# Towards Automated Verification IP Instantiation via LLMs

Ghaith Bany Hamad, Michael Marcotte, Syed Suhaib
Nvidia Corp., Santa Clara, USA

**Abstract—Recent advancements in large language models (LLMs) have opened new possibilities for automation in hardware verification, including tasks such as generating formal assertions from natural language specifications and register-transfer level (RTL) implementations. However, despite these promising developments, current LLM-generated assertions often fall short of human-level quality, limiting their practical use in formal verification (FV) workflows. On the other hand, Verification IPs (VIPs) are comprehensive, incorporating a complete set of checkers that systematically validate different aspects of the design under test. However, deploying VIPs has traditionally been challenging for designers due to limited understanding and time constraints, which can lead to delays and increase the risk of missed bugs. To address these issues, our work introduces LLMs as both assistants and agents for automating VIP instantiation, specifically designed to aid verification engineers. Our solution overcomes existing VIP deployment challenges by streamlining and automating the integration process, providing detailed guidance on VIP usage, and optimizing resource allocation—key advantages for meeting tight project timelines. Our results demonstrate that Retrieval-Augmented Generation (RAG) enables robust interpretation of Verilog RTL, producing syntactically and logically sound VIP instantiations essential for maintaining verification integrity.**

## I. INTRODUCTION

Formal verification accounts for, on average, 56% of time spent on ASIC design and plays a critical role in the design process in verifying the correctness of chip design [1], [13]. While FV provides notable benefits, such as catching elusive bugs that may be missed with simulation-based methods, it also has limitations: FV workflows demand significant engineering effort to develop formal testbenches manually and supporting collateral, a task handled by skilled experts. This requirement makes it challenging to scale FV adoption and expand its coverage across a larger number of subsystems.

Recent advancements in large language models (LLMs) [2], [3], [4], [5], [6] have led researchers to explore their potential for automating and enhancing productivity in hardware verification [7], [8], [9], [10]. Specifically, recent studies have examined the use of LLMs for tasks in formal verification, such as generating formal assertions from natural language (NL) specifications [11] and from the register-transfer level (RTL) descriptions of the design-under-test (DUT) [12]. The rationale behind this exploration is clear: the inputs handled by human FV engineers—such as design RTL and specifications—and the outputs they generate (FV testbench implementations) are all inherently represented as text and code, which aligns well with LLMs' strengths in reasoning and content generation [14], [15], [16]. Although prior work has demonstrated that LLMs can indeed generate hardware assertions, the quality of these assertions remains limited and falls short of human-level expectations. Due to this limitation, using LLMs for such tasks is not yet viable for formal verification teams.

Many verification teams have developed Verification IPs (VIPs) by packaging the most frequently used verification checkers to facilitate reuse across projects and enable deployment by engineers who are not verification experts. VIPs enhance productivity by enabling engineers to verify design compliance with various protocols and standards effectively. However, the deployment of VIPs has been fraught with challenges, such as tight project schedules or a lack of expertise regarding the optimal use of each VIP. A more effective solution of automating common VIP instantiations by exploiting patterns of common modules has immense potential in ensuring thorough coverage and rapid development for new projects, greatly accelerating the verification process and chip production. Our main novel contributions to this work are:

- **Data Processing Pipeline for Large Verilog Code Files:**
  A novel data processing pipeline is proposed to manage large-scale Verilog code files effectively, facilitating the creation of optimized prompts for LLMs. The pipeline begins with cleaning RTL SystemVerilog code to remove unnecessary elements, followed by segmenting the code into manageable chunks. A key innovation in this pipeline is a method to process these chunks in a manner that minimizes LLM agent calls, thereby optimizing runtime performance. Additionally, a multi-threading approach is integrated into the pipeline to handle large RTL files with numerous VIP instantiations efficiently. This ensures that the system can scale to meet the demands of complex verification tasks involving substantial codebases.

- **LLM-Based Automation of VIP Instantiation:**
  An LLM-based flow is developed to automate the instantiation of various Verification IPs (VIPs), streamlining the verification process. Within this flow, multiple augmentation methods were explored to optimize LLM performance.

