An Iteratively-refined Dataset for High-Level Synthesis Functional Verification through LLM-Aided Bug Injection

Lily Jiaxin Wan, Hanchen Ye, Jinghua Wang, Manvi Jha, Deming Chen
University of Illinois Urbana-Champaign
{wan25, hanchen8, jinghua3, manvij2, dchen}@illinois.edu

***Abstract*—This paper explores the application of Large Language Models (LLMs) in the domain of High-Level Synthesis (HLS) for hardware design and verification, focusing on functional verification challenges. The scarcity of open-source HLS codebases, especially those containing bugs, poses a significant challenge, as LLMs require extensive datasets for efficient fine-tuning and evaluation. To tackle this, we introduce an innovative bug injection methodology working with a new dataset that we curated from a wide range of open-source HLS benchmark suites. This dataset features over 1,500 designs, with both the version injected with bugs and the corresponding bug-free version. Our bug injection method synergizes In-Context Learning (ICL) with Retrieval Augmented Generation (RAG), and Chain of Thought (CoT). This approach significantly boosts the dataset's overall validity rate for single-bug injections. We demonstrate our solution quality using GPT-4 Turbo for injecting either logic bugs or non-ideal pragmas (compiler directives) into HLS designs. For logic bugs, we achieve an 84.8% ratio for valid injection attempts. Furthermore, our approach maintains an 88.0% dataset validity rate (the valid bug injection rate). In addition, we also evaluate the quality of HLS pragma injections (focusing on non-ideal pragmas), and achieve a 74.0% attempt and an 87.9% valid injection ratio. Compared to brute-force prompting, our strategy yields a 20.4% and a 54.0% validity improvement for the bug and non-ideal pragma injection, respectively. The *Chrysalis* dataset is accessible at https://github.com/UIUC-ChenLab/Chrysalis-HLS.**

***Index Terms*—Large Language Models, High-Level Synthesis, functional verification, dataset**

I. Introduction

In the rapidly evolving hardware industry, a notable productivity gap has arisen, propelled by the escalating complexity of integrated circuits (ICs) as predicted by Moore's Law [1] and the limitations of traditional verification methods. Simulation-based verification, the industry standard, is highly time-consuming and accounts for about 45% to 55% of the total design cycle [2]. It requires intensive effort from engineers, making it slow and resource-heavy in hardware development. This gap is growing due to rising system complexities and the approaching limits of physical properties.

To address the widening productivity gap in hardware design, particularly in functional verification, the industry increasingly adopts innovative solutions. Among these, Large Language Models (LLMs) have emerged as a significant breakthrough. Their capability to interpret human-generated text and produce contextually relevant and logically coherent outputs across a broad spectrum of prompts marks a paradigm shift in the approach to Electronic Design Automation (EDA) tasks, including functional verification. For instance, ChipNeMo [3], a chip-design-specific assistant chatbot fine-tuned from LLaMa2, utilizes data from NVBugs, bug summaries, design sources, and documentation for verification, significantly accelerating chip production processes widely used by hardware engineers. RTLFixer [4] proposes a framework for rectifying erroneous Verilog code using existing LLM tools like GPT-3.5, augmented by Retrieval Augmented Generation (RAG) and ReAct prompting mechanisms. In parallel, LLMs have also aided in enhancing hardware security by identifying vulnerabilities through SystemVerilog Assertions (SVA) generated for Register Transfer Level (RTL) verification [5] [6] [7].

979-8-3503-7608-1/24$31.00 ©2024 IEEE

Despite the considerable advancements in employing LLMs for RTL code verification and analysis, the domain of LLM-aided design for High-Level Synthesis (HLS) has remained under-explored. HLS simplifies the design process by automating the conversion of high-level programming languages, such as C or C++, into RTL code guided by compiler directives known as pragmas [8] [9] [10] [11]. These directives are specific to HLS to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code [12]. Operating at a more abstract level, HLS presents an opportunity to significantly shorten the design and verification cycle, thereby boosting overall design productivity [13] [14]. However, there remains a notable absence of datasets geared specifically towards HLS functional verification [15] [16]. This paper seeks to bridge this gap by introducing a new dataset, the most comprehensive compilation of HLS buggy code available to date. This dataset is meticulously curated to empower researchers to evaluate and fine-tune HLS-specific verification tools, representing a pivotal advancement in harnessing LLMs for HLS code verification. The key contributions of our work include the following:

