Benchmarking Open-Source Large Language Models for Verilog Code Generation and Synthesis

***Abstract*—Recent advancese language models (LLMs) have demonstrated their potential to automto automateration across domains. This paper investigates the applicability of open source LLMs for hardware description language (HDL) tasks, specifically the generation of Verilog modules. We benchmark three open-source models: Qwen2.5 (14B), Gemma2 (9B), and LLaMA3 (8B) versus a closed-source baseline (DeepSeek-6.7B), evaluating performance in 33 Verilog design tasks. We employ zero-shot, few-shot, and chain-of-thought prompting, multiple quantization formats, and fine-tuning via SFT and QLoRA. Our results show that open-source models, when appropriately tuned, can achieve high synthesizability and approach the reliability of proprietary models. We also automate testbench and formal verification (SVA) generation, with correctness checking scheduled for future work. This work demonstrates the feasibility of democratizing Electronic Design Automation (EDA) through accessible generative AI.**

***Index Terms*—LLM, Verilog, QLoRA, Fine-tuning, HDL Generation, Testbench, Zero-shot Prompting, Few-shot Prompting, Chain of Thought Prompting, Open-Source AI, EDA Automation**

I. Introduction

Electronic Design Automation (EDA) tools form the core of modern digital system design. However, tasks such as writing Verilog modules, creating testbenches, and generating formal assertions remain largely manual, time-consuming, and prone to error. With the emergence of generative AI, especially transformer-based large language models (LLMs), there is growing interest in automating hardware development workflows. Although commercial LLMs have demonstrated strong capabilities, they remain closed-source and cost-restrictive. This work explores whether fine-tuned open-source LLMs can perform comparably in Verilog code generation. We benchmark three open-source models and evaluate their performance against a closed-source baseline across diverse design tasks from the MG-Verilog dataset.

II. Background and Related Work

Generative closed-source LLMs like DeepSeek have set new benchmarks in code generation. For hardware tasks, existing studies on HDL generation have relied on a narrow set of prompting techniques and a limited range of LLMs. Fine-tuning methods such as Supervised Fine-Tuning (SFT) and Quantized Low-Rank Adaptation (QLoRA) have made it feasible to train large models on domain-specific tasks using commodity GPUs. Prompting methods, including zero-shot, few-shot, and chain-of-thought (CoT) prompting, have been effective in shaping model behavior.

III. Dataset and Models

A. Dataset

This study employs the MG-Verilog dataset, publicly available via Hugging Face Datasets, which comprises more than 11,000 Verilog design tasks. The dataset includes a diverse mix of combinational and sequential logic problems, ranging from simple arithmetic units like adders and multiplexers to more complex modules involving counters and state machines. Each entry in the dataset consists of a high-level design prompt (written in natural language), a corresponding Verilog implementation, and optionally, a testbench and formal assertions. This dataset serves as the foundation for fine-tuning the language models for generation quality and synthesizability.

B. Models

Four large language models were selected for this study: three open-source models and one closed-source baseline. The open-source models include Qwen2.5 (14B parameters), Gemma2 (9B parameters), and LLaMA3 (8B parameters). These models were fine-tuned on the MG-Verilog dataset using both Supervised Fine-Tuning (SFT) and Parameter-Efficient Fine-Tuning (QLoRA) techniques, as discussed in Section V. In contrast, the DeepSeek-6.7B model, a closed-source offering, was used without fine-tuning and accessed via its official API. It serves as a baseline for evaluating how well fine-tuned open-source models can perform relative to a proprietary model with a comparable parameter count. All open-source models were loaded and integrated using the Hugging Face Transformers library. This enabled uniform inference pipelines, quantization strategies, and prompting mechanisms across the models.

IV. Model Architecture and Code Generation Pipeline

The backbone of this project is based on decoder-only transformer architectures, commonly referred to as causal language models (CLMs). Transformers revolutionized sequence modeling by replacing recurrence with the self-attention mechanism, enabling the model to weigh the relevance of every token in a sequence relative to others. Unlike RNNs, transformers process entire sequences in parallel, allowing for efficient training and superior capture of long-range dependencies, crucial in tasks like Verilog generation, where the relationship between declarations and logic spans many lines.

