HDLxGraph: Bridging Large Language Models and HDL Repositories via HDL Graph Databases

Pingqing Zheng
University of Minnesota, Twin Cities
Minneapolis, MN, USA
pingqingzheng13@gmail.com

Jiayin Qin
University of Minnesota, Twin Cities
Minneapolis, MN, USA
qin00162@umn.edu

Fuqi Zhang
University of Minnesota, Twin Cities
Minneapolis, MN, USA
zhan7076@umn.edu

Niraj Chitla
University of Minnesota, Twin Cities
Minneapolis, MN, USA
chitl013@umn.edu

Zishen Wan
Georgia Institute of Technology
Atlanta, GA, USA
zishenwan@gatech.edu

Shang Wu
Northwestern University
Evanston, USA
swu@u.northwestern.edu

Yu (Kevin) Cao
University of Minnesota, Twin Cities
Minneapolis, MN, USA
yucao@umn.edu

Caiwen Ding
University of Minnesota, Twin Cities
Minneapolis, MN, USA
dingc@umn.edu

Yang (Katie) Zhao
University of Minnesota, Twin Cities
Minneapolis, MN, USA
yangzhao@umn.edu

***Abstract*—Retrieval Augmented Generation (RAG) is an essential agent for Large Language Model (LLM) aided Description Language (HDL) tasks, addressing the challenges of limited training data and prohibitively long prompts. However, its performance in handling ambiguous queries and real-world, repository-level HDL projects containing thousands or even tens of thousands of code lines remains limited. Our analysis demonstrates two fundamental mismatches, structural and vocabulary, between conventional semantic similarity-based RAGs and HDL codes. To this end, we propose HDLxGraph, the first framework that integrates the inherent graph characteristics of HDLs with RAGs for LLM-assisted tasks. Specifically, HDLxGraph incorporates Abstract Syntax Trees (ASTs) to capture HDLs' hierarchical structures and Data Flow Graphs (DFGs) to address the vocabulary mismatch. In addition, to overcome the lack of comprehensive HDL search benchmarks, we introduce HDLSearch, an LLM-generated dataset derived from real-world, repository-level HDL projects. Evaluations show that HDLxGraph improves search, debugging, and completion accuracy by 12.04%/12.22%/5.04% and by 11.59%/8.18%/4.07% over state-of-the-art similarity-based RAG and software-code Graph RAG baselines, respectively. The code of HDLxGraph and HDLSearch benchmark are available at https://github.com/UMN-ZhaoLab/HDLxGraph.**

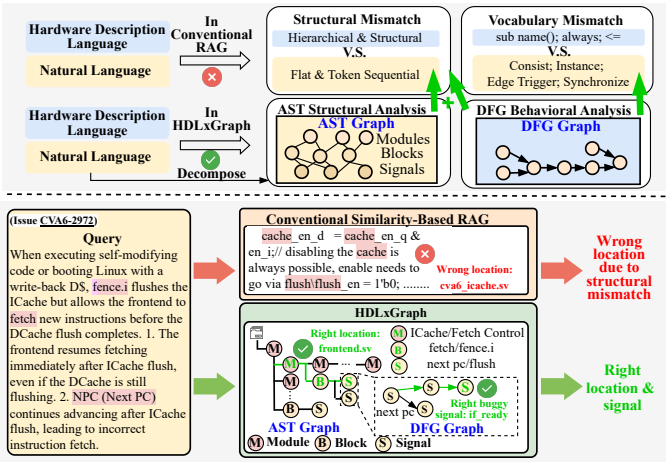

Fig. 1. (Top) An illustration of two fundamental mismatch between HDL and natural language in conventional RAG, including *structural* and *vocabulary* mismatches. And (Bottom) a demonstration of HDLxGraph's efficiency in bridging these mismatches by incorporating graph information, using an HDL debugging example for a CVA6 RISC-V HDL implementation [37].

I. Introduction

Recent advances in Large Language Models (LLMs) for software language understanding and generation [20], [34] have inspired efforts to extend their capabilities to facilitate Hardware Description Language (HDL) code designs. Prior works have demonstrated LLMs' potential in generating [31], [33], [39] and debugging HDL code [25], [28], [36]. However, LLM performance in HDL-related tasks remains hindered by limited training data and degradation caused by long prompts. To address these issues, researchers have integrated *Retrieval-Augmented Generation (RAG)*, which retrieves relevant HDL fragments from high-quality HDL codes to supplement knowledge gaps and reduce prompt length [10], [25].

