MetRex: A Benchmark for Verilog Code Metric Reasoning Using LLMs

Manar Abdelatty
manar_abdelatty@brown.edu
Brown University
School of Engineering
Providence, RI, USA

Jingxiao Ma
jingxiao_ma@brown.edu
Brown University
School of Engineering
Providence, RI, USA

Sherief Reda
sherief_reda@brown.edu
Brown University
School of Engineering
Providence, RI, USA

Abstract

Large Language Models (LLMs) have been applied to various hardware design tasks, including Verilog code generation, EDA tool scripting, and RTL bug fixing. Despite this extensive exploration, LLMs are yet to be used for the task of post-synthesis metric reasoning and estimation of HDL designs. In this paper, we assess the ability of LLMs to reason about post-synthesis metrics of Verilog designs. We introduce MetRex, a large-scale dataset comprising 25,868 Verilog HDL designs and their corresponding post-synthesis metrics, namely area, delay, and static power. MetRex incorporates a Chain of Thought (CoT) template to enhance LLMs' reasoning about these metrics. Extensive experiments show that Supervised Fine-Tuning (SFT) boosts the LLM's reasoning capabilities on average by 37.0%, 25.3%, and 25.7% on the area, delay, and static power, respectively. While SFT improves performance on our benchmark, it remains far from achieving optimal results, especially on complex problems. Comparing to state-of-the-art regression models, our approach delivers accurate post-synthesis predictions for 17.4% more designs (within a 5% error margin), in addition to offering a 1.7x speedup by eliminating the need for pre-processing. This work lays the groundwork for advancing LLM-based Verilog code metric reasoning.

Keywords: LLM, Verilog, Metrics, Post-synthesis, Reasoning, Chain-of-Thought

ACM Reference Format:
Manar Abdelatty, Jingxiao Ma, and Sherief Reda. 2025. MetRex: A Benchmark for Verilog Code Metric Reasoning Using LLMs. In *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3658617.3697625

ASPDAC'25, January 20–23, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-0635-6/25/01
https://doi.org/10.1145/3658617.3697625

1 Introduction

Recent advancements in Large Language Models (LLMs) have demonstrated remarkable potential to transform the field of hardware design, across a wide range of tasks such as Verilog code generation [1–5], EDA tools scripting [6], designing AI accelerators [7], and fixing RTL syntax errors [8]. However, an area yet to be explored is the application of LLMs for reasoning and estimation of post-synthesis metrics of HDL designs. Given HDL code as input, LLMs could potentially infer gate-level details and estimate key metrics, such as area, delay, and static power.

While current LLMs can generate raw Verilog code, they lack awareness of post-synthesis metrics and struggle to reason about them effectively. Prior works utilized LLMs to tweak Verilog code to meet area, delay, and power requirements using prompting methods [9] or search methods like Monte-Carlo tree search [10]. However, these approaches mainly focus on refining Verilog code, and they do not fundamentally enhance the LLM's understanding of how different design choices impact post-synthesis metrics. Thus, there is a need for approaches that empower LLMs with deeper insights into the underlying relationships between HDL code and post-synthesis metrics.

In light of this, we introduce *MetRex*, an LLM-based framework for high level metric estimation of HDL designs. *MetRex* encompasses a large-scale dataset of 25, 868 HDL designs, each annotated with post-synthesis metrics on area, delay, and static power. To enhance the LLM's capability to understand and reason about these metrics, we propose a Chain of Thought (CoT) template that details the logical steps necessary for computing these metrics. To the best of our knowledge, *MetRex* is the first framework that addresses the task of LLM-based code analysis for metric estimation of HDL designs.

Our contributions are summarized as follows:

- We introduce a new dataset, *MetRex*[1], for benchmarking Large Language Models (LLMs) for the task of reasoning about post-synthesis metrics of HDL designs. The dataset comprises 25,868 Verilog designs, each annotated with area, delay, and static power metrics.

