LLM-VeriPPA: Power, Performance, and Area Optimization aware Verilog Code Generation with Large Language Models

***Abstract*— Large Language Models (LLMs) are gaining prominence in various fields, thanks to their ability to generate high-quality content from human instructions. This paper delves into the field of chip design using LLMs, specifically in Power-Performance-Area (PPA) optimization and the generation of accurate Verilog codes for circuit designs. We introduce a novel framework VeriPPA designed to optimize PPA and generate Verilog code using LLMs. Our method includes a two-stage process where the first stage focuses on improving the functional and syntactic correctness of the generated Verilog codes, while the second stage focuses on optimizing the Verilog codes to meet PPA constraints of circuit designs, a crucial element of chip design. Our framework achieves an 81.37% success rate in syntactic correctness and 62.06% in functional correctness for code generation, outperforming current state-of-the-art (SOTA) methods. On the RTLLM dataset. On the VerilogEval dataset, our framework achieves 99.56% syntactic correctness and 43.79% functional correctness, also surpassing SOTA, which stands at 92.11% for syntactic correctness and 33.57% for functional correctness. Furthermore, Our framework able to optimize the PPA of the designs. These results highlight the potential of LLMs in handling complex technical areas and indicate an encouraging development in the automation of chip design processes. Our source codes are here [1].**

I. Introduction

As Moore's law continues to drive design complexity and scaling in chip design, it pushes chip design tools like Electronic Design Automation (EDA) to their limits. These traditional tools are time-consuming and rely on human experts. Machine learning (ML) has been successfully integrated into chip design for logic synthesis [9], [11], placement [36], routing [13], [19], testing [5], and verification [6], [12]. The popularity of agile hardware design exploration has been on the rise due to the growth of large language models (LLMs). A promising direction is using natural language instruction to generate hardware description language (HDL), e.g., Verilog, aiming to greatly lower hardware design barriers and increase design productivity, especially for users who do not possess extensive expertise in chip design [16], [28], [29], [38].

Despite various efforts, optimizing PPA remains the most critical task in chip design, and to the best of our knowledge, no existing methods support PPA optimization. Before we perform PPA optimization, we must generate correct Verilog code. Recent work in correct Verilog code generation falls into two categories: prompt engineering and fine-tuning. Prompt engineering improves Verilog code generation by adjusting descriptions and prompt structures. For example, hierarchical prompting [23] generates hierarchical code, ChipGPT [4] applies prompt engineering for automatic chip generation, and RTLLM [18] uses self-planning prompt engineering to enhance correctness. Fine-tuning improves Verilog code generation by modifying model parameters. VeriGen [32] uses fine-tuning on a collected dataset from GitHub, but lacks data cleaning and task-specific training, which reduces functional accuracy. ChipNeMo [14] performs a two-round fine-tuning with in-house data, although only the first round benefits RTL code generation. BetterV [28] fine-tunes the model alongside a generative discriminator, which increases deployment complexity. VerilogEval [15] and RTLCoder [16] provide benchmark datasets for single-round fine-tuning. In summary, no existing work addresses PPA optimization a very crucial aspect of chip design. There are two existing works that provides the PPA result (Vanilla) without optimization. In regards to the correct Verilog generation existing methods do not use the exact error details from the integrated simulator and multi-round conversation to understand the Verilog code for the circuit design. We highlighted the recent work in the comparison our VeriPPA framework in Table I. In this work, we propose VeriPPA, a systematic open-source framework that makes LLM capable of PPA optimization and strengthens LLM's capability generation of Verilog code, as shown in Figure 1. Our key contributions are summarized here:

- We introduce PPA optimizations to ensure that the generated Verilog codes meet design specification (i.e., PPA) is optimized, we use Synopsys Design Compiler to perform logic synthesis and use the open source ASAP 7nm Predictive PDK to obtain PPA reports.
- We propose an effective method for Verilog code generation using refinement of the errors by enabling the LLM to understand Verilog code for circuit design. We use the detailed error diagnostics from the iverilog simulator [37], and pinpoint the exact location of syntactic or functional discrepancies as indicated by testbench failures as new

[1] https://anonymous.4open.science/r/LLM-VeriPPA-B016

prompts. We use multi-round generation to enhance the syntax and functionality correctness.

- We incorporate in-context learning (ICL) in the PPA domain to improve the LLM's understanding of PPA optimizations, especially when labeled data are scarce. By carefully creating diverse Verilog to PPA report pairs using different optimization strategies. Further, we create PPA aware prompt and corresponding strategy testbench for the Verilog design.
  Compared with state-of-the-arts (SOTAs), e.g., RTLLM [18], VerilogEval [15], our VeriPPA achieves a success rate of 62.0% (+16%) for functional accuracy and 81.37% (+8.3%) for syntactic correctness in Verilog code generation on RTLLM dataset. On the VerilogEval dataset, our framework achieves 99.56% syntactic correctness and 43.79% functional correctness surpassing current SOTA methods.