Despite their potential, existing RAG approaches for HDL primarily rely on semantic similarity-based retrieval, which yield low recall when handling intricate and ambiguous queries or large and complex HDL repositories with thousands or even tens of thousands of code lines. Through analyzing semantic similarity-based RAGs and HDL codes, we find **two fundamental mismatches** between them (as illustrated in Figure 1 (top)). First, natural language queries are flat and sequential while HDL designs have a hierarchical structure

with multi-level entities, i.e., modules, blocks, and signals. We summarize this as the *structural mismatch*. In addition, cross-file and cross-module analysis is required to capture this hierarchical structure. Second, domain-specific HDL terminologies, such as the operators and key words, differ from their natural language descriptions. This *vocabulary mismatch* further prevents conventional RAGs from effectively learning HDL code semantics.

Inspired by recent advances in *Graph Retrieval-Augmented Generation* (Graph RAG) [7], [12] and the inherent graphic characteristics of HDL codes, we propose integrating HDL-specific graph structures into HDL-specific RAGs to address the aforementioned mismatches. To this end, we introduce HDLxGraph, a novel hybrid graph-enhanced RAG framework, which incorporates two HDL-specific graphs: Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs). Using ASTs, we parse a HDL repository, even with tens of thousands of code lines, into a **code graph view** containing multi-level entity hierarchy, including modules, blocks and signals. By abstracting DFGs from the signal-level connections in the ASTs, we further construct a precise **hardware signal graph view** representing the signal-level data flow within the hardware design.

We use an HDL debugging example for a CVA6 RISC-V implementation, which contains over 30 modules [37], to demonstrate the advantages of HDLxGraph (shown in Figure 1 (bottom)). Conventional similarity-based RAGs often mislead retrieval to `cva6_icache.sv` due to the frequent presence of keywords containing "Cache" in the natural language query. In contrast, HDLxGraph can address this issue through AST- and DGS-integrated retrieval: 1) The AST explicitly reveals the hierarchical hardware structure. When analyzing natural language queries, HDLxGraph can align the flat and sequential descriptions with the hierarchical HDL structure, even if the keywords indicating hierarchy appear only once or out of structural order. Consequently, HDLxGraph accurately locates the relevant file `frontend.sv`. 2) To identify and correct the bug, HDLxGraph then traces through the DFG from the queried buggy signal description to the actual buggy signal, `if_ready` in this example, and guides the HDL correction for debugging.

Besides developing HDLxGraph, we also identify the lack of comprehensive HDL code search benchmarks that include question–answer pairs derived from repository-level HDL projects. To address this gap, we further extend HDLxGraph to construct a new benchmark, dubbed HDLSearch.

In summary, our key contributions are as follows:

- We propose HDLxGraph, a novel RAG framework that integrates HDL-specific ASTs and DFGs to address the inherent mismatches between conventional semantic similarity-based RAGs and HDL codes. To the best of our knowledge, HDLxGraph is **the first** framework to integrate HDLs' inherent graph structures with RAGs.
- HDLxGraph is a hybrid graph RAG framework with enhanced understanding on HDLs' hierarchical structures and hardware data flows. Specifically, HDLxGraph aligns flat and sequential natural language queries across hierarchical structures containing multi-level entities in the ASTs. Building on this hierarchical alignment, HDLxGraph overcomes the vocabulary mismatch and traces through the DFGs to locate signals for downstream dubugging and completion tasks.
- Based on HDLxGraph, we further construct a new LLM-generated dataset for HDL code search, called HDLSearch, which derives query benchmark from real-world repository-level HDL projects, to address the gap of insufficient search datasets for HDL codes.
- By integrating HDLxGraph with three commonly-used LLMs of different scales and coding abilities, we demonstrate its adaptability on code search, debugging, and completion tasks. HDLxGraph improves the accuracy of search, debugging, and completion tasks by 12.04%/12.22%/5.04% and 11.59%/8.18%/4.07% over state-of-the-art similarity-based RAG and software-code Graph RAG, respectively.

II. Preliminaries

A. LLM-aided HDL Tasks

Generation. Although LLMs excel at generating simple HDL designs, they still struggle with complex repository-level chip designs [9], [18], [30], [38]. For example, state-of-the-art (SOTA) works [9], [10], [32] often rely on predefined templates or customized RAG datasets created by human experts, where LLMs primarily fill in fixed reference code segments rather than performing general HDL generation.

Debugging. Though LLMs perform well on simple debugging tasks, they fail to achieve high coverage in more complex fixing tasks [2], [8], [26], [35]. The study in [26] integrates LLMs with RAG to patch functional HDL bugs. However, it still depends on manually defined error types, reflecting the limitations of current approaches in handling real-world, complex bug fixing.

