ResBench: Benchmarking LLM-Generated FPGA Designs with Resource Awareness

Ce Guo
Imperial College London
United Kingdom
c.guo@imperial.ac.uk

Tong Zhao
Imperial College London
United Kingdom
tong.zhao24@imperial.ac.uk

Abstract

Field-Programmable Gate Arrays (FPGAs) are widely used in modern hardware design, yet writing Hardware Description Language (HDL) code for FPGA implementation remains a complex and time-consuming task. Large Language Models (LLMs) have emerged as a promising tool for HDL generation, but existing benchmarks for LLM-based code generation primarily focus on functional correctness while overlooking hardware resource usage. Furthermore, current benchmarks offer limited diversity and do not fully represent the wide range of real-world FPGA applications. To address these shortcomings, we introduce ResBench, the first resource-focused benchmark explicitly designed to distinguish between resource-optimized and inefficient LLM-generated HDL code. ResBench consists of 56 problems across 12 categories, covering applications from finite state machines to financial computing. Our open-source evaluation framework automatically tests LLMs by generating Verilog code, verifying correctness, and measuring resource usage. The experiments, which primarily analyze Lookup Table (LUT) usage, reveal significant differences among LLMs, demonstrating ResBench's capability to identify models that generate more resource-optimized FPGA designs.

CCS Concepts

• **Hardware → Board- and system-level test**; **Reconfigurable logic applications**; *Functional verification*; **Physical verification**; • **Software and its engineering → Source code generation**.

Keywords

Large Language Models (LLMs), Hardware Description Languages (HDLs), Verilog Code Generation, FPGA Resource Utilization, Automated Benchmarking, Empirical Evaluation of LLMs

ACM Reference Format:
Ce Guo and Tong Zhao. 2025. ResBench: Benchmarking LLM-Generated FPGA Designs with Resource Awareness. In *The International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies 2025 (HEART 2025), May 26–28, 2025, Kumamoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3728179.3728192

HEART 2025, Kumamoto, Japan

ACM ISBN 979-8-4007-1432-0/25/05
https://doi.org/10.1145/3728179.3728192

1 Introduction

Field-Programmable Gate Arrays (FPGAs) are widely used in reconfigurable computing, providing flexible and high-performance hardware implementations for applications such as artificial intelligence acceleration, financial computing, and embedded systems. However, FPGA development traditionally requires manual coding in hardware description languages (HDLs), a process that is both time-consuming and prone to errors. Large Language Models (LLMs) have recently shown potential in automating HDL generation, offering a way to improve FPGA design productivity.

While studies such as VeriGen [32] and RTLLM [23] have examined the feasibility of generating Verilog with LLMs, most existing benchmarks focus primarily on functional correctness while paying little attention to resource constraints, which is an essential consideration in FPGA design.

FPGA designs are subject to strict hardware resource constraints. Even when HDL code passes functional correctness tests, its efficiency in utilizing these resources can vary significantly depending on how logic is optimized and mapped to FPGA hardware. In reconfigurable computing, resource-aware optimizations play a crucial role in determining whether a design can be practically deployed and the degree of parallelism achievable. However, existing benchmarks for LLM-generated hardware designs typically evaluate the generated code based only on syntax and functional correctness, overlooking resource-related issues.

To address this limitation, we introduce **ResBench**, the first FPGA-resource-focused benchmark specifically designed to evaluate LLM-generated designs based on resource usage. Unlike previous benchmarks that focus primarily on syntax and functional correctness, ResBench highlights how well LLMs generate Verilog code optimized for FPGA resource utilization. Our key contributions are:

- A resource-focused benchmark featuring 56 problems across 12 categories, covering real-world FPGA workloads such as combinational logic, state machines, AI accelerators, and financial computing applications. (Section 3)
- An open-source automated evaluation framework that performs LLM querying, functional correctness testing, FPGA synthesis, and resource measurement[1]. The framework automatically generates Verilog code using LLMs and evaluates its correctness and resource usage. (Section 4)
- A detailed study of nine LLMs, comparing their performance in functional correctness and FPGA resource usage. The results reveal substantial differences in how various models generate resource-conscious designs. (Section 5)

[1]Code repository: https://github.com/jultrishyyy/ResBench

By integrating FPGA resource awareness into benchmarking, ResBench provides a practical evaluation of LLM-generated HDL. This benchmark establishes a foundation for advancing AI-driven FPGA design, encouraging the development of more resource-efficient models optimized for FPGAs.

2 Background

This section explores the evolution of large language models (LLMs) from general code generation to hardware design generation using hardware description languages (HDLs). We examine their capabilities and limitations, as well as existing benchmarks for LLM-generated hardware design.

2.1 Code-specialized and HDL-specialized LLMs

Language models have seen significant advancements, particularly with the introduction of Transformers [36]. Large-scale pre-trained models such as BERT [8], the GPT series [5, 28, 44], and PaLM 2 [7] have expanded their capabilities across various tasks, including code generation.

Code-specialized LLMs are models designed to generate computer programs based on human-language prompts. Surveys such as [16, 39, 45] review the latest techniques and model architectures developed for this purpose. In general, a code-specialized LLM can be created using two different approaches.

- The first approach trains models from scratch on large-scale open-source code datasets spanning multiple programming languages, as demonstrated by CodeGen [26], InCoder [9], and StarCoder [19]. These code-first LLMs excel at code completion and multi-language generation. However, this method demands extensive computational resources and high-quality datasets. Additionally, these models may struggle with instruction-driven tasks due to their limited ability to process human language instructions.
- The second approach fine-tunes general-purpose LLMs for coding tasks, as seen in Codex [6], which is derived from GPT-3, and Code Llama [30], which builds on Meta's Llama. Fine-tuning incorporates the linguistic knowledge of general LLMs, allowing them to handle natural language prompts while maintaining strong coding capabilities. This method is more computationally efficient than training from scratch, but it may lack the precision of code-first models in understanding programming language syntax.

