2026 ASEE ANNUAL CONFERENCE June 21–24, 2026 | Charlotte Convention Center, NC
Host Sponsor UNC Charlotte
ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION
Paper ID #53418

Evaluating AI-Based Tools for Learning Digital Logic Design with FPGA Laboratory

Miss Yung-Ching Liang, San Jose State University

A recent graduate of San Jose State University, Electrical Engineering Department.

Evaluating AI-Based Tools for Learning Digital Logic Design with FPGA Laboratory

Yung-Ching Liang, Chihyu Pai, PingShan Huang, Norman Wang, Chang "Charles" Choo
{ yung-ching.liang, chihyu.pai, ping-shan.huang, chenghsun.wang, chang.choo}@sjsu.edu
AI/DL FPGA/DSP Systems Laboratory
Department of Electrical Engineering
San Jose State University
San Jose, California, USA

Abstract – This paper presents an empirical study investigating the ability of large language models (LLMs) to generate complex hardware description language (HDL) code across a range of electrical engineering design tasks in the classroom. We evaluate and compare the performance of four representative LLMs—ChatGPT, LLaMA, Gemini, and Claude—on HDL design and verification problems. Eleven evaluation criteria, including response time, token usage, code length, and functional correctness, are defined and used to assess model performance. The evaluation framework incorporates task-dependent weighting and automated API-based testing, with simulation-based verification to ensure correctness. In addition, Retrieval-Augmented Generation (RAG) is integrated to enhance model response accuracy. Experimental results show that ChatGPT and Claude achieve the best overall balance of speed, accuracy, and efficiency for HDL code generation, while Gemini demonstrates significant accuracy improvements when combined with RAG. This work provides a reproducible, simulation-verified benchmarking framework for evaluating LLM-generated HDL code on realistic FPGA design tasks.

INTRODUCTION

In recent years, large language models (LLMs) have rapidly expanded beyond natural language tasks into domains traditionally dominated by specialized Electronic Design Automation (EDA) tools, including hardware description language (HDL) design and verification. Researchers are increasingly exploring how LLMs can automatically generate RTL code (e.g., Verilog and VHDL) from high-level or natural-language specifications, reducing dependence on manual coding and accelerating early design stages [1-4].

In this paper, our main objective is to evaluate the performance of four different types of large language models (LLMs): ChatGPT, Gemini, LLaMA, and Claude. To assess their performance, we focus on having these AI models generate Verilog code for hardware circuit design and FPGA development. The evaluation criteria include response time, code correctness, total number of lines generated, and token count. We use these metrics to compare the performance of each model. The main contributions of this paper are:

- A comprehensive empirical evaluation of multiple LLMs on diverse HDL design tasks validated through simulation and synthesis.
- An automated benchmarking and feedback framework integrating RAG (Retrieval-Augmented Generation) to improve HDL code correctness [5-7].
- A comparative analysis highlighting trade-offs between correctness, verbosity, token usage, and response time.

I. METHODOLOGY

To evaluate the performance of each LLM, we design a comprehensive benchmarking framework with multiple evaluation criteria. These criteria help to visualize the performance and easier to compare with different LLMs. The criteria are listed as follows: correctness, debugging capability, ability of simulation, adherence to constraints, FPGA synthesis compatibility, code readability, response time, prompt token usage, code compactness, and overall simplicity [8-10]. Table 1 shows all the criteria that are used in this project. The ability to generate the testbench is also an important criterion but is evaluated separately in the testing.

The Application Programming Interface (API) is used for interacting with LLM and allows users to customize settings and responses. In this paper, the API is used for sending a prompt to the LLM and receiving responses. We designed an automation system to control the procedure for each HDL design task. The flow chart is shown in Figure 1. First, we input the prompt that contains the description of the task, some restrictions, such as generating a testbench. After these steps, LLM will generate the response and then send the code to Xilinx Vivado tool for simulation. If the code is fully functional and no error is reported, the test ends, and the data is ready for evaluation. If there is any error reported, the error message will be fed back to LLM to correct

TABLE 1. CRITERIA FOR EVALUATIONS.

