Lorecast: <u>Layout-Aware Performance and</u> Power Forecasting from Natural Language

Runzhi Wang*,Prianka Sengupta*,Cristhian Roman-Vicharra*,Yiran Chen[†], Jiang Hu*

*Texas A&M University

[†]Duke University

Abstract—In chip design planning, obtaining reliable performance and power forecasts for various design options is of critical importance. Traditionally, this involves using system-level models, which often lack accuracy, or trial synthesis, which is both labor-intensive and time-consuming. We introduce a new methodology, called Lorecast, which accepts English prompts as input to rapidly generate layout-aware performance and power estimates. This approach bypasses the need for HDL code development and synthesis, making it both fast and user-friendly. Experimental results demonstrate that Lorecast achieves accuracy within a few percent of error compared to post-layout analysis, while significantly reducing turnaround time.

I. Introduction

Chip design planning is the process of evaluating and optimizing key metrics such as performance and power during the early stages of hardware development. In this process, predicting the performance and power of various design options is a critical yet challenging task. Engineers rely on these performance metrics to make informed decisions about architectural choices, resource allocation, and overall design optimization. For instance, designers might pose questions such as, "If the unfolding factor for an Infinite Impulse Response (IIR) filter is adjusted from x to y, will the block-level power constraint be violated?" or "Will replacing a carry-ripple adder with a carry-lookahead adder create significant challenges for timing closure?"

Although system-level models and transaction-level models [1] are useful in planning, they are loosely timed or approximately timed and thus incapable of providing sufficiently accurate estimates. Architecture-level models, such as McPAT [2], are mostly restricted to microprocessor designs and can have very large errors. For example, the power estimate error can be as large as 200% [3]. Errors at early design stages can propagate throughout the design process, resulting in costly revisions, delays, or suboptimal designs. In competitive markets such as Artificial Intelligence (AI) accelerators, Internet of Things (IoT), and high-performance computing, time-to-market is crucial, and slow iterations can hinder innovation.

Alternatively, designers can write Hardware Description Language (HDL) code, followed by logic and layout synthesis, which demands significant time and labor investments. The accuracy of architecture-level models can also be improved by machine learning-based calibration [4]. However, the calibration still requires expensive Register Transfer Level (RTL) implementation and synthesis. While some attempts have been made to predict performance and power using designer-written HDL code, creating the HDL code itself is a time-consuming and challenging process. When a designer has an idea, it often takes several times longer to translate that idea into HDL code. This creates a bottleneck, especially in the early design stages, where rapid iterations are needed to efficiently explore the design space. The challenge lies in providing a quick and reliable way to estimate performance and power based on a designer's idea.

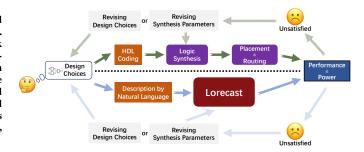


Fig. 1. Traditional flow vs. proposed flow with Lorecast approach: from natural language to performance and power.

To address this problem, we propose an approach called Lorecast, which transforms a designer's natural language description of their idea directly into layout-aware performance and power estimates. Figure 1 illustrates how Lorecast streamlines the traditional flow by avoiding manual HDL coding, logic and layout synthesis, while enabling natural-language-driven performance and power estimation in early design stages. The flow with Lorecast accelerates design planning, enabling faster and more efficient exploration of design options. Moreover, Lorecast reduces the need for system/architecture-level designers to have in-depth knowledge of HDL code. A key component of Lorecast is Large Language Model (LLM)-based automatic Verilog code generation. Distinguished from existing approaches, Lorecast significantly relaxes the requirement for functional correctness of the generated Verilog code. Functional correctness has been a difficult challenge to the practical and large-scale adoption of LLM-based Verilog code generation. However, Lorecast mitigates this challenge by using syntax-correct and structurally similar Verilog code generated by LLMs, which is sufficient for its purposes. The novelty of Lorecast primarily lies in its integration of LLM-based Verilog code generation with Machine Learning (ML)-based predictive models, effectively mitigating the limitations of standalone LLM-generated code. Without solving the challenge of functional correctness, the practical utility of LLM-generated Verilog code remains extremely limited, if not entirely nonexistent. Lorecast significantly broadens the applicability of such code by relaxing the requirement of functional correctness, effectively breathing life into LLM-generated Verilog designs.

The contributions of this work are summarized as follows.

- Performance and power prediction from natural language.
 The Lorecast framework allows designers to obtain performance and power estimates directly from English descriptions of their ideas, bypassing the need for HDL coding.
- High forecast accuracy. Lorecast achieves accurate estimates, with an average percentage error of only 2% for both power and Total Negative Slack (TNS), compared to the post-layout analysis by a commercial tool. Even for cases where the Verilog codes generated by the LLM are functionally incorrect, the average forecast errors are 5% and 7% for power and TNS, respectively,

compared to analysis of functionally correct designs.

- Acceleration of the design planning process. Lorecast accelerates the performance and power estimation process by $4.6\times$ compared to conventional methods that involve manually writing Verilog code, logic and layout synthesis.
- Relaxation of functional correctness requirement. We provide analysis and experimental evidence demonstrating that Lorecast remains effective even when the generated Verilog code is functionally incorrect, primarily due to its structural similarity to functionally correct designs.
- Enhancement of syntax correctness through prompting techniques. We propose two complementary techniques to improve the syntax correctness of LLM-generated code. The first technique introduces pseudocode as an intermediate reasoning step to guide HDL code generation, resulting in a 33% to 43% improvement in syntax correctness compared to free narrative-style prompting. Building on this, the second technique employs iterative prompting with regulated feedback, improving syntax correctness by 6% over naïve error-based iteration and 11% over direct generation without feedback.
- Evaluation with circuits significantly larger than existing ones. So far, LLM-based Verilog code generation techniques are mostly restricted to small circuits. We developed circuit cases that double the sizes in terms of cell count compared to the latest publicly released testcases. On these larger cases, Lorecast achieved 100% syntax correctness.
- Superiority over direct LLM-based forecasting methods. Experimental results show that Lorecast is a much more promising approach than direct LLM-based (including a fine-tuned LLM) forecasting without generating Verilog code.

II. RELATED WORKS

To the best of our knowledge, there is no prior study on forecasting circuit performance and power from natural language. However, there have been related works, which are briefly reviewed as follows.

Leveraging LLMs to generate HDL code. Previous research has explored the feasibility of using LLMs for design tasks. ChatChisel [5] leveraged LLMs alongside collaboration and Retrieval-Augmented Generation (RAG) [6] techniques to enhance code generation performance, successfully producing a RISC-V CPU and demonstrating the potential of LLMs in generating complex circuits. ChatCPU [7] was a framework combining LLMs with CPU design automation, which has been used to successfully design a CPU and complete its tape-out. BetterV [8] utilized a discriminator to guide Verilog code generation. Other studies focused on using LLMs to assist in hardware design. ChipGPT [9] demonstrated that LLMs can aid users in understanding complex designs. VGV [10] took advantage of LLMs' capabilities in computer vision to generate Verilog code directly from circuit diagrams.