While significant research has explored the use of language models for general software code generation, the application of LLMs to HDL code has not received comparable attention. Existing work on HDL code generation primarily focuses on improving general-purpose or software-code-specialized LLMs. In particular, these studies aim to improve these LLMs' understanding of hardware description tasks by training them on HDL datasets and benchmarking frameworks. Notable efforts in this area include VeriGen, MEV-LLM, and AutoVCoder.

- VeriGen [33] fine-tunes CodeGen (2B, 6B, 16B) on a Verilog dataset collected from GitHub repositories and textbooks. It employs supervised fine-tuning with problem-code pairs, validating functional correctness using a benchmark problem set and problems from HDLBits tutorials [31].
- MEV-LLM [25] trains on 31,104 source code files from GitHub, labeled by GPT-3.5, to fine-tune CodeGen (2B, 6B, 16B) and GEMMA (2B, 7B) models. This approach yields an improvement of up to 23.9% in the Pass@k metric [6] over VeriGen.
- AutoVCoder [10] also uses Verilog code from GitHub, filtering high-quality samples with ChatGPT-3.5. It applies a two-stage fine-tuning process to improve generalization, with final evaluation conducted on a real-world benchmark.

Beyond direct fine-tuning, reinforcement learning approaches like Golden Code Feedback is used to refine models iteratively using user feedback [40]. Similarly, multi-modal techniques such as VGV [41] integrate circuit diagrams with textual data during training, allowing models to understand spatial and parallel aspects of circuit design.

2.2 Benchmarks for LLM-Generated Software

The research community has recognized the need for standardized benchmarks to rigorously evaluate LLM-generated code in terms of design correctness.

Most existing benchmarks for code generation are tailored for software development rather than hardware design. For instance, HumanEval [6] and MBPP (Mostly Basic Python Problems) [20] are widely used to evaluate LLMs for software code generation. HumanEval consists of 164 Python programming tasks, each with a function signature and a set of test cases to validate correctness. While relatively small in scope, this benchmark is carefully designed, making it well-suited for quick functional correctness evaluations of Python-based code generation. Several extensions have expanded its coverage, including HumanEval+ [20], which increases the number of test cases by 80 times, and HumanEvalPack [24], which extends the benchmark to six programming languages.

Similarly, MBPP comprises approximately 974 short Python programming tasks, each including a task description prompt, a code solution, and three automated test cases. It emphasizes both correctness and clarity. Its enhanced version, MBPP+ [20], refines flawed implementations and expands the number of test cases to improve robustness.

While HumanEval and MBPP provide fundamental benchmarks, they primarily focus on entry-level programming tasks and do not always reflect the complexity of real-world software development. To address this limitation, benchmarks with more intricate problem sets have been introduced. For example, DSP-1000 [18] contains 1,000 science-related programming tasks from seven Python libraries, covering a diverse set of topics and incorporating multi-criteria evaluation metrics to provide a more realistic assessment of code generation models.

2.3 Benchmarks for LLM-Generated Hardware

Existing benchmarks for hardware design generation in HDL are often based on experiences and lessons learned in software code benchmarking, particularly in designing diverse problem sets and developing automated evaluation frameworks. Similar to their software counterparts, performance metrics for hardware design generation frequently focus on design correctness measures such as Pass@k. However, benchmarking hardware designs typically involves simulation-based hardware verification [27].

Table 1: Benchmarks for LLM HDL Generation

Benchmark	Size	PL	Type	Features
VerilogEval [21]	156	Verilog	Verilog code generation tasks	Covers a wide range of tasks from simple combinational circuits to finite state machines; includes automatic functional correctness testing.
HDLEval [17]	100	Multiple	Language-agnostic HDL evaluation	Evaluates LLMs across multiple HDLs using standardized testbenches and formal verification; categorizes problems into combinational and pipelined tests.
PyHDL-Eval [4]	168	Python-embedded DSLs	Specification-to-RTL tasks	Focuses on Python-embedded DSLs for hardware design; includes Verilog reference solutions and testbenches; evaluates LLMs' ability to handle specification-to-RTL translations.
RTLLM [22]	50	Verilog, VHDL, Chisel	Design RTL generation	Supports evaluation across multiple HDL formats; spans various design complexities and scales; includes an automated evaluation framework.
VHDL-Eval [37]	202	VHDL	VHDL code generation tasks	Aggregates translated Verilog problems and publicly available VHDL problems; utilizes self-verifying testbenches for functional correctness validation.
GenBen [38]	351	Verilog	Fundamental hardware design and debugging tasks	Evaluates synthesizability, power consumption, area utilization, and timing performance to ensure real-world applicability.

Unlike software benchmarks, which evaluate correctness by directly executing code, hardware designs require dedicated testbench scripts to simulate hardware behavior and validate functionality. A typical evaluation process involves querying the LLM for HDL code, executing the generated HDL within a hardware simulation environment, and comparing the outputs to determine correctness.

Some benchmarks have emerged to address these challenges, as presented in Table 1. Among them, VerilogEval, HDLEval, and PyHDL-Eval are widely recognized.

- VerilogEval [21] is a widely adopted benchmark for evaluating LLMs in Verilog code generation [1, 10, 43]. It consists of 156 problems taken from the HDLBits tutorial website [31], covering a range of Verilog tasks from combinational circuits to finite state machines. The framework automates functional correctness testing by comparing the simulation outputs of generated designs against predefined golden solutions.
- HDLEval [17] follows a language-agnostic approach. In particular, this benchmark allows the same set of problems, formulated in plain English, to be evaluated across different HDLs. The benchmark consists of 100 problems systematically categorized into combinational and pipelined designs, covering fundamental hardware components such as logic gates, arithmetic operations, and pipelined processing units. A prominent feature of this benchmark is the use of formal verification instead of unit tests. This feature ensures that the generated HDL code is functionally correct and maintains logical equivalence with reference implementations.
- PyHDL-Eval [4] is a framework for evaluating LLMs on specification-to-Register Transfer Level (RTL) tasks within Python-embedded domain-specific languages (DSLs). It includes 168 problems across 19 subclasses, covering combinational logic and sequential logic. The evaluation process involves executing the generated code in Python-embedded HDLs (e.g., PyMTL3, PyRTL) and measuring functional correctness based on pass rates.