- Build a new flow that leverages LLMs for injecting either logic bugs or non-ideal pragmas into a working HLS design.
- Leveraging this new flow, we create a new dataset: an extensive compilation of over 1,500 function-level designs, each containing one of 8 realistic logic bugs or one of 9 non-ideal pragmas practices, sourced from 13 open-source HLS benchmarks. This dataset contains both the version injected with bugs and the corresponding bug-free version, serving as a valuable asset for training domain-specific LLMs that can help engineers to debug their HLS code, advancing automation in code verification.
- Design custom prompts for GPT-4 Turbo to generate EDA-standard-compliant buggy codes, together with a novel methodology combining In-Context Learning (ICL), Retrieval Augmented Generation (RAG), and Chain-of-Thought (CoT) to enhance dataset quality. This approach resulted in a 16.5% and 32.2% enhancement for the ratio of valid bug injection attempts, indicating the percentage of bug or non-ideal pragma injections passing automatic checks. Additionally, our approach achieves a 20.4% and 54.0% validity improvement for bug or non-ideal pragma injection, respectively, through a manual validation process.
- Provide insights into the effectiveness of RAG, ICL, and CoT for HLS bug injection tasks. The combination of RAG and ICL equips the LLM with contextually relevant and tailored examples to tackle HLS-specific challenges. Meanwhile, CoT shows significant effectiveness in managing complex tasks such as complicated pragma insertions.

II. Dataset Collection

During the dataset development, we highlight code issues that are particularly intractable as they cannot be detected by compilers and present significant debugging challenges during the HLS design phase.

A. HLS Designs Collection

Building upon our comprehensive effort, the dataset aggregates a wide range of synthesizable HLS applications sourced from esteemed projects including: CHStone [17], FINN [18], GNNBuilder [19], H.264 [20], HLS4ML [21], MachSuite [22], Open-Source-IPs [23], Polybench [24], Rosetta [25], Vitis HLS introductory examples [26], Vitis libraries [27], Tacle-Bench [28] and HIDA [29]. Our dataset, emphasizing function-level tasks, comprises over 1,500 HLS function-level designs from reputable sources. In constructing the dataset, we thoroughly addressed the complexities of HLS functions, including header files or function calls as shown in Fig. 3 in the appendix.

B. Bugs and Non-idealities Simulation

TABLE I: Logic bug types with corresponding description

Type	Bug Description
OOB	Out-of-bounds array access
INIT	Uninitialized variable access
SHFT	Bit shift by an out-of-bounds amount
ZERO	Variable with a nonzero value initialized to zero
INF	An infinite loop due to incorrect loop termination
MLU	Errors in manual loop unrolling, omitting an iteration
BUF	Copying from the wrong half of a split buffer
USE	Unintended sign extension

TABLE II: Non-ideality types with corresponding pragmas and description

Type	Pragma	Non-ideality Description
APT	array_partition	Array partition type mismatch
FND	array_partition	Factor not divisible
DID	array_partition	Dim incorrect dimension
DFP	dataflow	Dataflow misalignment
IDAP	interface	Incorrect data access pattern
RAMB	interface	Random access to the M_axi interface resulting in non-burst AXI read/write
SMA	interface	Scalar values mapped to array interfaces
AMS	interface	Array value mapped to scalar interfaces
MLP	pipeline	Multi-level pipelining

In this section, we have delineated imperfection types into two main categories: logic bugs and pragma non-idealities. The classification of logic bugs is based on the framework introduced by Campbell et al. [30]. Meanwhile, we present a novel categorization for pragma non-idealities, emphasizing the degradation of non-ideal pragma injections on hardware performance. The specific types of logic bugs and pragma non-idealities are detailed in Tables I and II, respectively.

III. Dataset Construction

This section explores dataset construction using advanced LLM techniques like Chain of Thought (CoT), In-Context Learning (ICL), and Retrieval-Augmented Generation (RAG). After providing an overview of these techniques, we delve into our dataset construction framework, demonstrating their application.