Evaluating Verilog code generation capabilities of LLMs. Some studies focused on evaluating the code generation capabilities of LLMs, primarily in terms of syntax and functional correctness. In [9], the code generation capabilities of ChatGPT were compared to other LLMs. VerilogEval [11] introduced a benchmark for evaluating the correctness of Verilog code generation, along with an automated evaluation framework. RTLLM [12] proposed a benchmark and evaluated how prompt styles impact code generation accuracy, also presenting a prompting technique to improve correctness. RTL-Repo [13] developed a large-scale benchmark for assessing Verilog code generation capabilities of LLMs, containing over 4,000 Verilog code

samples. CreativEval [14] proposed a new perspective by focusing not on correctness but on fluency, flexibility, originality, and refinement.

Enhancing Verilog code generation capabilities of LLMs. To improve the accuracy of code generation, some researchers have experimented with fine-tuning open-source and lightweight LLM models [15] [16] [17]. In [18], researchers used a framework that integrates datasets categorized by different design complexities to fine-tune LLMs for specific tasks. RTLLM [12] proposed a structured prompt technique that guides LLMs in generating HDL code. Autochip [19] proposed a framework that uses feedback from syntax-checking tools to correct syntax errors in LLM-generated code, however it still can not guarantee the syntax correctness. OriGen [20] used feedbackbased correction to collect datasets for augmentation and improve code generation. Both RTLFixer [21] and AutoVCoder [22] utilized RAG to enhance syntax error correction in code. EDA Corpus [23] and MG-Verilog [24] proposed datasets tailored to various application scenarios to improve Verilog code generation capabilities in LLMs. Additionally, other researchers have proposed frameworks for generating datasets specifically for fine-tuning LLMs [25]. VerilogReader [26] proposed a framework to expand the comprehension scope of LLMs for improved generation of Verilog test code.

Leveraging LLMs to generate HDL testbenches. Recent studies have explored using LLMs to automate Verilog testbench generation. AutoBench [27] generated self-checking testbenches from design descriptions via scenario extraction and syntax correction. CorrectBench [28] was built on AutoBench by adding automatic validation and iterative self-correction using behavior comparisons across RTL variants. Although CorrectBench introduced enhancements to improve functional accuracy and coverage, it ultimately inherited the same fundamental limitation as AutoBench: the generated testbenches often failed to ensure correctness.

Predicting PPA from HDL code. Some studies focused on obtaining design evaluations during the early design stages. In [29], a machine learning method was proposed to perform Verilog-based evaluations without requiring synthesis. Some researchers concentrated on predictions specifically based on synthesis results [30]. MasterRTL [31] introduced an approach for design evaluation using a simple operator graph to describe relationships between logic gates. In [32], researchers leveraged the Look-Up Table (LUT) Graph from Verilog code to predict. Additionally, some researchers employed Graph Neural Networks (GNNs) to estimate the design's maximum arrival time by predicting component delays and slopes [33].

III. BACKGROUND

A. Large Language Models

Large Language Models (LLMs) are advanced AI systems designed to understand and generate human-like text by processing vast amounts of language data [34]. These models are typically based on Transformer architectures [35] and employ billions of parameters to capture complex patterns and relationships within language, enabling them to perform tasks such as text generation, translation, summarization, and question answering [36]. Prominent examples include OpenAI's GPT-4 [37], Google's Gemini 1.5 Pro [38], and DeepSeek [39], showcasing the immense capabilities of LLMs. With 175 billion parameters and a context capacity of 1 million tokens, these models achieve near-human performance across a variety of Natural Language Processing (NLP) tasks. The primary strength of LLMs lies in their pretraining on extensive text corpora, which gives them a general understanding of language that can be fine-tuned for specific tasks or domains [15] [16] [40]. As LLMs

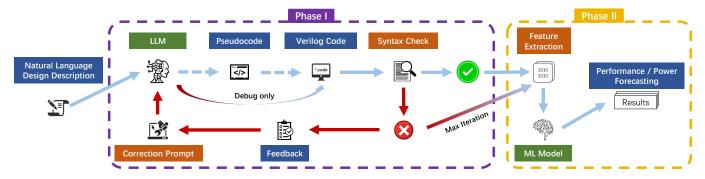


Fig. 2. Overview of the Lorecast methodology.

continue to evolve, their applications are expanding across various fields, including EDA, where they are increasingly employed for tasks such as code generation [7] [8] [10].

B. Verilog and Code Correctness

Verilog is a hardware description language (HDL) widely used in digital design and circuit development for specifying and modeling electronic systems at various abstraction levels, from highlevel functional descriptions to detailed structural representations [41]. Ensuring code correctness in Verilog is crucial because errors at the HDL level can lead to significant functional failures, performance inefficiencies, and increased costs when translated to physical hardware. Code correctness in Verilog includes both syntax correctness, ensuring code is free from syntax errors, and functional correctness, validating that the code's behavior aligns with design specifications [42]. While recent developments [7] [8] in LLM-based code generation have sought to improve Verilog code correctness via automated synthesis and error detection, their effectiveness remains limited—correctness rates often remain low even after such enhancements, due to the lack of deeper semantic understanding, global planning, generalizable control and checking mechanisms.

IV. THE PROPOSED LORECAST METHODOLOGY

A. Overview

The goal of Lorecast is to take natural language prompts as input and produce layout-aware performance and power forecasts for the circuit corresponding to the prompts. An overview of the Lorecast methodology is provided in Figure 2. It consists of two phases. Phase I is LLM-based Verilog code generation, and Phase II is performance/power forecasting according to the generated Verilog code. Although LLM-based Verilog code generation has been studied in prior works, our approach differs significantly in a crucial aspect: functional correctness is far less critical in our case. In conventional approaches [11] [12], functional correctness is essential because the generated Verilog code is typically intended for synthesis. By contrast, in our methodology, functional errors usually have a very small impact on performance and power forecasting. Functional correctness remains a significant challenge for LLM-based Verilog code generation and is far from being well solved. Our innovative use of LLM-based Verilog code generation largely bypasses this challenge. As such, we can focus on syntax correctness, which is much more achievable. Additionally, using Verilog code as an intermediate representation enables significantly better forecasting accuracy compared to direct performance and power predictions using LLMs.

B. Phase I: LLM-Based Verilog Code Generation

As shown in Figure 2, the LLM-based Verilog code generation primarily has two components: (1) The LLM takes English prompts

as input and produces corresponding Verilog code; (2) Syntax check is performed on the code and corrective prompts with feedback are fed to the LLM again for producing improved code. The LLMs for Verilog code generation can be obtained from either existing closed-source models, such as ChatGPT [37] and Gemini [38], or fine-tuning open-source models, such as Llama [43]. A recent analysis in [16] shows that the best results so far were obtained from GPT4 [37], a closed-source model. Therefore, we focus on using closed-source models with prompt engineering enhancements. Although the focus of our study here is closed-source models, our methodology is general and can work with fine-tuned open-source models as well.