For this study, we utilize decoder-only transformers, which generate text autoregressively—one token at a time—by conditioning only on the preceding tokens. All selected mod-

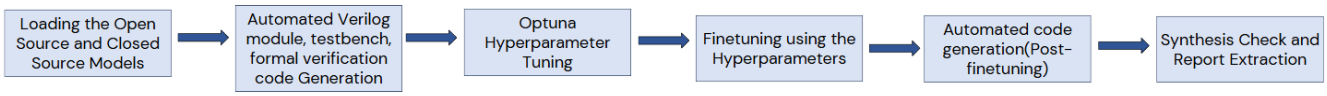

Fig. 1: Pipeline

els, including Qwen2.5-14B, Gemma2-9B, LLaMA3-8B, and DeepSeek-6.7B, fall under this category. Before text is processed by the model, it must be tokenized. We use Hugging Face's `AutoTokenizer` class to automatically match the tokenizer with its corresponding model. Tokenization converts input text into integer token IDs, which the model consumes during inference.

Model inference is handled via the `AutoModelForCausalLM` class from the Hugging Face Transformers library. This class loads the pre-trained models and executes generation using `model.generate()`, which predicts the next token sequence based on a given input prompt. After generation, token sequences are decoded back into human-readable text using `tokenizer.decode()`.

To ensure usable HDL output, a regex-based cleaning step is applied post-generation. This removes Markdown artifacts such as triple backticks (“‘) and isolates valid Verilog code blocks beginning with the `module` keyword. This step is critical to prepare the generated code for downstream synthesis and validation.

Overall, this pipeline—consisting of tokenization, inference, decoding, and cleanup—forms the foundation for automated Verilog generation using large language models.

V. Methodology

A. Prompting Strategies

To guide large language models (LLMs) in generating correct and synthesizable Verilog HDL code, three prompting techniques were systematically employed: zero-shot, few-shot, and chain-of-thought (CoT) prompting. Each method differs in the type and structure of contextual information provided to the model prior to generation.

In the zero-shot prompting approach, the model receives only a natural language description of the design task, relying entirely on its pretrained knowledge to interpret and complete the code. A typical zero-shot prompt used in this study was:

Zero-shot Prompt Example:

```
Write a synthesizable Verilog HDL module
for the following design: {design_code}.
Output only the Verilog code.
```

For example, when generating a behavioral counter:

Zero-shot Prompt Example:

```
Write a synthesizable Verilog HDL module
for the following design:
A behavioral counter that increments on
each clock cycle.
Module Name: behav_counter
Timescale: 1ns / 1ps
Parameters: None
Ports: Input: clk, Input: d[7:0], Input:
load, Input: clear, Output: cnt.
Output only the Verilog code.
```

In the few-shot prompting technique, the model is provided with one or two example input–output pairs before the actual design prompt. These help the model learn syntactic and structural patterns by analogy. Specifically, the Half Adder and Parameterized Adder were used as illustrative examples in the few-shot template due to their simplicity and clarity. These examples precede the new prompt to guide the model's behavior.

The chain-of-thought (CoT) prompting strategy introduces guided reasoning steps to structure the model's thought process before generating code. These intermediate instructions include analyzing the design, identifying ports, and describing internal behavior. A representative CoT prompt used in this work is:

Chain-of-Thought Prompt Example:

- Step 1: Analyze the design requirements for {design_code}.
- Step 2: Write the synthesizable Verilog HDL code.
- Output only the Verilog module without explanations.

This technique proved particularly effective for tasks involving hierarchical or stateful logic, where decomposition of the problem improved generation completeness and reduced structural errors.

B. Quantization and Inference

To ensure efficient inference on GPU-constrained environments, all models were deployed with Post-Training Quantization (PTQ) using the `bitsandbytes` library. PTQ is a compression technique that converts high-precision weights and activations (such as FP32) into lower-precision formats like FP16, INT8, or INT4. This dramatically reduces memory usage and inference time while retaining most of the original performance.

All inference experiments were conducted on an RTX A6000 GPU (48GB VRAM), which, while powerful, still necessitated the use of parameter-efficient techniques due to the large size of open-weight LLMs. By leveraging quantization, we made it feasible to run models such as Qwen2.5–14B, Gemma2-9B, and LLaMA3-8B within this constraint. PTQ enabled reduced computational demand and minimized deployment barriers, especially important for scenarios targeting hardware-limited environments like FPGAs or embedded systems.

C. Fine-Tuning Methods

To improve the performance of large language models on Verilog code generation tasks, a combination of Super-

vised Fine-Tuning (SFT) and Parameter-Efficient Fine-Tuning (PEFT) techniques was employed. SFT involved training the models on the MG-Verilog dataset in a fully supervised manner, allowing them to internalize Verilog-specific syntax, design conventions, and module structures.

As part of PEFT, Low-Rank Adaptation (LoRA) was applied to reduce the number of trainable parameters by decomposing weight updates into low-rank matrices, significantly lowering the computational burden.