A. Brief Introduction to ICL, RAG and CoT

ICL, RAG, and CoT represent pivotal techniques in enhancing the performance of LLMs and mitigating the issue of hallucinations, ensuring the generation of more reliable and contextually relevant content. ICL harnesses the LLM's ability to learn directly from the examples embedded within the prompt, allowing it to adapt its responses that align closely with the specific examples of a given task [31]. RAG targets on overcoming the inherent challenges LLMs face

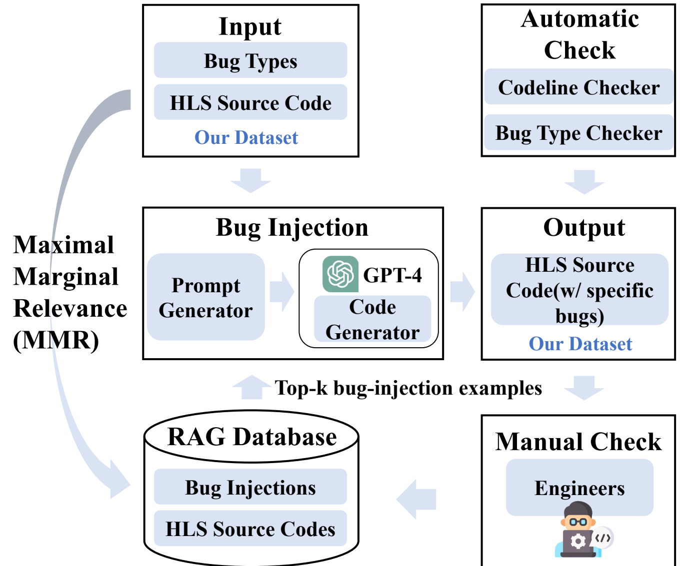


Fig. 1: Overview of Dataset Construction and Iterative Upgrade Process: The dataset is generated from and contributes to the development of the RAG database, allowing iterative enhancements through manual verification and correction of generated code with bugs.

with knowledge-intensive tasks [32]. It integrates external knowledge by dynamically retrieving relevant information during the generation process. It enriches the model's capability by integrating external knowledge sources into the generation process, dynamically retrieving and incorporating relevant information. The CoT methodology, in particular, enhances this framework by prompting the model to articulate intermediate steps or reasoning paths [33]. This approach aids in solving complex problems more effectively.

Our methodology utilizes RAG to select pertinent HLS source codes and adds the corresponding bug-injection examples to the prompt for ICL training. In our CoT approach, we employ two main strategies: integrating bug injection steps into the prompting process and prompting the LLM to not only introduce bugs but also provide commentary on the rationale behind its implementation choices.

B. Framework of Dataset Construction

Our dataset construction framework, as shown in Fig. 1, integrates RAG, ICL, and CoT methodologies. Initially, RAG is used to identify the top-k most relevant HLS source codes and associated bug-injection examples from the RAG database. These examples are then integrated into prompts using the ICL approach, with CoT methodology incorporated to enhance the prompt's structure. Upon submitting these prompts to an LLM, an automatic check is conducted to prune out invalid attempts. Further refinement is achieved through manual verification, where a sample of the outputs is examined by hardware engineers to assess the dataset's validity. Valid examples with bug description alignment are subsequently added to the RAG database for ongoing enhancement. The following sections detail each component of this framework, outlining their roles and contributions to the dataset's construction.

1) RAG Database: The RAG database forms the foundation of our work, containing a vast collection of HLS source codes and valid bug injection examples. This database is compiled through a dual approach: firstly, by incorporating bug injection generated by the LLMs and subsequently verified through automatic and manual checks, and secondly, by integrating bug injection manually curated by human engineers. The latter one is very important for the LLMs'

production on those complex bugs difficult for LLM to generate. Then the codes and bug injection will be further leveraged by RAG to do ICL prompting. In specific, our approach using algorithms to search for the most k relevant HLS source codes. Guided by Maximal Marginal Relevance (MMR), this process ensures the selection of pertinent and diverse examples, optimizing relevance and minimizing redundancy.

2) Prompt Generator: The prompt template has been meticulously crafted to include five critical sections—Context, Requirement, Steps, Examples, and Complementary Rules—to ensure the accurate generation of buggy code. Detailed explanations of these components, along with illustrative examples, are provided in Part B of the appendix.

3) Automatic Check: Automatic check first verifies the bug injection being correctly cross-referenced and aligned with the corresponding line in the original HLS code and adhering to structural rules before bug injection. These rules are designed to exclude bug injections where pragmas are introduced outside of function bodies. Even though the automatic check is passed, it does not mean that it is always a valid bug. For example, consider an attempted out-of-bound bug injections where the code line `'if (dataSize > 0 && dataSize < 1 + MIN_BLK_SIZE) rawBlockFlagStream << true;'` modified to `'if (dataSize > 0 && dataSize <= MIN_BLK_SIZE) rawBlockFlagStream << true;'` in the design inputBufferMinBlock. Although the original code line exists in the original design and it would pass the automatic check, it does not alter the condition's functionality or lead to any out-of-bound variables. Therefore, such a modification cannot be considered a valid bug in this case.