TABLE I
COMPARISON OF PPA AND VERILOG CODE GENERATION USING LLM WORKS .

Work	Vanilla PPA	PPA Optimization	Accuracy
VeriGen [32]	No	No	Moderate
RTLCoder [16]	No	No	High
ChipNeMo [14]	No	No	Moderate
VerilogEval [15]	No	No	Moderate
BetterV [28]	No	No	High
Revisiting VerilogEval [29]	No	No	High
RTLLM [18]	Yes	No	Moderate
ChipGPT [4]	Yes	No	Moderate
VerilogCoder [10]	No	NO	High
LLM-VeriPPA (Our Work)	**Yes**	**Yes**	**High**

II. BACKGOUND AND RELATED WORK

Register Transfer Logic (RTL) is an critical abstraction in chip design that outlines how data moves between logical operations and registers. RTL is typically described using HDLs such as Verilog. In modern chip design workflows, human engineers manually convert design specifications into HDL before synthesizing them into circuits [1]. This manual translation process is time-consuming and susceptible to errors, which can lead to potential flaws in the hardware circuit designs. Recent advancements in Artificial Intelligence (AI), particularly LLMs, have enabled the automation of translating design specifications into HDL by understanding the instructions and generating codes in HDLs such as Verilog. The ability to generate HDL that meets specific design requirements, such as PPA, is crucial in chip design.

Finetune LLMs. The LLMs have demonstrated various capabilities such as comprehension, reasoning, instruction following, and coding [22]. However, their capability in generating practical hardware Verilog codes is limited because of insufficient available Verilog codes due to propriety natures of circuit designs [7]. To address these challenges researches fine-tuned LLMs on hardware datasets. Thakur *et al.* [33] advocate for the fine-tuning of open-source LLMs such as CodeGen [24] to specifically generate Verilog code tailored for target designs. Subsequently, Chip-Chat [2] delves into the intricacies of hardware design using LLMs, highlighting the markedly superior performance of ChatGPT compared to other open-source LLMs. Chip-GPT [3] also focuses on the task of register-transfer level (RTL) design by leveraging the capabilities of ChatGPT. However, these works mainly target the scale of simple and small circuits (e.g., < 20 designs with a average of < 45 Verilog code lines), as pointed out in [18].

Enrich Verilog Source. Several recent efforts focus on enriching Verilog codes. RTLLM [18] introduces a benchmarking framework consisting of 30 designs that are specifically aimed at enhancing the scalability of benchmark designs. Furthermore, it utilizes effective prompt engineering techniques to improve the generation quality. MG-Verilog [39] provides the multi-level descriptions along side with code sample but its reliance on Llama-2-70B-chat [34] for annotation raises quality concerns about the dataset. VerilogEval [15] assesses the performance of LLM in the realm of Verilog code generation for hardware design and verification. It comprises 156 problems from the Verilog instructional website HDLBits. However, VerilogEval [15] does not offer PPA analysis for the generated codes. In RTLLM, the generated Verilog codes are directly extracted and synthesized using commercial tools to obtain PPA results, without PPA constraint-based feedback. Thus they suffer from limited generation quality.

Verilog Code Agents VerilogCoder [10] introduces the multiple autonomous AI agents based on Abstract Syntax Tree (AST) based waveform tracing, graph planner, and other tools. These highly domain specific AI agent's output does not work on smaller LLMs and models goes to hallucination. Further, It does not give provide any PPA optimizations.

III. FRAMEWORK

A. Design Overview

In our VeriPPA framework, as illustrated in Figure 1, we use a text-based description ($.txt$ file) of hardware design, designated as L, to serve as input/prompt for the LLMs. L details the module name, and specifies both input and output signals with the corresponding bit widths.

In the first stage, highlighted in light red, we use LLM to parse the text-based description L and generate the corresponding Verilog code V. V is then subjected to syntax and functionality checks using the ICARUS Verilog simulator [37] and a design-specific testbench T. Should V fail these checks, we utilize VeriRectify (Section III-C) to provide an automated prompt to the LLM to correct the errors. If V passes, it is synthesized to evaluate the Power, Performance, and Area (PPA) of the design. The second stage, highlighted in light green, assesses the design-specific PPA requirements. We compare the PPA metrics of the synthesized code (after design compiler) against the design constraints. If not meeting these constraints, a PPA-aware prompt (using in context learning) is fed back into the LLM for further optimization. Otherwise, it is saved as part of the dataset. The details of each technique are described in the subsequent subsections.