TABLE I
EFFECT OF PROMPT AND FEEDBACK DESIGN ON LLM STABILITY AND
CODE GENERATION QUALITY.

Methodology	Prompt structured level	Intermediate representation	Feedback mechanism	LLM non-determinism	Syntax correctness
Free-form	✗ None	✗ None	✗ None	Very high No constrained	*******
VerilogEval [11]	✓ Basic	✗ None	✗ None	High Partially constrained	*Adalaiaic
RTLLM [12]	✓ Highly	✗ None	✗ None	Moderate Clearly constrained	*AAAAA
Autochip [19]	✓ Basic	✗ None	✓ Unstructured	Moderate Errors can be fixed	*AAAAAA
Lorecast	✓ Highly	✓ Pseudocode	✓ Structured	Very low Fully constrained	*AAAAA

Non-determinism is a fundamental challenge underlying the instability of LLM-generated outputs [44] [45]. Structured prompting [11] [12] improves the likelihood of syntactically valid outputs by reducing ambiguity, while feedback-based supervision [19] enforces correctness through error detection and correction. However, during the reasoning process of LLMs, there is often a lack of effective reasoning path planning-while goals may be specified, the intermediate steps are typically unguided or loosely inferred, leading to inconsistencies in reasoning. Chain-of-Thought (CoT) prompting, which encourages LLMs to reason step by step through intermediate reasoning traces, has been employed to improve reasoning in non-HDL code generation [46]. However, its application in hardware design remains largely unexplored. A comparative summary of these methods in terms of prompting strategy, feedback mechanism, LLM non-determinism, and output quality is provided in Table I. By combining these approaches, their strengths are leveraged to enhance the overall quality and stability of the generated code.

We propose a new prompting methodology inspired by existing techniques. The first component of our methodology is **Regulated Prompting with Implicit CoT (RePIC)**, a structured prompting strategy that introduces pseudocode as an intermediate step to guide implicit CoT reasoning. Examples of the original description and the RePIC template are provided in Figure 3. Since Lorecast is to provide circuit designs with quick estimates of performance and power, instead of helping a layman to design circuits, its users normally have sufficient experience to supply the essential design information in natural language. This information is automatically regularized

by our system into a structured prompt, as illustrated in Figure 3. The regulated prompt then guides the LLM to implicitly generate pseudocode from the design description, which is subsequently converted into corresponding Verilog code, as shown in Figure 4.

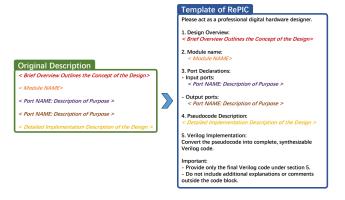


Fig. 3. The original description (left) and the template of RePIC (right).

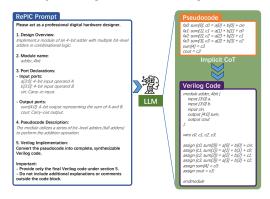


Fig. 4. RePIC pipeline: from RePIC prompt to Implicit CoT-guided Verilog code generation.

The second component is **Iterative Prompting with Regulated Error Feedback (I-PREF)**. I-PREF is performed when there are syntax errors in the Verilog code generated by the LLM. Various tools are available for checking Verilog code syntax correctness, including Icarus Verilog [47], Synopsys VCS [48], Xcelium [49], PyVerilog [50], and Verilator [51]. In this work, we adopt Icarus Verilog for syntax checking. In I-PREF, the error messages along with the Verilog code with syntax errors are sent back to the LLM as a new prompt for generating updated Verilog code. This process is repeated until there is no syntax error or the maximum limit *N* is reached. Usually, *N* is set to be 10 as improvement can rarely be obtained after 10 iterations. The original idea of taking error feedback for iterative prompting was introduced in [19]. However, its feedback prompts are not regulated. By contrast, we propose regulated feedback prompting and an example of such a regulation template is provided in Figure 5.

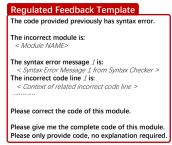


Fig. 5. An example of a template for regulated feedback prompting.

C. Phase II: Performance and Power Forecasting from Verilog Code

In Phase II, the Verilog code generated from Phase I is utilized for performance and power forecasting. In general, the code should have no syntax errors. On the other hand, our method tolerates limited functional errors, i.e., even if the code cannot be synthesized to correct circuits, it can still be applied to provide reasonable performance/power estimates. Several previous works have attempted to make performance and power predictions based on Verilog code [29] [30] [31] [33], and we adopt the approach from [29] with one modification. The models of [29] are trained on post-placement analysis data while the models used by Lorecast are trained on post-routing analysis data. Thus, Lorecast is expected to provide a more accurate forecast in capturing the layout impact. As shown in Figure 6, the input Verilog code is first parsed to obtain an Abstract Syntax Tree (AST) using an off-the-shelf software tool [50]. Next, features are extracted from the obtained AST. Then, an XGBoost model is applied to obtain the performance and power forecast for the corresponding Verilog code with the AST features and EDA tool parameters as input.

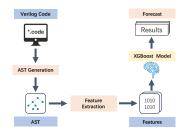


Fig. 6. Performance and power forecasting from Verilog code.

D. Functional Correctness versus Structural Similarity

Lorecast requires the LLM-generated Verilog code to be syntactically correct but not necessarily functionally correct. Why, then, can Lorecast still be effective despite functional inaccuracies? A key reason lies in the AST structure of the generated code. The Verilog code produced by Lorecast not only achieves a high rate of syntax correctness but also exhibits an AST structure that closely resembles that of functionally correct Verilog code. This is illustrated in Figure 7: the AST of Lorecast-generated code in (a) is highly similar to the AST of the functionally correct Verilog code in (b). In contrast, the AST produced by another code generator [11] in (c) differs significantly from that of the correct version shown in (d).

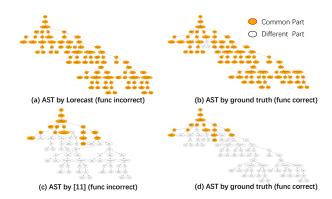


Fig. 7. ASTs of functionally correct and incorrect Verilog code for the same testcase. (d) same as (b). Orange color highlights common subtrees between (a)-(b) and (c)-(d).

V. EXPERIMENTAL RESULTS

A. Experiment Setup

Testcases. In addition to the dataset of RTLLM [12], we have prepared new designs, many of which are larger than those in [12], for training the XGBoost model and testing the techniques. The testcases are distinct from the designs in the training dataset. Seven of the testcases are from [12], one is from [52], and the other seven are newly produced by us. Please note that most testcases from [52] are very small, with a median of only 10 cells, and repetitions with small variances. The statistics of our testcases are shown in Table II in comparison with testcases of previous works. As shown, our testcases are significantly larger than the previous ones. Moreover, our cases cover a wide variety of designs, including datapath designs, such as multipliers and FFT units, and control logic intensive circuits, such as signal generator and Huffman decoder.

