to 81.93%, indicating that HDLdebugger could automate and streamline HDL debugging for chip design. In addition, we provide in-depth experimental analysis and outline potential future directions for HDL debugging with LLMs in Appendix D.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Aleman, et al. 2023. Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023).

[2] Akiko Aizawa. 2003. An information-theoretic perspective of tf–idf measures. Information Processing & Management 39, 1 (2003), 45–65.

[3] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, et al. 2024. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism. arXiv preprint arXiv:2401.02954 (2024).

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. Advances in neural information processing systems 33 (2020), 1877–1901.

[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] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 785–794.

[7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128 (2023).

[8] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. IEEE Transactions on Software Engineering 49, 1 (2022), 147–165.

[9] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30, 4 (2011), 473–491.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. Advances in Neural Information Processing Systems 35 (2022), 16344–16359.

[11] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 935–947.

[12] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997 (2023).

[13] Mike Gordon. 1995. The semantic challenge of Verilog HDL. In Proceedings of tenth annual IEEE symposium on logic in computer science. IEEE, 136–145.

[14] Dorit S Hochbaum. 1996. Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. In Approximation algorithms for NP-hard problems. 94–143.

[15] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. arXiv preprint arXiv:2303.18184 (2023).

[16] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. arXiv preprint arXiv:2310.06825 (2023).

[17] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis. 298–309.

[18] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! arXiv preprint arXiv:2305.06161 (2023).

[19] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, et al. 2022. Competition-level code generation with alphacode. Science 378, 6624 (2022), 1092–1097.

[20] Kui Liu, Anil Koyuncu, et al. 2019. TBar: Revisiting template-based automated program repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 31–42.

[21] 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).

[22] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2023. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. arXiv preprint arXiv:2312.08617 (2023).

[23] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, et al. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv preprint arXiv:2306.08568 (2023).

[24] Arindam Mitra, Luciano Del Corro, Shweti Mahajan, Andres Codas, Clarisse Simoes, Sahaj Agarwal, Xuxi Chen, Anastasia Razdaibiedina, Erik Jones, Kriti Aggarwal, et al. 2023. Orca 2: Teaching small language models how to reason. arXiv preprint arXiv:2311.11045 (2023).

[25] Martin Monperrus. 2018. Automatic software repair: A bibliography. ACM Computing Surveys (CSUR) 51, 1 (2018), 1–24.

[26] Gonzalo Navarro. 2001. A guided tour to approximate string matching. ACM computing surveys (CSUR) 33, 1 (2001), 31–88.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems 32 (2019).

[28] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 3505–3506.

[29] Steven J Rigatti. 2017. Random forest. Journal of Insurance Medicine 47, 1 (2017), 31–39.

[30] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. 2004. Simple BM25 extension to multiple weighted fields. In Proceedings of the thirteenth ACM international conference on Information and knowledge management. 42–49.

[31] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 10684–10695.

[32] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023).

[33] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

[34] 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).

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

[36] Charalampos Tsourakakis. 2015. The k-clique densest subgraph problem. In Proceedings of the 24th international conference on world wide web. 1122–1132.

[37] Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. 2023. Openchat: Advancing open-source language models with mixed-quality data. arXiv preprint arXiv:2309.11235 (2023).

[38] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software testing with large language model: Survey, landscape, and vision. arXiv preprint arXiv:2307.07221 (2023).

[39] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. arXiv preprint arXiv:2212.10560 (2022).

[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] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, et al. 2018. Context-aware patch generation for better automated program repair. In Proceedings of the 40th international conference on software engineering. 1–11.

[42] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2019).

[43] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 416–426.

[44] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Transactions on Software Engineering 43, 1 (2016), 34–55.

[45] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. arXiv preprint arXiv:2305.10601 (2023).

[46] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629 (2022).

[47] Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of java programs via multi-objective genetic programming. IEEE Transactions on software engineering 46, 10 (2018), 1040–1067.

[48] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays. 161–170.

[49] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. Autotrainer: An automatic dnn training problem detection and repair system. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 359–371.

A ADDITIONAL MATERIALS FOR DATA GENERATION

Table 6 indicates the common error type in HDL code summarized by LLM. By locating the main cause of errors, we can design multiple modification functions to recur the errors in a given HDL code. It is easy to reveal that some simple and similar modifications can result in totally different errors when matching the error causes and modification functions. These modification functions then become an essential part of the reverse engineering pipeline and the foundation of solutions in the RAG search engine.