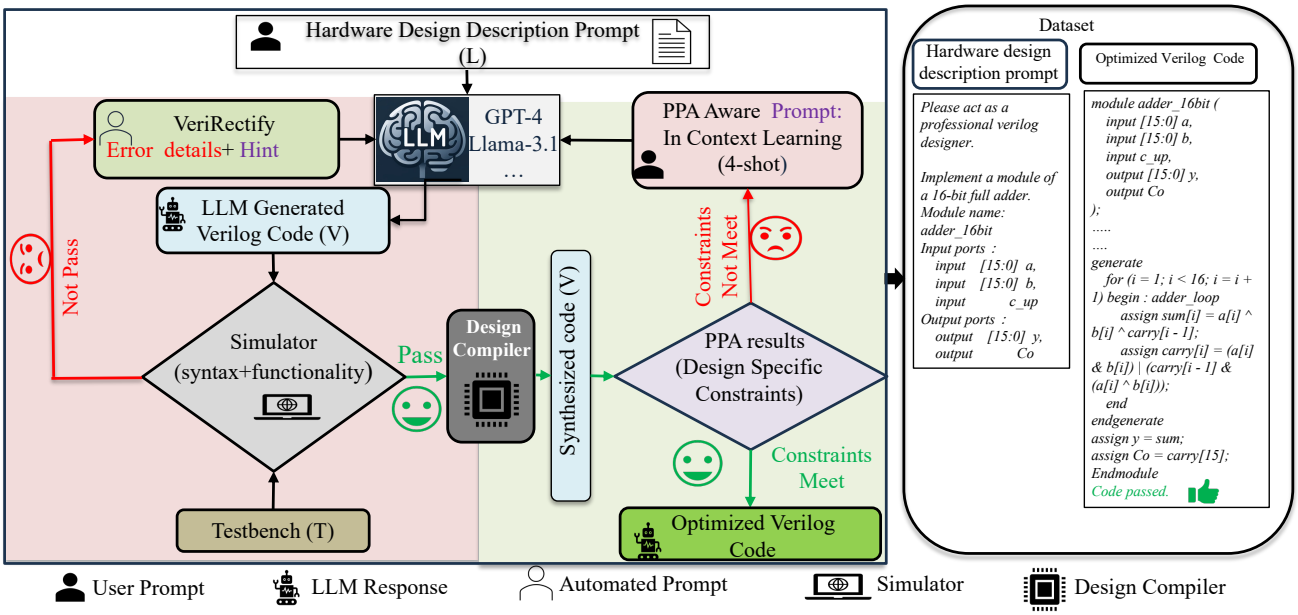

Fig. 1. VeriPPA framework.

B. Code Generation and Testing

VeriPPA incorporates the ICARUS Verilog simulator [37] to automate the evaluation (testing) of the generated codes. In contrast to high-level program languages such as Python, Verilog requires the use of testbenches, $T = \{T_1, T_2, \ldots, T_m\}$, to systematically assess the code's functionality, encompassing a wide array of test scenarios. Integrating the ICARUS Verilog simulator into VeriPPA provides immediate feedback on the code's syntactical and operational integrity. The ICARUS Verilog simulator could pinpoint the exact location of syntactic errors or functional fails based on testbench test case failures. This integrated approach contrasts with frameworks such as RTLLM [18], where an external simulator is used to check the correctness of the generated Verilog codes.

C. VeriRectify

We create a multi-iteration dialogue with an error feedback mechanism (Figure 2 (a) and (b)), analogous to human problem-solving techniques. This method is designed as a recursive function that improves the output by carefully analyzing and correcting the errors found in previous iterations. Let V_i denote the Verilog code resultant from the i^{th} iteration, and E_i represent the associated set of identified errors at this stage. Initially, V_0 is the first generated code accompanied by its detected errors E_0. Then the refinement function, $R(V_i, E_i)$, which takes as input V_i and E_i, and yields an enhanced code version V_{i+1} as output. Simultaneously, an error detection function $D(V_i)$ is employed to identify errors within V_i, generating E_i. The iterative process can be viewed as follows:

$$V_{i+1} = R(V_i, E_i) \quad \text{and} \quad E_{i+1} = D(V_{i+1}) \tag{1}$$

This process repeats until either no errors are detected or a predefined iteration limit, K is reached, i.e., the iteration halts if, $D(V_{i+1})=\emptyset$ or $i = K$. K is empirically adjustable (say 4) based on observed results of code generation. Thus, the multi-round conversation method enhances code quality with each iteration until an optimal or satisfactory solution is reached within the bounds of K. in this context is a systematic, iterative algorithm aimed at progressively minimizing the error in the generated Verilog code, enhancing code quality with each iteration until an optimal or satisfactory solution is reached within the bounds of K.

D. Power Performance and Area (PPA).

RTL simulation does not guarantee that the design (generated Verilog code) meet the design specification after we fabricate. Furthermore, the quality of the hardware design must be measured by its power, performance, and area metrics.