TABLE II
CELL COUNT STATISTICS OF TESTCASES USED BY LORECAST IN COMPARISON WITH TESTCASES OF RELATED WORKS.

Work	Num of	Number of cells						
WOIK	designs	Median	Mean	Max				
Chip-Chat [53]	8	37	44	110				
Thakur, et al. [15]	17	9.5	45	335				
RTLLM [12]	30	121	408	2435				
Lorecast testcases	15	890	1806	12031				

Techniques evaluated. Several LLMs are evaluated, including GPT3.5, GPT4, GPT40 [37], Llama3, Llama3.1 [43], Gemini1.5, Gemini1.5Pro [38], and DeepSeek V3. Lorecast is examined with GPT4, GPT40, Gemini1.5Pro, and DeepSeek V3, the four best-performing LLMs for Verilog code generation. In addition, the RePIC technique and I-PREF technique (Section IV-B) are also assessed.

Metrics for evaluating Verilog code generation. In the experiments, we evaluate syntax correctness, functional correctness of the generated Verilog code, and the accuracy of Lorecast forecasting. Syntax checking for LLM-generated Verilog code is performed using Icarus [47]. Functional correctness is assessed through RTL simulation using Icarus. In previous works [11], [12], syntax/functional correctness is evaluated by pass@k, which means the probability of any syntax/functional correct generation in k attempts for a design. Such a metric generally means multiple attempts are needed to produce syntax/functionally correct code. Since the goal of Lorecast is to obtain a quick estimation and we try to avoid multiple attempts. Hence, we adopt a simpler and significantly stricter metric. For each design, we report correct if all attempts, whether single or multiple, lead to syntax/functional correctness, i.e., the "correct" here is a binary indicator instead of probability. We also report the correct rate, which is the ratio of the number of designs where the generated codes are correct versus the total number of designs in the testcases.

Forecast accuracy metrics. The accuracy of TNS and power forecast is evaluated by **Absolute Percentage Mean Error (APME)**. All TNS values are reported as their absolute values. Let \bar{y} be the average forecast result among all designs, and \bar{y}^* be the ground truth average among all designs. Then, APME is defined by

$$\mathcal{E} = \frac{|\bar{y} - \bar{y}^*|}{\bar{y}^*} \times 100\% \tag{1}$$

The reason that we could not use Mean Absolute Percentage Error (MAPE) is that some ground truth TNS values are 0. We also report the **Normalized Root Mean Square Error (NRMSE)**. Due to large variations in data magnitude, RMSE can be hard to interpret directly.

To address this, we normalize it using the mean of the ground truth values. NRMSE is defined as

$$\mathcal{E}_{\text{NRMSE}} = \frac{\sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}}{\bar{y}} \times 100\%$$
 (2)

In addition, the accuracy is assessed by the R^2 correlation factor. In comparison with other workflows, we observe that the others often fail to produce valid forecasts due to syntax errors in the generated designs. To quantify this, we define the Syntax Correctness Rate, $\rho_{\rm syntax}$, as the ratio of syntactically correct designs to the total number of generated designs. Furthermore, we compute Conditional Accuracy and Conditional Error for the subset of designs with correct syntax, as defined below

$$\mathcal{A}_{\text{cond}} = \rho_{\text{syntax}} \cdot (1 - \mathcal{E}) \times 100\% \tag{3}$$

$$\mathcal{E}_{cond} = 1 - \mathcal{A}_{cond} = 1 - (\rho_{syntax} \cdot (1 - \mathcal{E})) \times 100\%$$
 (4)

Ground truth and computing platform. All ground truth data are based on manually written and functionally correct Verilog code. Logic synthesis is performed on the codes by Synopsys Design Compiler with a 45nm cell library [54]. The layout, including placement and routing, is obtained using Cadence Innovus. The ground truth timing and power results are obtained through post-layout analysis. Logic and layout synthesis run on a Linux x86_64 machine with AMD EPYC 7443 24-Core processors (48 cores in total), while ML model predictions are performed on a Windows 10 computer with an 11th Gen Intel(R) Core(TM) i7-11800H processor at 2.30GHz and 32GB RAM.

B. Main Results

The main results of performance and power forecasting are shown in Table III. In addition, the table includes syntax and functional correctness results from different LLMs, where one attempt is made for each design. Columns 4 and 5 are the forecasting results from manually written functionally correct Verilog code. Using the RePIC and I-PREF, all four LLMs can achieve 100% syntax correctness, which is required for Lorecast. None of the LLMs delivered 100% functional correctness as expected. Among them, GPT4 with RePIC and I-PREF achieves 87% functional correctness rate, aligns with or improves upon the state-of-the-art methods [12] [22]. GPT4-based Lorecast also achieves the lowest APME.

Figure 8 provides the scatter plots of forecast versus ground truth. GPT4-based Lorecast achieves \mathbb{R}^2 correlations of 0.99 for power and TNS forecast, respectively.

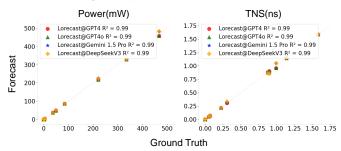


Fig. 8. Power and TNS forecasts from different LLMs vs. ground truth.

Beyond accuracy metrics, we further examine the alignment between generated and ground truth behaviors. Figure 9(a) demonstrates the impact of varying logic/layout synthesis tool parameters for design "pe_64bit". Since the post-layout analysis of the LLM-generated code highly overlaps with the ground truth under different

TABLE III PERFORMANCE AND POWER FORECASTING RESULTS.