4) Manual Check: Due to the vast scale of the dataset, a comprehensive review by human engineers is infeasible, necessitating a random sampling of LLM-generated solutions. These selected bug injections undergo a rigorous manual evaluation by experts, who assess them based on their functional alignment with the described bug and whether each pragma non-ideality injection leads to performance degradation. Thus, following a positive manual check, it can ensure a 100% guarantee that the bug is valid. This methodology not only enhances the dataset's integrity but also contributes to the enrichment of our RAG database with valid and diverse cases, thereby amplifying the RAG's effectiveness.

IV. EVALUATION

A. Evaluation Metric

This subsection introduces three crucial metrics for evaluating LLMs' effectiveness in bug injection within HLS source code.

1) Bug Injection Ratio (Ratio1): This ratio evaluates LLMs' efficiency in embedding bugs into HLS codes, calculated as the number of LLMs' successful bug injection divided by their total bug injection attempts. "Successful bug injection" happens when LLMs integrate a designated bug type into HLS code. "Total bug attempts" encompass all instances where LLMs try to inject a bug.

2) Ratio of Valid Bug Injection Attempts (Ratio2): This ratio measures the precision of bug injection, calculated as the number of bugs after automatic check divided by the total number of attempts. This metric ultimately determines the dataset's final size.

3) Ratio of Successful Bug Injection (Ratio3): This metric measures the percentage of bug injections that, upon manual review, are confirmed to function as intended within the code's specific context, leading to incorrect results for logic bugs or performance degradation for pragma non-idealities.

4) Validity: To evaluate the actual validity of correct bug injections over the number of injection attempts using our sampling data, we calculate the metric using the following formula:

$\text{Validity} = \sum_{\text{bug types}}(\text{Ratio3}) / \sum_{\text{bug types}}(\text{Ratio2})$.

B. Experiments

We conducted various experiments to explore the capabilities of LLM techniques. These experiments, designed as ablation studies, focused on single-bug injection scenarios to precisely evaluate the impact of these techniques on various bug types. All experiments were performed using GPT-4 Turbo (specifically *gpt-4-0125-preview*) via OpenAI APIs [34], with the *text-embedding-ada-002* API for MMR embedding. We set the GPT-4 Turbo temperature to 0.7 mostly, using a k=4 setting for ICL. For complex bugs like RAMB and IDAP, the temperature was reduced to 0.2 and 0.4 respectively for the ICL+RAG+CoT strategy to enhance validity while reducing creativity.