[1]https://github.com/scale-lab/MetRex

- We developed an automated flow using a Verilog compiler, a synthesis tool, and an LLM agent to detect and resolve syntax and synthesis errors, ensuring a dataset of clean, synthesizable designs.
- We introduce a Chain of Thought (CoT) prompting technique that improves the LLM's reasoning and understanding of post-synthesis metrics by 5.1%, 5.4%, and 8.9% on the area, delay, and static power metrics, respectively, compared to direct prompting methods.
- We employ the *MetRex* dataset in extensive Supervised Fine-Tuning (SFT) experiments, demonstrating that SFT can significantly improve the LLM performance in reasoning and estimating post-synthesis metrics on average by 37.0%, 25.3%, and 25.7% on the area, delay, and static power, compared to few-shot prompting techniques.
- We compare the LLM estimation accuracy to regression-based models [11], highlighting their potential for this task in offering insightful and direct analysis of HDL code without the need for intermediary formats. LLMs improve the rate of obtaining accurate estimates within a 5% error margin by 17.4% while offering 1.7x faster analysis by eliminating the need for feature extraction and pre-processing.

This paper is organized as follows. Section 2 discusses related work. Section 3 presents a general problem formulation of the metric reasoning task with LLMs. Section 4 discusses the *MetRex* dataset. Section 5 presents experimental results. Section 6 discusses current limitations and future directions. Finally, Section 7 concludes the paper.

2 Related Work

Large Language Models (LLMs) have demonstrated strong potential in code generation tasks, where they can generate logically consistent code across various programming languages [12]. Their utility extends beyond mere code generation to include code reasoning and understanding, where they can repair bugs in codebases [13], reason about code execution [14], and perform compiler optimizations [15].

In the hardware domain, LLMs have been extensively applied to Verilog code generation [1–5]. VeriGen [2, 5], for instance, finetuned code LLMs for generating Verilog. Several benchmarks, such as RTLLM [4] and VerilogEval [3], have been presented to standardize the evaluation of LLMs in Verilog code generation tasks. LLMs have also shown promise in code reasoning tasks, such as identifying and rectifying bugs in RTL designs [8] and optimizing Verilog code [10]. Despite these advancements, the application of LLMs in reasoning about post-synthesis metrics of HDL designs remains largely unexplored. This represents a significant research opportunity, especially in enhancing the LLM's understanding of how different design choices impact post-synthesis metrics.

The metric estimation of RTL designs has also been a subject of research for conventional machine learning techniques. Studies presented in [11, 16–19] aim to provide an early estimate of the RTL post-synthesis or post-layout metrics to help accelerate the hardware design exploration process. These techniques typically transform the RTL design into different representational formats. MasterRTL [11] proposes using simple operator graphs (SOGs) since it is closer to the synthesized netlist than abstract syntax trees (ASTs) [16, 17]. Manually engineered features are then extracted from these representations and used as inputs to regression-based models, such as XGBoost and graph neural networks, to predict post-synthesis metrics.

In contrast, leveraging LLMs for this task offers a unique advantage. Unlike traditional methods, LLMs can process Verilog code directly, a lossless representation, thereby bypassing the need for manual feature extraction or transformation into intermediary formats. This direct processing enables LLMs to autonomously identify and extract features and patterns closely associated with synthesis outcomes, leading to potentially more insightful and faster analyses. Our study specifically aims to assess LLM reasoning capabilities about post-synthesis metrics of Verilog code, to broaden our understanding of the utility of these models in HDL design methodologies.

3 Problem Formulation

In this section, we provide a general formulation of the HDL metric estimation task based on natural language instructions. Given a Verilog HDL design $\mathcal{V}$, the objective is to design an LLM-based model f_{reason} to estimate the post-synthesis metrics $\mathcal{M}_{synth}$, where $\mathcal{M}_{synth} = f_{reason}(\mathcal{V})$. However, directly predicting the final metrics from HDL code is a complex reasoning task. This is mainly because LLMs are optimized for understanding and generating text in context rather than performing numerical calculations or predictions. LLMs are good at reasoning about a problem when it is described in words, but may struggle with abstract numerical predictions without explicit reasoning steps [20].

Therefore, we task the LLM with reasoning about the post-synthesis metrics through intermediate steps, $\mathcal{I}$, which include gate-level details from the synthesized netlist and natural language descriptions of how to calculate these metrics. These intermediate steps guide the LLM in a Chain of Thought (CoT) manner, helping it predict the final metrics. The CoT template includes information such as gate counts, area and power characteristics per gate, and critical path stages, all described in natural language. Thus, the task becomes predicting both the reasoning steps and the final post-synthesis metrics, expressed as $\langle(\mathcal{I}, \mathcal{M}_{\text{synth}})\rangle = f_{reason}(\mathcal{V})$.