Search. Precise code search serves as the foundation for both HDL generation and debugging. Although no prior work has directly targeted HDL search, recent studies have explored LLMs' potential in HDL summarization [39], a preliminary step toward effective HDL search. However, these approaches overlook the inherent hierarchical structure of HDL, limiting their applicability to precise code search tasks.

B. Graph Retrieval-Augmented Generation for Code Tasks

Graph RAG leverages structured knowledge graphs to enhance the complex reasoning and context-awareness capabilities of RAG, and has demonstrated promising performance in various software code tasks [7]. However, despite syntactic similarities between HDLs and software languages, HDLs exhibit fundamental differ in abstraction hierarchy and behavioral modeling [1], [6], [19], rendering existing software-code Graph RAG suboptimal for HDL-related tasks.

First, at the abstraction level, HDLs are specification languages rather than general-purpose programming languages.

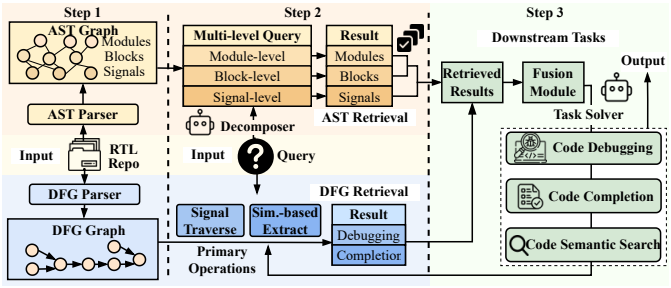

Fig. 2. The overview of our proposed HDLxGraph framework.

To facilitate circuit optimization, HDL designs typically require explicit control over hardware components, with the module serving as the primary design unit. In contrast, software languages offer multi-level abstractions (e.g., functions, classes, packages). Second, in terms of behavioral modeling, HDLs construct concurrently executed blocks using always blocks and model combinational signal propagation using assign statements (using Verilog as an example), both of which diverge from software's sequential control flow. As a result, Graph RAGs tailored for software fail to capture the true structure and semantics present in HDL code.

To integrate HDL-specific abstraction structures and behavioral patterns, we incorporate HDL-dedicated graph characteristics and customize the retrieval process accordingly to enable precise HDL queries.

C. Benchmarks/Datasets for LLM-aided Tasks

For HDL code generation benchmarks, RTLLM [21] consists of 30 designs; and VerilogEval [16] presents an evaluation dataset consisting of 156 problems from HDLBits. For HDL code debugging benchmarks, LLM4SecHW [8] contains a repository-level bug localization and repair sets from the version control data in Github; RTLFixer [32] introduces a Verilog syntax debugging dataset, derived from VerilogEval [16]; and CirFix [3] includes a repair benchmark with testbenches.

No existing benchmark has been established for the HDL search, which is an essential step for downstream tasks such as generation and debugging. Therefore, we propose HDLSearch, the first benchmark for HDL code search, which derives query benchmark from real-world repository-level HDL projects.

III. Methodology

Figure 2 illustrates the overview and workflow of our proposed HDLxGraph framework, which consists of three steps: 1) Graph Database Preparation, 2) Multi-level Retrieval, and 3) Downstream Task Completion. Beginning with **Step 1**, we extract ASTs and DFGs from the input code repositories through the AST and DFG parsers, then store HDL entities and relationships as nodes and edges in a graph database (see Section III-A). In **Step 2** (see Section III-B), HDLxGraph utilizes an LLM called Decomposer in AST retrieval to extract the input query into structural levels, which are later sent to pre-defined searching paradigms to retrieve relevant fine-grained code snippets. Additionally, code debugging and completion tasks trigger DFG retrieval in parallel to narrow the search space (for code debugging) or enable similarity matching between incomplete and complete code snippets (for code completion). **Step 3** utilizes LLMs to fuse the retrieved code snippets for code debugging, completion, or search, which further demonstrates the generality of our framework (see Section III-B). In addition, due to the lack of dedicated code search benchmarks in HDL repositories, we build a new benchmark, HDLSearch, using an automated benchmarking build pipeline. Details of benchmark generation are presented in Section III-C.