- In-Context Learning (ICL): This method enhances LLM responses by including demonstrations within the prompt, enabling the model to infer mechanisms and generalize them to new queries. While effective in improving accuracy, ICL requires extensive prompt engineering and becomes rigid and unscalable when applied to diverse VIPs.
- Retrieval-Augmented Generation (RAG): To address ICL's limitations, a RAG-based approach is introduced. RAG consists of two key steps: first, constructing a vectorized database of knowledge chunks derived from prior VIP examples; second, employing a retrieval function to extract the most relevant knowledge based on the user query. These retrieved chunks are appended to the LLM prompt, embedding specialized knowledge into the response without requiring additional model training. This flexible and cost-effective method enhances the adaptability of the LLM for various verification contexts.

- **Post-Response Data Correction and Hallucination Detection:**
  A post-response data correction pipeline is designed to ensure the reliability and precision of the LLM-generated outputs. This pipeline identifies and corrects inconsistencies in the generated content, mitigating the risk of errors in VIP instantiation. Furthermore, a hallucination detection mechanism is implemented to identify and flag instances where the LLM generates content that deviates from the expected or factual basis. Together, these components enhance the overall robustness and accuracy of the automated verification flow.

## II. PRELIMINARIES

### *A. Formal Hardware Verification*

Hardware verification is a critical phase in the chip design process, essential for ensuring that the final product performs according to its specifications. Despite the increasing allocation of resources to verification compared to hardware design, recent industry data indicates a growing number of projects where critical bugs evade detection during verification, requiring costly re-spins to address them [19].
Two primary approaches to hardware verification are widely recognized: traditional simulation-based verification and formal verification (FV), which has gained considerable traction over the past decade. Unlike simulation-based methods, FV seeks to rigorously prove that a digital circuit's design adheres to a specified set of requirements [20]. In FV, the specification is defined by a set of properties constraining the search space (assumptions) or identifying the design's expected behavior (assertions). Digital circuits are well-suited to representation as state transition systems, while the properties are expressed in temporal logic [21], [22]. This setup enables model checkers to traverse the state transition graph, searching for any states that violate these properties [23]. When a model checker identifies such a state, it provides a counterexample, or trace, that shows where the property fails. In contrast, simulation-based verification tests properties only for a subset of states based on specific input stimuli. If a model checker can demonstrate that a property holds across all states, it effectively provides formal proof that the property is universally valid for the design.

### *B. Retrieval Augmented Generation (RAG)*

Retrieval-Augmented Generation (RAG), shown in Fig. 1, is a cutting-edge and computationally efficient method that enhances an LLM's responses by drawing from a specialized vector database of external knowledge. To set up RAG, a vector database must first be created, storing a knowledge corpus as semantic vectors. This corpus can consist of various document types, such as webpages, PDFs, and text files. To accommodate storage requirements, each document is broken into smaller, manageable chunks that adhere to a token limit. Building this vector database is a one-time process, although it can be refreshed as new documents are added or existing ones are updated.
Once the vector database is prepared, the RAG system is ready for querying. When a user poses a question to the RAG system, it first converts the query into a semantic vector. RAG then retrieves the top-k chunks from the vector database that are most relevant to the user query. At a high level, this process is performed by calculating a similarity index between the query and each of the chunks that come from the specialized RAG database. These selected chunks are appended to the user's query, which is then passed to the LLM for response generation. Consequently, the LLM's answer is enriched with precise information from the RAG vector database, producing responses that are both accurate and contextually informed.