4 MetRex Benchmark

4.1 Data Collection and Cleaning

The initial phase in the dataset creation involved collecting a diverse range of HDL designs from various sources, as detailed in Table 1. We mainly relied on publicly available

Table 1. HDL design sources.

Source	Designs[1] (Count)	Complexity (Code Length) {Min, Median, Max}
RTL-Coder[2][21]	18,450	{3, 29, 918}
VeriGen [5]	7,292	{5, 69, 27,025}
ISCAS'89 [22]	29	{53, 530, 54,778}
ISCAS'85 [22]	10	{17, 1225, 3925}
OpenCores [23]	54	{1, 103, 2716}
NVLDA [24]	33	{19, 1333, 42,051}
MetRex (ours)	25,868 (Train) 138*(Test)	{3, 35, 54,778}

[1] Number of designs after cleaning
[2] RTL-Coder dataset is LLM-generated.
* Test set derived from VerilogEval benchmark [3].

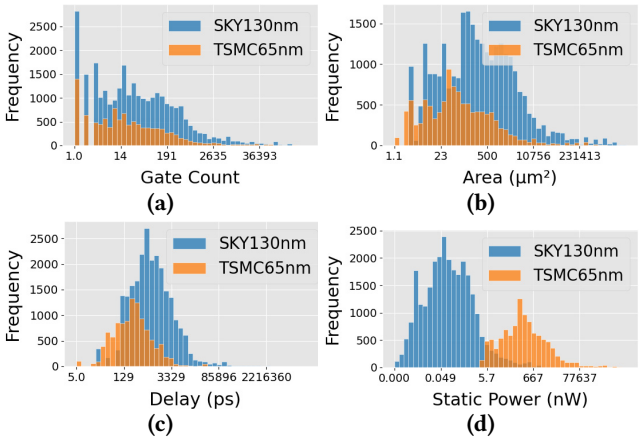


Figure 1. Dataset analysis showing: (a) gate count, (b) area, (c) delay, (d) static power distribution for both Skywater 130nm and TSMC 65nm

dataset sources that are used for evaluating LLMs for the task of Verilog code generation. Key sources include the RTL-Coder dataset [21], which contains designs generated by a GPT model, and VeriGen dataset [2, 5], which contains Verilog code extracted from GitHub repositories and academic textbooks. Additionally, we incorporated designs from ISCAS [22], OpenCores [23], and NVLDA [24]. Altogether, we collected 25, 868 designs, the majority of which are self-contained modules. These designs form the training split of our dataset. For the test set, we derived it from the VerilogEval benchmark [3].

Since the collected dataset contained LLM-generated and web-scraped Verilog code, it was important to clean the dataset to ensure the usability of these designs for further analysis and benchmarking. We undertook a comprehensive cleaning process that involved removing duplicate entries, filtering out non-synthesizable elements such as test benches and gate-level netlists, and rectifying errors and warnings detected during synthesis.

We automated the data cleaning flow by integrating an LLM agent with a synthesis tool and a Verilog compiler in an interactive feedback loop, inspired by the RTLFixer flow [8]. The LLM agent resolves errors and warnings flagged during

RTL

```
module adder (
    input a, b, cin,
    output cout, sum
);
    assign sum = a ^ b ^ cin;
    assign cout = a & b | a & cin | b & cin;
endmodule
```

Synthesis Thought
After synthesis, this design has **1 maj3_1**, **1 xor3_1**.

Area Thought
Area of **maj3_1** is **10.01**. Area of **xor3_1** is **23.77**. In total, we can compute **1*10.01 + 1*23.77 = 33.78**. Thus, total area is **33.78**.

Delay Thought
The critical path goes through the following gates: **xor3_1** with a fanout of **2** and a load capacitance of **0.01**. Delay of **xor3_1** is **0.34**. In total, we can compute **0.34 = 0.34**. Thus, total delay is **0.34**.

Static Power Thought
Leakage power of **maj3_1** is **2.09**. Leakage power of **xor3_1** is **15.22**. In total, we can compute **1*2.09 + 1*15.22 = 17.31**. Thus, the total static power is **17.31**.