Our approach takes a step further by inspecting PPA of the design V which passes the *VeriRectify* process as the following:

$$V = \begin{cases} V & \text{if PPA}(V) \text{ satisfies,} \\ \text{VeriRectify}(V, \text{PPA}(V)) & \text{otherwise.} \end{cases} \tag{2}$$

Our PPA check calls Synopsys Design Compiler to perform logic synthesis (and technology mapping) on the open-source ASAP 7nm Predictive PDK [35]. We check all designs' warning/error messages during the logic synthesis, and the power (μW), area (μm^2), and clock (ps) for quality. When the Verilog design can be synthesized and meets the PPA goal, it results in a pass. Otherwise, both the design and its corresponding PPA report will be fed back to the VeriRectify (Section III-C) for refinement.

The aim of PPA checking is to ensure the created design operates within a reasonable clock period, with acceptable

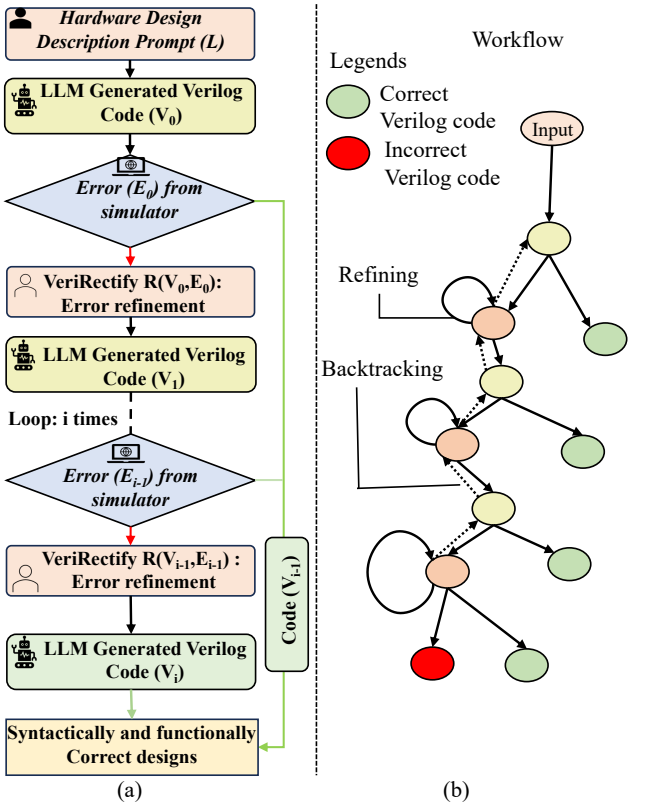

Fig. 2. (a) Multi-round conversation with error feedback; (b) Workflow of the process.

power and area. This requires determining the power and area under optimal timing, or the smallest clock period.

IV. Evaluation

A. Datasets

In assessing our VeriPPA framework, We utilize two benchmark datasets, the RTLLM dataset [18] includes 29 designs, and the VerilogEval dataset [15], which comprises two subsets: VerilogEval-human, featuring 156 designs, and VerilogEval-machine, consisting of 108 designs.

B. Experimental Setup

We demonstrate the effectiveness of VeriPPA for generating PPA-optimized Verilog for the given designs. We adopt GPT-3.5 [25], GPT-4 [26], GPT-4o [27], Llama-2-7B [34], Llama-3-8B [20], Codellama-7B [31], Llama3.1-405B [21], RTL-Coder [17], and DeepSeek Coder [8] as our LLM models. We use n=1, temperature temp = 0.7, and a context length of 2,048. Further, we incorporate the ICARUS Verilog simulator [37] to automate the testing of the generated code. For PPA check, we perform the logic synthesis using Synopsys Design Compiler with `compile_ultra` command and we use the ASAP 7nm Predictive PDK [35]. We implement an in-house simulator to sweep the timing constraints to find the fastest achievable clock frequency for all the generated designs. All experiments are conducted on a Linux- based host with AMD EPYC 7543 32-Core Processor and an NVIDIA A100-SXM 80 GB.