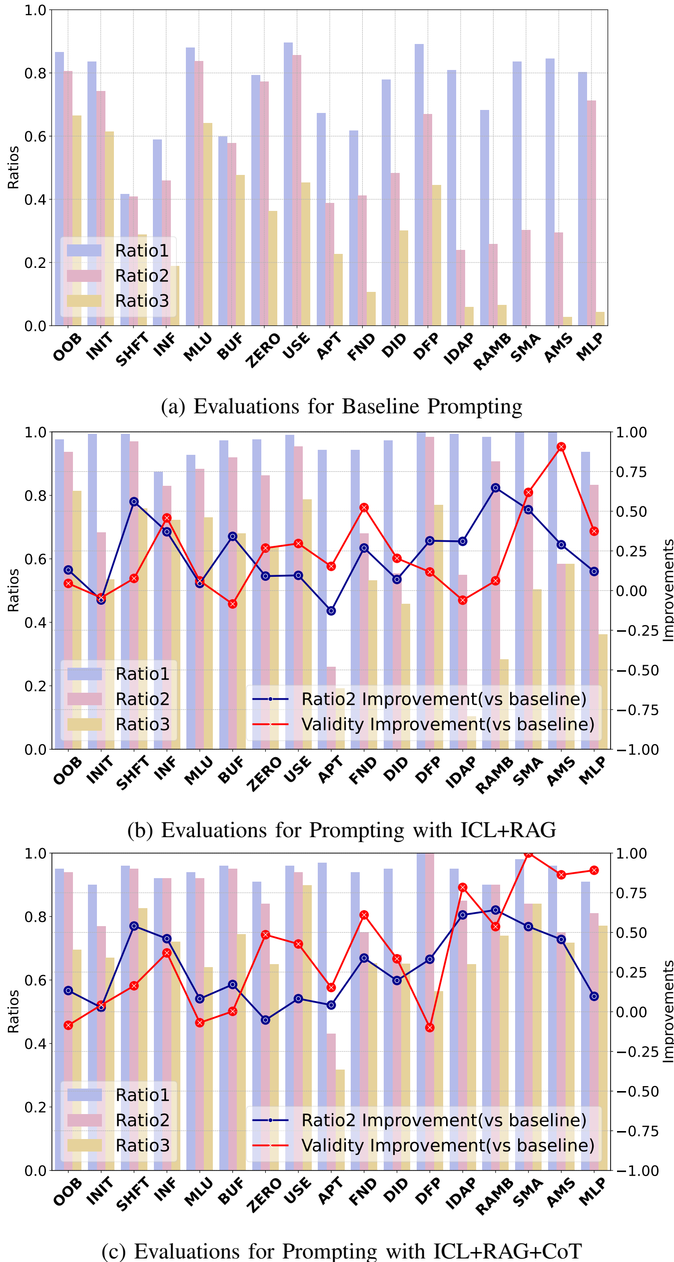
We manually reviewed approximately 450 bug injection instances in all for Ratio3 calculations. For the experimental phase using **Baseline Prompting**, we employed a brute-force approach to prompt generation, establishing a foundational framework for subsequent refinement. Meanwhile, for **Prompting with ICL+RAG**, we developed a comprehensive database with 280 cases using the ICL and RAG techniques. This database includes manually verified cases and those crafted by hardware engineers to address challenging bugs. When employing **Prompting with ICL+RAG+CoT**, we expanded the database to 459 cases and improved the approach by integrating CoT techniques, incorporating bug-injection steps in prompts, and requiring comments in both the database and bug-injection outputs. However, we observed that different types of bugs often require different handling strategies, which is why we use two distinct approaches. However, different types of bugs often require different strategies, which is why we use two distinct approaches. Adding CoT can sometimes introduce complexity that leads to overfitting or obscures simpler bug patterns, making it less effective for certain bugs. Finally, we employed hybrid bug injection strategies combining the above two tailored to different bug types based on validity results, further enhancing the quality of the dataset.

C. Analysis

1) Analysis on Prompting Technique Effectiveness: The evaluation results of injecting one bug or non-ideal pragma for each type is shown in Fig. 2 and Table III. In Figure 2, bar plots illustrate the ratios, with the red and blue lines representing the improvements in Ratio2 and Validity, respectively, over the baseline across various bug types. Table III presents an average evaluation of the different prompting strategies across all bug types. Our approach with ICL+RAG+CoT brings a 15.7% and a 52.5% validity improvement for the bug and non-ideal pragma injection, respectively, compared to brute-force prompting. Also, this approach resulted in an 18.1% and 34.1% enhancement for the ratio of valid bug injection attempts.

Integrating RAG-selected examples into the ICL framework significantly improves the LLM's understanding and problem-solving capabilities in the HLS domain. This approach boosts the logic bugs' Ratio2 by 19.7% and their Validity by 12.9%. Additionally, it enhances the Ratio2 for pragma non-idealities by 26.7% and improves their Validity by 27.5%. This innovative combination ensures that the LLM is equipped with examples that are not only relevant but also carefully tailored to address specific coding challenges, markedly enhancing its practical comprehension and problem-solving skills.

The adoption of CoT instructs the model incrementally, guiding it towards step-by-step actions to achieve reasonable outcomes. When comparing the efficacy of models prompted with ICL+RAG+CoT against those prompted with ICL+RAG, the inclusion of CoT offers a marginal improvement—less than 3%—for logic bugs in Validity. Nonetheless, CoT notably enhances the injection of bugs for pragma non-idealities, with a 7.4% increase in Ratio2 and a 25.0% enhancement in Validity. The discrepancy likely stems from the fact that the



(a) Evaluations for Baseline Prompting

(b) Evaluations for Prompting with ICL+RAG

(c) Evaluations for Prompting with ICL+RAG+CoT

Fig. 2: Comparison of the evaluation ratios among Baseline Prompting, Prompting with ICL+RAG, and Prompting with ICL+RAG+CoT

non-ideal pragma insertions are more complex and difficult than logic bug injection. CoT excels in decomposing tasks into simpler steps to achieve a reasonable result. However, its effectiveness is less apparent for simpler tasks, where the existing capabilities of LLMs saturate to tackle the problem, and the additional decomposition does not contribute to improved outcomes.