Figure 2. Dataset sample, showing the Chain of Thought (CoT) template for estimating area, delay, and static power.

synthesis or syntax checking. Warnings, such as unused signals, are included in the loop to prevent them from obscuring the relationship between Verilog code and post-synthesis metrics. Our workflow uses Icarus Verilog [25], Yosys [26], and Cadence Genus for syntax and synthesis verification.

Lastly, the cleaned dataset was taken through the synthesis flow to report the area, delay, and power metrics. We used Yosys [26] for synthesizing the designs and reporting the area, and OpenSTA [27] for reporting the delay and power metrics. The designs were synthesized using the Skywater 130nm Process Design Kit (PDK) [28]. We also synthesized the designs using Cadence Genus Synthesis Solution and TSMC 65nm technology. Fig. 1a shows the gate count distribution after synthesis and Fig. 1b-d shows the distribution of the collected metrics in both technologies.

4.2 Chain of Thought (CoT) Template

To construct the intermediate reasoning steps for metric computation, we first parse the gates and their corresponding metrics from the synthesis reports. The parsed information is then used to construct natural language reasoning thoughts. Fig. 2 displays a sample from the *MetRex* dataset of a full adder design synthesized using Skywater 130nm technology. The CoT template includes four primary reasoning thoughts: synthesis, area, delay, and static power. The synthesis thought includes a breakdown of the gate types and their count in the synthesized netlist. The area thought details the calculation of the total area by summing the individual areas of each gate type identified in the synthesis thought. The delay thought breaks down the stages of the critical path, including the type of gate, fanout, capacitive load, and delay for each stage. It then sums the delay per stage to calculate the total critical path delay. The static

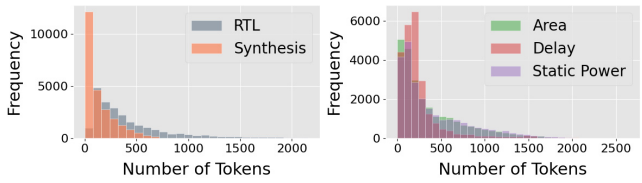


Figure 3. Dataset analysis showing token count distribution for (a) RTL and synthesis thoughts and (b) area, delay, and static power thoughts in the Skywater 130nm dataset.

Table 2. Test dataset derived from the VerilogEval benchmark [3], categorized by difficulty level.

Difficulty	Description	#	Gate Count {Min, Med, Max}
Level-1 (L1)	Basic logic gates	10	{1, 1, 1}
	Multi-bit gates	9	{2, 2, 100}
	1-bit comb. circuits	5	{2, 2, 4}
	Total #	**23**	**{1, 2, 100}**
Level-2 (L2)	Adder circuits	4	{2, 6, 15}
	Multi-bit comb. circuits	23	{1, 3, 11}
	Flip-Flop registers	14	{1, 3, 24}
	Basic Seq. circuits	2	{8, 8, 8}
	Total #	**43**	**{1, 3, 24}**
Level-3 (L3)	Finite state machines	24	{3, 11, 57}
	Counters	9	{10, 14, 48}
	Complex comb. logic	29	{1, 7, 580}
	Advanced Seq. circuits	9	{11, 67, 607}
	Total #	**72**	**{1, 14, 607}**

power thought outlines the leakage power for each gate type identified in the synthesis thought and sums these values to compute the total static power. Fig. 3a shows the distribution of the number of tokens for the RTL and synthesis reasoning thought and Fig. 3b shows the distribution of the number of tokens for the area, delay, and static power thoughts.

5 Experimental Results

5.1 Evaluation Setup

We conducted fine-tuning experiments using the train split of *MetRex*, comprising 25, 868 designs and the Skywater version of the dataset for our experiments. Our evaluation set was derived from the VerilogEval benchmark [3], containing 138 designs after excluding those with zero area or delay. The designs are categorized by difficulty in Table 2, where Level-1 includes simple combinational circuits with no more than 2-bit operators, Level-2 comprises moderate circuits such as adders, flip-flops, and shift registers, and Level-3 contains sophisticated designs like finite state machines and multi-bit arithmetic units. We measure performance using Mean Relative Error (MRE), defined in Eq. 1, where N is the number of designs in the test set, $\hat{R}_i$ is the LLM-estimated metric, and R_i is the ground truth metric reported from the EDA tool.