Score	5	4	3	2	1
Correctness	100%	<25%	<50%	<75%	>75%
Debug Ability	No Debugging	1 Debugging	2 Debugging	3 Debugging	>3 Debugging
Simulation	Passed				Failed
Adherence	0 extra	1 extra	2 extra	3 extra	4 extra
FPGA Synthesis	Passed				Failed
Code Readability	0 line	1 line	2 lines	3 lines	4 lines
Response Time	Second(s)				
Token	Token Used				
Compactness	Number of Lines				
Simplicity	0	1	2	3	4

and debug. In the feedback loop, a RAG system will engage in this process. Adding the RAG system to the feedback loop can improve the overall performance and prevent LLM from generating incorrect code for simulation and eventually failing to finish the task.

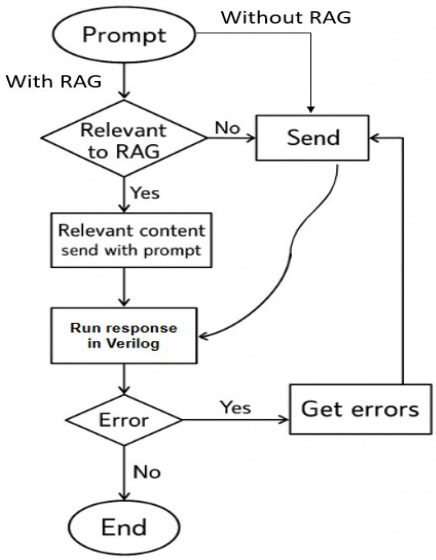

FIGURE 1. FLOW CHART

One of the major issues with LLM is its tendency to over-generate content. Even when given a simple and direct question, LLM often responds with excessive or unrelated information. This behavior increases the likelihood of incorrect or confusing code output. This might be one of the key reasons why LLM frequently produces inaccurate Verilog code. To address this problem, we decided to integrate RAG into our system to enhance the performance of the original LLM model. The goal was to improve the relevance and correctness of the generated code and to reduce the amount of irrelevant information in the chatbot's responses. We used Python to develop an online chatbot that leverages RAG. Unlike a standard LLM, this RAG-enhanced chatbot retrieves answers from a custom-built database. This allowed precise control of the information that the chatbot can access. The database includes content from LLM as well as PDF files from trusted sources such as research papers and technical articles related to the project. For example, some PDF files explain how to build a finite state machine using Verilog, how to write testbenches, and the theory of finite state machines. Curating such custom-built database ensures that the chatbot retrieves only relevant, accurate, and project-specific information, facilitating obtaining correct and useful code outputs.

In addition, we incorporated off-topic detection in the chatbot using a tool called Coralogix. This feature

monitors the chatbot's responses and detects when an answer is not relevant to the question. If off-topic content is detected, the system immediately notifies the user. This helps keep responses more accurate and aligned with user intent, especially critical when the chatbot is being used to generate Verilog code.

II. Evaluation Design Tasks

Following the benchmark design and test flow, a series of tasks were assigned to LLMs for evaluation, starting with fundamental digital logic circuits and progressing to more complex designs. The tasks included MUX, DFF, Finite State Machine, Bilinear/Bicubic/Bilateral/Gaussian filter, 3x3 matrix multiplication in both fixed-point and floating-point representation, 8-Tap FIR filter, and 32-bit DIT-FFT.

Finally, the evaluation results were observed at each step to determine which model demonstrated the best performance in HDL design tasks, and how many levels of learning were necessary for the model to meet the requirements of each design task. To do a composite score for LLMs, we assign a weight scale from 1 to 3 based on the difficulty of the task. We then normalized the result and took the average to calculate the composite score.