C. Generation Correctness

We evaluate Verilog generation accuracy using two primary metrics: syntax checking and functionality verification. Table II shows results from our methodology of correcting Verilog code through multiple correction attempts. For each design description, five codes are generated, with up to four corrections per generation. The number of correction attempts is set to four because after this point, correction efficiency decreases due to repetitive model outputs. For the GPT-3.5 model, initial syntax correctness is 44.13% and functionality correctness is 24.13%. Applying the VeriPPA framework changes syntax correctness to 65.51% and functionality correctness to 31.03%. For the RTLLM baseline, syntax correctness is 32.41% and functionality correctness is 20.68%. With the VeriPPA framework, these become 55.17% and 31.03%. For the GPT-4 model, syntax correctness is 66.20% and functionality correctness is 37.93%. With VeriPPA, syntax correctness is 81.37% and functionality correctness is 48.27%. RTLLM baseline scores are 60.00% for syntax and 34.48% for functionality. After applying VeriPPA, these become 77.93% and 48.27%. Testing GPT-4 with four-shot learning, syntax correctness is 70.34% and functionality correctness is 37.93%. With VeriPPA, syntax correctness is 79.31% and functionality correctness is 41.37%. For RTLLM, syntax correctness is 66.89% and functionality correctness is 44.82%. With VeriPPA, syntax correctness is 81.37% and functionality correctness is 62.06%. For GPT-4o, syntax correctness is 75.17% and functionality correctness is 44.82%. With VeriPPA, syntax correctness is 86.20% and functionality correctness is 48.27%. RTLLM baseline scores are 75.86% for syntax and 41.37% for functionality. After applying VeriPPA, syntax correctness is 82.06% and functionality correctness is 44.82%.

These results show that applying the VeriPPA framework to both GPT and RTLLM models changes both syntax and functionality correctness across all tested models and settings.

We evaluated VeriPPA using open-source LLMs, demonstrating its strong effectiveness with these models. Starting with the large Llama 3.1-405B model [21], VeriPPA increases syntax accuracy from 31.72% to 80.68% and functionality accuracy from 20.68% to 44.82%, as shown in Table III. For smaller models, VeriPPA continues to enhance syntax correctness. With Llama 2-7B [34] , syntax correctness rises from 20.68% to 27.58%. In the case of Llama 3-8B [20], syntax correctness improves from 3.4% to 17.5%. Codellama-7B [31] shows an increase in syntax correctness from 16.2% to 28.35%. Despite these syntax improvements, these smaller models do not achieve functionality correctness due to the rigorous tests in our test benches. As demonstrated in Table III, VeriPPA is highly effective with larger open-source LLMs.

We evaluate the VeriPPA framework using the VerilogEval-Machine and VerilogEval-Human datasets. Table IV summarizes results for VerilogEval-Machine. For GPT-4, syntax correctness with Revisiting VerilogEval [30] is 92.11%, and with

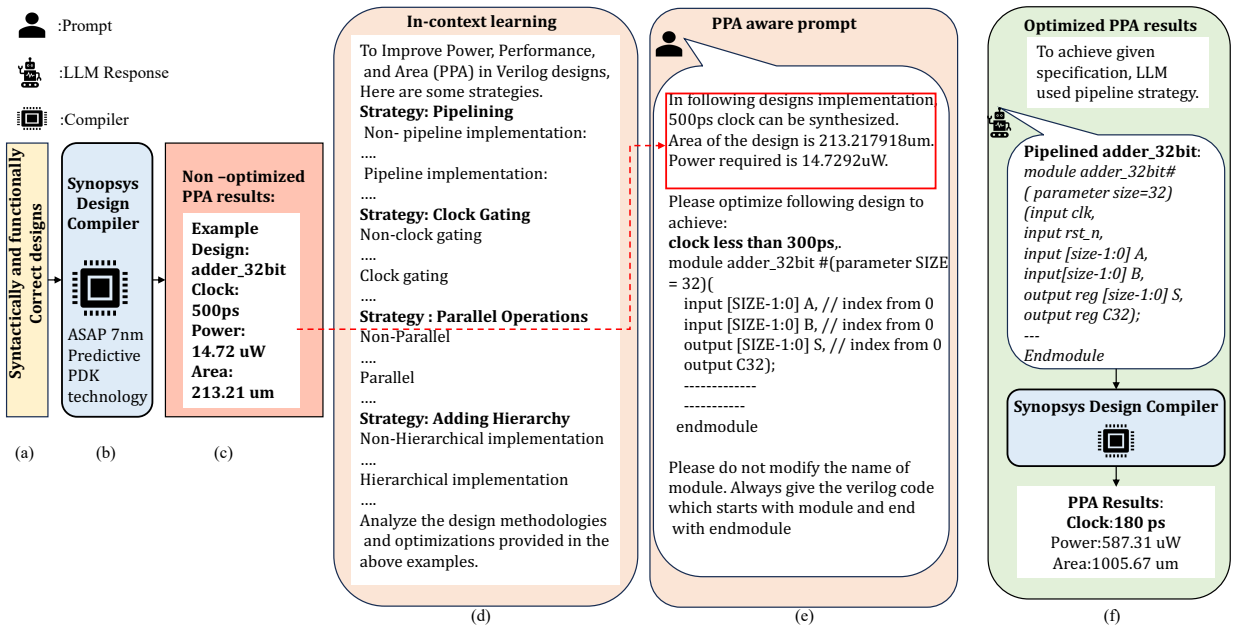

Fig. 3. Optimization Flow; (a) Syntactically and functionally correct designs, (b) Synopsis compiler, (c) Non-optimized PPA results based on 7nm ASAP technology, (d) In-context learning to optimize PPA, (e) PPA aware prompt, (e) Optimized results

TABLE II
COMPARISON OF DIFFERENT MODELS AND RTLLMS METHODS ON RTLLM DATASET.