2) Task Complexity vs. LLMs' Capability: GPT-4 Turbo's proficiency in generating realistic logic bugs is significantly higher than its ability to handle pragma non-idealities without applying any technique, primarily due to its extensive exposure to C code during training. This background enables it to identify and manipulate common logic bug patterns effectively. However, its performance on pragma non-idealities is less robust, hindered by limited training on HLS pragmas and the nuanced complexity of these pragmas, including understanding their specific requirements and identifying precise injection sites.

Despite these challenges, the ICL+RAG+CoT strategy has led to a marked improvement in handling pragma non-idealities, outpacing gains seen in logic bug injection. This strategy leverages context-rich examples and external knowledge to enhance the LLM's understanding of pragma non-idealities, effectively compensating for its initial limitations. This demonstrates the adaptability of LLMs to specialized tasks when provided with sufficient and relevant context.

TABLE III: Average Evaluation on Different Prompting Strategies

Prompting Strategy	**Ratio1**	**Ratio2**	**Ratio3**	**Validity**
Baseline (Logic)	73.5%	68.3%	46.2%	67.6%
ICL+RAG (Logic)	**96.3%**	**88.0%**	70.8%	80.5%
ICL+RAG+CoT (Logic)	93.8%	86.4%	72.0%	83.3%
Hybrid (Logic)	93.3%	84.8%	**74.6%**	**88.0%**
Baseline (Pragma)	77.1%	41.8%	14.2%	33.9%
ICL+RAG (Pragma)	97.5%	68.5%	42.1%	61.4%
ICL+RAG+CoT (Pragma)	**96.0%**	**75.9%**	**65.6%**	86.4%
Hybrid (Pragma)	94.1%	74.0%	65.0%	**87.9%**

V. Future Work

Our dataset presents a promising platform for evaluating the proficiency of existing LLMs in HLS bug localization. Our research outlines three primary directions for future exploration: Firstly, multiple challenges are discovered in using LLMs to inject certain types of bugs. Notably, bugs related to operator precedence misunderstandings are difficult for human evaluators to assess due to sparse code patterns and the inherent complexity of creating representative examples from scratch. Additionally, misunderstandings and pragma non-idealities (e.g., inner loops not fully-unrolled) are difficult in bug injection, due to either the sparsity of examples in existing codebase or the difficulty of creating representative examples from scratch. These categories need further investigation to develop refined inclusion methodologies. Secondly, developing an automated performance estimation workflow could expedite the identification and elimination of pragma non-idealities, thereby improving the dataset's quality and quantity. Finally, we aim to fine-tune an open-source LLM by using the dataset. This model would not only identify anomalies but also offer potential fixes, serving as a powerful tool for hardware engineers. Our approach could further enhance the debugging process, leveraging the specificity of the dataset to address the unique challenges of HLS code development.

VI. Conclusion

The introduction of the dataset marks a pivotal advancement in utilizing LLMs for HLS bug localization to significantly enhance debugging and verification processes. By employing GPT-4 Turbo and integrating methodologies of ICL, RAG and CoT, this work demonstrates a notable improvement in injecting compiler-challenging intent bugs, achieving an 84.8% ratio for valid bug injection attempts and an 88.0% validity — an increase of 16.5% in valid attempts and 20.4% in the valid injection ratio. Moreover, the work records an advancement in addressing pragma non-idealities, with a 74.0% ratio for valid injection attempts and an 87.9% validity — an increase of 32.2% for attempts and 54.0% for validity. This work not only demonstrates the benefits of the dataset but also outlines the potential future enhancements of the dataset and its applications in fine-tuning LLMs for HLS bug localization and correction.

Acknowledgements

This work was supported in part by the Semiconductor Research Corporation (SRC) under the grant number 2023-CT-3175, and by the AMD Center of Excellence at UIUC. We would also like to extend our appreciation to Xiaofan Zhang from Google for the valuable discussions and insights provided.

References

[1] R. R. Schaller, "Moore's law: past, present and future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[2] H. Foster, "Part 8: The 2020 wilson research group functional verification study," https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/, 2021.

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

[4] Y. Tsai *et al.*, "Rtlfixer: Automatically fixing rtl syntax errors with large language models," *arXiv preprint arXiv:2311.16543*, 2023.

[5] Orenes-Vera *et al.*, "Using llms to facilitate formal verification of rtl," *arXiv e-prints*, pp. arXiv–2309, 2023.

[6] R. Kande *et al.*, "Llm-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.

[7] X. Meng *et al.*, "Unlocking hardware security assurance: The potential of llms," *arXiv preprint arXiv:2308.11042*, 2023.

[8] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *2011 9th IEEE International Conference on ASIC*, 2011, pp. 1102–1105.

[9] R. Kastner *et al.*, "Parallel programming for fpgas," *arXiv preprint arXiv:1805.03648*, 2018.

[10] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xpilot: A platform-based behavioral synthesis system," *SRC TechCon*, vol. 5, p. 54, 2005.

[11] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "Fcuda: Enabling efficient compilation of cuda kernels onto fpgas," in *2009 IEEE 7th Symposium on Application Specific Processors*. IEEE, 2009, pp. 35–42.

[12] A. M. Devices, "Vitis high-level synthesis pragmas guide," https://docs.amd.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas, 2023.

[13] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 741–755.

[14] A. Papakonstantinou, Y. Liang, J. A. Stratton, K. Gururaj, D. Chen, W.-M. W. Hwu, and J. Cong, "Multilevel granularity parallelism synthesis on fpgas," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 178–185.

[15] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "Fpga hls today: successes, challenges, and opportunities," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–42, 2022.

[16] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *2011 9th IEEE International Conference on ASIC*. IEEE, 2011, pp. 1102–1105.

[17] Y. Hara *et al.*, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2008, pp. 1192–1195.

[18] Y. Umuroglu *et al.*, "Finn: A framework for fast, scalable binarized neural network inference," in *Proc. of FPGA*, 2017.

[19] S. Abi-Karam *et al.*, "GNNBuilder: An Automated Framework for Generic Graph Neural Network Accelerator Generation, Simulation, and Optimization," *arXiv preprint arXiv:2303.16459*, 2023.

[20] X. Liu *et al.*, "High level synthesis of complex applications: An H. 264 video decoder," in *Proc. of FPGA*, 2016.

[21] F. Fahim *et al.*, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," *arXiv preprint arXiv:2103.05579*, 2021.

[22] B. Reagen *et al.*, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proc. of IISWC*, 2014.

[23] X. Liu *et al.*, "HLS based Open-Source IPs for Deep Neural Network Acceleration," https://github.com/DNN-Accelerators/Open-Source-IPs, 2019.

[24] J. Karimov *et al.*, "Polybench: The first benchmark for polystores," in *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence: 10th TPC Technology Conference, TPCTC 2018, Rio de Janeiro, Brazil, August 27–31, 2018, Revised Selected Papers 10*, 2019.

[25] Y. Zhou *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proc. of FPGA*, 2018.

[26] Xilinx, "Vitis-HLS-Introductory-Examples," https://github.com/Xilinx/Vitis-HLS-Introductory-Examples, 2023.

[27] Xilinx, "Vitis libraries," https://github.com/Xilinx/Vitis_Libraries, 2019.

[28] Tacle, "Tacle Bench," https://github.com/tacle/tacle-bench, 2017.

[29] H. Ye *et al.*, "Hida: A hierarchical dataflow compiler for high-level synthesis," *arXiv preprint arXiv:2311.03379*, 2023.

[30] K. A. Campbell, "Robust and reliable hardware accelerator design through high-level synthesis," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2017.

[31] S. Min *et al.*, "Rethinking the role of demonstrations: What makes in-context learning work?" *arXiv preprint arXiv:2202.12837*, 2022.

[32] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[33] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[34] OpenAI, "Openai models api," https://platform.openai.com/docs/models, 2024.

Appendix

A. Benchmark-to-design Conversion

Figure 3 illustrates the conversion process from benchmarks in HLS projects to function-level designs. Each function is treated as an individual design, and the file includes all functions that the design invokes.