1) Bilinear, Bicubic, Bilateral, Gaussian filter Testing the LLM performance on designing the graphics processing filter, including bilinear, bicubic, bilateral, and Gaussian filters. The basic idea of assigning these tasks is to test whether the LLM can respond with a functional code that can implement the graphics processing filter. Each LLM performance:

- **ChatGPT:** Decent performance, needs several debugs to get the functional code.
- **Gemini:** Decent performance, needs several debugs to get the functional code.
- **Claude:** Best performance among the four LLMs.
- **LLaMA:** Can not respond with a functional code, and can not correct any errors and debug the code by providing feedback on the problems.

By evaluating the performance of designing a graphics processing filter, LLaMA has the worst performance on designing these filters. ChatGPT, Gemini, and Claude are able to complete the task, but their performance does not meet expectations. Two major suspects are that LLM lacks training on these specific filter designs, and the other is that LLM can not design tasks that are related to matrix multiplications. To understand the reasons, further research on matrix multiplication tasks is needed.

2) Fixed-point matrix, 32-bit floating-point matrix Develop a parameterized Verilog module to perform IEEE-754 floating-point multiplication and accumulation for the 3×3 matrices across half-precision (16-bit),
single-precision (32-bit), and double-precision (64-bit) formats, and implement a comprehensive testbench to verify correct functionality for each precision. Verilog code for multiplying two 3×3 floating-point matrices. Both ChatGPT-4o and Gemini both achieve correct ("Match") first-attempt results, while both Claude and Llama-3 cannot fix the major bugs. Although Claude achieves the highest score in the capability to debug, its code is way too long (423 lines) compared to that of ChatGPT-4o (≈65 lines) and Gemini (50 lines).

3) 8-Tap FIR Filter

An 8-tap FIR filter works by storing the input's last eight most recent samples in a shift-register, multiplying each sample in parallel by the corresponding coefficient, and summing all eight products to produce the filtered output; in Verilog you do this using an array of delay registers, eight floating-point multipliers, and an adder tree to add up the results on each clock cycle.

- **ChatGPT:** Generates fully synthesizable, simulatable FIR code along with an integrated testbench.
- **Gemini:** Generates Verilog code that can be compiled but not simulated. It has the lowest scores for every category, although it generates the shortest runtime (10 s) and medium token utilization.
- **Claude:** Generates complete synthesizable FIR code for FPGAs with the best correctness and debug scores. Runs in a reasonably fast (15.2 s) and consumes the fewest tokens among all models.
- **LlaMA:** Incorrect result and is not able to detect its own errors when debugging. Generates synthesizable code with the shortest time (2.9 s)

4) 32-bit DIT-FFT

The Decimation-in-Time Fast Fourier Transform (DIT-FFT) is an efficient algorithm used to compute the Discrete Fourier Transform (DFT) by breaking it down into smaller DFTs recursively. To implement this task in HDL, a butterfly computation unit is needed to perform complex additions and multiplications using precomputed twiddle factors. Next, a bit-reversal module to reorder input indices appropriately and a control FSM to manage the sequencing of stages. 32-bits DIT-FFT is the most difficult design that we had among all. Most of the LLMs struggle to produce a functional code. As we investigated through the code, we discovered that they all failed to use correct twiddle factors in each stage. Additionally, we compared these programs to the correct code found on the internet and observed that these LLMs attempt to create a function for calculating the twiddle factors, whereas the functional code simply declares them as fixed numbers.

III. Evaluation Result

To streamline the analysis, we established four principal evaluation criteria: response time, accuracy, token used, and generation time, each converted into a quantitative score.

- **Comparing the response time:** LLaMA is the fastest and most consistent in response time. ChatGPT has the slowest response time on

average and has the largest variation. Gemini and Claude are in the middle, with Claude having a more symmetric distribution. Depending on whether speed or consistency is more important, LLaMA has the fastest response time, while ChatGPT needs more time to respond.

- **Comparing the code-generating accuracy:** ChatGPT is clearly the top performer for code generation across all measured tasks. Claude performs well, especially in mathematical transformations. Gemini is middling, and LLaMA shows the lowest code accuracy in this test set.