$$MRE = \frac{1}{N}\Sigma_{i=1}^{N}\frac{|\hat{R}_i - R_i|}{R_i} \times 100\% \quad (1)$$

However, the MRE can be heavily biased by outliers and it only evaluates the accuracy of the top-1 LLM-generated answer. Therefore, we introduce a new metric, *acc@k*, inspired by the *pass@k* metric [12]. The *acc@k*, defined in Eq. 2, measures the percentage of designs that meet a specific accuracy threshold when considering the median of the top-k LLM predictions. Specifically, for each design in the test set, we compute the relative error between the ground truth (R_i) and the median of the first k predictions ($\hat{R}_{i,1:k}$), then check if this error falls within a specified margin t.

$$acc@k(MRE \leq t) = E_N\left[\mathbb{1}\left(\frac{|median(\hat{R}_{i,1:k}) - R_i|}{R_i} \leq t\right)\right] \quad (2)$$

Here, N is the total number of designs in the evaluation set, and the indicator function $\mathbb{1}(.)$ returns 1 if the relative error between the median prediction and the ground truth is within the margin t, and 0 otherwise. We assess the *acc@k* with k values of 1, 5, 10, and MRE thresholds t of 10% and 20%. A higher *acc@k* indicates better model performance.

5.2 In-Context Learning (ICL)

In-Context Learning (ICL), also known as few-shot prompting, is a prompting technique used to extrapolate LLM's knowledge to new tasks by learning from a small number of context-specific examples [29]. In this study, we use ICL as a baseline to evaluate the base LLMs' ability to reason about post-synthesis metrics and to evaluate the impact of using chain-of-thought prompts on their reasoning capabilities. We use 10 few-shot examples, mainly composed of RTL descriptions of basic gates such as AND, NOR, and OR, and their respective post-synthesis metrics. These examples are designed to enhance the LLM's understanding of how basic Verilog logic operators are translated into logic gates within the Standard Cell Library (SCL), including their area, delay, and power characteristics.

Table 3. Impact of using chain-of-thought (CoT) prompt on the *acc@5* of the test set, using in-context learning. × means without CoT, while ✓ means with CoT. The Δ values in **bold** represent the improvements from using the CoT prompt.

Margin	Model	With CoT?	acc@5 ↑		
			Area	Delay	Static Power
10%	**Mixtral-8x7b**	×	19.6%	19.6%	13.0%
		✓	19.6%	22.5%	18.8%
		Δ	**+0.0%**	**+2.9%**	**+5.8%**
	LLama3-8B	×	10.1%	15.2%	6.5%
		✓	18.8%	23.9%	15.2%
		Δ	**+8.7%**	**+8.7%**	**+8.7%**
20%	**Mixtral-8x7b**	×	25.4%	26.1%	15.9%
		✓	26.1%	29.0%	26.1%
		Δ	**+0.7%**	**+2.9%**	**+10.1%**
	LLama3-8B	×	13.0%	21.0%	10.9%
		✓	23.9%	28.3%	21.7%
		Δ	**+10.9%**	**+7.2%**	**+10.9%**

Table 4. Finetuning results using Skywater 130nm instruction datasets on the area, delay, and static power metrics. Results show improvements of supervised fine-tuning as measured by the *acc@k* value, described in Section 5.1. × means the model is not fine-tuned and uses only in-context learning, while ✓ means the model is fine-tuned using an instruction dataset of RTL code and metric reasoning pair. **Bolded** values highlight best-performing accuracy for a given metric and error margin.