Building on this, Quantized Low-Rank Adaptation (QLoRA) further extended LoRA by enabling fine-tuning under 4-bit quantization. QLoRA inserts trainable low-rank adapter layers into otherwise frozen base models, allowing large models like Gemma2-9B to be fine-tuned efficiently using modest GPU resources while preserving output quality. This combined approach of SFT and PEFT ensured both scalability and high-quality Verilog generation.

D. Hyperparameter Tuning

To optimize fine-tuning performance, Bayesian hyperparameter optimization was performed using the Optuna framework. The optimization objective was to minimize the training loss obtained during a single epoch of fine-tuning. Each trial explored a combination of six key hyperparameters: learning rate, per-device batch size, gradient accumulation steps, LoRA rank, LoRA alpha, and LoRA dropout.

All models were fine-tuned using parameter-efficient Quantized Low-Rank Adaptation (QLoRA) with 4-bit quantization to reduce memory usage. The base models were prepared using the Hugging Face Transformers and `bitsandbytes` libraries, and fine-tuned using the TRL framework's `SFTTrainer`. For each Optuna trial, a different hyperparameter configuration was used to train the model on a subset of the MG-Verilog dataset, and the final training loss was recorded from the trainer's log history.

Ten trials were conducted for each model, and the configuration yielding the lowest training loss was selected for final fine-tuning and code generation. This approach ensured that the models were trained under optimal settings for Verilog code generation tasks.

VI. System Design and Automation Pipeline

To manage the combinatorial complexity of evaluating multiple models, prompting techniques, quantization formats, and 33 Verilog design tasks, a fully automated generation and validation pipeline was developed. This system allowed reproducible and scalable experimentation with minimal manual intervention.

The automation begins with a command-line interface built using the `argparse` library in Python. This interface enables users to flexibly specify the model name, prompting strategy (zero-shot, few-shot, or chain-of-thought), and quantization level (int4, int8, or float16) at runtime. This approach provides modularity and reduces the need for hard-coded configuration changes between runs.

To manage and validate structured experiment configurations, the `Pydantic` library was used. It provides type-checked configuration classes that ensure all required parameters are specified and conform to expected formats. This helped prevent runtime errors due to malformed input configurations and facilitated clean experiment tracking.

Following model inference, the generated outputs often contain verbose or unstructured text, such as commentary, duplicated code blocks, or irrelevant content. Regular expressions (regex) were used extensively to post-process and clean the raw LLM outputs. Specifically, regex patterns were employed to extract the first valid `module ... endmodule` block from the generated response, remove extraneous lines, and standardize indentation and formatting for synthesis readiness.

To automate the execution of generation experiments across all combinations of model, prompt type, and quantization setting, a series of Bash scripts was developed. These scripts loop through each configuration, invoke the generation scripts, and organize outputs into structured directories. Over 90% of the entire Verilog code generation and processing workflow was automated through these scripts, significantly reducing manual effort and ensuring consistency.

A. Hardware Synthesis and Evaluation

To evaluate the hardware implementability of the generated Verilog modules, an automated synthesis pipeline was developed using Xilinx Vivado. Each design was inserted into a standardized synthesis project targeting a specific FPGA architecture. For every module, synthesis and implementation runs were executed in batch mode to extract metrics such as timing delay, power consumption, and resource utilization.

The entire process was automated using a combination of scripting and programmatic control. Modules were added to the synthesis environment and synthesized sequentially. After completion, reports on timing, power and utilization were generated and parsed to extract relevant performance metrics. Synthesis success was verified by checking status logs and only successfully synthesized modules were considered for analysis.

All results were compiled into structured CSV files for downstream comparison and benchmarking across different models and fine-tuning strategies.

VII. Testbench and Formal Verification Generation

LLMs were also used to generate testbenches and SystemVerilog Assertions (SVAs) for each design. These were structured to include clock/reset logic, stimulus generation, and assertion-based checking. While the generation is complete, correctness validation via simulation and property checking is pending.

VIII. Results and Analysis

To systematically evaluate the performance of large-language models in real-world hardware contexts, we synthesized the Verilog modules generated by each model configuration across all prompting strategies and quantization levels.