The error message and solution database of the RAG search engine is built on the foundation of the modification functions. Table 7 indicates information in the database. For each common error code in HDL code compilation, it provides related descriptions and error root reasons that help debug. And most essentially, the database collects recommending solutions for a given error code. When retrieving knowledge with the RAG search engine, the root reason and solution together with the error message and solution will be returned to form LLM input.

B ADDITIONAL MATERIALS FOR RAG

B.1 BERT-LSTM

The BERT-LSTM model synergizes the contextual embedding capabilities of BERT (Bidirectional Encoder Representations from Transformers) with the sequential data processing strength of LSTM (Long Short-Term Memory) networks. Specifically, given a buggy code b and its error message m, we first concatenate them into a single sequence $qs = [CLS, b, SEP, m]$. Since the token length of buggy code and error message is too long, we separate the query sequence qs into a set of sub-sequences $\{s_k\}_{k=1}^{|qs|/n_s}$, where each subsequence has n_s tokens. Then, we fed each subsequence s_k into BERT, and BERT will output a sentence-level vector $\mathbf{h}_k = BERT(s_k)$, where $\mathbf{h}_k$ captures the context information of tokens in s_k. After obtaining all subsequence embeddings $\{\mathbf{h}_k\}_{k=1}^{|qs|/n_s}$ for $\{s_k\}_{k=1}^{|qs|/n_s}$, we employ a Bi-LSTM to capture long-range dependencies and intricate patterns across different subsequences as $\{\mathbf{h}'_k\}_{k=1}^{|qs|/n_s} = $ BiLSTM($\{\mathbf{h}_k\}_{k=1}^{|qs|/n_s}$). Upon acquiring the sequence of Bi-LSTM outputs, we use a self-attention mechanism to generate the final representation of qs as $\mathbf{z}^s = $ Self_Attention($\{\mathbf{h}'_k\}_{k=1}^{|q|/n_s}$). The self-attention blocks attend to different parts of the LSTM sequence, enabling the model to focus on the most relevant features for classification. Then, we feed $\mathbf{z}^s$ to a linear layer to predict buggy pattern probability for the buggy code query (b, e).

B.2 Greedy Algorithm

As introduced in Sec. 2.2, the top-k buggy code problem in Eq. (2) is NP-hard, indicating that we cannot obtain the optimal top-k buggy code instances D_c^q in any polynomial time. Thus, we propose a greedy algorithm in Alg. 1 with a theoretical guarantee to optimize the re-rank objective in Eq. (2). For clarification, we denote the Eq. (2) by $S(D_c^q)$ as follows.

$$S(D_c^q) = \max_{D_c^q \subseteq \hat{D}_c^q} \sum_{I_i \in D_c^q} sim(q, I_i) + \frac{1}{k} \cdot \sum_{I_i \in D_c^q} dis(I_i, D_c^q)$$

The basic idea of Alg. 1 is to greedily select the buggy code instance that can bring the maximum information gain into the selected instance set D_c^q until it exceeds the number k. Specifically, given the instance set D_c^q, we first define the marginal information gain of I_j as follows:

$$\triangle S(I_j | D_c^q) = S(I_j \cup D_c^q) - S(D_c^q) \tag{9}$$

As illustrated in Alg. 1, we first initialize the instance set D_c^q as $\emptyset$ (line 1). Then, we compute the similarity score $sim(q, I_j)$ for each $I_j \in \hat{D}_c^q$ (lines 2-4). Next, we compute the marginal information gain of each $I_j \in \hat{D}_c^q$ and select node I^* with the maximum $\triangle S(I_j | D_c^q)$ following Eq. (9) (line 6-9). Then, we add I^* into D_c^q, and remove it from $\hat{D}_c^q$ (lines 10-11). We repeat the selection procedure until we have selected k buggy code instances (lines 5-12).