Model	Syntax (%)		Functionality (%)		Syntax (%)		Functionality (%)	
	w/o VeriPPA	w/ VeriPPA	w/o VeriPPA	w/ VeriPPA	RTLLM	RTLLM w/ VeriPPA	RTLLM	RTLLM w/ VeriPPA
GPT-3.5	44.13	65.51	24.13	31.03	32.41	55.17	20.68	31.03
GPT-4	66.20	81.37	37.93	48.27	60.00	77.93	34.48	48.27
GPT-4 (4-shot)	70.34	79.31	37.93	41.37	66.89	81.37	44.82	62.06
GPT-4o	75.17	86.20	44.82	48.27	75.86	82.06	41.37	44.82

TABLE III
OPEN SOURCE LLM RESULTS.

Model	Without VeriPPA		With VeriPPA	
	Synt. (%)	Funct. (%)	Synt. (%)	Funct. (%)
Llama3.1-405B	31.72	20.68	80.68	44.82
Llama-2-7B	20.68	0	27.58	0
Llama-3-8B	3.4	0	17.24	0
CodeLlama 7B	16.2	0	28.35	0

TABLE IV
COMPARISON ON VERILOGEVAL-MACHINE DATASET: REVISITING VERILOGEVAL [30] VS. VERIPPA.

Model	Revisiting VerilogEval [30]		VeriPPA	
	Syntax (%)	Function (%)	Syntax (%)	Function (%)
GPT-4	92.11	33.57	99.56	43.79
GPT-4 (4-shot)	90.21	35.76	95.91	45.25
RTL-Coder	0.38	0.64	27.94	1.28
DeepSeek-Coder-67B	55.12	16.66	78.97	24.35

VeriPPA it is 99.56%. Functionality correctness for GPT-4 is 33.57% for Revisiting VerilogEval and 43.79% for VeriPPA. Using GPT-4 with four-shot learning, syntax correctness is 90.21% (Revisiting VerilogEval) and 95.91% (VeriPPA), while functionality correctness is 35.76% and 45.25%, respectively. For the VerilogEval-Human dataset, as shown in Table V, GPT-4 syntax correctness is 91.28% with Revisiting VerilogEval and 97.17% with VeriPPA. Functionality correctness is 29.48% (Revisiting VerilogEval) and 39.74% (VeriPPA). The four-shot learning variant of GPT-4 shows similar results, with syntax correctness at 88.97% and 95.76%, and functionality correctness at 29.4% and 39.74% for Revisiting VerilogEval and VeriPPA, respectively.

We also evaluate the VeriPPA framework using RTL-Coder [17] and DeepSeek Coder [8]. For DeepSeek Coder on the VerilogEval-Machine dataset, syntax correctness is 55.12% with Revisiting VerilogEval and 78.97% with VeriPPA. Functionality correctness is 16.66% (Revisiting VerilogEval) and 24.35% (VeriPPA). For RTL-Coder, syntax correctness is 0.38% with Revisiting VerilogEval and 27.94% with VeriPPA, while functionality correctness is 0.64% and 1.28%, respectively. These results show that applying VeriPPA with the VeriPPA framework changes both syntax and functionality correctness across different models and datasets. Using four-shot learning also changes functionality correctness, indicating the benefit of multi-sample correction for Verilog code generation.

D. PPA Optimization

In this section, we shift focus from verifying the correctness of the generated Verilog codes to optimizing its quality. In VeriPPA, We use the Synopsys Design Compiler to synthesize our designs and generate PPA reports. Table VI shows the

TABLE V
COMPARISON ON VERILOGEVAL-HUMAN DATASET: REVISITING VERILOGEVAL [30] VS. VERIPPA.

Model	Revisiting VerilogEval [30]		VeriPPA	
	Syntax (%)	Function (%)	Syntax (%)	Function (%)
GPT-4	91.28	29.48	97.17	39.74
GPT-4 (4-shot)	88.97	29.48	95.76	39.74