TABLE I: Verilog Designs Used for Benchmarking

Half Adder	Full Adder	Parameterized Adder	Parameterized Subtractor	Parameterized Multiplier	Parameterized Divider	32-bit Register File	Parameterized Shift Register	4-bit ALU	RAM	ROM
FSM (1100 pattern)	Fibonacci Number	Binary Adder Tree	Ternary Adder Tree	Dual Clock Synchronous RAM	Single Clock Synchronous RAM	RAM with Separate Input and Output Ports	Single-Port RAM	Counter with Asynchronous Reset	Bidirectional Pin	UART
FFT Module	Digital Filter	BCD to Gray converter	7 Segment LCD	Gray Counter (Design 1)	Behavioral Counter (Design 2)	Parameterized Comparator (Design 3)	Modulation and Demodulation (Design 4)	Parameterized Counter (Design 5)	True Dual-Port RAM with a Single Clock (Design 6)	8x64 Shift Register with Taps (Design 7)

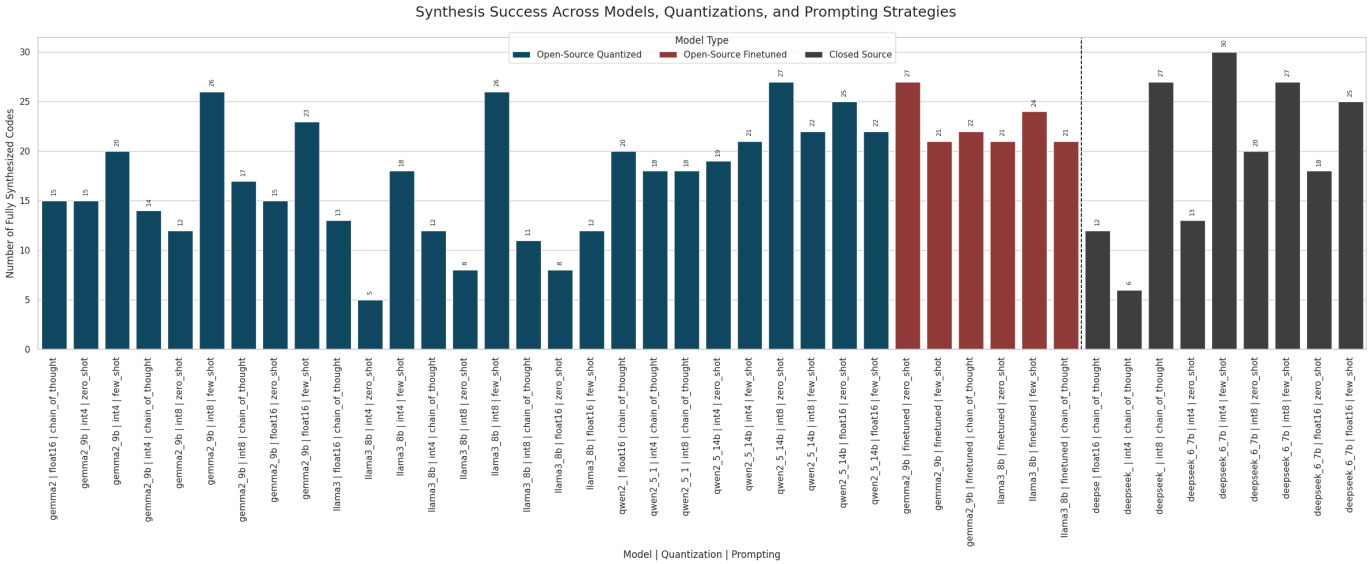

Fig. 2: Synthesis Results of Verilog Designs Generated

The synthesis process was carried out using the Xilinx Vivado, with critical metrics extracted from the generated reports on timing, power, and resource utilization.

These metrics enabled an objective assessment of both functional correctness—measured by synthesis success—and hardware quality, including efficiency, and performance. The success rate across the 33 benchmark designs, along with timing slack, LUT usage, and power estimates, formed the core evaluation criteria for each model-prompt-quantization combination.

A. Synthesis Pass Rate

The synthesis success rate—measured as the number of Verilog modules that compiled successfully out of 33 design tasks—varied notably across different model configurations. Fine-tuned open-source models consistently outperformed both their quantized counterparts and, in several cases, the closed-source DeepSeek model. Notably, Gemma2-9B (fine-tuned) with zero-shot prompting achieved a synthesis success rate of 81.82% (27/33), comparable to DeepSeek's best configuration, which required few-shot prompting with int4 quantization. Similarly, LLaMA3-8B (fine-tuned) achieved up to 72.73%, demonstrating strong performance across prompting strategies.

Even without fine-tuning, quantized open-source models exhibited competitive results. For instance, Gemma2-9B + int8 + few-shot and LLaMA3-8B + int8 + few-shot each synthesized 26/33 (78.79%) of the modules, rivaling DeepSeek's best-performing float16 and int8 configurations. This suggests that open-source models, even in their pre-fine-tuned form, can generalize effectively to hardware design tasks when paired with well-crafted prompts.

While few-shot prompting was the most consistently successful approach overall, zero-shot prompting also delivered top-tier results in specific cases, especially for fine-tuned or well-pretrained models. For example, Qwen2.5-14B + int8 + zero-shot achieved 27/33 (81.82%) synthesis success, despite not being fine-tuned. This highlights that well-trained base models can perform strongly even without explicit task demonstrations.