Margin (t) MRE ≤ t	Model	Finetuned ?	Area (acc@k) ↑			Delay (acc@k) ↑			Static Power (acc@k) ↑		
			acc@1	acc@5	acc@10	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10
10%	Mixtral-MetRex-8x7b	×	19.6%	19.6%	21.0%	23.9%	22.5%	22.5%	20.3%	18.8%	19.6%
		✓	43.5%	46.4%	45.7%	42.8%	45.7%	45.7%	39.1%	39.9%	38.4%
		Δ	+23.9%	+26.8%	+24.6%	+18.8%	+23.2%	+23.2%	+18.8%	+21.0%	+18.8%
	LLama3-MetRex-8b	×	17.4%	18.8%	18.1%	20.3%	23.9%	22.5%	15.9%	15.2%	15.2%
		✓	**58.0%**	**58.0%**	**58.7%**	**47.8%**	**47.1%**	**47.8%**	**42.0%**	**42.8%**	**41.3%**
		Δ	+40.6%	+39.1%	+40.6%	+27.5%	+23.2%	+25.4%	+26.1%	+27.5%	+26.1%
20%	Mixtral-MetRex-8x7b	×	25.4%	26.1%	26.1%	31.9%	29.0%	29.7%	25.4%	26.1%	28.3%
		✓	58.0%	61.6%	60.9%	50.7%	53.6%	55.8%	**53.6%**	**54.3%**	**52.2%**
		Δ	+32.6%	+35.5%	+34.8%	+18.8%	+24.6%	+26.1%	+28.3%	+28.3%	+23.9%
	LLama3-MetRex-8b	×	22.5%	23.9%	22.5%	25.4%	28.3%	28.3%	22.5%	21.7%	21.0%
		✓	**73.2%**	**76.1%**	**74.6%**	**61.6%**	**64.5%**	**63.8%**	52.2%	49.3%	47.1%
		Δ	+50.7%	+52.2%	+52.2%	+36.2%	+36.2%	+35.5%	+29.7%	+27.5%	+26.1%

Using these 10 few-shot examples, we evaluated the ability of different LLMs to reason about the post-synthesis metrics of the test set. We ran the experiments in two modes: one using the chain-of-thought (CoT) template shown in Fig. 2, and the second without CoT, where the total area, delay, static power of the few-shot examples are given directly without intermediate reasoning steps. The models tested include `Mixtral-8x7b` [30], and `Llama3-8b` [31]. The models were run locally with 4-bit quantization on a single A6000 GPU and prompted at a sampling temperature of 0. Results, summarized in Table 3, indicate that CoT prompting enhanced performance on average by 5.1%, 5.4%, and 8.9% on the area, delay, and static power metrics respectively.

5.3 Supervised Fine-tuning (SFT)

While few-shot prompting can help extrapolate LLM knowledge to new tasks, it is limited by how many examples we can fit in the context window and does not intrinsically instill knowledge in the LLM weights. Supervised fine-tuning (SFT) can help align the LLM to a specific downstream task by adjusting the LLM weights according to an instruction dataset. Therefore, we conduct extensive experiments on supervised fine-tuning. We utilize the train split of the *MetRex* dataset for instruction tuning and evaluate on the test dataset derived from the VerilogEval benchmark [3].

Our fine-tuning experiments are mainly focused on the `Mixtral-8x7b` and `Llama3-8b` models. We employ LoRA [32], a parameter-efficient fine-tuning technique that decomposes the weight matrices into smaller, manageable low-rank matrices. This approach significantly reduces the computational load and memory demands associated with fine-tuning LLMs. We fine-tune a LoRA adapter per metric using an instruction dataset of Verilog code and metric reasoning pair. Additionally, we quantize the LLMs to 4 bits to reduce the memory footprint. The models were fine-tuned using a maximum sequence length of 1048 tokens on a single A40 GPU.

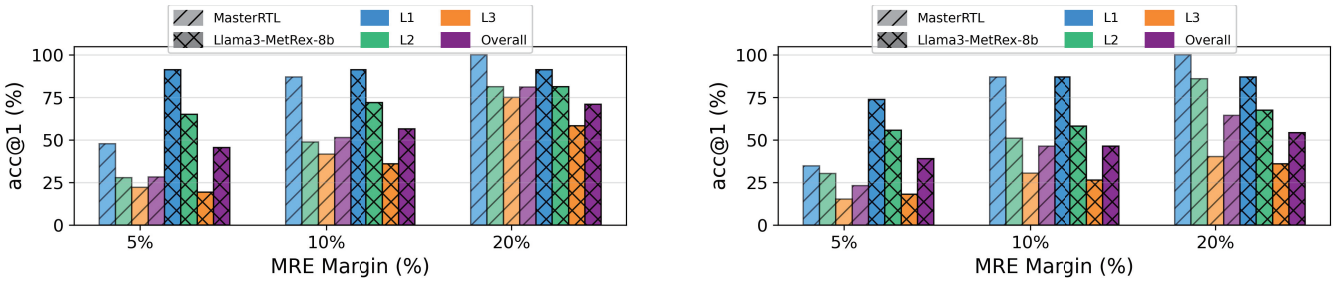
Table 4 shows the *acc@k* metric for the fine-tuned models using a LoRA rank of 128. All LLMs are evaluated at a sampling temperature of 0.4. Supervised fine-tuning significantly boosts the LLMs' estimation accuracy of the final metrics compared to their pre-trained few-shot prompted counterparts. SFT improved the *acc@1* on average by 37.0%, 25.3%, and 25.7% for the area, delay, and static power metrics, respectively. `Llama3-MetRex-8b` shows better performance on the area and delay metrics compared to `Mixtral-MetRex-8x7b`, which only outperforms `Llama3-MetRex-8b` in the static power estimates within the 20% error margin.