TABLE VI
PPA RESULTS OF GENERATED VERILOG CODE

Design Name	GPT-4			GPT-4 (4-shot)		
	Clock (ps)	Power (μW)	Area (μm^2)	Clock (ps)	Power (μW)	Area (μm^2)
adder_8bit	318.5	6.3	38.5	333.1	6.1	42.9
adder_16bit	342.2	10.9	104.5	135.1	41.1	152.8
adder_32bit	500.0	14.2	211.6	500.0	14.7	213.2
multi_booth	409.0	112.1	526.0	409.0	112.1	526.0
right_shifter	47.5	144.3	42.9	47.5	144.3	42.9
width_8to16	74.1	223.2	145.8	145.6	128.7	157.2
edge_detect	61.5	49.0	23.3	61.5	49.0	23.3
mux	54.7	215.3	86.1	54.7	215.3	86.1
pe	500.0	552.5	2546.5	500.0	541.0	2488.6
asyn_fifo	295.2	406.4	1279.3	228.3	526.6	1295.4
counter_12	134.4	33.1	40.6	124.5	34.6	36.4
fsm	88.3	32.7	31.5	68.7	49.0	50.2
multi_pipe_4bit	254.7	40.7	131.3	-	-	-
pulse_detect	10.3	187.5	13.5	32.7	59.1	13.5
calendar	-	-	-	208.6	86.6	199.0

results of different designs with different LLM models. It shows the without optimization PPA (Vanilla) result of each design. To demonstrate VeriPPA ability to optimize PPA later, we show best Vanilla PPA from multiple PPA reports. For example, the $pulse\_detect$ design passes five times functionally and syntactically. Therefore, in post-synthesis, we collect five PPA reports for the $pulse\_detect$ design, and we select the best PPA result to include in Table VI. However, these best PPA results do not meet design-specific PPA requirements,

To address this, We further perform the PPA constraint-based feedback mechanism, integrated with ICL, as illustrated in Figure 3. This approach represents a significant step towards aligning LLM-generated codes with application-specific PPA requirements. Figure 3 demonstrates our process, starting with the collection of syntactically and functionally correct designs and generating non-optimized PPA results as shown in Figure 3 (a), (b), and (c). The non-optimized PPA results do not meet application-specific PPA requirements. For example, $adder\_32bit$, can be synthesized with a 500ps clock as shown in Figure 3 (c). However, this clock speed does not fulfill the rapid clock requirements necessary for some applications, such as cryptographic hardware which consists of adders, where a fast clock is crucial, but area and power constraints are less critical. To enhance the speed of $adder\_32bit$, we impose a clock constraint, aiming for a clock speed of less than 300ps, as outlined in the PPA constraint-based prompt in Figure 3 (e). The framework instructs the LLM to consider various optimization strategies, including Pipelining, Clock Gating, Parallel Operation, and Hierarchical Design as depicted in Figure 3 (d).

Upon providing the PPA-based constraint prompt and context to the LLM, we analyze the resultant Verilog code for syntax and functional accuracy, making corrections where necessary. If the code passes both checks, we proceed to its final synthesis, achieving an optimized Verilog code as shown in Figure 3 (f), where the $adder\_32bit$ operates at an improved 180ps clock. In Table VII, we present the results of selected optimized designs. Notably, no design from the VerilogEval [15] dataset is present in Table VII, as those designs did not require complex optimization.

TABLE VII
PPA OPTIMIZED VERILOG DESIGN RESULTS

Design Name	Clock (ps)	Power (μW)	Area (μm)
adder_32bit	180.0	587.31	1005.67
multi_booth	123.2	42.39	42.92
pe	325.0	1206.0	4863.88
asyn_fifo	114.8	988.92	1344.86

TABLE VIII
ANALYSIS TABLE - ONE ITERATION COMPARISON USING GPT-4O [27] MODEL

Method	MACs	Tokens	Accuracy (%)	
			Syntax	Functionality
Self-panning	647589.84	317446	77.93	41.37
VeriPPA	552317.76	270744	80.68	41.37

E. Computational cost analysis between self-planning (SOTA) and VeriPPA

. To provide a fair comparison, we first limit both methods to a single iteration: the self-planning (SOTA) and our VeriPPA approach. As shown in Table VIII, VeriPPA used 46,702 fewer tokens (a 14.71% reduction) compared to SOTA, while also achieving better syntax accuracy 80.68% for VeriPPA versus 77.93% for SOTA, with the same functionality accuracy 41.37%. Our estimation shows, for a GPT-4o VeriPPA requires 95272.08 trillion less Multiply-Accumulate Operations (MACs) than the SOTA . Overall, results demonstrate that VeriPPA has lower computational costs than the SOTA method while maintaining or improving accuracy.

V. CONCLUSION

In this paper, we introduce a novel framework VeriPPA, designed to assess and enhance LLM efficiency in this specialized area. Our method includes generating initial Verilog code using LLMs, followed by a unique two-stage refinement process. The first stage focuses on improving the functional and syntactic integrity of the code, while the second stage aims to optimize the code in line with Power-Performance-Area (PPA) constraints, an essential aspect of effective hardware design. This dual-phase approach of error correction and PPA optimization has led to notable improvements in the quality of LLM-generated Verilog code. Our framework achieves 62.0% (+16%) for functional accuracy and 81.37% (+8.3%) for syntactic correctness in Verilog code generation, compared to SOTAs.