Design	Ground	Truth+	Forecast	from manual]	Lorecast	with GPT	1	Lore	cast with	Gemini 1.	5 Pro	I	orecast	with GPT4	ю	Lorecast with DeepSeek V3			
Design	Power*	TNS*	Power	TNS	Syntax	Func	Power	TNS	Syntax	Func	Power	TNS	Syntax	Func	Power	TNS	Syntax	Func	Power	TNS
right_shifter	992	0.069	963	0.0691	/	_	748	0.0689	/	/	927	0.0689	/	/	748	0.0689	/	_	683	0.0711
adder_bcd	8	0	22	0.0200	/	/	24	0	/	/	23	0	/	/	25	0	/	/	26	0.0014
signal_generator	1,508	0.225	1,347	0.2249	/	/	1,547	0.2135	/	Х	1,547	0.2135	/	Х	1,547	0.2184	/	X	1,547	0.2184
multiply	35	0	26	0.0002	/	/	37	0.0001	/	/	69	0.0015	/	/	34	0.0008	/	/	27	0.0027
accu	4,042	0.306	3,774	0.3043	1	/	3,996	0.3062	/	/	4,316	0.3049	/	/	3,996	0.3318	/	/	3,996	0.3318
LIFObuffer	29	0	39	0.0010	/	/	33	0.0001	/	/	44	0.0006	/	/	48	0.0027	/	/	20	0.0009
asyn_fifo	100	0	106	0.0010	/	X	160	0.0003	/	Х	210	0	/	Х	180	0.0003	/	X	148	0.0003
sobel_filter	49,475	0.995	49,226	0.9842	/	X	47,813	0.9541	/	Х	51,574	1.0499	/	X	47,088	0.9594	/	X	50,885	1.0505
matmul22	307	0	312	0.0002	/	/	326	0	/	/	376	0	/	/	326	0	/	/	371	0.0004
mux256to1v	79	0	36	0.0001	/	/	45	0.0002	/	/	32	0.0002	/	/	32	0.0002	/	/	43	0.0002
pe_32bit	84,818	1.139	86,900	1.1384	/	/	86,442	1.1432	/	/	86,442	1.1432	/	/	86,442	1.1432	/	/	86,442	1.1432
izigzag	221,131	0.05	221,992	0.0496	/	/	217,893	0.0501	/	/	226,328	0.0495	/	/	217,893	0.0501	/	/	223,398	0.0497
huffmandecode	37,461	0.877	36,499	0.8363	/	/	37,102	0.8751	/	/	36,658	0.8855	/	/	37,102	0.8751	/	/	35,958	0.8644
pe_64bit	334,799	1.584	333,869	1.5933	/	/	329,173	1.5870	/	/	329,173	1.5870	/	/	329,173	1.5870	/	/	329,173	1.5870
fft_16bit	466,462	0.9	466,382	0.8741	/	/	458,226	0.9019	/	/	483,266	0.8551	1	/	458,226	0.9019	/	/	483,925	0.855
Average	80,083	0.41	80,080	0.4063			78,904	0.4067			81,399	0.4107			78,857	0.4093			81,109	0.4118
APME			(1%)	(1%)			(1%)	(1%)			(2%)	(1%)			(2%)	(1%)			(1%)	(1%)
NRSME			(1%)	(4%)			(3%)	(3%)			(6%)	(5%)			(4%)	(3%)			(6%)	(5%)

⁺Ground truth is obtained by manually *Power unit is μW and TNS unit is ns

TABLE IV TIME COST* OF VERILOG CODE WRITING/DEBUG + SYNTHESIS IN COMPARISON WITH LORECAST.

Design	Manual	Verilog +	synthesis			Loreca	st				Speed	dup
Design	Verilog coding	Logic	Layout	Total	Prompt writing	LLM code generation	ML prediction	Total	Manual time	CPU time	End-to-end time	Incremental change time (synthesis param update only)
right_shifter	1489	215	37	1741	270	5	0.0017	275	5.5X	50.4X	6.3X	$148 \times 10^{3} \text{ X}$
adder_bcd	1120	223	43	1386	420	5	0.0013	425	2.7X	53.2X	3.3X	$205 \times 10^{3} \text{ X}$
signal_generator	1990	220	38	2248	439	6	0.0164	445	4.5X	43.0X	5.1X	$16 \times 10^{3} \text{ X}$
accu	1129	223	55	1407	250	8	0.023	258	4.5X	34.8X	5.5X	$12 \times 10^{3} \text{X}$
LIFObuffer	1640	223	44	1907	560	9	0.0021	569	2.9X	29.7X	3.4X	$127 \times 10^{3} \text{ X}$
multiply	1780	217	40	2037	287	5	0.0243	292	6.2X	51.4X	7.0X	$11 \times 10^{3} \text{ X}$
asyn_fifo	5320	225	41	5586	1030	21	0.0259	1051	5.2X	12.7X	5.3X	$10 \times 10^{3} \text{X}$
sobel_filter	2820	235	51	3106	660	26	0.022	686	4.3X	11.0X	4.5X	$13 \times 10^{3} \text{ X}$
matmul22	2459	224	41	2724	490	9	0.0285	499	5.0X	29.4X	5.5X	$9 \times 10^{3} \text{ X}$
mux256to1v	739	221	49	1009	320	3	0.0014	323	2.3X	90X	3.1X	$193 \times 10^{3} \text{ X}$
pe_32bit	1650	231	52	1933	469	5	0.0292	474	3.5X	56.7X	4.1X	$10 \times 10^{3} \text{ X}$
huffmandecode	2200	270	56	2526	550	32	0.0089	582	4X	10.2X	4.3X	$37 \times 10^{3} \text{ X}$
izigzag	1819	265	49	2133	480	38	0.0132	518	3.8X	8.3X	4.1X	$24 \times 10^{3} \text{ X}$
pe_64bit	1588	252	69	1909	512	5	0.013	517	3.1X	64.2X	3.7X	$25 \times 10^{3} \text{ X}$
fft 16bit	3600	343	94	4037	990	52	0.0247	1060	3.6X	8.4X	3.8X	$17 \times 10^{3} \text{ X}$
Average	2090	239	51	2379	515	15	0.0157	530	4X	19.3X	4.6X	$57 \times 10^{3} \text{X}$

^{*}The time unit is s

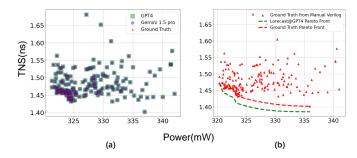
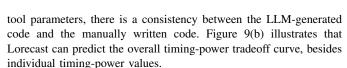


Fig. 9. (a): post-layout TNS and power from Verilog code generated by LLMs and ground truth. (b): performance-power tradeoff curves predicted by Lorecast@GPT4 and the ground truth from manually written Verilog code.



Runtime comparison between Lorecast and RTL-synthesis based estimation. We compared the time cost of two different flows: one is the performance/power estimation based on manually written Verilog code and logic/layout synthesis, and the other is the manually written prompts followed by Lorecast. The Verilog coding/debug and prompt writing/debug time are estimated by asking six participants to do both for each design. Please note that these participants generally know Verilog coding and prompting, but with different skill levels. Figure 10 presents the mean, min, max, and standard deviation range of the time for all the testcase designs. Due to the limited sample size and data skew, the mean minus one standard deviation occasionally

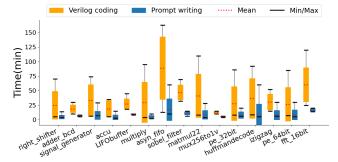


Fig. 10. Verilog code writing and debug time in yellow bars; prompt writing and check time in blue bars. Each bar indicates $\mu \pm \sigma$ range, where μ is the mean and σ is the standard deviation.

falls below the observed minimum. Overall, prompt writing/debug time is significantly shorter than Verilog code writing/debug time. It also has a smaller variance, so that time budgeting becomes easy.

Other components of the time cost are summarized in Table IV, where the coding/debugging time results are the mean values. The ML prediction time is obtained using the model [29]. The time is dominated by Verilog code writing/debugging and prompt writing/debugging. Overall, Lorecast can achieve $4.6 \times$ speedup. Manual (Verilog coding/debugging or prompt writing/debugging) time has been reduced by 4x. For CPU time, LLM-based code generation and ML prediction is 19.3× faster than traditional logic and layout synthesis. For incremental design changes, where only synthesis parameters are modified, Lorecast re-inference achieves a speedup exceeding 50,000x compared to re-running traditional logic and layout synthesis. Furthermore, Lorecast reduces the requirement for HDL coding skills for the performance/power estimation.

TABLE V
RESULTS OF LLM-BASED REASONING WITHOUT GENERATING VERILOG CODE.

Design	GPT4		GP'	GPT40		Llama3.1 405B		Gemini 1.5 Pro		ek R1 [55]		tuned a3 8B
Ü	Power*	TNS*	Power	TNS	Power	TNS	Power	TNS	Power	TNS	Power	TNS
right_shifter	20	0.005	5.4	0.02	20	0.1	Х	Х	65	0.25	113	0
adder_bcd	1.44	0	8.4	0	35	0.37	0.6	0.2	42	0.45	90	0
signal_generator	10	0	6.8	0.015	33	0.02	5	0.5	85	0.35	103	0
multiply	20	0.025	8.2	0.018	76	0.05	20	2	120	0.35	105	4.5
accu	20	0.04	7.5	0.012	49	0.03	10	1	95	0.62	113	0.6
LIFObuffer	5	0	46.2	0	29	1.03	2	0.3	78	0.32	125	0.5
asyn_fifo	30	0.005	9.3	0.02	128	0.07	30	3	220	1.2	120	6.8
sobel_filter	30	2	93.6	0	51	2.51	10	0.8	115	0.58	310	8.2
matmul22	40	0.035	10.1	0.025	262	0.1	50	5	380	1.8	240	3.4
mux256to1v	10	0	12.8	0	0.38	0.23	25	1.2	155	0.67	110	0.5
pe_32bit	30	0.035	12.7	0.03	189	0.08	60	4	180	0.85	180	2.9
huffmandecode	30	0.025	14.3	0.035	336	0.12	40	6	115	0.75	240	4.1
izigzag	30	0.025	13.5	0.028	231	0.06	25	3	210	1.1	220	3.5
pe_64bit	40	0.065	18.5	0.04	472	0.15	100	8	420	1.6	320	4.8
fft_16bit	10	0	327.5	0	138	3.19	70	2.2	260	1.8	290	8.8
Average	22	0.15	40	0.016	137	0.54	32	2.7	169	0.85	179	3.2
APME	(99%)	(63%)	(99%)	(96%)	(99%)	(32%)	(99%)	(559%)	(99%)	(107%)	(99%)	(680%

^{*}Power unit is μW and TNS unit is ns.

TABLE VI
IMPACT OF LORECAST PROMPTING IN COMPARISON WITH OTHER PROMPTING TECHNIQUES WITH THE SAME VERILOG CODE TO POWER/TNS
PREDICTION TECHNIQUE.

Design	Ground	Truth	Lorecast				рт		ast with 2] Prompt	ina	Voril	Forecast with VerilogEval [11] Prompting			
Design	Power*	TNS*	Syntax	Func	Power	TNS	Syntax	Func	Power	TNS	Syntax	Func	Power	TNS	
right_shifter	992	0.069	/	/	748	0.0689	/	/	773	0.0690	Х	Х	-	-	
adder_bcd	8	0	1	1	24	0	1	1	37	0.0005	X	X	-	-	
signal_generator	1508	0.225	1	1	1547	0.2135	1	Х	1346	0.2130	/	X	4774	0.0433	
multiply	35	0	1	1	37	0.0001	1	1	48	0.0002	Х	X	-	-	
accu	4042	0.306	/	1	3996	0.3062	1	1	3717	0.2822	×	X	-	-	
LIFObuffer	29	0	1	1	33	0.0001	×	X	-	-	×	X	-	-	
asyn_fifo	100	0	/	Х	160	0.0003	×	Х	-	-	×	X	-	-	
sobel_filter	49475	0.995	/	Х	47813	0.9541	X	X	-	-	X	X	-	-	
matmul22	307	0	1	1	326	0	×	Х	-	-	×	X	-	-	
mux256to1v	79	0	1	1	45	0.0002	/	1	45	0.0001	/	1	36	0.0001	
pe_32bit	84818	1.139	1	1	86442	1.1432	1	1	87456	1.1463	/	X	2003	0.0421	
izigzag	221131	0.05	1	1	217893	0.0501	×	Х	-	-	×	X	-	-	
huffmandecode	37461	0.877	1	1	37102	0.8751	×	Х	-	-	×	X	-	-	
pe_64bit	334799	1.584	1	1	329173	1.5870	1	1	332914	1.6127	X	X	-	-	
fft_16bit	466462	0.9	1	1	458226	0.9019	X	Х	-	-	Х	X	-	-	
Conditional Error					(1%)	(1%)			(48%)	(48%)			(98%)	(99%)	

⁻ Forecast can not be performed due to syntax errors in Verilog code generated by LLMs.

Direct LLM-based forecasting without Verilog code generation.One may ask: why not let an LLM directly forecast performance and

One may ask: why not let an LLM directly forecast performance and power without generating Verilog code. An experiment is performed to verify this concept and the results from 6 different LLMs are shown in Table V. The Llama 3-8B model has been fine-tuned with the training dataset completely separated from the testcases. One can see that their errors are much greater than Lorecast. For the design "right_shift", Gemini 1.5 Pro could not even produce legal results. The results confirm that direct forecasting using LLMs without Verilog code generation is a significantly more difficult path than Lorecast.

C. Importance of Syntax Correctness and Structural Similarity

In Table VI, RTLLM [12] and VerilogEval [11] are two previous works on LLM-based Verilog code generation. All forecast techniques in Table VI use the same ML power and TNS prediction technique [29] based on the generated Verilog code. There are two observations here. First, achieving a high syntax correctness rate is not straightforward, as both RTLLM [12] and VerilogEval [11] fail to achieve syntax correctness for a significant number of cases. In contrast, Lorecast achieves 100% syntax correctness in all cases. Second, syntax correctness is critical for the forecast. Without syntax correctness, the ML technique [29] is not able to generate power/TNS prediction results.

The results of Table VI also highlight the importance of structural similarity, which is described in Section IV-D. For the two cases "signal_generator" and "pe_32bit", VerilogEval [11] produces syntax correct but functionally incorrect Verilog code. This code leads to

much greater prediction errors than Lorecast and RTLLM [12] due to their AST structures not being similar to the functionally correct ground truth designs. Specifically, for the "signal generator" case, VerilogEval's AST achieves only 20.8% Subtree Match Rate [56] with the ground truth, while RTLLM and Lorecast exceed 95%, with Lorecast also being functionally correct. For "pe_32bit", both Lorecast and RTLLM produce functionally correct code with match rates exceeding 85%, whereas VerilogEval remains both functionally incorrect and structurally misaligned, with only 36% match. These results suggest an association between AST-level structural similarity and the accuracy of performance and power predictions.

D. Impact of Functional Incorrectness

Comparing the GPT4-based Lorecast and Gemini1.5Pro-based Lorecast in Table III, their errors are similar, although Gemini1.5Pro results in 3 functionally incorrect designs while GPT4 has only 2. In general, the impact of functional incorrectness on Lorecast accuracy is small due to the structural similarity described in Section IV-D.

We also examine the impact of functionally incorrect Verilog code on logic and layout synthesis outcomes, as shown in Table VII, where power and TNS values are obtained through actual synthesis rather than ML prediction. This experiment is conducted using various LLMs with Lorecast-style prompting. Except for "asyn_fifo", where some generated Verilog code is not synthesizable, the functionally incorrect Verilog code generally yields synthesis results, both in terms of power and TNS, that closely match those of functionally correct Verilog implementations. This outcome can be attributed to the structural similarity between the Verilog code generated by

^{*}Power unit is μW and TNS unit is ns.

Lorecast and its functionally correct counterparts, as discussed in Section IV-D.

TABLE VII

EFFECT OF FUNCTIONALLY INCORRECT VERILOG CODE GENERATED BY LORECAST ON LOGIC/LAYOUT SYNTHESIS (NOT FORECAST) RESULTS.

	Functiona	lly Correct	Functionally Incorrect												
Design	Ma	Manual		GPT4		Gemini 1.5 Pro		Г4о	DeepSeek V3						
	Power*	TNS*	Power	TNS	Power	TNS	Power	TNS	Power	TNS					
signal generator	1508	0.225	Ø	Ø	1346	0.279	1467	0.253	1367	0.263					
asyn_fifo	100	0	X	X	X	Х	125	0	X	X					
sobel_filter	49475	0.995	48737	0.981	50537	1.029	47415	1.011	46734	1.026					

Ø: functionally correct and thus out of the scope of this experiment

E. Ablation Studies

To assess the general applicability of Lorecast, we additionally conduct ablation studies on the RTLLM benchmark [12], a set of design tasks introduced in prior work, as a complementary evaluation beyond our primary testcases.

Impact of different templates in prompting correctness without I-PREF. Although template-based structured prompting has been proposed in previous works [11] [12], different template styles also matter. In Table VIII, we show that our template style can improve the syntax correctness rate by 33%-43%. Please note that I-PREF is not performed in these cases.

TABLE VIII IMPACT OF DIFFERENT TEMPLATES ON SYNTAX AND FUNCTIONAL CORRECTNESS ACROSS LLMs (WITHOUT I-PREF).

Prompt	Correctness Rate(%)													
Technology	GP	Γ4	GPT	'4o	Gemini	1.5 Pro	DeepSeek V3							
recnnology	Syntax	Func	Syntax	Func	Syntax	Func	Syntax	Func						
VerilogEval [11]	66.7	21.4	50	21.4	42.9	17.9	55.2	25						
RTLLM [12]	86.2	39.3	82.8	42.9	71.4	42.9	79.3	53.6						
Lorecast	100	57	93.1	50	86.2	57	93.1	57						

Effect of functional correctness across LLMs. Sometimes, an LLM cannot produce syntax correct Verilog code for a design. As shown in Figure 11, the results indicate a correlation between functional correctness(represented by the hatched bars) and conditional accuracy, although functional errors can be tolerated. For Lorecast@GPT4, power and performance \mathcal{E} is 1% for functionally correct code, rising to 5% and 7% for functionally incorrect one.

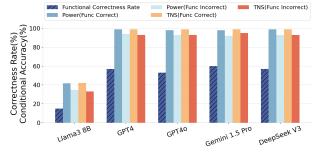


Fig. 11. Comparison of LLMs on forecasting accuracy and functional correctness

Impact of I-PREF on correctness across LLMs. Syntax/functional correctness rates for 8 LLMs with and without I-PREF are depicted in Figure 12. Here, syntax/functional correctness is asserted for a design if the results from all three attempts are correct.

We can obtain the following observations.

- Syntax correctness rate is always significantly higher than functional correctness rate. This confirms that syntax correctness is a much more achievable goal than functional correctness for LLM-generated Verilog code.
- I-PREF can always improve syntax correctness rate, which is fundamental for Lorecast.

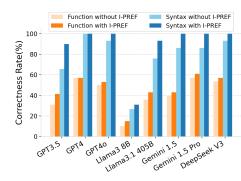


Fig. 12. Syntax/functional correctness of LLMs with and without I-PREF.

Effect of regulation in iterative feedback prompting on correctness. Although iterative feedback prompting was proposed in [19], its feedback prompting is not regulated. Figure 13 shows that our regulated feedback prompting in Lorecast can improve the syntax correctness rate by about 6%.

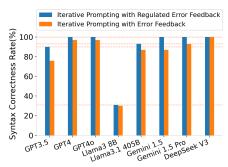


Fig. 13. Effect of regulation in iterative feedback prompting.

Syntax correctness rate versus max I-PREF iterations. In Figure 14, we vary the maximum number of I-PREF iterations and observe the impact on syntax correctness rate. Initially, increasing the maximum number of iterations does help in improving the correctness rate. However, the benefits diminish at around 8 iterations. This is why we set the limit of I-PREF iterations to 10.

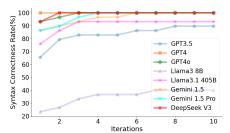


Fig. 14. Syntax correctness rate vs. the max I-PREF iterations.

VI. CONCLUSIONS

We propose a methodology for forecasting circuit performance and power from natural language. It leverages LLM-based Verilog code generation techniques and the Verilog-based ML prediction technique to produce forecasts with approximately 2% error compared to postlayout analysis. Lorecast accelerates the performance and power estimation process by 4.6x compared to conventional methods that involve manually writing Verilog code, logic and layout synthesis. The LLM-based Verilog code generation technique here is customized to improve syntax correctness rate and reduce the requirement for functional correctness. The validation is performed on circuits significantly larger than previous works on LLM-based Verilog code generation.

X: syntax correct but not synthesizable
*Power unit is "IN" -- 1 m--

wer unit is μW and TNS unit is ns

REFERENCES

- D. C. Black and J. Donovan, SystemC: From the ground up. Springer, 2004
- [2] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.
- [3] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *HPCA*, 2015.
- [4] J. Zhai, C. Bai, B. Zhu, Y. Cai, Q. Zhou, and B. Yu, "McPAT-Calib: A microarchitecture power modeling framework for modern CPUs," in ICCAD, 2021.
- [5] T. Liu, Q. Tian, J. Ye, L. Fu, S. Su, J. Li, G.-W. Wan, L. Zhang, S.-Z. Wong, X. Wang, and J. Yang, "ChatChisel: Enabling agile hardware design with large language models," in *ISEDA*, 2024.
- [6] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in NIPS, 2020.
- [7] X. Wang, G.-W. Wan, S.-Z. Wong, L. Zhang, T. Liu, Q. Tian, and J. Ye, "ChatCPU: An agile CPU design & verification platform with LLM," in DAC, 2024.
- [8] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "BetterV: Controlled verilog generation with discriminative guidance," in *ICML*, 2024.
- [9] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, and X. Li, "ChipGPT: How far are we from natural language hardware design," in arXiv, 2023.
- [10] S.-Z. Wong, G.-W. Wan, D. Liu, and X. Wang, "VGV: Verilog generation using visual capabilities of multi-modal large language models," in *LAD Workshop*, 2024.
- [11] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "VerilogEval: Evaluating large language models for verilog code generation," in ICCAD, 2023.
- [12] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "RTLLM: An open-source benchmark for design RTL generation with large language model," in ASP-DAC, 2024.
- [13] A. Allam and M. Shalan, "RTL-Repo: A benchmark for evaluating LLMs on large-scale RTL design projects," in *LAD Workshop*, 2024.
- [14] M. DeLorenzo, V. Gohil, and J. Rajendran, "CreativEval: Evaluating creativity of LLM-based hardware code generation," in MLCAD, 2024.
- [15] 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 *DATE*, 2023.
- [16] 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 *LAD Workshop*, 2024.
- [17] 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," in *TODAES*, 2024.
- [18] B. Nadimi and H. Zheng, "A multi-expert large language model architecture for Verilog code generation," in LAD Workshop, 2024.
- [19] Blocklove, Jason, Thakur, Shailja, Tan, Benjamin, Pearce, Hammond, Garg, Siddharth, Karri, and Ramesh, "Automatically improving llmbased verilog generation using eda tool feedback," in TODAES, 2025.
- [20] F. Cui, C. Yin, K. Zhou, Y. Xiao, G. Sun, Q. Xu, Q. Guo, Y. Liang, X. Zhang, D. Song et al., "OriGen: Enhancing RTL code generation with code-to-code augmentation and self-reflection," in ICCAD, 2024.
- [21] Y.-D. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically fixing RTL syntax errors with large language models," in DAC, 2024.
- [22] 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," in *ICCD*, 2024.
- [23] B.-Y. Wu, U. Sharma, S. R. D. Kankipati, A. Yadav, B. K. George, S. R. Guntupalli, A. Rovinski, and V. A. Chhabria, "EDA Corpus: A large language model dataset for enhanced interaction with OpenROAD," in *LAD Workshop*, 2024.
- [24] Y. Zhang, Z. Yu, Y. Fu, C. Wan, and Y. C. Lin, "MG-Verilog: Multi-grained dataset towards enhanced LLM-assisted Verilog generation," in LAD Workshop, 2024.
- [25] K. Chang, K. Wang, N. Yang, Y. Wang, D. Jin, W. Zhu, Z. Chen, C. Li, H. Yan, Y. Zhou, Z. Zhao, Y. Cheng, Y. Pan, Y. Liu, M. Wang, S. Liang, Y. Han, H. Li, and X. Li, "Data is all you need: Finetuning LLMs for chip design via an automated design-data augmentation framework," in DAC, 2024.

- [26] R. Ma, Y. Yang, Z. Liu, J. Zhang, M. Li, J. Huang, and G. Luo, "VerilogReader: LLM-aided hardware test generation," in *LAD Workshop*, 2024.
- [27] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "Autobench: Automatic testbench generation and evaluation using llms for hdl design," in MLCAD, 2024.
- [28] —, "Correctbench: Automatic testbench generation with functional self-correction using Ilms for hdl design," in arXiv, 2024.
- [29] P. Sengupta, A. Tyagi, Y. Chen, and J. Hu, "How good is your Verilog RTL code?: A quick answer from machine learning," in *ICCAD*, 2022.
- [30] C. Xu, C. Kjellqvist, and L. W. Wills, "SNS's not a synthesizer: a deep-learning-based synthesis predictor," in ISCA, 2022.
- [31] W. Fang, Y. Lu, S. Liu, Q. Zhang, C. Xu, L. W. Wills, H. Zhang, and Z. Xie, "MasterRTL: A pre-synthesis PPA estimation framework for any RTL design," in *ICCAD*, 2023.
- [32] R. Moravej, S. Bodhe, Z. Zhang, D. Chetelat, D. Tsaras, Y. Zhang, H.-L. Zhen, J. Hao, and M. Yuan, "The graph's apprentice: Teaching an LLM low level knowledge for circuit quality estimation," in arXiv, 2024.
- [33] D. S. Lopera, I. Subedi, and W. Ecker, "Using graph neural networks for timing estimations of RTL intermediate representations," in MLCAD Workshop, 2024.
- [34] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," in *OpenAI blog*, 2019.
- [35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, 2017.
- [36] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in NIPS, 2020.
- [37] OpenAI, "GPT-4 technical report," in arXiv, 2023.
- [38] Gemini Team, Google, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," in arXiv, 2024.
- [39] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan et al., "Deepseek-v3 technical report," in arXiv, 2024.
- [40] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," in *arXiv*, 2021.
- [41] S. Palnitkar, in Verilog HDL: a guide to digital design and synthesis, 2003.
- [42] D. Harris and N. Weste, in CMOS VLSI Design: A Circuits and Systems Perspective, 2010.
- [43] Llama Team, "The Llama 3 herd of models," in arXiv, 2024.
- [44] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al., "Evaluating large language models trained on code," in arXiv, 2021.
- [45] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of chatgpt in code generation," in *TOSEM*, 2025.
- [46] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," in *TOSEM*, 2025.
- [47] S. Williams, in The Icarus Verilog Compilation System, 2024.
- [48] Synopsys, Inc, in Synopsys VCS, 2023.
- [49] Cadence Design Systems, Inc, in Xcelium, 2023.
- [50] S. Takamaeda-Yamazak, "Pyverilog: A python-based hardware design processing toolkit for Verilog HDL," in ARC, 2015.
- [51] W. Snyder, "Verilator and SystemPerl," in DAC, 2004
- [52] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting verilogeval: A year of improvements in large-language models for hardware code generation," in *TODAES*, 2025.
- [53] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-Chat: Challenges and opportunities in conversational hardware design," in MLCAD Workshop, 2023.
- [54] Silicon Integration Initiative (Si2), "NanGate open cell library and free PDK libraries," https://si2.org/open-cell-and-free-pdk-libraries/, 2024, accessed: 2024-10-29.
- [55] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," in *arXiv*, 2025.
- [56] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSME*, 1998.