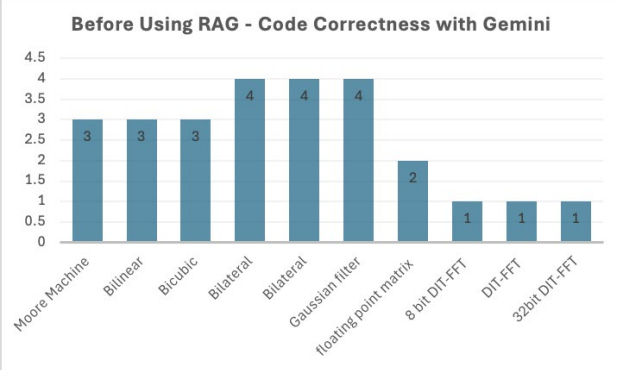

FIGURE 2. CODE CORRECTNESS (WITHOUT GEMINI)

- **Comparing the token used:** ChatGPT is the most token-expensive model, which can be beneficial for complex explanations but will also potentially have higher usage charges or latency. Claude is the most efficient, and this can pay off with performance in low-bandwidth or real-time situations. Gemini and LLaMA have similar average token counts used.
- **Comparing the total and average line generated:** Claude is the most verbose model for code writing across a wide range of tasks, both in total and in average. ChatGPT is a consistent performer, balancing verbosity and brevity equally across tasks. Gemini will probably prefer brevity, with the exception of where complexity demands otherwise (e.g., 32-bit FFT optimizations). LLaMA is most brevity-oriented or underperforms in this code-writing task.

In the overall comparison by normalizing the data of response time, accuracy, tokens used, and total lines generated. Claude and ChatGPT offer the best composite performance, with corresponding and highest composite scores. Both models offer high overall balance, suggesting consistent performance across a variety of assessment measures (e.g., completeness, accuracy, and code generation). Gemini offers the third, moderate composite score, suggesting good performance but potentially poor performance in specific areas. LLaMA offers the lowest composite score, suggesting poor performance in code amount, accuracy, or task coverage. The performance gap between the top (Claude/ChatGPT) and bottom (LLaMA) is considerable, indicating wide variability in model performance. ChatGPT and Claude are the best overall for general HDL code design tasks based on normalized performance metrics.

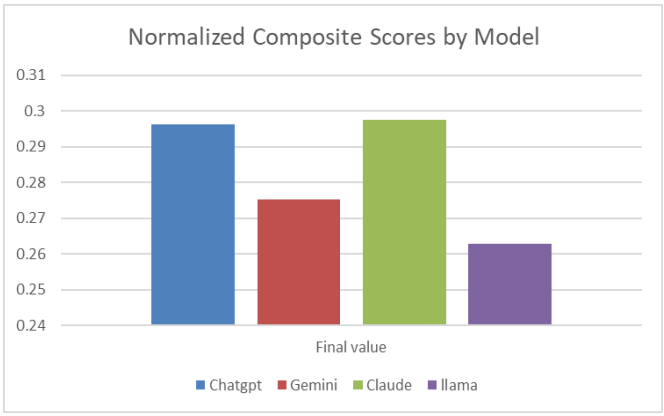

FIGURE 3. EVALUATION RESULT OF THE HDL DESIGN TASK

IV. RESULTS AND DISCUSSION

We noticed that one of the major issues with Gemini is its tendency to over-generate content. Even when given a simple and direct question, Gemini often responds with excessive or unrelated information. This behavior increases the likelihood of generating incorrect or confusing code. We believe this over-generation is one of the key reasons why Gemini frequently produces inaccurate Verilog. To address this issue of Gemini producing incorrect or confusing code output, we decided to integrate RAG (Ret into our system to enhance the performance of the original LLM model. Our goal was to improve the relevance and correctness of the generated code while reducing the amount of irrelevant information in the chatbot's responses.

Our evaluation of the original Gemini model revealed serious limitations in code correctness, largely due to the over-generation of irrelevant information. By implementing a RAG-based solution, along with a carefully curated database, chunking, and off-topic detection, we significantly improved the accuracy and usefulness of the Verilog code generated by the chatbot. These enhancements make the chatbot a more effective tool for our senior project and a more reliable source of technical support for coding tasks.