A. Graph Database Preparation

As shown in **Step 1** of Figure 2, the HDLxGraph RAG framework begins with an off-line graph database construction. Without losing representability, we focus on Verilog in our implementation. Please note that, although different HDLs have different syntactic properties, they share the same three-level structural abstraction as `module` → `block` → `signal` in Verilog. Specifically, we use an AST to support the **code graph view** that emphasizes multi-level structural relationships in HDL, and a DFG to facilitate the **hardware graph view** focusing on signal flow reflecting circuit topology, providing a comprehensive and tailored representation of the HDL repository. The AST graph incorporates node types such as `MODULE`, `BLOCK`, and `SIGNAL` connected through `CONTAINS` and `INSTANTIATE` edge types, whereas the DFG graph introduces `TEMP` nodes alongside `SIGNAL` nodes, connected via `FLOWS_TO`, `TRUE`, `FALSE`, and `COND` edges. Figure 3 demonstrates a toy example of Verilog source code, its corresponding nodes and edges, and the constructed graph database. When constructing the entire graph database, we perform the following three sub-steps:

1) Parsing. The graph database construction begins with analyzing individual HDL file in the repository using a Verilator-based [29] AST and DFG parser. For AST parsing, we extract the cross-level dependancy information of `MODULE`, `BLOCK`, and `SIGNAL` from each Verilog file to represent the fine-grained hierarchical code structure. Note that block level (always, assign, initial) here represent behavioral abstraction at the register-transfer level, defining concurrent hardware operations. Concurrently, we generate the hardware signal flow for DFG parsing, which characterizes the transmission and interaction between signals. The DFG graph incorporates both the signal directions and the dependency relationships between signals, reflecting the functionality and processing

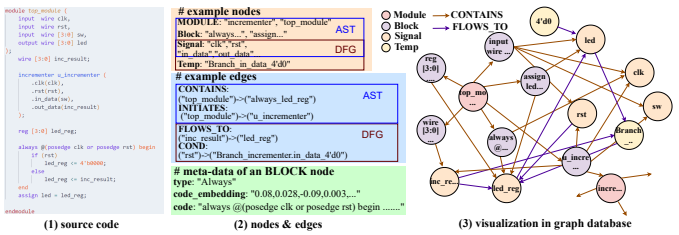

Fig. 3. Visualization of an example in the graph database.

data flow of a circuit. This multi-granularity representation enables our database to store both the code structure and the hardware behavior of a single HDL file, thereby facilitating a comprehensive graph abstraction of the HDL, as shown in Figure 3 (2) and (3).

2) Meta-data Generation. After the parsing of graph data, we generate embeddings for nodes (both `MODULE` and `BLOCK`) via code encoding to facilitate semantic search. These embeddings and parser-extracted attributes (e.g., block type, code) are stored as node meta-data, as illustrated by the meta-data on the nodes and edge types in Figure 3 (3). We use CodeT5+ [34], a SOTA code LLM model, to directly generate the embeddings for our parsers, thereby eliminating the need for intermediate description generation.

3) Cross-file Relationship Construction. Finally, we address the absense of cross-file relationship, which is the module `INSTANTIATE` relationships. We search for the module node with same name recorded in the meta-info instance block to establish the cross-file and cross-module relationships.

The developed graph database provides a multi-level code exploration spanning from module-level abstractions to signal-level implementations, thereby positioning our HDL graph database as a extensible framework for multiple downstream tasks thanks to the modular fashion of database schema management.

B. Multi-level Retrieval for Downstream Task Completion

Multiple-level Retrieval. In real-world hardware project issues, user queries always contain rich contextual cues, such as module names, functional descriptions, and sometimes brief code snippets, offering hints for retrieval. Therefore, user queries can be used to extract multi-level structural information and then guide the following multiple-level AST retrieval. In addition, signal-level flow through DFG retrieval is adopted for code completion and debugging tasks.

AST Retrieval. HDLxGraph constructs a hierarchical representation of HDL codebases through an AST-based graph, enabling multi-level HDL retrieval as depicted in Figure 4. For AST retrieval, we follow three sub-steps:

1) Query Decomposition. HDLxGraph employs an LLM, called *Decomposer*, to decompose the original query into three abstraction levels: module, block, and signal, thereby extracting structural information. It supports intricate queries from various downstream tasks such as: `'Find some certain blocks under a certain module'` in Search, or `'Some functions in some certain modules have led to the following errors ...'` Therefore, we obtain multi-level queries which have captured inherent structural information in the original query.