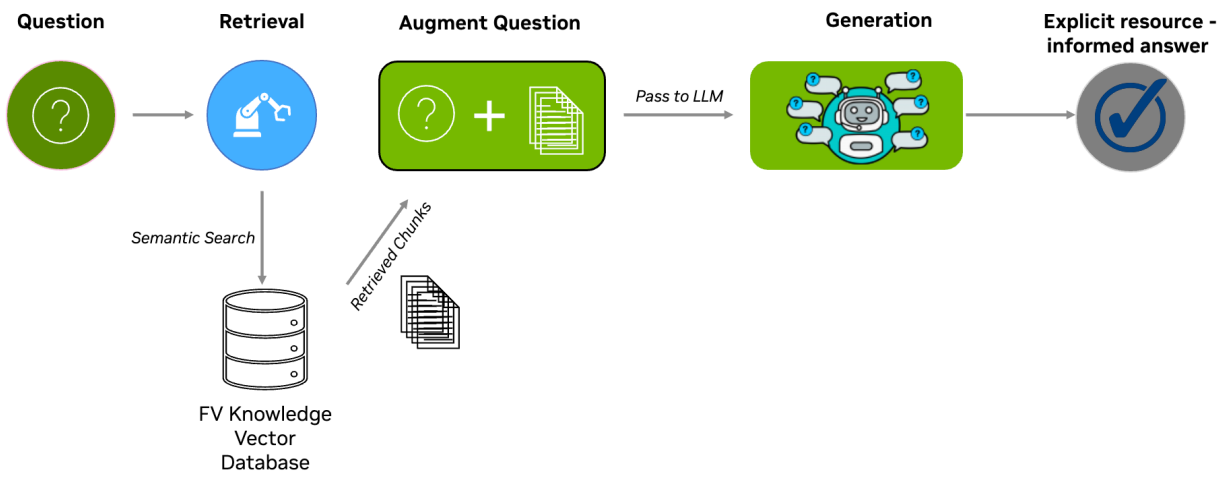


Fig. 1- General Flow for Retrieval Augmented Generation (RAG) System.

## III. Motivation and Prior Work

In this paper, we introduce a novel application of large language models (LLMs) as agents for the automatic instantiation of verification intellectual properties (VIPs), a topic that remains unexplored in formal verification research. Prior work has evaluated LLMs for tasks such as SystemVerilog Assertion (SVA) code generation [12], [17] and assertion generation with AssertLLM [1], both demonstrating promising results in generating functional, high-quality code. However, in practical settings, VIPs serve as effective checkers to verify repeated RTL (Register Transfer Level) modules, containing numerous assertions that are crucial to validating module correctness. The potential benefits of automating VIP instantiation across different designs are substantial, motivating us to explore an agent-based approach in which an LLM is trained specifically as an expert for VIP instantiations. This approach aligns with a growing trend of optimizing LLMs for specialized domains to enhance accuracy and reduce hallucinations, focusing their capabilities within narrowly defined areas [18].

Using a generic, out-of-the-box LLM, such as a GPT model or Meta's Llama2/3, is inadequate for our objectives; nor do internal LLMs currently have the necessary proficiency for targeted VIP instantiation. This limitation stems from the absence of VIP-related data in the models' pre-training datasets. In other words, foundational LLMs lack the specific knowledge needed to understand and instantiate VIPs accurately. While fine-tuning LLMs to integrate formal verification (FV) knowledge related to VIPs could address this gap, generating new datasets and fine-tuning models is both time-intensive—often taking weeks—and computationally costly. Thus, fine-tuning becomes an impractical solution for scalable VIP instantiation, highlighting the need for approaches that are efficient, cost-effective, and scalable to future iterations of the automatic VIP instantiation project.

A significant challenge remains without fine-tuning: the LLM lacks any inherent knowledge of our specialized formal verification (FV) VIP instantiation task. One possible approach is to include all necessary information for VIP instantiation directly in the query; however, LLMs have a limited token context window. For instance, Llama2 has a context window of 2048 tokens, meaning the combined length of the prompt and the response must not exceed this limit, or else the response is truncated, leading to incomplete or inaccurate VIP instantiations. This constraint presents a bottleneck when working with large files, as we must carefully manage the information passed to the LLM to avoid wasting valuable tokens on irrelevant details. To maximize the effectiveness of the LLM within these limitations, it is essential to ensure that only the most pertinent information is provided, specifically tailored to the VIP type being instantiated. Developing a strategy to handle large files efficiently allows us to streamline VIP instantiation by including only the necessary context, thus avoiding truncation issues and ensuring that the LLM's token window is utilized optimally. This approach is critical for accurately instantiating VIPs across different designs while maintaining efficiency and effectiveness within the constraints of the LLM's token limit.