THEOREM B.1. *Alg. 1 can achieve a $1 - 1/e$ approximation ratio.*

PROOF. The $S(D_c^q)$ in Eq. (2) is monotone and submodular.
- Monotone: Given any $I_i \in \hat{D}_q^c$ and selected instances set D_q^c, we can obtain $S(D_c^q \cup I_i) - S(D_c^q) \geq 0$. Thus, $S(D_c^q)$ is monotone increasing.
- Submodularity: Given $\tilde{D}_c^q \subset D_c^q$, and a buggy code instance $I_i \notin \tilde{D}_c^q, D_c^q$, we can obtain:

$$S(\tilde{D}_c^q \cup I_i) - S(\tilde{D}_c^q) = sim(q, I_i) + \frac{1}{k} dis(I_i, \tilde{D}_c^q) \tag{10}$$

$$S(D_c^q \cup I_i) - S(D_c^q) = sim(q, I_i) + \frac{1}{k} dis(I_i, D_c^q) \tag{11}$$

Since $dis(I_i, \tilde{D}_c^q) = \min_{I_j \in \tilde{D}_c^q}(2 - sim(I_i, I_j))$ and $\tilde{D}_c^q \subset D_c^q$, thus we can obtain the inequality as:

$$S(\tilde{D}_c^q \cup I_i) - S(\tilde{D}_c^q) \geq S(D_c^q \cup I_i) - S(D_c^q).$$

It demonstrates that $S(D_c^q)$ in Eq. (2) is submodular.

Since $S(D_c^q)$ is monotone and submodular, according to [14], the approximation ratio of Alg. 1 is $1 - 1/e$. □

Time Complexity. Assume the dimension of $\mathbf{z}^w$ and $\mathbf{z}^s$ are d_w and d_s, respectively. First, it takes $O(N(d_w + d_s))$ to compute the similarity score $sim(q, I_j)$ for each $I_j \in \hat{D}_c^q$ (line 2-4). Then, it takes $O(k^2(d_w + d_s))$ to select the top-k buggy code instances from $\hat{D}_c^q$ (line 5-12). Thus, the time complexity of the top-k buggy code selection algorithm is $O((N + k^2)(d_w + d_s))$ in total.

C RETRIEVAL-AUGMENTED LLM FINE-TUNING

C.1 Implementation Details

We implement our approach in PyTorch [27], and fine-tune on CodeLlama [32] which is provided by huggingface [42] model zoo. The foundation of our code is built upon the FastChat and Alpaca frameworks, incorporating cutting-edge technologies such as flashattention [10], deepspeed [28], to enhance effectiveness during both training and inference phases. Our experimental setup utilizes eight NVIDIA-GTX A100 GPUs with 80G memory to ensure enough computational capacity. For training, we primarily adhere to the default hyperparameters. During the inference stage, we employ a greedy decoding strategy, akin to the approach used in ChipNemo [21], to mitigate the significant compilation costs associated with this process.

Table 6: Sample HDL error and modification functions.

Error Type	Error Cause	Modification Function
Memory	RAM connection or RAM port	Pulse signal modification
Clock	Clock unit connection and availability	Pulse signal or assignment modification
Trace	Trace unit availability and usage	Assignment modification
Compression	Compression unit definition and I/O	Register or assignment modification
Data	Data Value	Register or pulse signal modification
Test Point	Procedures completeness	All variables modification
Pattern Reform	Pattern equivalence	Assignment and probe modification
Syntax E	Sscript syntax error	Syntax modification
Netlist E	Netlist attribute	Netlist modification
Simulation E	Simulation config	Config modification

[1] The errors here are abstracted and modified for information security, but still reflect the categories and causes without revealing details.

Table 7: Examples of error information.

Error code	Descriptions	Root-Reasons	Solutions
C-error-1	Netlist not correctly obtain defined signal	Definition not match netlist	Modify definition or netlist
C-error-2	Not define top port as output	Signal not output	Modify top port as output
C-error-6	Clock signal not off during **	** get wrong signal	Correctly off clock signal
T-error-2	Clock definition duplicate	clock name conflict	Delete one definition
T-error-4	Probe initilized as 0 in **	Invalid initialization	Modify probe initialization to non-0
T-error-18	Assignment in non-initialization stage	Wrong assignment	Delete assignment
T-error-27	Pulse non-exist variable	wrong pulse	Delete pulse
M-error-1	Memory close when read	Wrongly off memory	Correct memory and constraint
M-error-17	Logical loop in netlist	Logical loop in netlist	Modify netlist
P-error-8	Definition lack of shift	Wrong definition	Modify definition

[1] The rules here are abstracted and modified for information security, but still reflect the meanings, causes and solutions without revealing details.

Algorithm 2 Thoughts Generation Flow

Input: Thoughts generation prompt p_t, correct response prompt p_c, buggy code b, correct code c, error message as m, retrieved information as rag^d, large language model as LLM, Number of thoughts to generate k.

Output: Generated thoughts t_i.