2) Top-k Selection and Filtering. Leveraging Verilog's inherent three-level abstraction `module` → `block` → `signal`, HDLxGraph first retrieves top-k candidate modules and blocks which have the highest similarity scores with the decomposed query in corresponding levels based on semantic matching, then filters valid module-block pairs through containment relationships. To facilitate precise code retrieval in

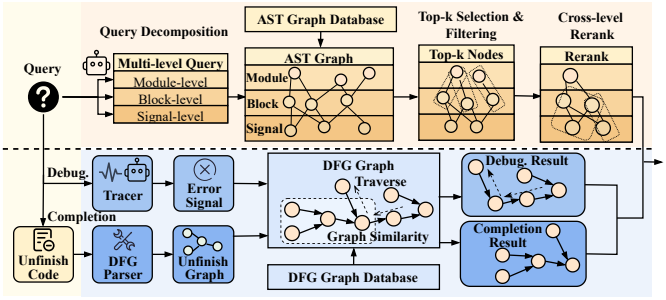

Fig. 4. Flow of multi-level retrieval containing AST and DFG retrieval.

different levels, a suite of retrieval APIs is introduced. Since we select Neo4j as the graph database, the query APIs are written in Cypher to interact with the database.

3) Cross-level Rerank. Finally, we rerank results by averaging the similarity scores of module-block pairs containing each signal, since signal-level queries lack sufficient context for direct scoring. Therefore, HDLxGraph extracts all filtered module-block pairs that contain the signal and computes their average similarity scores as the signal-level similarity score so that we prioritize signal with the highest similarity score to be the retrieved signal. This hierarchical approach ensures fine-grained retrieval of HDL's structural information across multiple abstraction layers while maintaining compatibility with similarity-based semantic analysis, balancing precision and scalability in hardware database exploration.

DFG Retrieval. The DFG captures signal-level variables and their relationships, making it especially useful for signal-sensitive downstream tasks. In this work, we leverage it to enhance code completion and debugging, as shown in Figure 2. There are two primary operations for DFG retrieval of different tasks, which are *Signal Traverse* and *Similarity-based Extract*:

1) Debugging. For debugging tasks, if a signal mismatch is detected, the debugging process can iteratively traverse the DFG upstream with *Signal Traverse* operation from the error signal, inspecting each node (e.g., operators, multiplexers, or instance outputs) to identify where the dataflow diverges from expected behavior. This approach guides LLM debugging by focusing only on the subgraph directly influencing the problematic signal, filtering out irrelevant code regions. By extracting the immediate upstream nodes and their associated code blocks, the system can find a concise, context-rich error candidate set.

2) Completion. While we want to retrieve the similar code with the unfinished code, some reference code may look different but still have similar functionality because the hardware (i.e., dataflow) described is very similar. Graph embedding offers a viable approach for Verilog code completion by translating code's structural and semantic relationships into a unified mathematical framework. By leveraging GraphSAGE [13], these graphs are compressed into low-dimensional vector representations that preserve contextual patterns, such as recurring HDL constructs (e.g., finite state machines, pipelined operations) or common coding idioms (e.g., non-blocking assignments in clock-driven blocks). This allows real-time

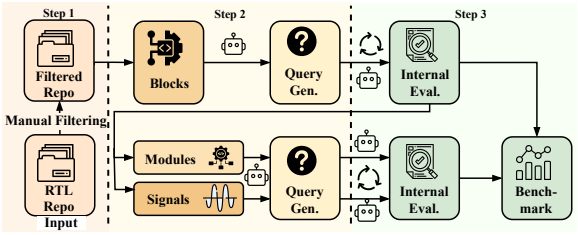

Fig. 5. HDLSearch benchmark generation flow.

retrieval of relevant patterns from large codebases while adhering to Verilog-specific constraints.

C. HDLSearch Benchmark

While existing HDL code benchmarks inherently require HDL code search capabilities to evaluate task performance, they were not explicitly designed to assess these search capabilities in isolation. However, manually creating an expert-annotated benchmark is time-consuming and labor-intensive. Therefore, we propose an automated benchmark build pipeline that can automatically analyze, divide, and pair documentation texts and code snippets and construct a benchmark, dubbed HDLSearch. As shown in Figure 5, the benchmark pipeline contains three main steps:

1) Manual Filtering. Our corpus originates from RTL-Repo [4], a collection of publicly accessible GitHub repositories specializing in HDLs. Some HDL projects lack structured documentation and standardized code organization. Therefore, we first implement a manual filtering and select 10 representative repositories at different difficulty levels, ranging from educational FPGA projects, interconnection protocols to commercial CPUs.

2) Query Generation. Adopting a hierarchical framework where **block serves as the fundamental level**, we implement a multi-stage generation process. Initial functional block descriptions are first generated, then systematically propagated through two parallel pathways, which are 1) **Signal-level annotation:** through contextual information, the semantics of a functional block can be inherited by its associated signals, thereby effectively annotating these signals with specific functionalities, and 2) **Module-level abstraction** by designing a set of explicit and tailored prompts for the LLMs, we enable it to analyze and summarize the interactions among individual functional blocks as a module-level description. This dual-path flow ensures consistent semantic alignment between fine-grained signal behaviors and coarse-grained module operations. With all descriptions finished, repo-specific information such as module and signal's names are removed to generate a relatively ambiguous query.