Existing research offers two promising methods for equipping LLMs with the relevant information needed to effectively address challenges like those discussed here, especially the limited context window. These methods, known collectively as prompt augmentation, include in-context learning (ICL) [24] and retrieval-augmented generation (RAG). Rather than re-training an LLM, both methods build upon an out-of-the-box model, enabling a quicker implementation and alignment of the LLM agent with our domain-specific needs. Furthermore, as outlined in the pre-

vious section, RAG facilitates straightforward database creation and eliminates the need for prompt structure modifications; adding information to the vector database is sufficient to introduce new, specialized VIP knowledge to the LLM.

## IV. LLMs as Assistants and Agents

Our work introduces LLMs as agents driving the instantiation of VIPs for verification engineers. Our proposal outlines a procedure to take state-of-the-art foundational LLMs and apply them to focused and technical domains to accomplish tasks with performance far exceeding those of the base LLMs. Through retrieval augmented generation (RAG), we have successfully deployed a special Q&A bot for design and verification engineers that augments its natural language reasoning with the domain expertise from VIP documentation (details in Fig. 2). However, the focus of this proposal is on VIP instantiation where we present our modularized, scalable pipeline that allows for the instantiation of VIPs. We demonstrate in our experiments that RAG provides a more robust understanding of Verilog code compared to our baselines, creating far fewer hallucinations and exceedingly accurate and precise results necessary for the proper generation of VIP instantiations to be deployed in the formal verification pipeline.

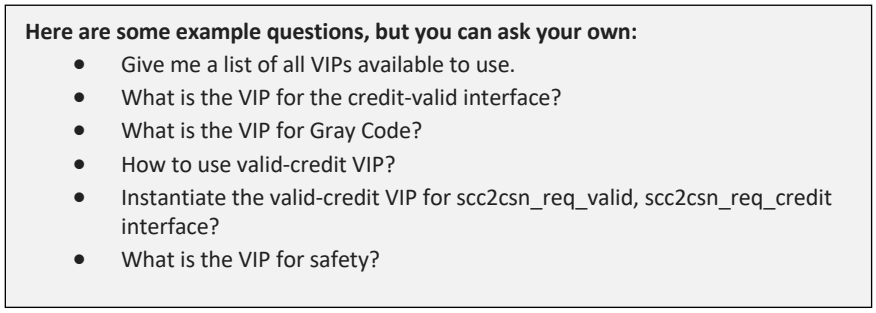


Fig. 2- Deployed Q&A ChatBot based on the RAG to help answer engineers' queries on VIPs.

## V. Methods

The proposed VIP instantiation flow is divided into four main steps: Data Pre-processing, Prompt Augmentation, VIP Instantiation, and LLM Output Post-Processing, as shown in Fig. 3. In the following section, the details of each stage will be explained.

### *A. RTL code pre-processing pipeline*

The first data pre-processing stage involves two main tasks: filtering the raw RTL design files and then partitioning code into sizable blocks for the prompts to the LLM. For the first step of cleaning the RTL design files, we have a deterministic algorithm containing procedures to filter out unnecessary artifacts from the code that are irrelevant to VIP instantiation. These artifacts include comments, 'ifdef' and 'ifndef' declarations, input/output port declarations with the keywords 'defines' and 'includes', and blank lines resulting from deleting these details in the code files. Following filtering these details, we are ready to process the cleaned code file into prompts. The second step, which is the partitioning of code blocks, is performed using our novel algorithm to ensure optimal chunking by maintaining the integrity of blocks of code while respecting the LLM token limit. This step is crucial to formulate raw data into reasonable prompts so we can leverage LLMs in our flow with minimal hallucinations. To do so, we use our proprietary algorithm that recursively splits cleaned Verilog code into partitions that obey a maximum token count we must respect to query the target LLMs accurately. This step aims to ensure each query to the LLM contains a partition of the cleaned code file that is syntactically complete so we can process each section independently through the LLM and instantiate relevant VIPs using the cleaned information. We have verified that our data pre-processing techniques work reliably for our instantiation of FIFO and arbiter VIPs and designed our algorithms to cover cases for these modules. However, our data-preprocessing pipeline, along with the rest of our flow, was designed to scale the implementation to any number of VIPs, which we have done by adding to the list of rules we want to be followed in the driver functions of our cleaning and prompt creating processes. The execution of these steps results in cleaned prompts of partitioned code that we can iteratively feed to our LLMs to instantiate our VIPs. The code partitions have no overlap and collectively span the entire cleaned code file.