2.4 Challenges in LLM Benchmarking for FPGA Design

Despite advancements in LLM-driven Verilog generation, existing models primarily focus on producing syntactically and functionally correct HDL but fail to address critical hardware constraints essential for FPGA deployment, such as resource efficiency, timing constraints, and power consumption. Unlike ASIC design, FPGA-based development demands careful consideration of resource usage, including lookup tables (LUTs), flip-flops (FFs), block RAM (BRAM), and digital signal processing (DSP) blocks. However, current LLMs for Verilog generation lack an understanding of FPGA-specific requirements, often producing designs that are functionally correct but inefficient and impractical for real-world FPGA deployment.

The inability to evaluate and optimize LLM-generated Verilog for FPGA resource constraints highlights the need for advancing resource-aware Verilog generation and motivates this study. To establish LLMs as a practical solution for HDL automation, it is essential to equip them with a deeper understanding of FPGA design constraints. Achieving this requires developing new training datasets, designing robust evaluation frameworks, and refining LLM training strategies to enhance their capability in hardware-aware code generation. This paper focuses on constructing a benchmark that provides an evaluation of LLMs' performance in generating HDLs that are both functionally correct and optimized for FPGA-specific resource constraints.

3 Design of ResBench

This section presents the design of ResBench, outlining its guiding principles and structured problem set for evaluating LLM-generated Verilog code. Additionally, we compare ResBench with existing benchmarks for LLM-generated HDL code.

3.1 Design Principles and Benchmark Problems

ResBench is designed to evaluate LLM-generated Verilog across a diverse range of FPGA applications, with a primary focus on resource optimization awareness. The benchmark consists of 56 problems categorized into 12 domains, each representing a key area of FPGA applications. The problems in ResBench are carefully structured to evaluate both functional correctness and resource efficiency. The benchmark covers a wide range of FPGA design tasks, from fundamental digital logic and mathematical computation to more complex, application-driven domains such as machine learning, cryptography, and financial computing. By spanning these diverse categories, ResBench ensures that LLMs are tested on both low-level design problems and high-level algorithmic implementations for real-world applications.

The design of ResBench is guided by two key principles:

- **The Principle of Resource Usage Differentiation** aims to highlight differences in how LLMs optimize FPGA resource usage. The benchmark includes problems that allow multiple resource-aware optimization strategies and require mathematical transformations. This approach makes it possible to distinguish between models that generate resource-efficient Verilog and those that do not.
- **The Principle of FPGA Application Diversity** recognizes the wide range of FPGA applications, particularly in computational acceleration and edge computing. ResBench spans various domains, including financial computing, climate modeling, and signal processing, allowing LLMs to be tested across a broad set of real-world FPGA workloads.

Table 2 provides an overview of the problem categories along with representative examples. The problems in the benchmark are designed to align with the two guiding principles. Specifically, ResBench addresses the principles as follows:

- ResBench addresses resource usage differentiation by introducing problems that require optimization techniques beyond simple code-level improvements. Fig. 1 illustrates this with a polynomial evaluation problem from the benchmark. Algebraic simplification in this case enables a more efficient Verilog implementation using fewer LUTs. LLMs that fail to apply this optimization generate designs with excessive LUT consumption.
- ResBench addresses FPGA application diversity by covering both foundational and application-driven workloads. Foundational problems include combinational logic, finite state machines, arithmetic operations, pipelining, polynomial evaluations, and mathematical functions. Application-driven problems span machine learning, encryption, and financial computing, emphasizing the role of FPGAs in AI acceleration, security, and high-speed data processing.

By adhering to these principles, the benchmark evaluates not only an LLM's ability to generate syntactically correct Verilog code but also its capability to produce hardware-efficient designs suited for FPGA deployment.

3.2 Comparison with Existing Benchmarks

Table 3 highlights key differences between ResBench and existing HDL benchmarks. The differences are particularly significant in FPGA resource optimization awareness and problem diversity:

- FPGA resource optimization awareness. Most existing benchmarks, such as VerilogEval, HDLEval, and GenBen, focus primarily on functional correctness and HDL syntax quality but do not explicitly account for FPGA resource usage. Consequently, these benchmarks cannot distinguish between functionally correct designs that differ significantly in hardware resource utilization. In contrast, ResBench introduces optimization-aware problems specifically designed to expose variations in resource usage. This enables a more practical comparison of LLMs based on their ability to generate resource-efficient designs for FPGAs.
- Problem diversity. Existing benchmarks primarily focus on fundamental HDL constructs such as basic logic, state machines, and arithmetic operations, with limited diversity in FPGA applications. For example, VerilogEval emphasizes control logic and arithmetic, while HDLEval mainly evaluates digital circuits and state machines. In contrast, ResBench encompasses a significantly broader range of FPGA applications, including machine learning, encryption, financial computing, and physics-based modeling. These domains represent real-world FPGA workloads where resource efficiency is crucial for minimizing device cost and maximizing parallelism. By incorporating a diverse set of tasks, ResBench provides a more comprehensive evaluation of LLM-generated HDL in practical FPGA design scenarios.

4 Evaluation Framework for ResBench

To evaluate LLM-generated designs for FPGA design with ResBench, we implement a structured framework that examines both functional correctness and hardware efficiency.