3) Benchmark Refinement. To further ensure benchmark validity, we employ an iterative refinement process using templated instructions. Through multiple rounds of evaluation and regeneration, we gradually remove unsuitable queries and align the LLM-generated query outputs with practical requirements till it reaches the defined termination count K.

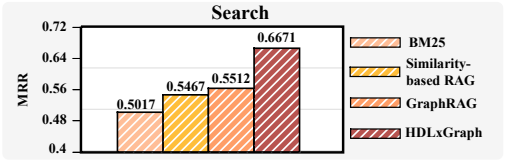

Fig. 6. HDL search MRR comparison with baselines.

IV. Experiments

A. Experimental Configuration and Platforms

We evaluate HDLxGraph from three perspectives: 1) effectiveness over traditional retrieval methods across code search, debugging, and completion; 2) comparison with SOTA software-code Graph RAG baselines; and 3) ablation of key components. We test HDLxGraph with three LLMs of varying scales: Claude-3.5-Sonnet [5] (large, strong coding ability), Qwen2.5-Coder-7B [14] (medium, coding-oriented model), and LLaMA-3.1 [11] (small, general-purpose). All experiments use top-p = 1.0, temperature = 0.7, and are conducted on a 2×A6000 Linux server with 10 independent runs per task for statistical robustness.

B. Code Semantic Search

Benchmark: Considering the absence of benchmarks for HDL-specific code search, we use our proposed HDLSearch (see Section III-C) as the benchmark with termination count $K = 7$ when generation. The generated benchmark comprises 50 module-level, 100 block-level, and 200 signal-level queries, with 6,300 code blocks from 10 repositories serving as distractor, i.e., retrieval scope. The evaluation focuses on block-level retrieval, which serves as a fundamental step to other downstream tasks such as debugging and completion.

Metric: We adopt the widely-used mean reciprocal rank (MRR) in RAG as the metric, which assesses whether the method returns correct results within the top-ranked outputs.

Baselines: We compare HDLxGraph against two traditional similarity-based RAG methods, BM25 [27] and CodeT5+ embeddings [34], as well as the SOTA software-code GraphRAG method proposed by Microsoft [7].

Evaluation Results: As shown in Figure 6, HDLxGraph achieves superior search performance across all 100 block-level queries than the similarity-based and software-code GraphRAG baselines. Notably, HDLxGraph improves the average MRR by 12.04% over the similarity-based baseline, whereas the sofware-code GraphRAG baseline achieves only a marginal improvement of 0.82%. This highlights the necessity of incorporating HDL-inherent structural hierarchy (AST) and data dependencies (DFG) into RAG, enabling fine-grained retrieval and alignment with underlying hardware semantics compared to general Graph RAG methods.

C. Code Debugging

Benchmark: We choose the mor1kx repository [24], a repository-level debugging challenge in the SOTA LLM4SecHW [8] benchmark, for our debugging evaluation.

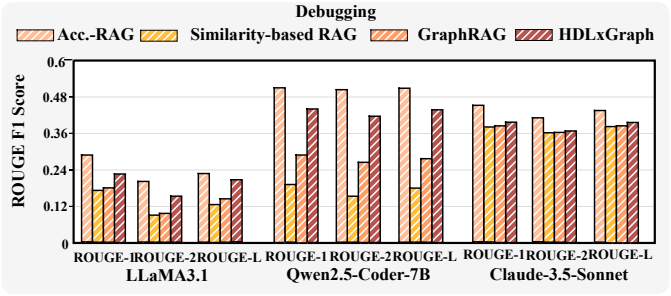

Fig. 7. HDL debugging comparison with baselines.

The mor1kx repository [24] is an OpenRISC processor IP core with 5 git commit SHAs covering various debugging issues.

Metric: We choose ROUGE-N F1 score [15] as the evaluation metric, which refers to the direct N-gram overlap between a prediction and a reference word considering precision and recall. The parameter N can be set to 1, 2 and L, corresponding to matching at unigram, bigram, and longest common subsequence gram, respectively.

Baselines: We compare our framework with three RAG strategies: the accurate-RAG debugging strategy, denoted as "Accurate-RAG", which relies on human effort to extract the exact buggy code segments to be modified as a theoretical top-tier RAG baseline, CodeT5+@ [34], denoted as "Similarity-based RAG", which represents the conventional similarity-based RAG approach, and "GraphRAG", which represents Microsoft's software-code GraphRAG.