1: **Initialize:** $G \leftarrow \{\}, C \leftarrow \{\}, D \leftarrow \{\}$.
2: **for** $j = 1$ **to** k **do**
3:    $t_j \leftarrow LLM(p_t \circ b \circ c \circ m \circ rag^d)$ //Generate thoughts.
4:    $G \leftarrow G \cup t_j$ //Append t_j to G.
5: **end for**
6: **for** each t_j in G **do**
7:    $c_j \leftarrow LLM(p_c \circ b \circ m \circ rag^d \circ t_j)$ //Predicted correct script.
8:    $C \leftarrow C \cup c_j$ //Append c_i to C.
9: **end for**
10: **for** each c_j in C **do**
11:    Calculate edit distance $d_j \leftarrow \text{EditDistance}(c, c_j)$
12:    $D \leftarrow D \cup d_j$ //Append d_j to D.
13: **end for**
14: Sort D and obtain related thoughts t_j^*,
15: **return** t_j^*.

Algorithm 1 Top-k relevant code selection

Input: Buggy code query $q = (b, e)$ and N code instances $\hat{D}_c^q$, and parameter k.

Output: Top-k instances $D_c^q \subseteq \hat{D}_c^q$.

1: **Initialize:** $\hat{D}_c^q \leftarrow \emptyset$.
2: **for** $I_j \in \hat{D}_c^q$ **do**
3:    $Sim(q, I_i) \leftarrow$ **Eq. (1)**
4: **end for**
5: **for** $i = 1$ **to** k **do**
6:    **for** $I_j \in \hat{D}_c^q$ **do**
7:       $\triangle S(I_j | D_c^q) = S(I_j \cup D_c^q) - S(D_c^q)$
8:    **end for**
9:    $I^* = \arg\max_{I_j \in \hat{D}_c^q} \triangle S(I_j | D_c^q)$
10:    $\hat{D}_c^q = \hat{D}_c^q \setminus I^*$
11:    $D_c^q = D_c^q \cup I^*$
12: **end for**
13: **return** D_c^q.

C.2 Thoughts Generation

As Algorithm 2 illustrate the flow of thoughts generation, For simplicity, we omit the sample index. The process mainly consist of 4 major steps. We firstly concatenate Thoughts generation prompt p_t, Buggy code b, Correct code c, Error message m, relevant information rag^d, and pass to LLM for inference, result of which stands for generated thoughts $t_1 \dots t_k$. After obtaining the thoughts for the error code, we then concatenate correct response prompt p_c, Buggy

#1: Task Requirement Prompt
Below is an error snippets, paired with an error message that provides error information.
Return the correct snippets based on the error message.

#2: Retrieved Code Instances
Example1:   Example2:   Example3:   Example4:

Buggy Code: Buggy Code: Buggy Code: Buggy Code: ⋯
⋯   ⋯   ⋯   ⋯
Correct Code: Correct Code: Correct Code: Correct Code:
⋯   ⋯   ⋯   ⋯

#3: Buggy Code
⋯
  assign_pi
  probe_po
  assign rst_n 1
  pulse_clocks
⋯

#4: Error Message &Retrieved Solutions
1 procedure rule TPT-83 occurred(Not allow to pulse non-clock pin in timeplate)
…
The reason is that the define_timeplate block –clocks contains a non-clock pin
…

#5: Prompt Instruction
Please return the correct snippets based on the current error snippets.

#6: Response
It appears that the primary issue in your HDL script is related to the incorrect use or definition of a clock signal. The error message "A floating pin is defined as a clock" suggests that there is a problem with how a clock It appears that the primary issue in your HDL script is related to the incorrect use or definition of a clock signal. The error message "A floating pin is defined as a clock" suggests that there is a problem with how a clock signal is being handled or defined in your script. The error seems to be associated with the **rst_n** signal. is being handled or defined in your script. The error seems to be associated with the **rst_n** signal.

The correct script is:
…
  assign_pi
  probe_po
  pulse_clocks
…

Figure 8: A case of HDL debugging.

code b, Error message m, , and generated thought t_j to construct the step 2 input for LLM. The inference result denotes the predicted correct code script $c_1 \ldots c_k$ for correction the given error code. Given above two steps, the code correction is prepared successfully. The rest steps will focus on finding the best suited correction for given error code. To accomplish that, for given predicted correct code script $c_1 \ldots c_k$, we obtain the edit distance $d_1 \ldots d_k$ from each to original correct code c. The final procedure is to sort the edit distances and retrieve the generated thought t_j^* corresponding to the lowest edit distance d_j. The final t_j^* serves as our though output for the flow.