We build the software for the evaluation framework based on the lessons learned in testing LLM-based software code generation. The benchmarks for LLM-based software share a common evaluation framework aimed at assessing whether generated code is both syntactically valid and functionally correct. In particular, each benchmark generally includes four key components:

(1) Prompts, which can be presented as a natural language description [11, 35] or both description and function signature [6, 20], guiding the model on what to generate.
(2) A reference solution, which serves as the correct implementation for comparison.
(3) Test cases, which are predefined inputs and expected outputs used to validate correctness
(4) Performance metrics, like Pass@k [6] and Code Similarity Scores [29], which estimates how effectively an LLM-generated solution satisfies the given problem constraints. A widely used metric is Pass@k, which measures the likelihood

Table 2: Summary of benchmark categories, including the number of problems and representative examples.

Category	# Problems	Example Problems
Combinational Logic	8	parity_8bit, mux4to1, bin_to_gray
Finite State Machines	4	fsm_3state, traffic_light, elevator_controller
Mathematical Functions	5	int_sqrt, fibonacci, mod_exp
Basic Arithmetic Operations	5	add_8bit, mult_4bit, abs_diff
Bitwise and Logical Operations	4	bitwise_ops, left_shift, rotate_left
Pipelining	5	pipelined_adder, pipelined_multiplier, pipelined_fir
Polynomial Evaluation	5	$(x+2)^2 + (x+2)^2 + (x+2)^2$, $(a+b)^2 - (a-b)^2$
Machine Learning	5	matrix_vector_mult, relu, mse_loss
Financial Computing	4	compound_interest, present_value, currency_converter
Encryption	3	caesar_cipher, modular_add_cipher, feistel_cipher
Physics	4	free_fall_distance, kinetic_energy, wavelength
Climate	4	carbon_footprint, heat_index, air_quality_index
Total	**56**	–

```verilog
module polynomial_5 (
    input signed [7:0] in_0,
    input signed [7:0] in_1,
    output signed [15:0] out
);
    wire signed [8:0] sum;
    wire signed [8:0] diff;
    wire signed [15:0] sum_squared;
    wire signed [15:0] diff_squared;
    assign sum = in_0 + in_1;
    assign diff = in_0 - in_1;
    assign sum_squared = sum * sum;
    assign diff_squared = diff * diff;
    assign out = sum_squared - diff_squared;
endmodule
```

(a) Design Generated by Qwen-2.5 (213 LUTs)

```verilog
module polynomial_5 (
    input signed [7:0] in_0,
    input signed [7:0] in_1,
    output signed [15:0] out
);
    assign out = 4 * in_0 * in_1;
endmodule
```

(b) Design Generated by GPT-4 (0 LUT + 1 DSP)

Figure 1: Benchmark example illustrating HDL optimization capability using the expression $(a+b)^2 - (a-b)^2$. (a) Qwen-2.5 computes the full expression directly, leading to high LUT usage. (b) GPT-4 simplifies the expression to $4ab$, significantly reducing resource usage by using a single DSP unit instead of LUTs. This example demonstrates ResBench's ability to differentiate LLMs based on resource optimization.

Table 3: Comparison of Benchmarks for LLM HDL Evaluation

Benchmark	Year	Hardware	Optimization Awareness	Problem Diversity
VerilogEval [33]	2023	General HDL	No	Logic, FSMs, arithmetic
HDLEval [17]	2024	General HDL	No	Digital circuits, control logic
PyHDL-Eval [4]	2024	General HDL	No	Python-based HDL, small designs
RTLLM [23]	2024	General HDL	No	RTL, bus protocols, DSP
VHDL-Eval [37]	2024	General HDL	No	VHDL logic, sequential circuits
GenBen [38]	2024	General HDL	No	Application-driven tasks
ResBench (This paper)	2025	FPGA	Resource Usage Optimizations	56 problems across 12 domains

that at least one of the top-k generated solutions passes all test cases.

Different from the evaluation of software code generation, our framework uses the resource usage count as a key metric to quantify the quality of resource-oriented optimization. The framework also