The single best-performing configuration was DeepSeek 6.7B + int4 + few-shot, which achieved a synthesis success rate of 90.91% (30/33). However, fine-tuned open-source models came close to this level of performance while using less aggressive quantization and simpler prompting strategies, narrowing the gap significantly.

Overall, these findings reinforce a central insight of this study: with targeted fine-tuning and prompting strategies,

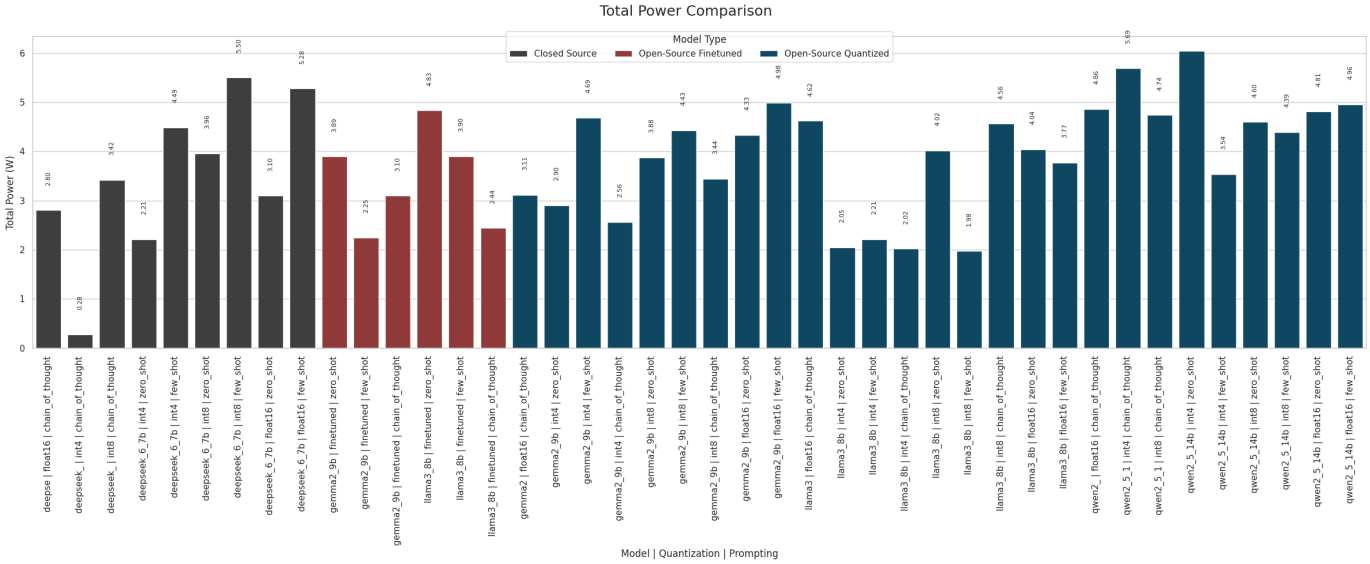

Fig. 3: Total Power Consumption Results of Verilog Design Generated

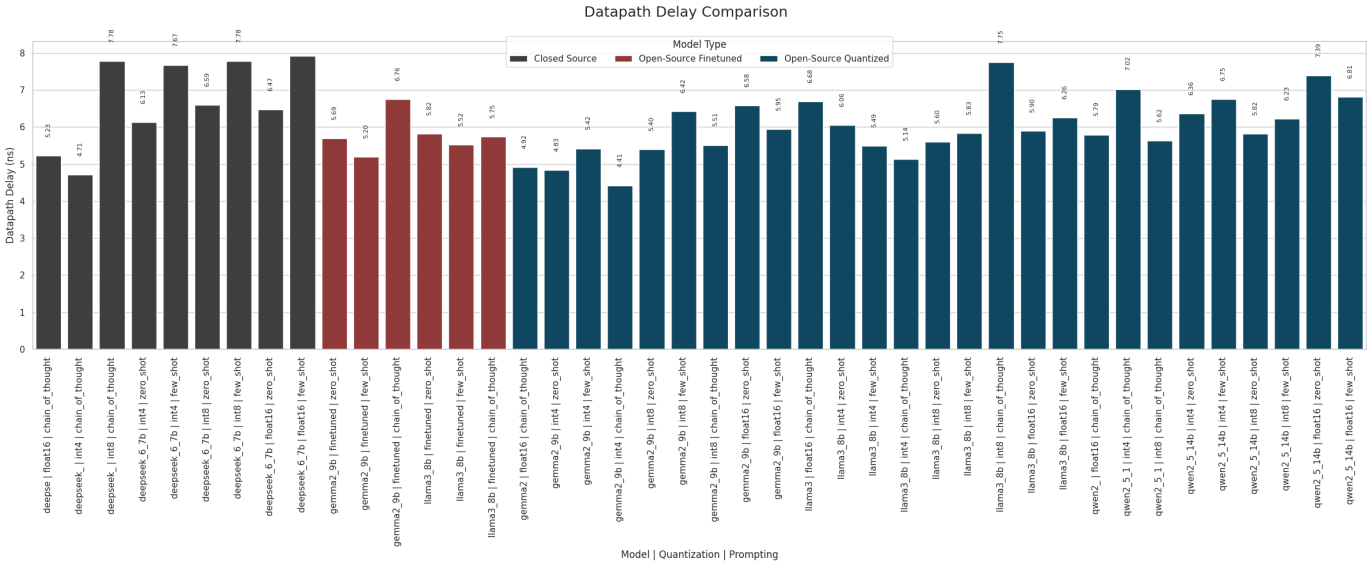

Fig. 4: Datapath Delay Results of Verilog Design Generated

open-source LLMs can generate Verilog code with hardware-level synthesizability that matches or even exceeds that of proprietary models. This makes them a compelling, transparent, and scalable alternative for domain-specific code generation in digital design.

B. Power Usage

The power consumption analysis revealed clear trade-offs between performance and efficiency across different model configurations. The lowest overall power usage was recorded by DeepSeek + int4 + chain-of-thought, consuming just 0.28 W, though this came with a low synthesis success rate (6/33). Among open-source models, fine-tuned Gemma2-9B stood out for its balanced efficiency, particularly in the few-shot setting, where it consumed only 2.25 W while synthesizing 21 designs. Similarly, quantized configurations of LLaMA3-8B, such as int8 + few-shot, delivered strong synthesis performance (26/33) with a power draw of just 1.98 W, highlighting the energy efficiency of lightweight open models. Overall, fine-tuned and quantized open-source models achieved competitive or superior power profiles compared to closed-source baselines, especially when paired with the right prompting strategy.

C. Datapath Delay

The datapath delay analysis highlights the timing efficiency of different model configurations. Open-source quantized models consistently outperformed closed-source counterparts

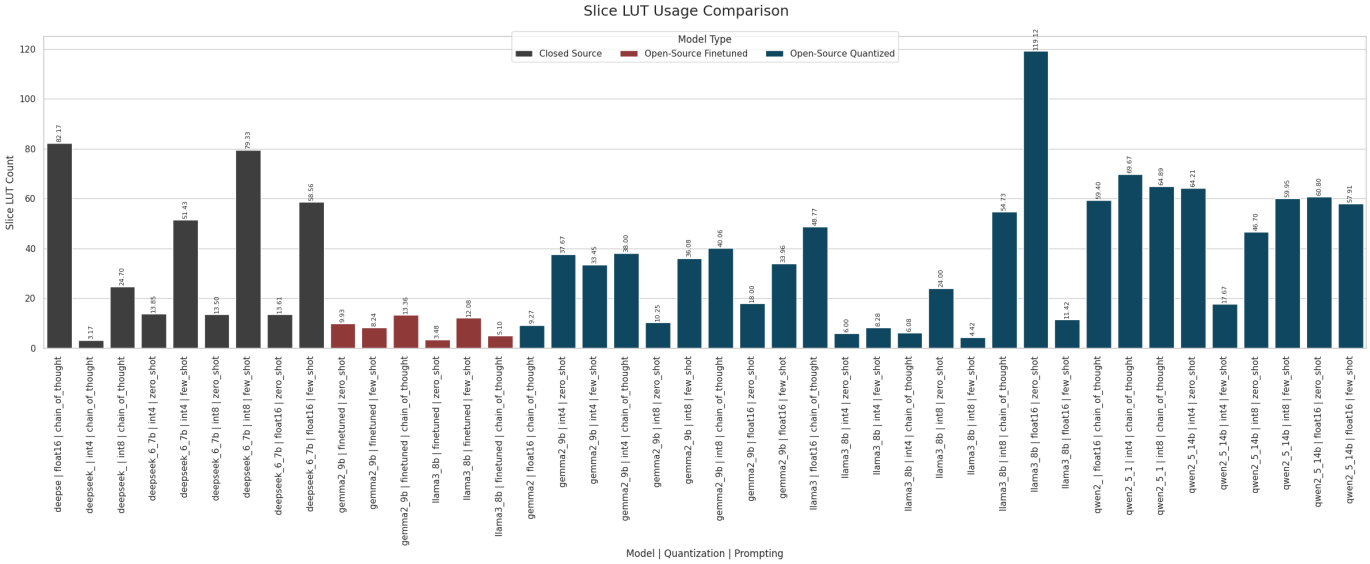

Fig. 5: Slice LUT Utilization Results of Verilog Design Generated

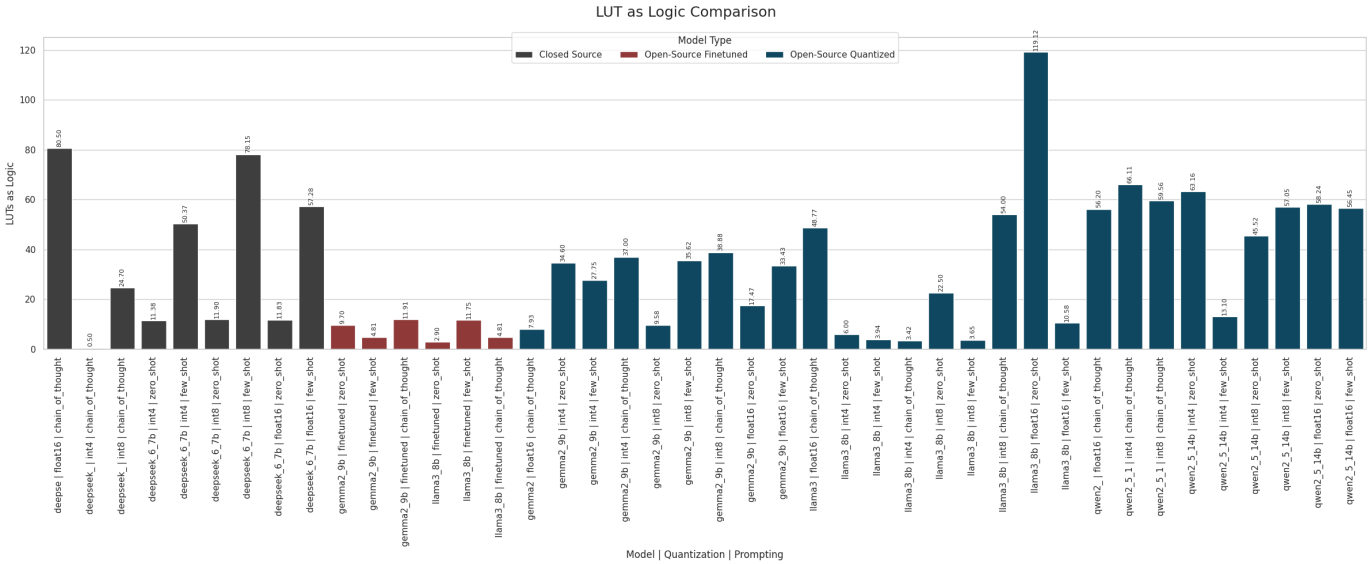

Fig. 6: LUT as Logic Utilization Results of Verilog Design Generated

in minimizing delay, with the best result from Gemma2-9B + int4 + chain-of-thought, achieving just 4.41 ns—the lowest observed. In contrast, DeepSeek models, particularly in int8 and float16 modes, exhibited significantly higher delays, often exceeding 7.7 ns, indicating suboptimal logic synthesis. Fine-tuned open-source models like Gemma2-9B and LLaMA3-8B achieved balanced performance, maintaining 5.2–5.8 ns delays while preserving high synthesis rates. Qwen2.5-14B models, though larger, showed competitive timing, with int8 + zero-shot achieving a respectable 5.82 ns delay. Overall, this demonstrates that quantized open-source models can deliver faster and more timing-efficient Verilog implementations, especially when combined with the right prompting strategy.

D. Slice LUT and LUT-as-Logic Utilization

The analysis of Slice LUT and LUT-as-Logic usage reveals important differences in design compactness across models. While fine-tuned open-source models like Gemma2-9B generally produced efficient designs—with many configurations under 15 LUTs—there were exceptions. For instance, LLaMA3-8B + float16 + zero-shot generated a highly verbose design consuming over 119 LUTs, the highest among all. Some Qwen2.5-14B configurations also exceeded 40 LUTs, particularly in zero-shot and chain-of-thought prompting, indicating bloated logic despite quantization. Meanwhile, DeepSeek showed inconsistent utilization, ranging from compact designs (int4 + CoT: 0.5 LUTs) to highly verbose outputs (float16 + CoT: 80+ LUTs), highlighting the variability in its code gener-

ation. In contrast, leaner configurations—such as LLaMA3-8B (int4 + few-shot: 3.94 LUTs) and Gemma2-9B (int8 + few-shot: 35 LUTs)—demonstrate how prompting strategy and quantization significantly impact resource usage.