We delve deeper into the process of self-guided thought generation The core principle of our approach involves submitting both the buggy code and its corrected version to the LLM, subsequently prompting the model to generate instructive guidance. We denote

the correct code as c_i and thoughts as t_i. Generally, compute $p(c_i|t_i)$ proves challenging due to the inaccessibility of suitable thoughts. By leveraging Bayes' rule, we can reparameterize the formulation as $p(c_i|t_i) \propto p(t_i|c_i)p(c_i)$. By focusing on $p(t_i|c_i)$, we encourage the LLM to align its outputs with verified correct code.

C.3 Temperature setting

By modifying temperature we can generate multiple thoughts differently. We use multinomial sampling to generate samples randomly. Define a sequence of input tokens $\mathbf{x} = \{x_1, x_2, ..., x_n\}$, denote π_θ as the LLM generation without passing final softmax layer. The next token x_{n+1} can be obtained by:

$$out = \pi_\theta(\mathbf{x})$$
$$pr = softmax_T(out) \quad (12)$$
$$x_{n+1} \sim Categorical(pr),$$

where out is the next token logits output. $softmax_T$ means the softmax function with temperature where the formula of probability pr_i is $\frac{e^{out_i/T}}{\sum_j e^{out_j/T}}$. T is the temperature parameter where higher T makes the output distribution more uniform, thus introducing more randomness. $Categorical$ means Categorical distribution. For example, assume the next token probability distribution is $\{dog : 0.4, cat : 0.5, bike : 0.1\}$, then the next token is selected according to their probabilities. We can generate K thoughts by this sampling strategy. Therefore, we can generate thoughts randomly by controlling the temperature parameter.

C.4 Debugging example

Figure 8 illustrates an example of HDL debugging. When given an error script and related error messages, we first use our search engine to retrieved related expert solutions and similar in-context samples (default 5 buggy code instances). Then we feed buggy codes, error messages, and related in-context samples and expert solutions to our fine-tuned LLMs. Finally, the LLM will produce a corresponding analysis and related correct script.

D ANALYSIS AND DISCUSSION

This section delves into a detailed examination of our research findings, focusing on the utility and implications of leveraging LLMs for debugging within HDL environments. We provide a thorough assessment of LLM-based debugging capabilities and the effects of iterative debugging procedures. Additionally, we explore both the challenges and prospects of incorporating LLMs into the HDL debugging framework.

D.1 Trade-offs in RAG and SFT

Typically, in a Retrieval-Augmented Generation (RAG) system, the LLM acts as an agent that remains untuned to preserve its generalization ability, which might be compromised by task-specific parameter optimization. However, to enhance its efficacy in debugging, we find it essential to fine-tune the LLM, which presents a significant challenge as LLMs are expected to address a broader array of problems. One solution incorporate a wider range of common instructional data during training to retain its general applicability. Another innovative strategy involves constructing a multi-LLM

architecture housing numerous "expert" models to tackle specific domain challenges. Additionally, HDL debugging encompasses various tasks across different stages of electronic design automation, prompting us to consider developing a comprehensive framework in future investigations.

D.2 Iterative Debugging

Our investigation predominantly concentrates on single-round debugging, noting that existing studies, such as [35], demonstrate that a vast majority of issues (about 90%) are resolvable in a single iteration. Nonetheless, complex scenarios necessitate multiple debugging rounds, posing a substantial challenge due to the high costs associated with gathering multi-round data. Moreover, effective iterative debugging demands LLMs capable of enhanced contextual analysis and comprehension. This area will be a focal point of our subsequent research endeavors.

D.3 Enhancing the User Experience in Debugging Systems

Within our debugging framework, we can quantitatively assess performance metrics such as pass rates and execution times. However, during the deployment phase, we observed that traditional debugging approaches—transforming erroneous scripts into correct ones are insufficient. In the context of HDL, the emphasis extends beyond mere error correction to enhancing the quality of the implementation, given the direct impact of hardware language on chip performance. Code that successfully compiles may not necessarily optimize chip design performance. Additionally, exact solutions are not always required; engineers often benefit from suggestive "hints" that inspire solutions to complex issues. Nonetheless, developing a metric to evaluate the quality of these hints represents a significant challenge.