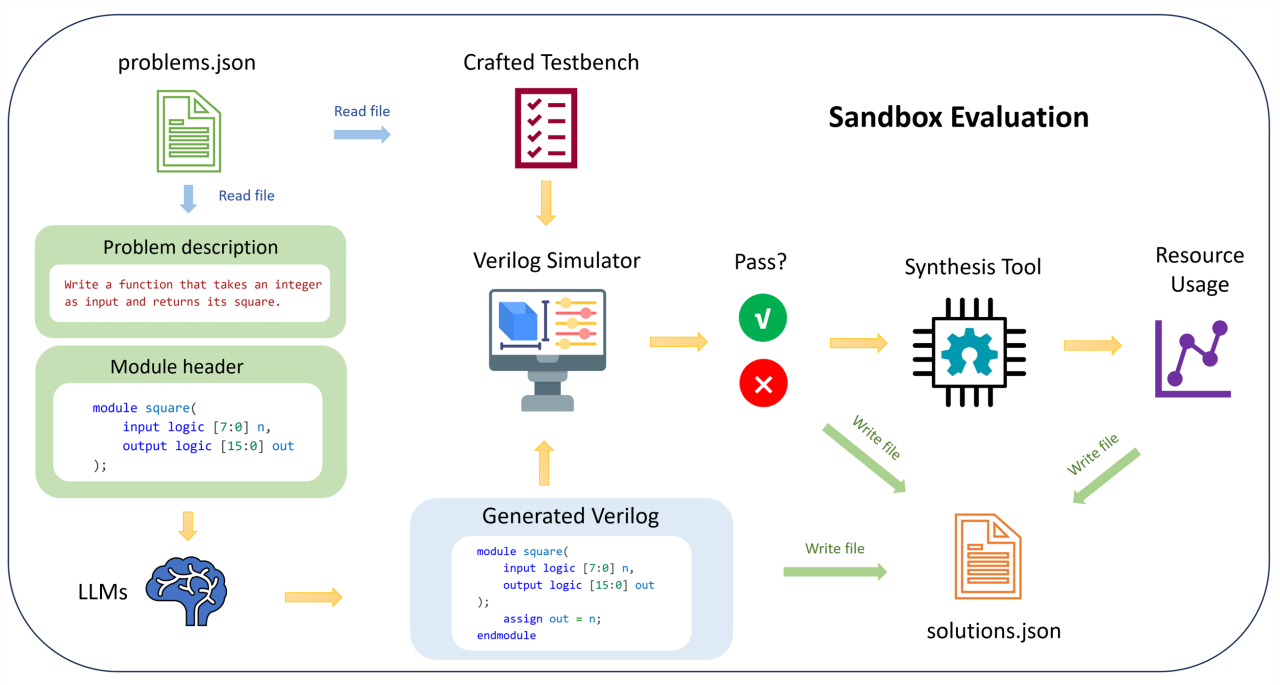

Figure 2: Overview of the software workflow. The process begins with Verilog generation using an LLM, followed by functional verification through testbenches. Functionally correct designs undergo FPGA synthesis to extract resource usage metrics, and the framework compiles performance reports comparing functional correctness and resource usage.

uses an automated benchmarking system that automates Verilog code generation, functional correctness testing, FPGA synthesis, and resource usage extraction.

The open-source software for ResBench automates the evaluation of LLM-generated Verilog, systematically measuring both functional correctness and FPGA resource usage with minimal manual intervention. An overview of the software's workflow is shown in Fig. 2.

The software accepts a user-specified LLM and generates Verilog solutions based on structured problem definitions. It produces detailed evaluation reports, indicating whether each design passes synthesis and functional correctness checks, along with a resource utilization summary. By automating the full evaluation pipeline, the framework facilitates large-scale benchmarking and comparative studies of different LLMs for HDL code generation.

To maintain consistency, each problem follows a structured format consisting of three components: a natural language problem description in plain English, a Verilog module header, and a predefined testbench. The problem description specifies the expected input-output format and functional constraints, ensuring that LLM-generated code aligns with real-world design requirements. The module header provides a consistent Verilog interface with defined input and output signals but leaves the internal logic for the LLM to fill. The testbench validates functional correctness through simulation by applying predefined test cases in the testbench and comparing outputs against a manually verified reference solution.

The evaluation framework follows a structured process to evaluate LLM-generated designs. The evaluation of an LLM on a benchmark problem consists of the following three steps:

(1) The framework queries the selected LLM to generate multiple Verilog code snippets for a given problem. These generated snippets are stored in text format along with references to their corresponding problem descriptions. Functional correctness is then verified using predefined testbenches. Designs that pass all test cases proceed to FPGA synthesis, while those that fail have their errors recorded for further analysis.
(2) FPGA synthesis is performed to determine resource usage metrics such as LUT count, DSP utilization, and register count. For designs that fail synthesis, the resource count is set to ∞, ensuring a consistent comparison framework.
(3) The framework generates structured reports summarizing pass rates, synthesis success rates, and resource usage statistics. Users can visualize model performance through automatically generated comparisons of functional correctness and resource usage.

By following this structured evaluation process, the framework provides a fully automated benchmarking solution that evaluates LLM-generated Verilog across all benchmark problems, focusing on both design correctness and resource usage.

5 Evaluation

This section presents our evaluation of LLM-generated FPGA designs using ResBench. The evaluation focuses not only on functional correctness but also on FPGA resource usage.

5.1 Experimental Setup and Metrics

Our experiments evaluate the capability of LLMs to generate both funtionally correct and resource-efficient Verilog code by examining the number of functionally correct designs. Also, we examine how well different models optimize FPGA resources.

We run the software proposed in Section 4 with all benchmark problems. For each LLM-generated design, we compile the testbench and simulate it to verify the functional correctness of the generated Verilog modules. If the simulation confirms that the design is correct, we use the Vivado synthesis tool to generate a resource report and assess resource usage.

To evaluate functional correctness, we generate the same number of designs from each LLM and count how many designs pass for each problem. In cases where two LLMs produce the same number of passing designs, we break the tie by considering the number of designs that pass synthesis but fail the functional correctness test. We do not use the Pass@k metric, which is commonly applied to LLM-generated software code, because we intend to distinguish designs that fail hardware synthesis from those that successfully synthesize but do not meet functional correctness requirements.

In this study, we quantify the capability of resource optimization by minimizing the LUT count. LUTs serve as the primary logic resource for implementing combinational operations and small memory elements. While FPGAs also provide other resources, these tend to be application-specific. For instance, DSPs and BRAMs are crucial for arithmetic-intensive and memory-heavy designs but are not universally required across all FPGA applications. In contrast, LUTs are a fundamental component in nearly every design, making them a consistent and reliable metric for evaluating different HDL implementations.

Each LLM-generated design d_i is evaluated based on its LUT usage. If a design successfully passes both synthesis and functional correctness testing, its LUT count is recorded as $\mathrm{LUT}(d_i)$. Otherwise, it is assigned ∞ to indicate that the design is either non-synthesizable or functionally incorrect:

$$\mathrm{LUT}(d_i) = \begin{cases} \text{LUT count}, & \text{if } d_i \text{ is synthesizable and correct} \\ \infty, & \text{otherwise} \end{cases} \tag{1}$$

$$\mathrm{LUT}_{\min} = \min\left(\mathrm{LUT}(d_0), \mathrm{LUT}(d_1), \ldots, \mathrm{LUT}(d_{n-1})\right) \tag{2}$$

By using ∞ for failed designs, our approach naturally excludes non-functional implementations. This setting maintains computational consistency and eliminates the need for explicit filtering in the evaluation of Equation 2. Note that while this study focuses on minimizing LUT usage, our framework is capable of extracting and analyzing other resource metrics with a different optimization objective.