Furthermore, as shown in Figure 3, Claude (≈0.2974) and ChatGPT (≈0.2936) lead the pack, delivering the best overall balance of fast response time, high correctness, compact code, and low token usage. Gemini (≈0.2752) performs moderately well, while Llama (≈0.2629) trails behind. These results show that Claude or ChatGPT (without RAG) is best for HDL-design support.

V. CONCLUSION

In this paper, we presented an empirical study investigating the ability of large language models (LLMs) to generate complex hardware description language (HDL) code for a wide range of classroom-oriented electrical engineering design tasks. These tasks span from simple digital logic circuits to advanced digital signal processing (DSP) hardware designs. The former includes combinational and sequential circuits such as adders, multipliers, flip-flops, counters, and finite state machines, while the latter encompasses matrix multipliers, interpolation circuits, finite impulse response (FIR) filters, and fast Fourier transform (FFT) hardware. We evaluate and compare the performance of four representative LLMs—ChatGPT, LLaMA, Gemini, and Claude—on HDL design and verification problems using eleven evaluation criteria, including response time, token usage, code length, and functional correctness. The proposed evaluation framework incorporates task-dependent weighting, automated API-based testing, and simulation-based verification to ensure correctness.

In addition, Retrieval-Augmented Generation (RAG) is integrated to enhance model response accuracy. Experimental results indicate that ChatGPT and Claude provide the best overall balance of speed, accuracy, and efficiency for HDL code generation, while Gemini exhibits significant accuracy improvements when combined with RAG. This work offers a reproducible, simulation-verified benchmarking framework for evaluating LLM-generated HDL code on realistic FPGA design tasks in both classroom and laboratory settings.

REFERENCES

[1] York, Jason Blocklove, New York University, et al. Evaluating LLMs for Hardware Design and Test, arxiv.org/html/2405.02326v1. Accessed 04 Dec. 2024.

[2] Fang, W., Li, M., Li, M., Yan, Z., Liu, S., Zhang, H., & Xie, Z. (2024). AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs. Proceedings of the IEEE.
https://doi.org/10.979-8-3503-7608-1/24

[3] Thakur, S., Blocklove, J., Pearce, H., Tan, B., Garg, S., & Karri, R. (2024). AutoChip: Automating HDL generation using LLM feedback. In Proceedings of the Design Automation Conference (pp. 1–7). ACM.
https://doi.org/10.48550/arXiv.2311.04887

[4] Liu, M., Pinckney, N., Khailany, B., & Ren, H. (2023, September 14). *VerilogEval: Evaluating large language models for Verilog Code Generation*. arXiv.org. https://arxiv.org/abs/2309.07544

[5] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020, May 22). *Retrieval-Augmented Generation for Knowledge-Intensive NLP tasks*. arXiv.org. https://arxiv.org/abs/2005.11401

[6] Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2023, December 18). *Retrieval-Augmented Generation for Large Language Models: A survey*. arXiv.org. https://arxiv.org/abs/2312.10997

[7] Zhang, P., Xiao, S., Liu, Z., Dou, Z., & Nie, J. (2023, October 11). *Retrieve anything to augment large language models*. arXiv.org. https://arxiv.org/abs/2310.07554

[8] Tsai, Y., Liu, M., & Ren, H. (2023, November 28). *RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models*. arXiv.org. https://arxiv.org/abs/2311.16543

[9] Thakur, S., Blocklove, J., Pearce, H., Tan, B., Garg, S., & Karri, R. (2023, November 8). *AutoChip: Automating HDL Generation using LLM Feedback*. arXiv.org. https://arxiv.org/abs/2311.04887

[10] Gao, M., Zhao, J., Lin, Z., Ding, W., Hou, X., Feng, Y., Li, C., & Guo, M. (2024, July 21). *AutoVCoder: A Systematic Framework for Automated Verilog Code Generation using LLMs*. arXiv.org. https://arxiv.org/abs/2407.18333