### *B. Prompt Augmentation*

The second stage of our VIP instantiation flow is the prompt augmentation stage which we vary in method depending on the experiments we conducted to determine the efficacy of Retrieval Augmented Generation (RAG) in instantiating VIPs compared to our baseline and In-Context Learning (ICL) methods of prompt augmentation. We set up our RAG's vectorized database to contain internal documents on VIP instantiation and examples of module code and VIP instantiation pairs demonstrating examples of proper VIP instantiation. Each experiment involves the LLM being iteratively fed each base prompt created from the data pre-processing stage. For baseline augmentation, we feed just the code partitions without further augmentation to evaluate VIP instantiation through an LLM as a control. For the ICL and RAG experiments, we developed a distinct series of rules formal verification experts would use to help instantiate VIPs for FIFOs and arbiters. If a FIFO is detected inside the code partition, our flow will append the FIFO ruleset. If an arbiter is detected, the arbiter ruleset will be appended to guide the LLM accordingly for these different modules. These rulesets remained the same between the ICL and RAG experiments and were appended after each code partition to augment the response. The ICL method prompts were also given a one-shot example of a VIP instantiation for the respective module, while the RAG method used RAG to augment the prompt with relevant chunks from the vectorized database of internal documents on formal verification. After the steps in prompt augmentation, the finalized prompts are ready to be fed through the LLMs to be queried for VIP instantiation.

### *C. Proposed LLM–based flow for VIP Instantiation*

The third stage involves VIP instantiation through an LLM agent, where augmented prompts are iteratively queried to generate the desired VIP instantiation for each respective code partition. This stage leverages the deployed augmentation methods to enhance the prompts and guide the LLM toward accurate and efficient VIP instantiation.

- **Context Learning Method:**
  In this approach, the deployed prompt incorporates a general system prompt tailored for each target VIP, such as FIFO or Arbiter VIPs. This prompt includes a set of general rules to instruct the LLM on how to instantiate the VIP, supplemented by specific instructions derived from experimental observations. The flow then combines these elements into a final prompt, integrating the In-Context Learning (ICL) component with RTL code chunk. This ensures that the LLM generates VIP instantiations that adhere to best practices and the requirements identified during earlier stages of experimentation.
- **Retrieval-Augmented Generation (RAG) Method:**
  For this method, the process begins by identifying the target VIP to be instantiated. Relevant examples and details are retrieved from the FV knowledge vector database using the RAG approach. These extracted elements are appended to the initial prompt for the specific RTL code chunk. By incorporating these precise, contextually relevant details into the prompt, the LLM's prompt is enriched, producing responses that are not only accurate but also well-informed by prior knowledge stored in the RAG vector database. This approach enhances the overall reliability and quality of the VIP instantiation process.

The LLM used for both the baseline and ICL experiment was a llama model fine-tuned on internal code documentation including Verilog code. For the RAG, we used the open-source Llama3-70b-instruct endpoint giving additional value to the RAG approach for VIP instantiation by demonstrating expensive fine-tuning to the domain is not necessary for correct VIP instantiation. Querying the LLM for each partition of the code is costly in many cases as some of the Verilog files we performed our experiment on contained hundreds of thousands of lines of code making the aggregate inference time hours. We optimized this runtime bottleneck by creating methods to identify with greater probability the partitions of code that contained the desired FIFO and arbiter modules and only processed these chosen chunks. These helper methods are designed to use the naming conventions of FIFOs and arbiters we can strongly assume formal verification engineers follow their code syntax and ignore all chunks that now have any module instantiations for FIFOs and arbiters. This function ensures every FIFO and arbiter will be processed, but false positives can occur where an LLM receives a code partition, hallucinates, and instantiates erroneous VIPs for the misidentified module instantiation. These false positives are easily detected in post-processing. Following the completion of the LLM queries, the aggregated output results are ready for the final post-processing stage of the flow.

### D. *Results Post-processing Pipeline*

The fourth and final stage of our VIP instantiation flow is the post-processing of the LLM VIP instantiations. Here, we augment and filter the LLM output responses to complete the VIP instantiation. First, we detect false positives by exploiting patterns in the outputted response that signal that the response was a hallucination. Specifically, in the prompts, LLMs are instructed when a signal needed for VIP instantiation is not found in the inputted code partition, to flag the signal as unknown. For the FIFO and Arbiter modules, most of the signal ports are defined in code. Seldom edge cases where a verification engineer needs to identify a signal for a VIP instantiation manually; 2 or more unknown signals mean the instantiated VIP was a false positive, and we can confidentially void the hallucination in the final output. After false positives are filtered, we are left with VIPs instantiated for their respective identified modules of either FIFO or arbiter. We developed driving post-processing functions for each module to complete the VIP instantiations and make them valid VIPs. For instance, instantiations of FIFOs sometimes need additional post-processing to fill in the FIFO depth and data width. These parameters are requirements for the VIP to be complete, but these values are sometimes not in the code file. To complete the VIPs for these cases, we developed an automatic retrieval process to extract the FIFO depth and data width for FIFO modules in the module's respective source code. These values are then injected into the LLM output to complete the instantiation, leaving us with a complete and correct VIP instantiation for a FIFO module. Finally, these post-process responses are aggregated and written to an output file and ready to be used in the next part of the formal verification process of verifying the Verilog code.