REFERENCES

[1] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Chip-chat: Challenges and opportunities in conversational hardware design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, page 1–6. IEEE, September 2023.

[2] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Chip-chat: Challenges and opportunities in conversational hardware design. *arXiv preprint arXiv:2305.13243*, 2023.

[3] Kaiyan Chang et al. Chipgpt: How far are we from natural language hardware design, 2023.

[4] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. Chipgpt: How far are we from natural language hardware design, 2023.

[5] Wen Chen et al. Novel test detection to improve simulation efficiency: A commercial experiment. In *ICCAD'12*, page 101–108, New York, NY, USA, 2012.

[6] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *DAC '03*, page 286–291, New York, NY, USA, 2003.

[7] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Celine Lin. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models, 2025.

[8] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[9] Winston Haaswijk et al. Deep learning for logic optimization algorithms. In *2018 ISCAS*, pages 1–4, 2018.

[10] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool, 2025.

[11] Abdelrahman Hosny et al. Drills: Deep reinforcement learning for logic synthesis. In *2020 25th ASP-DAC*, pages 581–586, 2020.

[12] Hanbin Hu et al. Hfmv: Hybridizing formal methods and machine learning for verification of analog and mixed-signal circuits. In *DAC '18*, New York, NY, USA, 2018.

[13] Rongjian Liang et al. Drc hotspot prediction at sub-10nm process nodes using customized convolutional network. In *ISPD '20*, page 135–142, New York, NY, USA, 2020.

[14] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, Bonita Bhaskaran, Bryan Catanzaro, Arjun Chaudhuri, Sharon Clay, Bill Dally, Laura Dang, Parikshit Deshpande, Siddhanth Dhodhi, Sameer Halepete, Eric Hill, Jiashang Hu, Sumit Jain, Ankit Jindal, Brucek Khailany, George Kokai, Kishor Kunal, Xiaowei Li, Charley Lind, Hao Liu, Stuart Oberman, Sujeet Omar, Ghasem Pasandi, Sreedhar Pratty, Jonathan Raiman, Ambar Sarkar, Zhengjiang Shao, Hanfei Sun, Pratik P Suthar, Varun Tej, Walker Turner, Kaizhe Xu, and Haoxing Ren. Chipnemo: Domain-adapted llms for chip design.

[15] Mingjie Liu et al. VerilogEval: evaluating large language models for verilog code generation. In *ICCAD'23*, 2023.

[16] Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2024.

[17] Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique, 2024.

[18] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model, 2023.

[19] Dani Maarouf et al. Machine-learning based congestion estimation for modern fpgas. In *FPL'18*, pages 427–4277, 2018.

[20] Meta. Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3/, 2024. Accessed: 2024-06-15.

[21] Meta AI. Meta llama 3.1. Meta AI Blog, July 2024. Accessed: October 15, 2024.

[22] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey, 2024.

[23] Andre Nakkab, Sai Qian Zhang, Ramesh Karri, and Siddharth 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*, MLCAD '24, New York, NY, USA, 2024. Association for Computing Machinery.

[24] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[25] OpenAI. Gpt-3.5. https://platform.openai.com/docs/models/gpt-3-5, 2023. Accessed on 15/11/2023.

[26] OpenAI. Gpt-4. https://platform.openai.com/docs/models/gpt-4, 2023. Accessed on 15/11/2023.

[27] OpenAI. Gpt-4o model documentation. https://platform.openai.com/docs/models/gpt-4o, 2024. Accessed: 2024-10-15.

[28] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. Betterv: Controlled verilog generation with discriminative guidance, 2024.

[29] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting verilogeval: Newer llms, in-context learning, and specification-to-rtl tasks. *arXiv preprint arXiv:2408.11053*, 2024.

[30] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting verilogeval: A year of improvements in large-language models for hardware code generation, 2025.

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

[32] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation, 2023.

[33] Shailja Thakur et al. Benchmarking large language models for automated verilog rtl code generation. In *DATE'23*, pages 1–6. IEEE, 2023.

[34] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[35] Vinay Vashishtha, Manoj Vangala, and Lawrence T Clark. Asap7 predictive design kit development and cell design technology co-optimization. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017.

[36] Samuel Ward et al. Pade: A high-performance placer with automatic datapath extraction and evaluation through high dimensional data learning. In *DAC'12*, pages 756–761.

[37] S. Williams. The icarus verilog compilation system, 2023. [Online]. Available: https://github.com/steveicarus/iverilog.

[38] Haoyuan Wu, Zhuolun He, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. Chateda: A large language model powered autonomous agent for eda. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(10):3184–3197, October 2024.

[39] Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation, 2024.