Evaluation Results: As illustrated in Figure 7, HDLxGraph achieves consistently higher scores than both the similarity-based RAG and software-code Graph RAG baselines across all scenarios, with performance approaching that of "Accurate-RAG". Notably, when using less powerful LLaMA 3.1 and Qwen2.5-Coder-7B models, HDLxGraph attains an average improvement of 8.18% in ROUGE-1, ROUGE-2, and ROUGE-L metrics over the best software Graph RAG baseline, demonstrating HDLxGraph's generalizability even under constrained hardware resources.

D. Code Completion

Benchmark: We evaluate code completion capabilities using VerilogEval-Human v2 [17] with RTLLM [22] as a reference implementation. To avoid dataset contamination, we manually remove similar examples between the two datasets.

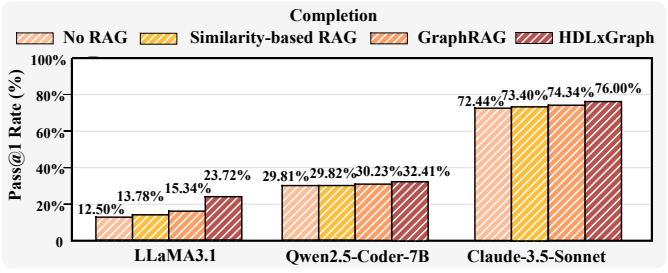

Fig. 8. HDL completion Pass@1 comparison with baselines.

TABLE I
ABLATION STUDY ON HDLXGRAPH COMPONENTS

Toolset Setting	Code Search MRR Score	Code completion Pass@1 Rate(%)	Code debugging ROUGE L F1
Ours (Full toolset)	0.6671	23.72	0.205
w/o AST analysis	0.1342	15.12	0.1438
w/o DFG behavior	0.6312	22.54	0.1561

Metric: We apply Pass@k metrics [23] to assess the generation pass rate. $Pass@1$ is used in our experiment.

Baselines: We compare HDLxGraph against three baselines: direct LLM completion without RAG, the similarity-based RAG using CodeT5+, and the software-code GraphRAG.

Evaluation Results: As shown in Figure 8, HDLxGraph consistently improves $Pass@1$ accuracy by 3-10% across various LLMs. While our evaluation framework operates at module granularity rather than full repository scope, we strategically employ the RTLLM [22] codebase as a RAG corpus, thereby maintaining repository-level evaluation. The higher accuracy suggests HDLxGraph's generalizability across different abstraction levels, highlighting that structural code understanding significantly benefits completion tasks, even at sub-repository granularity.

E. Ablation Study

To evaluate the effectiveness of each component of HDLxGraph, we conduct an ablation study of AST and DFG separately. We use LLaMA3.1 as the model for these experiments.

Evaluation Results: We evaluate the results for the same three downstream tasks: 1) In code search, the AST hierarchical analysis plays a primary role, whereas DFG behavior detection has limited impact, as natural language queries rarely align directly with DFG-level semantics; 2) In code completion, similarly, the AST hierarchical analysis proves more effective due to its strong alignment with structural retrieval. In contrast, DFG has limited impact, as incomplete code often lacks sufficient signal-level context to construct meaningful dataflow; 3) In code debugging, both AST and DFG provide essential information, reflecting the human debugging process, which often involves tracing actual signal waveforms while simultaneously searching for relevant abstract code structures.

V. CONCLUSION AND FUTURE WORK

We propose HDLxGraph, the first graph-enhanced RAG framework that combines AST-based structural matching with DFG-aware retrieval, for LLM-aided HDL tasks. Evaluations on search, debugging, and completion tasks show clear improvements over baselines, highlighting the effectiveness of HDLxGraph. Future work will focus on developing adaptive traversal mechanisms for dynamic AST and DFG exploration, enabling more flexible interaction beyond fixed-function interfaces. We also aim to investigate multi-view HDL representations to bridge the gap between natural language and circuit semantics.

REFERENCES

[1] I. Abdelaziz, J. Dolby, J. P. McCusker, and K. Srinivas, "A toolkit for generating code knowledge graphs," *The Eleventh International Conference on Knowledge Capture (K-CAP)*, 2021.

[2] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "On hardware security bug code fixes by prompting large language models," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4043–4057, 2024.

[3] H. Ahmad, Y. Huang, and W. Weimer, "Cirfix: automatically repairing defects in hardware design code," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, New York, NY, USA, 2022, p. 990–1003.