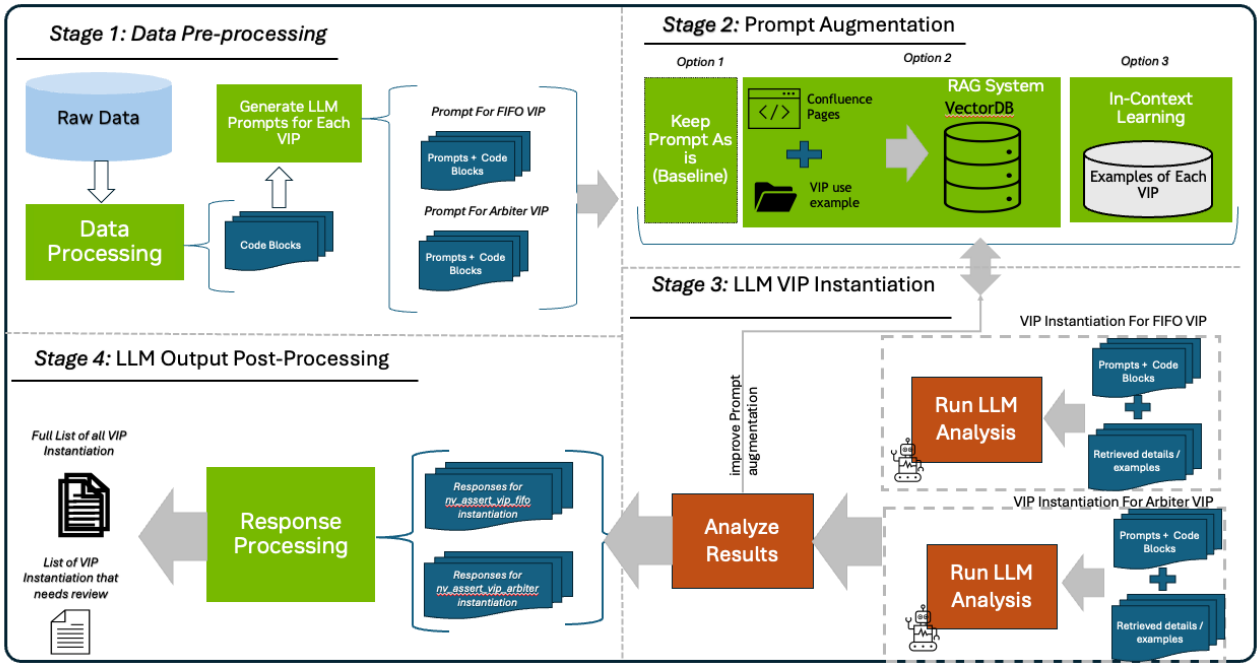


Fig. 3- Proposed LLM-based framework for code generation for VIP instantiation

## VI. RESULTS AND DISCUSSION

Starting from the RTL Verilog code of the target design, the proposed VIP instantiation flow has been used to instantiate FIFO and Arbiter VIPs for several DUTS. Table I compares the effectiveness of three prompt augmentation techniques—**Baseline**, **In-Context Learning (ICL)**, and **Retrieval-Augmented Generation (RAG)**—using two distinct LLMs, chipmixtral_8x7b_chat_tp4_trt_h100 and Llama-3-70b, for generating VIP assertions in FIFO and ARB components.

### A. Comparison of Different Prompt Augmentation Techniques

1. **Baseline**: This approach, tested with the chipmixtral_8x7b_chat_tp4_trt_h100 model, was not able to correctly identify and instantiate either of the VIPs. In Table I, all instances for assert_vip_fifo and assert_vip_arb were marked as "missing," indicating that baseline prompts alone are insufficient for generating correct VIP assertions.
2. **In-Context Learning (ICL)**: When enhanced with ICL, the chipmixtral_8x7b_chat_tp4_trt_h100 model showed moderate improvements. For assert_vip_fifo, ICL produced 4 correct, 1 incorrect, and 2 missing outputs. However, results were less consistent for assert_vip_arb, with only 1 correct output, 4 incorrect, and 7 missing instances. While ICL improved performance compared to Baseline, its results were variable, particularly with more complex designs.
3. **Retrieval-Augmented Generation (RAG)**: this method delivered the best performance. For FIFO VIP (assert_vip_fifo), RAG produced 7 out of 7 correct outputs, achieving a 100% accuracy rate. For arbiter VIP (assert_vip_arb), RAG generated 11 correct outputs with only one incorrect, showing strong reliability across both VIP assertion tasks.

*B. Comparison of the Performance of Different LLM Models*

1. **Chipmixtral_8x7b**: Tested with both Baseline and ICL prompt augmentations, this model struggled to deliver consistently accurate outputs. The Baseline approach yielded no correct results, while ICL showed some improvement with limited success, particularly in generating assert_vip_fifo assertions. However, its overall performance was inconsistent, especially for the more complex assert_vip_arb task.
2. **Llama-3-70b (with RAG)**: The Llama-3-70b model, paired with RAG, substantially outperformed chipmixtral_8x7b_chat in both assertion tasks. It achieved perfect accuracy for assert_vip_fifo and nearly perfect results for assert_vip_arb. This suggests that Llama-3-70b, particularly with RAG, has a stronger capability in generating accurate and reliable VIP assertions, demonstrating its suitability for complex verification tasks.

In summary, the RAG retrieval process provides a far greater benefit to the overall accuracy of instantiation compared to the baseline and ICL methods with few-shot learnings, as demonstrated in Table I below. Additionally, because RAG method used an open-source Llama3 endpoint instead of the finetuned chip model trained on internal data including Verilog code. This means that the successful results of the RAG have even more value as the approach does not rely on a fine-tuned model to execute its task, meaning that we can update our procedure with better general LLMs that only improve our VIP instantiation solution. In other words, our flow is designed to match the pace of ongoing LLM research and to be updated with the inference capabilities of the latest, strongest LLMs in the industry. Furthermore, due to our modularized procedure, our LLM-powered VIP instantiation process has the potential to support many more VIPs with fast development and is applicable beyond any project.