5.4 Comparison to Regression-based Models

In this section, we compare the accuracy of the finetuned `Llama3-MetRex-8b` (with a LoRA rank of 128 and 256 for area and static power, respectively) against regression-based machine learning approaches to highlight the opportunities and challenges of using LLMs for this task. We specifically compare to MasterRTL [11], which first converts the HDL code to a simple-operator graph (SOG) using Yosys, from which it extracts feature vectors for regression analysis.

Fig. 4 shows the comparative performance for the area and static power estimation, highlighting the percentage of designs with MRE less than 5%, 10%, and 20% across the three levels of complexity within our evaluation set. The data illustrates that the `Llama3-MetRex-8b` model can frequently generate more accurate answers than the regression-based model in less complex designs (level-1 and level-2) under the 5% and 10% error margins. However, it underperforms in level-3 primarily due to the increased reasoning complexity.

MasterRTL performs better under more relaxed error margins (20%), mainly because it utilizes detailed gate-type features from the extracted SOG, which helps stabilize the performance of the regression model across a broad range of problem complexities. However, the `Llama3-MetRex-8b` model operates directly on Verilog code, resulting in higher



(a) Area **(b)** Static Power

Figure 4. Comparison between MasterRTL [11] and finetuned `Llama3-MetRex-8b`, showing (a) area and (b) static power accuracy across the three levels in the test set.

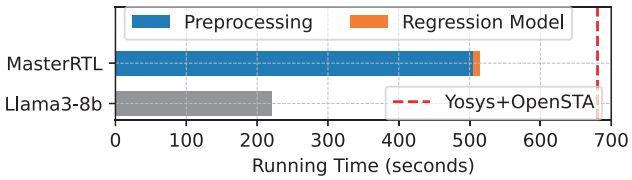


Figure 5. Run-time comparison between MasterRTL [11] and `Llama3-MetRex-8b`. Evaluation is done using the H100 GPU. Pre-processing is done using an Intel Xeon CPU.

sensitivity to variations in code design and susceptibility to generate extreme outliers.

Nonetheless, LLMs offer several advantages. First, they provide better overall estimates that are on average 17.4% and 2.5% more accurate within 5% and 10% error margins, respectively, compared to MasterRTL. Second, they provide interpretable results by explaining the breakdown of gates post-synthesis, offering insights beyond mere numerical predictions. Third, LLMs eliminate the need for preprocessing HDL code into intermediary formats and performing feature extraction, which significantly reduces runtime overhead. As shown in Fig. 5, the majority of MasterRTL's runtime is dedicated to generating the SOG and extracting feature vectors, totaling 505.3 seconds, whereas model inference time is minimal at just 8.5 seconds. Although LLMs inherently require substantial computational resources, leveraging GPU acceleration allows their runtime to be 2x faster than logic synthesis, and 1.7x faster than MasterRTL.

6 Discussion and Future Work

In this study, we aimed to assess the ability of LLMs to reason and predict post-synthesis metrics of Verilog code. Our collected dataset and evaluation framework mainly focused on self-contained and relatively small-scale designs, due to the limited fine-tuning context window. We aim to extend our dataset and fine-tuning experiments to include larger and more complex designs.