[4] A. Allam and M. Shalan, "Rtl-repo: A benchmark for evaluating llms on large-scale rtl design projects," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2405.17378

[5] S. Anthropic, "Model card addendum: Claude 3.5 haiku and upgraded claude 3.5 sonnet." [Online]. Available: https://api.semanticscholar.org/CorpusID:273639283

[6] K. Du, J. Chen, R. Rui, H. Chai, L. Fu, W. Xia, Y. Wang, R. Tang, Y. Yu, and W. Zhang, "Codegrag: Bridging the gap between natural language and programming language via graphical retrieval augmented generation," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2405.02355

[7] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, and J. Larson, "From local to global: A graph rag approach to query-focused summarization," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2404.16130

[8] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "Llm4sechw: Leveraging domain-specific large language model for hardware debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2023, pp. 1–6.

[9] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. C. Lin, "Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[10] M. Gao, J. Zhao, Z. Lin, W. Ding, X. Hou, Y. Feng, C. Li, and M. Guo, "Autovcoder: A systematic framework for automated verilog code generation using llms," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2407.18333

[11] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, and et al., "The llama 3 herd of models," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[12] Z. Guo, L. Xia, Y. Yu, T. Ao, and C. Huang, "Lightrag: Simple and fast retrieval-augmented generation," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2410.05779

[13] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018. [Online]. Available: https://arxiv.org/abs/1706.02216

[14] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-coder technical report," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2409.12186

[15] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013/

[16] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "VerilogEval: evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023.

[17] ——, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–8.

[18] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[19] X. Liu, B. Lan, Z. Hu, Y. Liu, Z. Zhang, F. Wang, M. Shieh, and W. Zhou, "Codexgraph: Bridging large language models and code repositories via code graph databases," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2408.03910

[20] A. Lozhkov, R. Li, L. B. Allal, and et al., "Starcoder 2 and the stack v2: The next generation," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2402.19173

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

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

[23] A. Nakkab, S. Q. Zhang, R. Karri, and S. Garg, "Rome was not built in a single step: Hierarchical prompting for llm-based chip design," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, ser. MLCAD '24. New York, NY, USA: Association for Computing Machinery, 2024.

[24] OpenRISC, "mor1kx," https://github.com/openrisc/mor1kx, 2022.

[25] Y. Pu, Z. He, T. Qiu, H. Wu, and B. Yu, "Customized retrieval augmented generation and benchmarking for eda tool documentation qa," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2407.15353

[26] K. Qayyum, M. Hassan, S. Ahmadi-Pour, C. K. Jha, and R. Drechsler, "From bugs to fixes: Hdl bug identification and patching using llms and rag," in *2024 IEEE LLM Aided Design Workshop (LAD)*, 2024, pp. 1–5.

[27] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[28] L. Shi, M. Kazda, B. Sears, N. Shropshire, and R. Puri, "Ask-eda: A design assistant empowered by llm, hybrid rag and abbreviation de-hallucination," in *2024 IEEE LLM Aided Design Workshop (LAD)*, 2024, pp. 1–5.

[29] W. Snyder, P. Wasson, D. Galbi, and et al, "Verilator." [Online]. Available: https://github.com/verilator/verilator

[30] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[31] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *arXiv*, 2023. [Online]. Available: https://arxiv.org/abs/2308.00708

[32] Y.-D. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language models," *arXiv*, 2023.

[33] X. Wang, G.-W. Wan, S.-Z. Wong, L. Zhang, T. Liu, Q. Tian, and J. Ye, "Chatcpu: An agile cpu design and verification platform with llm," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3658493

[34] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "CodeT5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds., Dec. 2023.

[35] K. Xu, J. Sun, Y. Hu, X. Fang, W. Shan, X. Wang, and Z. Jiang, "Meic: Re-thinking rtl debug automation using llms," *arXiv*, 2024.

[36] X. Yao, H. Li, T. H. Chan, W. Xiao, M. Yuan, Y. Huang, L. Chen, and B. Yu, "Hdldebugger: Streamlining hdl debugging with large language models," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2403.11671

[37] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[38] Y. Zhang, Z. Yu, Y. Fu, C. Wan, and Y. C. Lin, "MG-Verilog: multi-grained dataset towards enhanced llm-assisted verilog generation," in *The First IEEE International Workshop on LLM-Aided Design (LAD'24)*, 2024.

[39] Y. Zhao, D. Huang, C. Li, P. Jin, Z. Nan, T. Ma, L. Qi, Y. Pan, Z. Zhang, R. Zhang, X. Zhang, Z. Du, Q. Guo, X. Hu, and Y. Chen, "Codev: Empowering llms for verilog generation through multi-level summarization," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2407.10424