TABLE I: RESULTS OF THE VIP INSTANTIATION FOR BOTH ASSERT_VIP_FIFO AND ASSERT_VIP_ARB.

| FV VIP | LLM Model | Prompt Augmentation | Correct | Incorrect | Missing | Total # of instances |
|---|---|---|---|---|---|---|
| assert_vip_fifo | chipmixtral_8x7b | Baseline | 0 | 0 | 7 | 7 |
| | chipmixtral_8x7b | ICL | 4 | 1 | 2 | 7 |
| | Llama-3-70b | RAG | 7 | 0 | 0 | 7 |
| assert_vip_arb | chipmixtral_8x7b | Baseline | 0 | 0 | 12 | 12 |
| | chipmixtral_8x7b | ICL | 1 | 4 | 7 | 12 |
| | Llama-3-70b | RAG | 11 | 1 | 0 | 12 |

Key for Table I:
**Correct**: This column means completely correct instantiation, all signals correct
**Incorrect**: This column means VIP instantiation for modules present with either incomplete or wrong signals
**Missing**: This column means missing VIP instantiation for the module

## VII. REFERENCES


[1] Fang, W., Li, M., Li, M., Yan, Z., Liu, S., Zhang, H., & Xie, Z. (2024). Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms. *arXiv preprint arXiv:2402.00386*.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. arXiv preprint arXiv:2303.08774, 2023.

[3] Meta. Meta llama 3, 2024.

[4] Hugo Touvron and et al. Llama 2: Open foundation and fine-tuned chat models, 2023.

[5] Albert Q. Jiang, et al. Mixtral of experts. ArXiv, abs/2401.04088, 2024.

[6] Gemini Team Google. Gemini: A family of highly capable multimodal models. ArXiv, abs/2312.11805, 2023.

[7] Mingjie Liu, et al. Chipnemo: Domain-adapted llms for chip design. ArXiv, abs/2311.00176, 2023

[8] Zhuolun He, Haoyuan Wu, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. Chateda: A large language model powered autonomous agent for eda. 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD), pages 1–6, 2023

[9] Kiran Thorat, Jiahui Zhao, Yaotian Liu, Hongwu Peng, Xi Xie, Bin Lei, Jeff Zhang, and Caiwen Ding. Advanced large language model (llm)-driven verilog development: Enhancing power, performance, and area optimization in code synthesis. ArXiv, abs/2312.01022, 2023.

[10] YunDa Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language models. ArXiv, abs/2311.16543, 2023

[11] Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms. ArXiv, abs/2402.00386, 2024

[12] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. Using llms to facilitate formal verification of rtl. ArXiv, abs/2309.09437, 2023.

[13] Foster, H. (2021). Part 8: The 2020 Wilson Research Group Functional Verification Study. Accessed: 2024-07-25.

[14] Mark Chen, et al. Evaluating large language models trained on code, 2021.

[15] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021

[16] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. ArXiv, abs/2108.07732, 2021.

[17] Orenes-Vera, M., Martonosi, M., & Wentzlaff, D. (2023). From rtl to sva: Llm-assisted generation of formal verification testbenches. *arXiv preprint arXiv:2309.09437*.

[18] Zhang, K., Li, J., Li, G., Shi, X., & Jin, Z. (2024). Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*.

[19] Harry Foster. "the 2022 wilson research group ic/asic functional verification treads". White Paper. Wilson Research Group and Mentor, A Siemens Business, 2022.

[20] Thomas Kropf. Introduction to formal hardware verification. Springer Science & Business Media, 1999.

[21] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, Logics of Programs, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[22] Edmund M. Clarke, E. Allen Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst., 8(2):244–263, apr 1986.

[23] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. Formal methods in system design, 19:7–34, 2001.

[24] Dong, Qingxiu, et al. "A survey on in-context learning." *arXiv preprint arXiv:2301.00234* (2022).