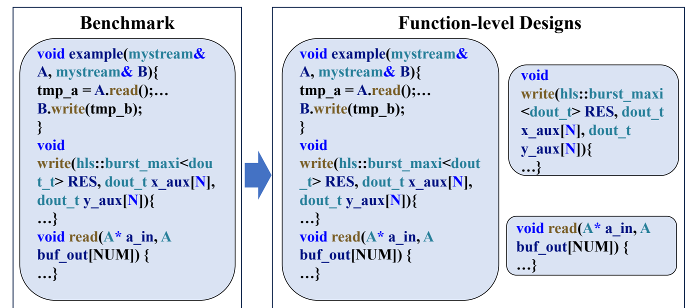


Fig. 3: An example of conversion from original benchmarks to the function-level designs we used for bug injection.

B. Prompting Examples

Figure 4 illustrates the methodology for inducing an Out-of-Bounds (OOB) error within the "gemm_4096" function, employing distinct color codes to delineate the advancement of the prompting mechanism.

1) Baseline Prompting: This approach utilizes the segment in blue boxes, focusing on a straightforward prompt. The key elements include Context, Requirement, and Complementary Rules.

2) Prompting with ICL+RAG: This technique expands upon the baseline by incorporating both the blue and orange boxes, integrating ICL and RAG methodologies for enhanced context understanding. It adds the Steps part and another complementary rule compared to baseline prompting.

3) Prompting with ICL+RAG+CoT: The most comprehensive strategy, this method amalgamates the blue, orange, and green boxes. It includes ICT, RAG, and Chain of Thought (CoT) prompting, offering a multi-faceted approach to error introduction. It includes the Example part in addition to what is used in ICL+RAG prompting.

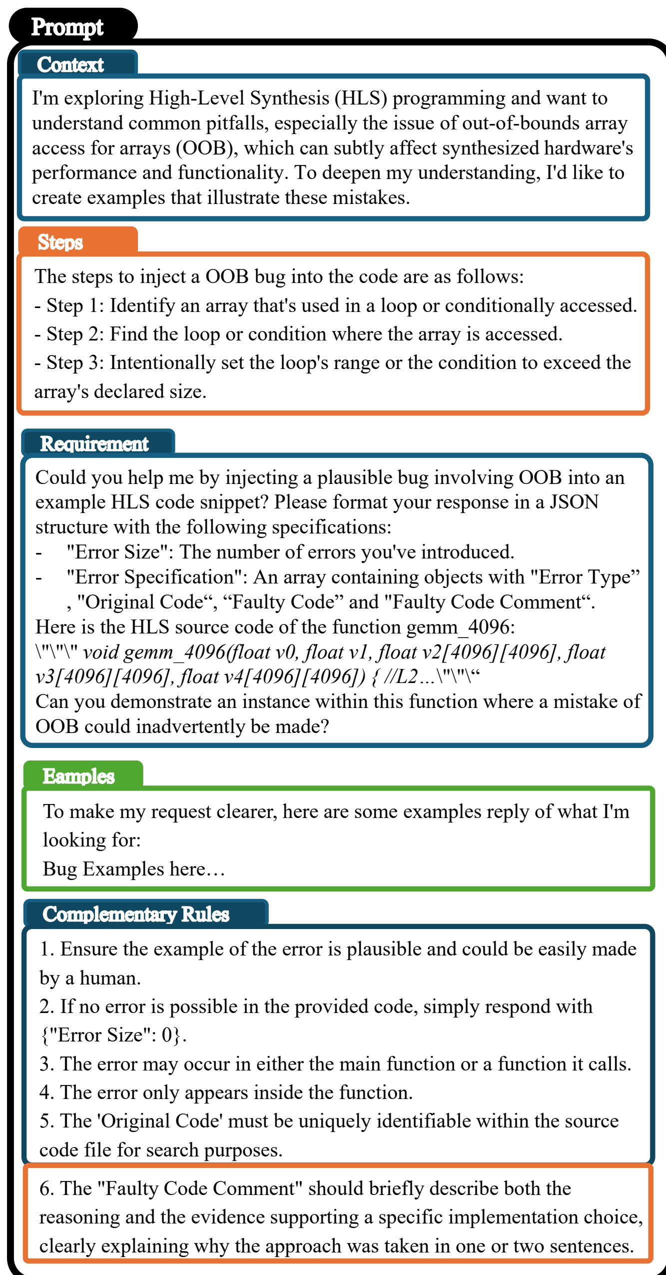


Fig. 4: An illustration showcasing the procedure for directing GPT-4 to introduce an Out-Of-Bounds (OOB) error into the 'gemm_4096' source function. The color-coded representation signifies the progressive sophistication of the prompting mechanisms employed, starting from basic to more complex, reflective of an iterative refinement in error injection techniques.