For correctness testing and resource usage evaluation, we use Vivado 2023.1 for simulation, synthesis, and analysis. The hardware implementation is targeted at the programmable logic section of the AMD Zynq 7000 XC7Z020CLG400-1 SoC, operating at its default clock frequency. While design correctness remains independent of the chosen tool, the LUT count is influenced by Vivado's synthesis capabilities and the type of LUTs on the target device. However, we expect the impact of FPGA software choice on relative resource efficiency to be small. In particular, for a given problem, the HDL designs with the smallest LUT count will likely stay unchanged even when evaluated with different FPGA tools.

For all the evaluated LLMs, we set the temperature parameter to 1.5 to encourage high diversity of the HDL code for each problem. The evaluation includes three types of models: general-purpose LLMs, code-specialized LLMs, and HDL-specialized LLMs. The general-purpose models we evaluate include GPT-3.5 [44], GPT-4o [13], GPT-4 [2], GPT-o1-mini [14], Llama3.1-450B [34], Qwen-Max [3], and Qwen-Plus [42]. The evaluated code-specialized models include Qwen2.5-Coder-32B-Instruct [12] and Codestral [15]. We also evaluate VeriGen [33], an HDL-specialized model. However, during the evaluation, VeriGen failed to generate legitimate Verilog code for all problems. As a result, we omit its results from further discussion.

5.2 Functional Correctness

Table 4 provides detailed pass counts across 12 categories of problems, with 15 designs generated for each problem. Each table cell follows the format: **pass / synthesis OK but incorrect design / synthesis error**. For example, if a category contains 5 problems, each LLM generates a total of 75 solutions (5 problems × 15 generated designs per problem), and the sum of the three numbers in each cell corresponds to this total. In this table, we also include the number of wins, which represents the number of categories in which each LLM achieved the highest pass count.

The results show that GPT-o1-mini is the leading model, achieving the highest pass counts in most categories. This suggests that reasoning-optimized models have an advantage in Verilog code generation. This is potentially because its reasoning capabilities contribute to more accurate outputs.

Table 4 shows a notable observation in finite state machines, mathematical functions, pipelining. The generated code can often pass synthesis but fail to function correctly. This observation suggests that while LLMs grasp basic syntax, they struggle with complex functional logic. In contrast, for more intricate problems with complex contexts, such as mathematical functions and financial computing, LLMs tend to produce syntactically incorrect code, reflecting challenges in understanding and reasoning within these contexts.

The results show that for every problem there is at least one model providing correct solutions. However, in categories such as pipelining, financial computing, and encryption, LLMs tend to underperform and show higher variability. For example, GPT-3.5 produced no passing solutions in pipelining, but LLaMA 3.1 achieved good results. A similar pattern is observed in the mathematical functions category. These results highlight the importance of evaluating both functional correctness and resource optimization in complex design scenarios.

Table 4: Design correctness of LLMs in generating Verilog code across different categories

Cell format: pass / synthesis OK but incorrect design / synthesis error

	GPT-3.5 turbo	GPT-4	GPT-4o	GPT-o1 mini	Llama3.1 405B	Qwen-max	Qwen-plus	Qwen2.5-coder 32B	Codestral
Combinational Logic	112 / 5 / 3	117 / 3 / 0	**120 / 0 / 0**	118 / 1 / 1	115 / 2 / 3	117 / 2 / 1	109 / 1 / 10	112 / 2 / 6	**120 / 0 / 0**
Finite State Machines	23 / 15 / 22	32 / 22 / 6	31 / 24 / 5	**39 / 18 / 3**	31 / 24 / 5	34 / 26 / 0	27 / 23 / 10	39 / 10 / 11	36 / 6 / 18
Mathematical Functions	13 / 19 / 43	6 / 39 / 30	36 / 10 / 29	**46 / 24 / 5**	7 / 6 / 62	26 / 27 / 22	20 / 26 / 29	5 / 8 / 62	0 / 3 / 72
Basic Arithmetic Ops	37 / 2 / 36	63 / 8 / 4	66 / 9 / 0	**68 / 4 / 3**	43 / 2 / 30	38 / 22 / 15	27 / 13 / 35	54 / 6 / 15	62 / 13 / 0
Bitwise & Logic Ops	35 / 0 / 25	55 / 0 / 5	58 / 2 / 0	**59 / 0 / 1**	52 / 0 / 8	47 / 0 / 13	33 / 11 / 16	36 / 0 / 24	55 / 0 / 5
Pipelining	0 / 59 / 16	11 / 54 / 10	26 / 49 / 0	15 / 38 / 22	7 / 38 / 30	15 / 32 / 28	16 / 26 / 33	**21 / 31 / 23**	6 / 56 / 13
Polynomial Evaluation	19 / 3 / 53	69 / 0 / 6	**74 / 1 / 0**	68 / 5 / 2	58 / 6 / 11	55 / 2 / 18	28 / 5 / 42	65 / 7 / 3	69 / 6 / 0
Machine Learning	31 / 3 / 41	60 / 8 / 7	60 / 13 / 2	**73 / 1 / 1**	45 / 28 / 2	63 / 12 / 0	61 / 12 / 2	57 / 2 / 16	64 / 8 / 3
Financial Computing	9 / 23 / 28	21 / 22 / 17	**29 / 13 / 18**	20 / 20 / 20	11 / 21 / 28	28 / 15 / 17	15 / 12 / 33	16 / 7 / 37	17 / 23 / 20
Encryption	30 / 0 / 15	**30 / 2 / 13**	25 / 20 / 0	30 / 0 / 15	26 / 0 / 19	25 / 9 / 11	30 / 1 / 14	30 / 0 / 15	30 / 0 / 15
Physics	45 / 3 / 12	**57 / 0 / 3**	53 / 4 / 3	54 / 5 / 1	41 / 11 / 8	49 / 7 / 4	40 / 17 / 3	38 / 15 / 7	55 / 2 / 3
Climate	8 / 15 / 37	21 / 30 / 9	**41 / 11 / 8**	41 / 15 / 4	24 / 23 / 13	38 / 19 / 3	19 / 31 / 10	32 / 14 / 14	28 / 19 / 13
Number of wins	**0**	**2**	**4**	**5**	**0**	**0**	**0**	**1**	**1**

5.3 Resource Usage

Table 5 presents the LUT usage results for the benchmark problems. In this table, problems where all LLMs yield identical LUT usage are excluded for brevity. The cell with the lowest resource usage in each category is highlighted in bold. The number of wins is determined by counting these highlighted cells.