IX. Discussion

Fine-tuned open-source models like Gemma2-9B and LLaMA3-8B matched or exceeded closed-source models across synthesis rate, power efficiency, timing, and utilization. Prompting strategies played a crucial role—few-shot was most effective, while chain-of-thought improved completeness in complex designs. Fine-tuning with MG-Verilog led to major gains in synthesizability and power efficiency, even on quantized models using QLoRA. Quantization also played a key role in enhancing efficiency: int4 and int8 models, when paired with the right prompting, synthesized over 20 modules with power usage as low as 2 W and competitive timing—making them ideal for resource-constrained or large-scale generation. Notably, Qwen2.5-14B, despite being unfine-tuned, achieved top-tier results (81.82%) with the right prompting and quantization. Automation using CLI tools and log parsing enabled scalable evaluation across 1,188 synthesis runs. These results position open-source LLMs as strong, cost-effective alternatives for domain-specific HDL generation.

X. Conclusion

This study shows that open-source LLMs, when paired with targeted fine-tuning and effective prompting, can serve as viable, high-performing alternatives to proprietary models for Verilog code generation. With competitive synthesis rates and hardware efficiency, they offer a scalable and transparent path forward for automating digital design workflows—especially when supported by quantization-aware techniques and automation pipelines.

XI. Future Work

Future extensions of this work include comprehensive correctness validation of the generated testbenches and SystemVerilog Assertions (SVAs), which would ensure functional accuracy beyond synthesizability. Additionally, integrating energy and performance modeling using frameworks like Timeloop and Accelergy could provide deeper insights into the downstream impact of generated designs. Expanding support to more complex hardware structures such as finite state machines (FSMs) and pipelined architectures is also a key direction. Finally, exploring reinforcement learning or reward modeling techniques may help better align generated Verilog code with hardware design objectives through feedback-driven optimization.

Acknowledgment

We express our sincere gratitude for the opportunity, guidance, and support extended by Professor Joycee Mekie throughout the research. We are also deeply thankful to Mihir Agarwal and Zaqi Momin, whose previous work laid the foundation for this study. Despite having graduated, their continued guidance and insights were invaluable in shaping the direction and execution of this research.

References

[1] M. Agarwal, Z. Momin, K. Prasad and J. Mekie, "VeriBench: Benchmarking Large Language Models for Verilog Code Generation and Design Synthesis," *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, London, United Kingdom, 2025, Available: https://doi.org/10.1109/ISCAS56072.2025.11044004.

[2] DataCamp, "Zero-Shot Prompting: Getting Started," [Online]. Available: https://www.datacamp.com/tutorial/zero-shot-prompting

[3] DataCamp, "Few-Shot Prompting: A Practical Guide," [Online]. Available: https://www.datacamp.com/tutorial/few-shot-prompting

[4] Prompting Guide, "Chain-of-Thought Prompting," [Online]. Available: https://www.promptingguide.ai/techniques/cot

[5] Hugging Face, "Free Course on Large Language Models," [Online]. Available: https://huggingface.co/learn/llm-course

[6] Nipun Batra, "Machine Learning 2024 - Course Schedule," [Online]. Available: https://nipunbatra.github.io/ml2024/schedule.html

[7] Intel, "Verilog HDL Example Designs," [Online]. Available: https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/horizontal/verilog.html

[8] YouTube, "FPGA & Verilog Synthesis Tutorial," [Online]. Available: https://www.youtube.com/watch?v=bCz4OMemCcAt=1896s

[9] MIT Accelergy, "Accelergy Power Modeling Tool - Tutorial," [Online]. Available: https://accelergy.mit.edu/tutorial.html?authuser=0

[10] Zaqi Momin, "Hardware Accelerator Research Notes," [Online]. Available: https://zaqimomin.craft.me/HardwareAccelerator

[11] DeepSeek, "DeepSeek Coder 6.7B Instruct Model," [Online]. Available: https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct

[12] Qwen, "Qwen2.5-14B Instruct Model," [Online]. Available: https://huggingface.co/Qwen/Qwen2.5-14B-Instruct

[13] Google, "Gemma 2-9B Instruction-Tuned Model," [Online]. Available: https://huggingface.co/google/gemma-2-9b-it

[14] Meta, "Meta LLaMA 3 8B Instruct Model," [Online]. Available: https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct

[15] Hugging Face Datasets, "MG-Verilog Dataset by GaTech-EIC," [Online]. Available: https://huggingface.co/datasets/GaTech-EIC/MG-Verilog