Additionally, we focused on the area, delay, and static power due to their relatively direct relationship to HDL code. Breaking down the switching power calculations in natural language to the LLM is more complicated as it requires propagating the activity factor through the logic gates and would require the LLM to understand the synthesized circuit topology and edge connection between these gates. Recent advancements in LLM research are showing progress in encoding graph data as natural language sequences [33, 34], which will help with tackling the switching power reasoning. This could potentially improve the LLM accuracy on the delay estimation as well, as it will be able to reason about different paths in the circuit graph. Moreover, we assume that both the target technology node and synthesis strategy are fixed. We aim to investigate the influence of different synthesis strategies on the LLM estimation accuracy to offer insights into the utility of these models in different hardware design environments.

Nonetheless, exploiting the reasoning capabilities of LLMs presents an exciting research opportunity for the hardware community. Particularly, the metric reasoning and estimation problem of HDL code will pave the way for tackling more difficult tasks such as generating efficient hardware code and accelerating the design exploration process. This study lays the groundwork for such future explorations.

7 Conclusion

In this paper, we introduced *MetRex*, a new benchmark for evaluating LLMs for the task of reasoning about Verilog code post-synthesis metrics. *MetRex* includes a large-scale dataset with a wide variety of HDL designs, annotated with their post-synthesis metrics, and chain of thought templates that detail the reasoning steps on how to compute these metrics. Our best performing model achieves accuracy rates of 73.2%, 61.6%, and 52.2% when estimating area, delay, and static power metrics within a 20% error margin, respectively. This work lays the foundation for a new line of research that leverages the capability of LLM reasoning to estimate post-synthesis metrics of HDL designs.

Acknowledgments

This work is supported by NSF grant 2350180.

References

[1] Hammond Pearce, Benjamin Tan, and Ramesh Karri. Dave: Deriving automatically verilog from english. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 27–32, 2020.

[2] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[3] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Invited paper: Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8, 2023.

[4] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 722–727. IEEE, 2024.

[5] 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. *ACM Transactions on Design Automation of Electronic Systems*, 2023.

[6] 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*, 2024.

[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. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2023.

[8] YunDa Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv preprint arXiv:2311.16543*, 2023.

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

[10] Xufeng Yao, Yiwen Wang, Xing Li, Yingzhao Lian, Ran Chen, Lei Chen, Mingxuan Yuan, Hong Xu, and Bei Yu. Rtlrewriter: Methodologies for large models aided rtl code optimization. *arXiv preprint arXiv:2409.11414*, 2024.

[11] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. Masterrtl: A pre-synthesis ppa estimation framework for any rtl design. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[13] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023.

[14] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

[15] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*, 2024.

[16] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108, 2014.

[17] Qijun Zhang, Shiyu Li, Guanglei Zhou, Jingyu Pan, Chen-Chia Chang, Yiran Chen, and Zhiyao Xie. Panda: Architecture-level power evaluation by unifying analytical and machine learning solutions. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 01–09. IEEE, 2023.

[18] Yikang Ouyang, Sicheng Li, Dongsheng Zuo, Hanwei Fan, and Yuzhe Ma. Asap: Accurate synthesis analysis and prediction with multi-task learning. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2023.

[19] Prianka Sengupta, Aakash Tyagi, Yiran Chen, and Jiang Hu. Early identification of timing critical rtl components using ml based path delay prediction. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2023.

[20] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[21] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–5. IEEE, 2024.

[22] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *1989 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1929–1934 vol.3, 1989.

[23] C. Albrecht. Iwls 2005 benchmarks. *International Workshop for Logic Synthesis (IWLS)*, 2005.

[24] NVIDIA. NVDLA Hardware. https://github.com/nvdla/hw.

[25] Stephen Williams et al. Icarus verilog: open-source verilog more than a year later. *Linux Journal*, 2002.

[26] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, 2013.

[27] Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. Toward an open-source digital flow: First learnings from the openroad project. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Google. Skywater-PDK. https://github.com/google/skywater-pdk.

[29] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[30] Mixtral AI. Mixtral 9-7b model. https://mistral.ai/news/mixtral-of-experts/, 2023.

[31] Meta. Llama 3. https://ai.meta.com/blog/meta-llama-3/, 2024.

[32] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.

[33] Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. Graphgpt: Graph instruction tuning for large language models. *arXiv preprint arXiv:2310.13023*, 2023.

[34] Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. Talk like a graph: Encoding graphs for large language models. *arXiv preprint arXiv:2310.04560*, 2023.