The results suggest that different LLMs have varying levels of optimization capability within our benchmark framework. This highlights how the benchmark problems reveal differences in LLMs' ability to optimize resource usage. Moreover, we have the following observations based on the results:

(1) GPT-o1-mini leads with 19 wins, significantly outperforming the runner-up, GPT-4, which achieves 12 wins. This indicates that GPT-o1-mini generates Verilog designs with lower resource usage in most problems. Its strong performance suggests that advanced reasoning capabilities may enhance its understanding of problem requirements, the Verilog language, and the complexities of hardware design. In contrast, code-specialized LLMs, while demonstrating high accuracy in producing functionally correct Verilog, may lack the reasoning depth needed to optimize designs effectively for FPGA constraints. This difference highlights that generating syntactically correct HDL alone is insufficient for producing resource-efficient hardware, as true optimization demands a deeper understanding of both design requirements and FPGA-specific constraints.

(2) GPT-3.5-Turbo, Qwen2.5-Coder, and Codestral demonstrate the weakest resource optimization ability, achieving only 7, 7, and 8 wins, respectively. The poor resource optimization of GPT-3.5-turbo is potentially due to its model size and lack of updates. Qwen2.5-Coder and Codestral, the two code-specialized models in our evaluation, also struggle with resource optimization. One possible explanation is that these models are primarily trained on software code rather than HDL, which may limit their ability to account for FPGA resource constraints when generating and optimizing Verilog. Additionally, key optimization techniques, such as mathematical simplifications, are unlikely to be picked up effectively from software code data.

(3) The observed variations in resource usage across different problem types confirm that our benchmarks can lead to divergent hardware resource usage for Verilog designs generated by different LLMs. For simple tasks such as combinational logic and basic arithmetic operations, the differences tend to be less significant. This is likely because the training data of the models include well-established reference solution for these problems. However, for more complex problems, our benchmark problems lead to significantly greater divergence in LUT usage. This suggests that our benchmark problems effectively evaluate the ability of LLMs to optimize resource usage beyond learned patterns.

Considering both functional correctness and resource usage, we find that GPT-o1-mini achieves the highest performance in both aspects, while code-specialized models, including Qwen2.5-Coder and Codestral, perform the worst.

6 Conclusion and Future Work

LLMs provide a promising solution for automating HDL generation. However, most current benchmarks focus mainly on functional correctness while overlooking FPGA resource constraints. This lack of attention on FPGA resource efficiency underscores the need for resource-aware benchmarks to better evaluate LLM-generated HDL for real-world FPGA deployment. Additionally, current benchmarks lack problem diversity, limiting their effectiveness in evaluating real-world FPGA applications. To address these limitations, we introduce ResBench, the first resource-centric benchmark for LLM-generated HDL. ResBench features 56 problems spanning 12 categories. The benchmark problems are designed to expose LLMs'

Table 5: LUT_{min} for each LLM across categories

	GPT-3.5 turbo	GPT-4	GPT-4o	GPT-o1 mini	Llama3.1 405B	Qwen-max	Qwen-plus	Qwen2.5-coder 32B	Codestral
fsm 3state	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
traffic light	1	1	2	**0**	**0**	2	3	2	∞
elevator controller	3	3	**2**	**2**	**2**	**2**	**2**	**2**	**2**
vending machine	**1**	**1**	2	**1**	2	**1**	**1**	2	**1**
int sqrt	∞	∞	68	177	∞	**64**	229	173	∞
fibonacci	∞	56	**1**	56	56	56	∞	∞	∞
mod exp	∞	∞	4466	4669	∞	1911	**1678**	∞	∞
power	∞	**79**	93	93	∞	93	93	93	∞
log2 int	∞	∞	∞	**10**	20	∞	∞	12	∞
abs diff	**12**	**12**	14	**12**	**12**	∞	**12**	**12**	**12**
modulo op	**82**	**82**	**82**	**82**	111	∞	∞	∞	∞
left shift	**10**	**10**	**10**	**10**	**10**	12	12	**10**	**10**
pipelined adder	∞	**0**	16	∞	**0**	∞	**0**	15	∞
pipelined multiplier	∞	∞	77	70	**56**	∞	70	∞	∞
pipelined max finder	∞	**0**	24	**0**	24	24	24	24	24
$x^3 + 3x^2 + 3x + 1$	49	49	**0**	91	**0**	91	**0**	91	49
$(x+2)^2 + (x+2)^2 + (x+2)^2$	64	33	96	**11**	108	108	26	18	33
$(a+b)^2 - (a-b)^2$	∞	**0**	213	59	16	213	16	16	16
relu	**8**	**8**	**8**	**8**	**8**	16	**8**	**8**	16
mse loss	∞	216	**64**	**64**	216	**64**	216	**64**	**64**
compound interest	∞	13060	10135	10135	52950	**9247**	∞	10135	52950
currency converter	∞	∞	**0**	**0**	25	**0**	∞	∞	∞
free fall distance	**6**	**6**	64	**6**	**6**	64	67	64	**6**
kinetic energy	70	70	**54**	**54**	**54**	**54**	**54**	**54**	**54**
potential energy	6	6	84	**0**	6	6	6	6	6
carbon footprint	174	121	110	**92**	121	121	110	110	110
heat index	**16**	**16**	201	**16**	195	**16**	124	201	201
air quality index	∞	∞	128	**104**	∞	**104**	116	128	128
Number of wins	**7**	**12**	**10**	**19**	**11**	**10**	**9**	**7**	**8**

ability to generate Verilog designs optimized for FPGA resource usage.

While ResBench is not explicitly designed to emphasize combinational logic and arithmetic operations, the current problem set naturally includes a high proportion of such designs. Future work will expand the benchmark to include more sequential designs, such as pipelined architectures and state-driven circuits. Additionally, although the current evaluation focuses on Verilog, our framework is designed to support multiple HDLs. Future efforts will extend support to VHDL and high-level synthesis (HLS) tools.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and suggestions. The support of the United Kingdom EPSRC (grant number UKRI256, EP/V028251/1, EP/N031768/1, EP/S030069/1, and EP/X036006/1), Intel, and AMD is gratefully acknowledged.

AI Usage Statement: This work involves the use of generative AI in multiple aspects. The methodology presented in this paper focuses on evaluating the ability of AI models to generate HDL code. As such, all experimental results are based on Verilog designs produced by LLMs. For writing, ChatGPT-4 and Llama 3.1 were used to refine phrasing, improve clarity, and proofread the text.

References

[1] Manar Abdelatty, Jingxiao Ma, and Sherief Reda. 2024. MetRex: A Benchmark for Verilog Code Metric Reasoning Using LLMs.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 Technical Report.

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen Technical Report.

[4] Christopher Batten, Nathaniel Pinckney, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2024. PyHDL-Eval: An LLM Evaluation Framework for Hardware Design Using Python-Embedded DSLs. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. IEEE, 1–17.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[6] 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. 2021. Evaluating Large Language Models Trained on Code.

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[8] Jacob Devlin. 2018. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding.

[9] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A Generative Model for Code Infilling and Synthesis.

[10] Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. 2024. AutoVCoder: A Systematic Framework for Automated Verilog Code Generation using LLMs.

[11] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring Mathematical Problem Solving with the Math Dataset.

[12] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2.5-Coder Technical Report.

[13] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. GPT-4o System Card.

[14] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. OpenAI O1 System Card.

[15] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B.

[16] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515 [cs.CL] https://arxiv.org/abs/2406.00515

[17] Farzaneh Rabiei Kashanaki, Mark Zakharov, and Jose Renau. 2024. HDLEval Benchmarking LLMs for Multiple HDLs. In *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, IEEE, 1–5.

[18] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In *International Conference on Machine Learning*. PMLR, 18319–18345.

[19] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: May the Source Be with You!

[20] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *Advances in Neural Information Processing Systems* 36 (2024).

[21] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating Large Language Models for Verilog Code Generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.

[22] Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 2024. OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided Design RTL Generation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[23] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Models. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 722–727.

[24] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction Tuning Code Large Language Models.

[25] Bardia Nadimi and Hao Zheng. 2024. A Multi-Expert Large Language Model Architecture for Verilog Code Generation.

[26] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis.

[27] Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. 2024. Autobench: Automatic Testbench Generation and Evaluation Using LLMs for HDL Design. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–10.

[28] Alec Radford. 2018. Improving Language Understanding by Generative Pre-Training.

[29] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: A Method for Automatic Evaluation of Code Synthesis.

[30] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code Llama: Open Foundation Models for Code.

[31] Alvin Tan. 2017. HDLBits: Digital Circuits Exercises. https://hdlbits.01xz.net/wiki/Problem_sets Accessed: 2025.

[32] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[33] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 1–31.

[34] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LLaMA: Open and Efficient Foundation Language Models.

[35] Mansi Uniyal, Mukul Singh, Gust Verbruggen, Sumit Gulwani, and Vu Le. 2024. One-to-Many Testing for Code Generation from (Just) Natural Language. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 15397–15402.

[36] Ashish Vaswani. 2017. Attention is All You Need. *Advances in Neural Information Processing Systems* (2017).

[37] Prashanth Vijayaraghavan, Luyao Shi, Stefano Ambrogio, Charles Mackin, Apoorva Nitsure, David Beymer, and Ehsan Degan. 2024. VHDL-Eval: A Framework for Evaluating Large Language Models in VHDL Code Generation.

[38] Gwok-Waa Wan, Wang yubo, SamZaak Wong, jingyi zhang, Mengnv Xing, Zhe jiang, Nan Guan, ying wang, Ning Xu, Qiang Xu, and Xi Wang. 2025. GenBen:A Genarative Benchmark for LLM-Aided Design. https://openreview.net/forum?id=gtVo4xcpFI

[39] Jianxun Wang and Yixiang Chen. 2023. A Review on Code Generation with LLMs: Application and Evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 284–289.

[40] Ning Wang, Bingkun Yao, Jie Zhou, Xi Wang, Zhe Jiang, and Nan Guan. 2024. Large Language Model for Verilog Generation with Golden Code Feedback.

[41] Sam-Zaak Wong, Gwok-Waa Wan, Dongping Liu, and Xi Wang. 2024. VGV: Verilog Generation Using Visual Capabilities of Multi-Modal Large Language Models. In *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 1–5.

[42] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2.5 Technical Report.

[43] Yiyao Yang, Fu Teng, Pengju Liu, Mengnan Qi, Chenyang Lv, Ji Li, Xuhong Zhang, and Zhezhi He. 2025. HaVen: Hallucination-Mitigated LLM for Verilog Code Generation Aligned with HDL Engineers.

[44] Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, et al. 2023. A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models.

[45] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends.