CHIPBENCH: A NEXT-STEP BENCHMARK FOR EVALUATING LLM PERFORMANCE IN AI-AIDED CHIP DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

While Large Language Models (LLMs) show significant potential in hardware engineering, current benchmarks suffer from saturation and limited task diversity, failing to reflect LLMs' performance in real industrial workflows. To address this gap, we propose a comprehensive benchmark for AI-aided chip design that rigorously evaluates LLMs across three critical tasks: Verilog generation, debugging, and reference model generation. Our benchmark features 44 realistic modules with complex hierarchical structures, 89 systematic debugging cases, and 132 reference model samples across Python, SystemC, and CXXRTL. Evaluation results reveal substantial performance gaps, with state-of-the-art Claude-4.5-opus achieving only 30.74% on Verilog generation and 13.33% on Python reference model generation, demonstrating significant challenges compared to existing saturated benchmarks where SOTA models achieve over 95% pass rates. Additionally, to help enhance LLM reference model generation, we provide an automated toolbox for high-quality training data generation, facilitating future research in this underexplored domain. Our code is available at https://anonymous.4open.science/r/ChipBench-8766.

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated remarkable capabilities in conversational AI and code generation (Cai et al., 2024; Du et al., 2024; Gu et al., 2025; Li et al., 2025; Liu et al., 2023a; Nadimi & Zheng, 2024; Ross et al., 2023; Thakur et al., 2024; Wang et al., 2023; Zhong & Wang, 2024). The integration of agentic techniques has further amplified these abilities, with systems like Claude and Cursor excelling in high-level languages such as Python and C++ (Wang et al., 2025; Guo et al., 2025; Ishibashi & Nishimura, 2024; Islam et al., 2024; Dong et al., 2025). Recently, this success has extended to the semiconductor domain, where LLMs show promise in chip design (Gai et al., 2025; Abdollahi et al., 2024; Swaroopa et al., 2024; Hu et al., 2024; He et al., 2025; Firouzi et al., 2025; Ho et al., 2025; Yu et al., 2025; Thakur et al., 2024).

To evaluate LLM capabilities in chip design, benchmarks like VerilogEval (Liu et al., 2023b; Pinckney et al., 2025a) and RTLLM (Lu et al., 2024; Liu et al., 2024) have been proposed. However, these benchmarks have become inadequate due to rapid LLM advancement. State-of-the-art LLM-based Verilog generators, such as MAGE (Zhao et al., 2025b), have already achieved over 95% accuracy on these benchmarks, indicating saturation and highlighting the need for more challenging and comprehensive evaluations. We identify three critical limitations of existing benchmarks that hinder accurate assessment of LLM capabilities for industrial deployment:

1. Overly simplified benchmarks overestimate LLMs' capabilities for industrial deployment. Existing Verilog benchmarks contain only simple functional modules ranging from 10 to 76 lines, with all modules being self-contained and never instantiating sub-modules. This contrasts sharply with industrial reality, where Verilog modules typically exceed 10,000 lines with complex functionality and employ hierarchical designs that instantiate and interact with multiple sub-modules for code reuse and systematic verification (Keutzer et al., 2000; Jin et al., 2025). The gap is further widened by the source of these modules: most samples in existing benchmarks are derived from coding contests rather than real industrial projects, resulting in significant discrepancies in both functionality and coding style compared to production code. Consequently, even if LLMs excel in existing benchmarks,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

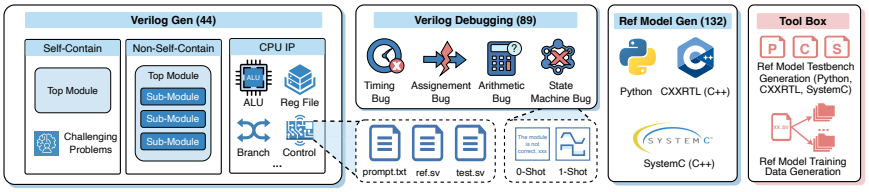


Figure 1: Overview of ChipBench.

they may still perform poorly in industrial-level deployment due to these substantial gaps between benchmark scenarios and production requirements.

2. The exclusive focus on Verilog code generation overlooks LLMs' debugging capabilities, which are even more critical for industrial development. The semiconductor industry's extreme caution stems from the fact that design bugs can result in billions of dollars in losses (Price, 1995), making companies reluctant to adopt LLMs for complete code generation due to their inability to produce consistently bug-free code. Given these constraints, LLM-assisted debugging represents a more practical and immediately deployable alternative that integrates seamlessly into existing workflows without disrupting established safety protocols. Despite debugging being more urgent and realistic than full generation, existing benchmarks focus exclusively on code generation, creating a critical gap in evaluating LLMs' practical utility for industrial integration.

3. The narrow focus on Verilog generation neglects reference model evaluation, which is equally vital for industrial chip design. Reference models, typically written in high-level languages (e.g., SystemC (Panda, 2001), Python, and CXXRTL (Wolf et al., 2013)), serve as predictors of hardware behavior and enable cross-comparison to eliminate potential bugs (Jiang et al., 2020; Zhao et al., 2025a; Hung & Narasimhan, 2004). In industrial workflows, reference model generation is even more resource-intensive than Verilog design, reflected in a 1:1 - 5:1 ratio of verification engineers (write reference model) to design engineers (write Verilog) (Foster et al., 2022). Despite this importance, existing benchmarks completely ignore this task, leading researchers to overlook this critical area and limiting LLMs' potential to enhance the chip design industry.

To address these gaps, we propose ChipBench, a comprehensive benchmark designed to bridge the gap between academic datasets and industrial complexity. By expanding evaluation beyond simple generation to include debugging and reference modeling, ChipBench provides a rigorous assessment of LLM capabilities. We summarize our contributions as follows:

- **A Challenging and Realistic Verilog Generation Benchmark.** We compile 44 modules from open-source CPU IPs and competitive platforms, categorized into self-contained, non-self-contained, and real-world CPU submodules. As shown in Table 1, our dataset features 3.8× longer code length and 13.9× more cells than VerilogEval. Test results show that the most powerful multi-agent system MAGE achieves only 37% pass rate, significantly lower than its 95% pass rate on VerilogEval, demonstrating our benchmark's potential as a robust milestone for future research.
- **A Systematic Verilog Debugging Benchmark.** We introduce a debugging suite with 89 test cases covering four error types: timing, arithmetic, assignment, and state machine bugs. By manually injecting these errors, we create a controlled environment to evaluate LLM repair capabilities. Test results show that LLMs generally demonstrate 5%-20% higher pass rates on debugging than code generation, demonstrating their potential to be integrated into industrial workflows.
- **A Heterogeneous Reference Model Generation Benchmark.** We introduce 132 samples (44×3), enabling evaluation of reference model generation across Python, SystemC, and CXXRTL. Our self-developed framework supports systematic comparison between reference models and golden Verilog implementations, and is extensible to any existing Verilog codebase. Evaluation reveals significant capability gaps, with top-tier Claude-4.5-opus (Anthropic, 2025b) achieving only 13.33% accuracy in Python, demonstrating substantial performance decline compared to other Python tasks and highlighting LLMs' unexplored potential in this critical area.

Table 1: Comparison with previous benchmarks.

Benchmarks	Verilog Gen				V-Debug	Ref-Gen
	Code	Cells	Subs	Tests	Tests	Tests
Thakur et al.	16.6	7.9	0	17	✗	✗
RTLLMV2	46.1	33.7	0.2	50	✗	✗
VerilogEvalV2	16.1	31.4	0	156	✗	✗
Self-Contn (ours)	47.8	323.3	0	29	✓	✓
Not-Self-Contn (ours)	119.5	361.2	1.8	6	✗	✓
Cpu-Design (ours)	68.1	862.3	0	9	✓	✓
ChipBench (ours)	**61.7**	**438.7**	**0.25**	**44**	**89**	**132+**
Total Test Cases				**264+**		

- **Automated Toolbox for Reference Model Verification and Training Dataset Generation.** To facilitate future research on reference model generation, we release an automated toolbox alongside ChipBench. This tool automates the verification of LLM-generated reference models and facilitates the generation of high-quality training datasets from existing Verilog datasets. We tested our toolbox on a subset of QiMeng CodeV-R1 training dataset (Zhu et al., 2025) and have successfully generated 2206 high-quality Python reference models from 10,000 Verilog training data.

2 CHIPBENCH

In this section, we will introduce the components and workflow of ChipBench in detail.

2.1 OVERVIEW

Our proposed ChipBench extends beyond a simple Verilog coding testbench to comprehensively evaluate LLMs across multiple aspects of the chip design workflow. Through this benchmark, we emphasize the importance of debugging and reference model generation, capabilities that could elevate AI-aided chip design to new heights. As illustrated in Figure 1 and Table 1, ChipBench encompasses three core tasks spanning over 264 test cases: 44 structurally complex and challenging Verilog generation cases, 89 Verilog debugging cases, and 132 reference model generation cases distributed across Python, SystemC, and CXXRTL (extension to infinite cases with our proposed toolbox). We summarized details of our modules in Table 7 in the Appendix.

In addition to the comprehensive test suite, we provide an integrated toolbox for reference model verification and automated training dataset generation. Given the current shortage of high-quality reference model training data, this toolbox serves as a valuable resource for the research community. To validate its effectiveness, we deployed the toolbox with DeepSeek V3.2 and successfully generated over 2,200 high-quality Python reference models from 10,000 Verilog training cases (Zhu et al., 2025). Given the abundance of existing Verilog training datasets, including QiMeng CodeV (87k samples; Zhu et al., 2025), Pyranet (692k samples; Nadimi et al., 2025), and VeriGen (50k samples; Thakur et al., 2024), our toolbox has the potential to generate extensive reference model datasets at scale, thereby significantly benefiting the research community.

2.2 VERILOG GENERATION TASKS

We developed a dataset of high-quality golden Verilog code sourced from open-source Online Judge (OJ) platforms and established CPU IP projects. Each test case was carefully selected to ensure appropriate difficulty levels that are neither trivial nor overly challenging. For each test case, we manually wrote description prompts rather than using LLM-generated ones to ensure accurate functional descriptions, consistent with recommendations from VerilogEval V2 (Pinckney et al., 2025a).

To conduct a comprehensive evaluation of LLM coding capabilities, we categorize the test cases into three distinct categories. The first category consists of self-contained module tests, where modules operate independently without instantiating external sub-modules. While this format aligns with existing benchmarks like VerilogEval, as shown in Table 1, ChipBench significantly elevates the

complexity, with samples averaging 47.8 lines of code and 323.3 cells compared to approximately 16.1 lines and 31.4 cells in VerilogEval V2, presenting a more rigorous challenge to the model's logic synthesis capabilities.

The second category covers hierarchical modules that are non-self-contained. These cases require the top-level module to instantiate and interact with sub-modules, reflecting real-world industrial scenarios where coding standards prioritize modularity and reuse to minimize verification overhead. To assist LLMs in understanding interface requirements, we include both the description and source code of sub-modules within the prompt. We only require LLMs to complete the top module, since all models failed when tasked with generating complete hierarchical designs, making such tests prohibitively challenging.

The third category involves CPU IP modules sourced directly from open-source CPU projects to reflect the demands of high-performance computing design. We manually selected representative functional units, including ALUs, controllers, branch predictors, etc, with moderate complexity from larger project contexts. These test cases assess the LLM's ability to generate functional Verilog code that meets professional engineering standards and real-world deployment requirements.

For every test case in ChipBench, we provide a golden Verilog implementation, a manually developed test file, and a high-quality prompt written by domain experts. The test files utilize a robust combination of directed tests and constrained random testing (over 1,000 iterations) to ensure high-fidelity evaluation.

2.3 Verilog Debugging Tasks

To evaluate LLMs' error correction capabilities, we constructed debugging tasks by manually injecting faults into golden Verilog modules. We identified four categories of bugs most prevalent in hardware engineering: arithmetic bugs, assignment bugs, timing bugs, and state machine bugs. As shown in Figure 6 in the Appendix, arithmetic bugs involve errors in operator usage, such as substituting multiplication operators for addition operators. Assignment bugs refer to incorrect constant value assignments to wires or registers. Timing bugs occur specifically in sequential logic, manifesting as values assigned in incorrect clock cycles or misuse of blocking versus non-blocking assignments. State machine bugs represent logic errors within state transition mechanisms, where the transition logic itself is flawed.

To simulate realistic debugging workflows, we designed two distinct testing modes: zero-shot and one-shot evaluation. The zero-shot test provides the model with the module description and buggy implementation, indicating that an error exists without providing localization details. The one-shot test provides identical information but supplements it with simulation waveform data (.vcd files). This replicates industrial workflows where design and verification engineers analyze simulation waveforms to isolate and diagnose faults.

2.4 Framework Workflow

We demonstrate the workflow for the Verilog generation task and the reference model generation task in Figure 2(a)(b).

Verilog Generation. Our evaluation framework is built upon VerilogEval, ensuring full functional compatibility. As shown in Figure 2(a), each test case comprises three files: a `prompt.txt` file describing the task, a `test.sv` harness for evaluation, and a `golden.sv` serving as the ground truth. Upon generating a solution (`generated.sv`), the system first performs a syntax check to ensure compilability. Subsequently, we execute `test.sv` using iVerilog to compare the outputs of the generated code against the golden reference. To ensure reliability, the test harness employs a combination of manually designed corner cases and over 1,000 randomly generated stimuli.

Reference Model Generation. Our framework supports verification of LLM-generated reference models through a custom Heterogeneous Test Engine (HTE). As illustrated in Figure 2(b), this workflow requires a module description file `prompt.txt` and a golden implementation file `golden.sv` as input and supports multiple reference model languages, including Python and C++ (via SystemC or CXXRTL extensions). Once the LLM generates a reference model, HTE performs a syntax check followed by functional verification. To validate the reference model, we extract I/O signals

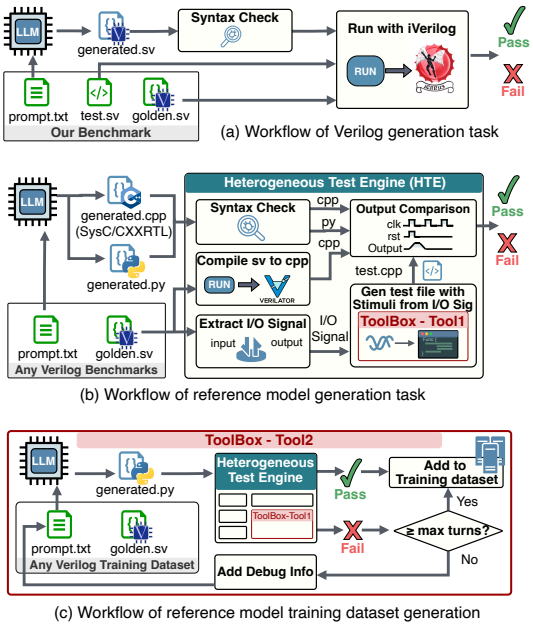


Figure 2: Workflow of ChipBench across different tasks: (a) evaluating LLM-generated Verilog code, (b) assessing LLM-generated reference models (supporting Python, SystemC, and CXXRTL), and (c) utilizing the toolbox to generate high-quality training datasets for reference models (supporting Python, SystemC, and CXXRTL, with Python shown as an example)

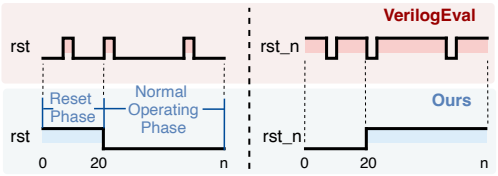


Figure 3: Comparison of reset signal patterns between our approach and VerilogEval. We toggle reset once and only compare outputs during the normal operating phase to better reflect real-world verification standards and eliminate false negatives.

from `golden.sv` and generate a heterogeneous test harness (`test.cpp`). Simultaneously, the `golden.sv` is compiled into C++ using Verilator (Snyder & contributors, 2025). The test harness then compares the outputs of the LLM-generated reference model against the transferred golden model using over 1,000 random stimuli. This automated flow is theoretically extensible to any testbench, as long as a prompt and a golden Verilog implementation are provided.

2.5 TOOLBOX

ChipBench includes a specialized Toolbox to facilitate future research in reference model generation, comprising two primary tools shown in Figure 2(b) and (c).

Tool 1: Automatic Testbench Generation. This tool automates the creation of test files of LLM-generated references based on the I/O signals extracted from the golden Verilog. It injects random input stimuli to robustly compare the golden implementation with the generated reference model.

For input signals, we carefully designed reset signal patterns to align with industrial verification standards. As shown in Figure 3, unlike VerilogEval test cases that toggle reset signals frequently, our tool identifies the reset polarity (active-high `rst` or active-low `rst_n`) and asserts system reset for the initial 20 cycles before transitioning to continuous operation. To prevent spurious failures, we mask output comparisons during the reset phase and only compare outputs during normal operation. This design addresses the fundamental discrepancy between Verilog, which may output 'X' (unknown) states before initialization, and high-level languages like Python/C++, which lack equivalent unknown states. This approach eliminates false negatives and better reflects real-world verification practices.

Tool 2: Automatic Training Dataset Generation. To address the shortage of training data for reference model generation, we extended the HTE to create an automated training dataset generation pipeline. As shown in Figure 2(c), this tool is compatible with any Verilog training dataset containing prompts (`prompt.txt`) and golden implementations (`golden.sv`).

Table 2: Test results for Verilog generation (red: low pass rate, green: high).

Model/Dataset	CPU IP			Non-Self-Contained			Self-Contained			Average		
	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
Claude 4.5 Haiku	11.11	11.11	11.11	16.67	33.33	33.33	30	36.67	36.67	19.26	27.04	27.04
Claude 4.5 Opus	22.22	22.22	22.22	33.33	50	50	36.67	43.33	50	30.74	38.52	40.74
Claude 4.5 Sonnet	11.11	11.11	11.11	16.67	33.33	33.33	10	20	26.67	12.59	21.48	23.70
DeepSeek-Coder	11.11	11.11	11.11	50	50	50	26.67	36.67	43.33	29.26	32.59	34.81
DeepSeek-R1	11.11	11.11	11.11	33.33	50	50	26.67	40	53.33	23.70	33.70	38.15
Gemini 2.5 Pro	0	11.11	11.11	0	16.67	16.67	3.33	10	13.33	1.11	12.59	13.70
Gemini 3 Flash	22.22	22.22	22.22	33.33	50	50	36.67	46.67	46.67	30.74	39.63	39.63
GPT-3.5 Turbo	11.11	11.11	11.11	0	0	0	13.33	26.67	26.67	8.15	12.59	12.59
GPT-4o	11.11	11.11	11.11	0	0	0	20	33.33	46.67	10.37	14.81	19.26
GPT-5	11.11	11.11	22.22	33.33	33.33	33.33	30	50	53.33	24.81	31.48	36.29
GPT-5.2	22.22	22.22	22.22	33.33	33.33	50	23.33	40	40	26.29	31.85	37.41
Llama 3.1 8B	11.11	11.11	11.11	0	0	0	10	13.33	16.67	7.04	8.15	9.26
Llama 3.3 70B	11.11	11.11	11.11	0	0	0	10	26.67	30	7.04	12.59	13.70
MAGE (DeepSeek-V3)	22.22	-	-	50	-	-	40	-	-	37.41	-	-

Table 3: Test results for Python reference model generation (red: low pass rate, green: high).

Model/Dataset	CPU IP			Non-Self-Contained			Self-Contained			Average		
	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
Claude 4.5 Haiku	0	0	0	0	0	0	40	40	43.33	13.33	13.33	14.44
Claude 4.5 Opus	0	0	0	0	0	0	40	46.67	46.67	13.33	15.56	15.56
Claude 4.5 Sonnet	11.11	11.11	11.11	0	0	11.11	36.67	36.67	36.67	15.93	15.93	19.63
DeepSeek-Coder	0	0	0	0	0	0	36.67	40	40	12.22	13.33	13.33
DeepSeek-R1	0	0	0	0	0	0	43.33	43.33	43.33	14.44	14.44	14.44
Gemini 2.5 Pro	0	0	0	0	0	0	40	46.67	46.67	13.33	15.56	15.56
Gemini 3 Flash	11.11	11.11	11.11	0	0	0	33.33	50	50	14.81	20.37	20.37
GPT-3.5 Turbo	0	0	0	0	0	0	16.67	20	23.33	5.56	6.67	7.78
GPT-4o	0	0	0	0	0	0	33.33	40	40	11.11	13.33	13.33
GPT-5	0	0	0	0	11.11	11.11	40	43.33	43.33	13.33	18.15	18.15
GPT-5.2	0	0	0	0	11.11	11.11	43.33	43.33	43.33	14.44	18.15	18.15
Llama 3.1 8B	0	0	0	0	0	0	6.67	16.67	20	2.22	5.56	6.67
Llama 3.3 70B	0	0	0	0	0	0	26.67	30	30	8.89	10	10

Our pipeline employs a multi-turn feedback loop: if an LLM-generated reference model fails verification, the error information is fed back to the model for self-correction. We validated this workflow using DeepSeek V3.2 on 10,000 randomly selected samples from the CodeV-R1 dataset. The process successfully generated 2,206 verified, high-quality reference models, demonstrating the tool's effectiveness in constructing robust training data for AI-aided chip design.

3 Evaluation

3.1 Evaluation Setup

Models. We evaluate ChipBench across multiple state-of-the-art LLMs, including GPT (3.5, 4o, 5, 5.2), Claude (Haiku, Opus, Sonnet), Gemini (2.5-pro, 3-flash), Llama (3.1-8B, 3.3-70B), DeepSeek-V3.2 (coder, reasoner). The detailed version information for each model is listed in Table 6 in the Appendix. For all models, we use the hyperparameters recommended by VerilogEval: `temperature=0.85` and `top_p=0.95`.

Multi-Agent Workflow. In addition to single-model evaluations, we test MAGE with DeepSeek-V3.2 backbone, a state-of-the-art multi-agent framework for Verilog generation. MAGE employs dedicated agents for sampling, debugging, and decision-making to enhance code quality.

Metrics. We report pass@1, pass@5, and pass@10 for three distinct tasks: Verilog generation, Verilog debugging, and Python reference model generation. For the multi-agent workflow, we report only pass@1, as the workflow inherently involves multi-turn iterations. Additionally, we track token consumption to estimate the API costs associated with running our benchmark.

3.2 Primary Results

We present the results for Verilog generation in Table 2, Verilog debugging in Table 4, and reference model generation in Table 3. Our key observations are as follows:

Significant performance drop compared to existing benchmarks. Despite testing on advanced models, the average pass@1 is relatively low. For instance, GPT-4o achieves ∼65% pass@1 on RTLLM V2 and VerilogEval V2, but only 10.37% on our dataset. Even MAGE, which exceeds 95%

Table 4: Test results for Verilog debugging (red: low pass rate, green: high).

Model/Dataset	Arithmetic Bug			Assignment Bug			State Machine Bug			Timing Bug			Average		
	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
Claude 4.5 Haiku	41.67	62.50	66.67	50	56.67	56.67	16.67	16.67	16.67	31.03	41.38	41.38	34.84	44.30	45.35
Claude 4.5 Opus	58.33	75	79.17	53.33	60	70	33.33	50	66.67	44.83	55.17	62.07	47.45	60.04	69.48
Claude 4.5 Sonnet	4.17	12.50	12.50	6.67	6.67	10	0	0	0	0	0	0	2.71	4.79	5.62
DeepSeek-Coder-V2.5	58.33	62.50	62.50	53.33	53.33	53.33	0	16.67	16.67	31.03	37.93	37.93	35.67	42.61	42.61
DeepSeek-R1	45.83	75	79.17	50	66.67	73.33	50	50	66.67	31.03	55.17	68.97	44.22	61.71	72.03
Gemini 2.5 Pro	0	0	0	3.33	3.33	3.33	0	0	0	0	0	0	0.83	0.83	0.83
Gemini 3 Flash	37.50	62.50	66.67	36.67	53.33	53.33	16.67	50	50	27.59	41.38	41.38	29.61	51.80	52.84
GPT-3.5 Turbo	16.67	25	25	6.67	26.67	46.67	16.67	33.33	33.33	0	0	0	10.00	21.25	26.25
GPT-4o	16.67	29.17	33.33	6.67	36.67	40	0	0	0	6.90	17.24	20.69	7.56	20.77	23.50
GPT-5	50	66.67	70.83	43.33	60	63.33	50	50	66.67	31.03	51.72	55.17	43.59	57.10	64
GPT-5.2	45.83	62.50	66.67	40	60	66.67	0	50	50	44.83	51.72	55.17	32.66	56.05	59.63
Llama 3.1 8B	4.17	8.33	20.83	3.33	23.33	36.67	16.67	16.67	16.67	6.90	10.34	17.24	7.77	14.67	22.85
Llama 3.3 70B	37.50	37.50	41.67	20	33.33	36.67	16.67	16.67	16.67	20.69	31.03	34.48	23.71	29.63	32.37

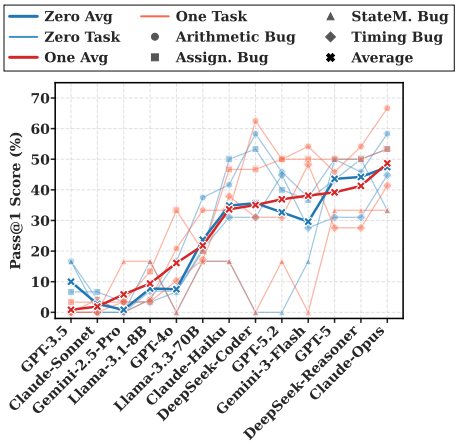


Figure 4: Performance comparison between zero-shot debug tasks and one-shot debug tasks. In zero-shot tests, LLMs are only aware of the buggy code and the existence of bugs, while in one-shot tests, LLMs have access to the complete waveform file.

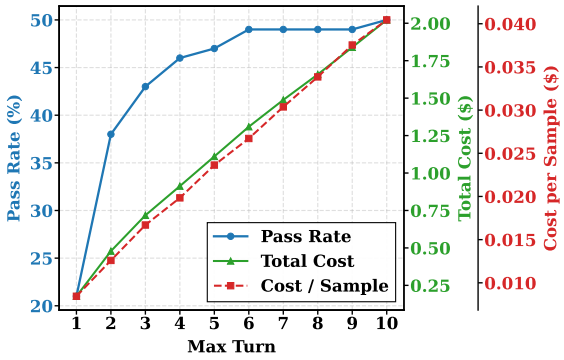


Figure 5: Performance and cost efficiency analysis of DeepSeek Reasoner across multiple turns. The blue line represents the cumulative pass rate, while the red dashed line indicates the average cost incurred per solved instance.

accuracy on VerilogEval, achieves only 37.41% on ChipBench, though it remains the top performer. These results underscore the difficulty of our dataset and demonstrate that ChipBench serves as a rigorous next-generation benchmark for AI-aided chip design. Notably, CPU IP design proves to be the most challenging task, with no model exceeding 22.22% accuracy, indicating that current LLMs require further optimization for complex Verilog generation.

Debugging capability exceeds generation capability. LLMs generally perform better on Verilog debugging than generation when given the same Verilog module. For example, Claude-opus achieves 47.45% on debugging compared to 30.74% on generation. This performance gap highlights the potential of LLMs in debugging tasks and suggests that multi-turn self-correction workflows could significantly enhance generation accuracy.

Python generation outperforms Verilog on simple modules, but underperforms on complex modules. As shown in Table 3, for simple self-contained modules, LLMs achieve 5%–20% higher accuracy on Python reference model generation compared to Verilog generation, likely due to Python's prevalence in LLM pre-training data. Although the accuracy is higher than Verilog generation, the absolute pass rate remains below 50%, far from the reliable golden reference models needed to validate Verilog code. This advantage further diminishes as module complexity increases: pass@10 rates drop to at most 11.11% for complex non-self-contained modules and CPU IP designs. This reveals that while LLMs possess strong Python syntax knowledge, they lack hardware-specific modeling capabilities. By improving these capabilities, we have the potential to significantly enhance LLMs' impact in the chip design industry.

Table 5: Cost efficiency analysis (Verilog pass@1). Cost calculations are based on standard API pricing per million tokens.

Model	Input (Tokens)	Output (Tokens)	Cost ($)	Pass@1 (%)	Cost/Pass@1 ($)
Claude 4.5 Haiku	25,076	24,925	0.037	19.26	0.194
Claude 4.5 Opus	25,076	32,221	2.793	30.74	9.085
Claude 4.5 Sonnet	25,076	29,479	0.517	12.59	4.109
DeepSeek-Coder	21,933	23,992	0.010	29.26	0.033
DeepSeek-R1	21,933	171,740	0.388	23.70	1.638
Gemini 2.5 Pro	23,409	299,064	3.222	1.11	290.283
Gemini 3 Flash	23,409	418,835	0.127	30.74	0.414
GPT-3.5 Turbo	21,710	14,030	0.032	5.56	0.574
GPT-4o	21,699	19,375	0.248	10.37	2.391
GPT-5	21,654	152,233	9.459	24.81	38.120
GPT-5.2	21,654	24,021	1.766	26.29	6.718
Llama 3.1 8B	22,699	32,271	0.003	7.04	0.039
Llama 3.3 70B	22,699	16,624	0.012	7.04	0.169

3.3 One-Shot Debugging Analysis

To evaluate whether LLMs can utilize waveform data for debugging like human engineers, we compare zero-shot and one-shot debugging in Figure 4. In zero-shot debugging, the LLM receives only the module description and the buggy implementation. In one-shot debugging, we additionally provide the waveform file (.vcd) in the prompt, mimicking real-world workflows where engineers analyze signals to detect mismatches. Our results show mixed performance: one-shot debugging outperforms zero-shot on 6 of the 13 models but underperforms on the other 7. This suggests that while waveform data contains critical debugging information, most current LLMs lack the capability to interpret it effectively. Consequently, we posit that developing agentic tools for waveform analysis and training LLMs specifically on waveform-aware debugging are promising future research directions.

3.4 Reference Model Training Dataset Generation

We evaluated our reference model training dataset generation tool using DeepSeek V3.2 on 100 Verilog samples from the QiMeng CodeV-R1 training dataset. We tested the pass rate and cost under different maximum debugging iterations. As shown in Figure 5, initially, the pass rate increases with maximum iteration count, reaching 50%. Beyond this point, the model reaches its debugging limitation and the pass rate plateaus. This means that with multi-turn debugging, our tool can generate approximately $n/2$ Python reference models from n Verilog training samples.

While higher iteration counts improve pass rates, they also increase API costs. Both the total cost and average cost per successful sample continue to rise with maximum iteration count. This suggests a trade-off: we should use lower iteration counts when Verilog data is abundant to minimize costs, and higher iteration counts when data is limited to maximize the number of generated reference models.

We further validated the scalability of our tool by processing 10,000 samples with a maximum iteration count of 1, successfully generating 2,206 verified Python reference models, demonstrating the effectiveness of our proposed tool.

3.5 Cost Analysis

We detail token consumption and associated costs in Table 5 to help researchers estimate budgets when evaluating their LLMs and agentic frameworks on ChipBench. Among all models, DeepSeek V3.2-coder emerges as the most cost-efficient model for Verilog generation, offering the lowest cost per pass@1 while maintaining relatively high code quality. Although Claude-4.5-opus achieves slightly higher pass@1, the high cost of the Claude API makes its Cost/Pass@1 over 275$\times$ higher.

4 Take-Aways

Based on our evaluation, we identify several promising future research directions:

- **Hierarchical Verilog Generation.** The success rate for generating only the top module is already below 22.22% and drops to 0% when generating complete hierarchical designs,

highlighting the critical need for hierarchical design capabilities that align with industrial practices.

- **Waveform-Aware Debugging.** While LLMs demonstrate stronger abilities in debugging than in Verilog generation, our experiments in Figure 4 reveal poor waveform interpretation capabilities. Since waveforms are crucial debugging information sources in industry, enhancing LLMs' waveform analysis could significantly improve their practical utility.

- **Reference Model Generation.** Despite LLMs' proficiency in other Python tasks, their pass rates for Python reference model generation are extremely low. This gap indicates that while LLMs understand Python syntax, they lack hardware behavior modeling knowledge, representing a significant unexplored opportunity for both chip design and verification workflows.

5 Related Work

Verilog Generation Benchmark. Early stage benchmarks VerilogEval (Liu et al., 2023b), proposed in 2023, pioneered LLM evaluation for Verilog code completion, while successive work, VerilogEval V2 (Pinckney et al., 2025a), introduced robust infrastructure for specification-to-RTL tasks. Contemporaneous benchmarks RTLLM (Lu et al., 2024) and RTLLM V2 (Liu et al., 2024) also focus on the same spec-RTL tasks and gained popularity. However, these benchmarks have key limitations compared to ChipBench: they exclusively target code generation while neglecting debugging and reference model generation, and their test cases present low complexity for current LLMs, with VerilogEval V2 containing only self-contained modules under 76 lines.

Recent ArXiv-released benchmarks in late 2025 address complexity differently. RealBench (Jin et al., 2025) features open-source IPs with extremely high complexity and high-quality specifications, but focuses only on generation and proves too difficult for current LLMs. CVDP (Pinckney et al., 2025b) expands to 13 task categories, including code modification and code improvement, but lacks comprehensive reference model evaluation and offers limited debugging cases. ChipBench, as a contemporary work, provides essential complementary capabilities with a specific emphasis on debugging and reference model generation overlooked by prior work. We offer distinctly different test cases for Verilog generation, and the combined usage of our benchmark with these contemporary works can provide a more comprehensive evaluation of LLMs' capabilities in chip design. Furthermore, we also provide a specialized toolbox to accelerate future research about reference model generation.

LLM for Testbench Generation. Recent works on LLM-assisted Verilog testbench generation, like AutoBench (Qiu et al., 2024), CorrectBench (Qiu et al., 2025), and PRO-V (Zhao et al., 2025a), also consider reference model generation as a sub-task. However, they only evaluate the correctness of the entire testbench and neglect the correctness of the reference model itself. Although they use 10-sampling and majority voting to increase accuracy, our evaluation shows that the pass@10 of the most advanced model is only around 20%, meaning they are highly likely to add stimulus signals to an incorrect reference model. Therefore, we believe that testbench generation could benefit significantly from accurate reference models, which further demonstrates the value of our benchmark in evaluating reference model generation capabilities.

6 Conclusion

This paper introduces ChipBench, a comprehensive benchmark that addresses critical gaps in evaluating LLMs for chip design applications. By expanding beyond simple Verilog generation to include complex modules, debugging tasks, and reference model generation, ChipBench provides a more realistic assessment framework that better reflects industrial deployment requirements. Evaluation of various models on ChipBench reveals that current models have significant limitations in AI-aided chip design and remain far from ready for real industrial workflow integration. We highlight the importance of debugging and reference model generation capabilities and provide a toolbox to support future research in reference model generation.

AI USE STATEMENT

In this work, we used generative AI tools for one task with required disclosure and for several tasks with recommended disclosure.

For the task with required disclosure, we used generative AI tools to generate synthetic data: the reference model training dataset described in Section 3.4 was produced by an LLM (DeepSeek V3.2) through our automated verification pipeline, and every retained sample was verified against the golden Verilog implementation by our test harness rather than accepted on the model's word.

For tasks with recommended disclosure, we used generative AI tools to convert the manuscript to the ICLR 2027 LaTeX template, including adjusting the layout of figures and tables, and for language editing to improve the readability of text written by the authors.

We have not used generative AI tools for research ideation, benchmark construction, methodology or experiment design, code implementation, running experiments, data analysis, or interpreting results, all of which were carried out by the authors. The 44 Verilog modules, their task descriptions, the injected bugs, and the test harnesses were written and curated by the authors. Large language models also appear in this work as the objects of evaluation; that use is described in Section 3 and is not AI assistance to the research process.

We have reviewed all AI-assisted work: the AI-assisted edits to the text and layout were checked by the authors, and the LLM-generated reference models retained in the training dataset were validated by differential simulation against the source Verilog rather than by manual inspection alone.

We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work introduces a benchmark for evaluating LLMs on chip design tasks. It does not involve human subjects, personally identifiable information, or sensitive data. The benchmark is built from Verilog designs collected from open-source online judge platforms and open-source CPU projects, with task descriptions written and bugs injected manually by the authors. The evaluated models were accessed through their public APIs.

REPRODUCIBILITY STATEMENT

Our code is available at https://anonymous.4open.science/r/ChipBench-8766. The exact model versions, release dates, and API identifiers behind every number we report are listed in Table 6. Each Verilog module in the benchmark, together with its code size and cell count, is described in Table 7. The evaluation protocol, including the pass@k metrics and the zero-shot and one-shot debugging settings, is described in Section 3.

REFERENCES

Meisam Abdollahi, Seyedeh Faegheh Yeganli, Mohammad Baharloo, and Amirali Baniasadi. Hardware design and verification with large language models: A scoping review, challenges, and open issues. *Electronics*, 14(1):120, 2024.

Anthropic. Introducing Claude Haiku 4.5: Speed and intelligence at scale. `https://www.anthropic.com/news/claude-haiku-4-5`, 2025a. Released: 2025-10-15.

Anthropic. Introducing Claude Opus 4.5: Our most intelligent model. `https://www.anthropic.com/news/claude-opus-4-5`, 2025b. Released: 2025-11-24.

Anthropic. Introducing Claude Sonnet 4.5: Setting new records in reasoning. `https://www.anthropic.com/news/claude-sonnet-4-5`, 2025c. Released: 2025-09-29.

Zhenyao Cai, Seehee Park, Nia Nixon, and Shayan Doroudi. Advancing knowledge together: integrating large language model-based conversational ai in small group collaborative learning.

In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, pp. 1–9, 2024.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, Krishna Haridasan, and Ahmed Omran. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

DeepSeek-AI. DeepSeek-V2.5 model upgrade: Merging chat and coder capabilities. `https://api-docs.deepseek.com/updates`, 2024. Released: 2024-09-05.

DeepSeek-AI. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025. doi: 10.1038/s41586-025-09422-z.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*, 2025.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

Farshad Firouzi, David Z Pan, Jiaqi Gu, Bahar Farahani, Jayeeta Chaudhuri, Ziang Yin, Pingchuan Ma, Peter Domanski, and Krishnendu Chakrabarty. Chipmnd: Llms for agile chip design. In *2025 IEEE 43rd VLSI Test Symposium (VTS)*, pp. 1–10. IEEE, 2025.

Harry Foster et al. The 2022 wilson research group functional verification study. *Siemens. com*, 2022.

Jiahao Gai, Hao Chen, Zhican Wang, Hongyu Zhou, Wanru Zhao, Nicholas Lane, and Hongxiang Fan. Exploring code language models for automated hls-based hardware generation: Benchmark, infrastructure and analysis. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, pp. 988–994, 2025.

Google DeepMind. Gemini 3 flash model card. Technical report, Google DeepMind, December 2025. URL `https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf`. Released: 2025-12-17.

Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22, 2025.

Jiale Guo, Suizhi Huang, Mei Li, Dong Huang, Xingsheng Chen, Regina Zhang, Zhijiang Guo, Han Yu, Siu-Ming Yiu, Pietro Lio, et al. A comprehensive survey on benchmarks and solutions in software engineering of llm-empowered agentic system. *arXiv preprint arXiv:2510.09721*, 2025.

Zhuolun He, Yuan Pu, Haoyuan Wu, Tairu Qiu, and Bei Yu. Large language models for eda: Future or mirage? *ACM Transactions on Design Automation of Electronic Systems*, 30(6):1–53, 2025.

Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 300–307, 2025.

Yuchen Hu, Junhao Ye, Ke Xu, Jialin Sun, Shiyue Zhang, Xinyao Jiao, Dingrong Pan, Jie Zhou, Ning Wang, Weiwei Shan, et al. Uvllm: An automated universal rtl verification framework using llms. *arXiv preprint arXiv:2411.16238*, 2024.

William NN Hung and Naren Narasimhan. Reference model based rtl verification: An integrated approach. In *Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No. 04EX940)*, pp. 9–13. IEEE, 2004.

Yoichi Ishibashi and Yoshimasa Nishimura. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization. *arXiv preprint arXiv:2404.02183*, 2024.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, 40(4): 58–66, 2020.

Pengwei Jin, Di Huang, Chongxiao Li, Shuyao Cheng, Yang Zhao, Xinyao Zheng, Jiaguo Zhu, Shuyi Xing, Bohan Dou, Rui Zhang, et al. Realbench: Benchmarking verilog generation models with real-world ip designs. *arXiv preprint arXiv:2507.16200*, 2025.

Kurt Keutzer, A Richard Newton, Jan M Rabaey, and Alberto Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE transactions on computer-aided design of integrated circuits and systems*, 19(12):1523–1543, 2000.

Jia Li, Chongyang Tao, Jia Li, Ge Li, Zhi Jin, Huangzhao Zhang, Zheng Fang, and Fang Liu. Large language model-aware in-context learning for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(7):1–33, 2025.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023a.

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

Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation(invited). In *Proceedings of 2024 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 2024.

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)*, pp. 722–727. IEEE, 2024.

Meta. The Llama 3.1 collection: 8B, 70B, and 405B. `https://ai.meta.com/blog/meta-llama-3-1/`, 2024a. Released: 2024-07-23.

Meta. Llama 3.3 model card. `https://github.com/meta-llama/llama-models/blob/main/models/llama3_3/MODEL_CARD.md`, 2024b. Released: 2024-12-06.

Bardia Nadimi and Hao Zheng. A multi-expert large language model architecture for verilog code generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–5. IEEE, 2024.

Bardia Nadimi, Ghali Omar Boutaib, and Hao Zheng. Pyranet: A multi-layered hierarchical dataset for verilog. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pp. 1–7. IEEE, 2025.

OpenAI. Openai unveils new models and lower price (gpt-3.5-turbo-0125). `https://openai.com/blog/new-embedding-models-and-api-updates`, 2024a. Released: 2024-01-25.

OpenAI. GPT-4o: Enhanced reasoning and multilingual performance. `https://openai.com/index/gpt-4o-2024-11-20/`, 2024b. Released: 2024-11-20.

OpenAI. Introducing GPT-5: A unified system for reasoning and multimodality. `https://openai.com/index/introducing-gpt-5/`, 2025a. Released: 2025-08-07.

OpenAI. Introducing GPT-5.2: Professional knowledge work and hallucination reduction. `https://openai.com/index/introducing-gpt-5-2/`, 2025b. Released: 2025-12-11.

Preeti Ranjan Panda. Systemc: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*, pp. 75–80, 2001.

Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting verilogeval: A year of improvements in large-language models for hardware code generation. *ACM Transactions on Design Automation of Electronic Systems*, 2025a.

Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on rtl design and verification. *arXiv preprint arXiv:2506.14074*, 2025b.

Dick Price. Pentium fdiv flaw-lessons learned. *IEEE Micro*, 15(2):86–88, 1995.

Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. 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*, pp. 1–10, 2024.

Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. Correctbench: Automatic testbench generation with functional self-correction using llms for hdl design. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pp. 1–7. IEEE, 2025.

Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, pp. 491–514, 2023.

Wilson Snyder and Verilator contributors. Verilator: Open-source systemverilog simulator and lint tool. `https://github.com/verilator/verilator`, 2025. Released: 2025-12-11.

Sneha Swaroopa, Rijoy Mukherjee, Anushka Debnath, and Rajat Subhra Chakraborty. Evaluating large language models for automatic register transfer logic generation via high-level synthesis. *arXiv preprint arXiv:2408.02793*, 2024.

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*, 29(3):1–31, 2024.

Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 1069–1088, 2023.

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

Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine, Yong Liu, and Haoxing Ren. Spec2rtl-agent: Automated hardware code generation from complex specifications using llm agent systems. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 37–43. IEEE, 2025.

Yujie Zhao, Zhijing Wu, Hejia Zhang, Zhongming Yu, Wentao Ni, Chia-Tung Ho, Haoxing Ren, and Jishen Zhao. Pro-v: An efficient program generation multi-agent system for automatic rtl verification. *arXiv preprint arXiv:2506.12200*, 2025a.

Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. Mage: A multi-agent engine for automated rtl code generation. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pp. 1–7. IEEE, 2025b.

Li Zhong and Zilong Wang. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 21841–21849, 2024.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Pengwei Jin, et al. Qimeng-codev-r1: Reasoning-enhanced verilog generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

A APPENDIX

A.1 DETAILED MODEL VERSION INFORMATION

Please refer to Table 6 for model versions we used for evaluation. We also list the release date, model size, and API names.

Table 6: Overview of evaluated models with verified release dates and parameter sizes. Model references cite technical reports; sizes with (*) denote industry estimates based on performance benchmarks.

Model	Reference	Release Date	Size	Model API Name
GPT-5.2	OpenAI (2025b)	2025-12-11	52.5T*	`gpt-5.2-2025-12-11`
GPT-5	OpenAI (2025a)	2025-08-07	52.5T*	`gpt-5-2025-08-07`
GPT-4o	OpenAI (2024b)	2024-11-20	∼200B*	`gpt-4o-2024-11-20`
GPT-3.5 Turbo	OpenAI (2024a)	2024-01-25	175B*	`gpt-3.5-turbo-0125`
Claude 4.5 Opus	Anthropic (2025b)	2025-11-24	>150B*	`claude-opus-4-5-20251101`
Claude 4.5 Haiku	Anthropic (2025a)	2025-10-15	<40B*	`claude-haiku-4-5-20251001`
Claude 4.5 Sonnet	Anthropic (2025c)	2025-09-29	150B*	`claude-sonnet-4-5-20250929`
Gemini 3 Flash	Google DeepMind (2025)	2025-12-17	∼10B*	`gemini-3-flash-preview`
Gemini 2.5 Pro	Comanici et al. (2025)	2025-06-17	400B*	`gemini-2.5-pro`
DeepSeek-R1	DeepSeek-AI (2025)	2025-01-20	671B	`deepseek-reasoner`
DeepSeek-Coder	DeepSeek-AI (2024)	2024-09-05	236B	`deepseek-coder`
Llama 3.3 70B	Meta (2024b)	2024-12-06	70B	`Llama-3.3-70B-Instruct-Turbo`
Llama 3.1 8B	Meta (2024a)	2024-07-23	8B	`Meta-Llama-3.1-8B-Instruct-Turbo`

A.2 DETAILED INFORMATION OF MODULES

We summarize the description of each module in Table 7.

Table 7: Verilog module descriptions.

Design	Module Name	Description	Code Lines	Circuit Cells
Self-Contained	continuous_input_sequence_detect	8-bit shift-register pattern detector	33	16
	extraneous_items_sequence_detect	9-bit shift-register detector	51	16
	non-overlapping_sequence_detect	Non-overlapping 6-bit group sequence checker	95	35
	continuous_input_sequence_detect2	FSM sequence detector	78	29
	signal_generator	5-bit waveform generator	48	112
	data_serial-to-parallel_circuit	Serial-to-parallel packer	50	28
	data_accumulation_output	Ready/valid accumulator	44	91
	FracWidthConv_24to128	Bit-pack a valid-only 24-bit input stream into 128-bit outputs	48	532
	FracWidthConv_8to12	Bit-pack a valid-only 8-bit input stream into 12-bit outputs	48	42
	IntWidthConv_8to16	Valid-only width converter that packs two 8-bit inputs into one 16-bit output	40	28

Table 7 (continued)

Design	Module Name	Description	Code Lines	Circuit Cells
	4-bit_carry_look-ahead_adder_circuit	4-bit carry look-ahead adder	18	20
	state_machine_overlap_sequence_detection	Overlapping sequence detector	40	19
	least_common_multiple	8-bit LCM/GCD calculator	92	7409
	sequence_generator	Cyclic sequence generator	21	7
	vending_machine2	Vending machine FSM	64	92
	OddDiv_HalfDuty	Divide-by-7 clock divider with ∼50% duty cycle	56	22
	arbitrary_fractional_frequency_division	Fractional clock divider achieving average /8.7	51	66
	write_state_machine_two_stage	Two-process Moore FSM	53	11
	gray_code_counter	4-bit Gray-code counter	44	25
	multiplication_and_bitwise_operations	Resource-minimized constant multiply compute	12	96
	pulse_synchronization_circuit	Clock-domain crossing pulse synchronizer	45	6
	simple_stopwatch	Stopwatch counter	38	48
	settable_counter	Hex counter	50	22
	up_and_down_counter	Decimal up/down counter	44	33
	simple_implementation_RAM	Simple dual-port RAM (depth 8, width 4)	31	189
	johnson_counter	4-bit Johnson counter	12	5
	pipeline_multiplier	4-bit unsigned pipelined multiplier	42	93
	traffic_lights	Traffic light controller	101	105
	gaming_machine_billing_program	Prepaid gaming billing FSM	36	178
Not-Self-Contained	submodules_to_comparison_of_three_inputs	A Comparator for the minimum of three 8-bit unsigned inputs	56	116
	full_subtractor_using_three_to_eight_decoder	A 1-bit full subtractor	109	28
	asynchronous_FIFO	Asynchronous FIFO (dual clock)	144	380
	synchronous_FIFO	Single-clock FIFO	88	367
	8_bit_alu	An 8-bit ALU	70	81
	cpu_top	A Top-level 16-bit CPU	250	1195
cpu-design	controller	A RISC-V main controller that decodes the 7-bit opcode into datapath control signals	52	42
	alu	Purely combinational 32-bit ALU with 4-bit ‘Operation‘ code	51	1456
	regfile	32×32-bit register file with 1 write port and 2 combinational read ports	39	3122
	branch_unit	A branch unit	26	206
	alu_controller	A RISC-V ALU-control decoder	45	69
	pc_reg	A PC register	45	180
	div	A Sequential 32-bit signed/unsigned division unit	99	1228
	ctrl	A Pipeline control Unit	73	213
	cp0_reg	A CP0 register file implementing Count/Compare/Status/Cause/EPC/etc.	183	1245

A.3 Examples of Injected Bugs

Figure 6 shows one example of each of the four bug types injected into a golden module.

Original

```
1 module TopModule(
2 input clk,
3 input rst_n,
4 output reg red,
......
10 );
......
38 s1_red:
39 begin:
40 p_red <= 1'b1;
41 p_green <= 1'b0;
42 p_yellow <= 1'b0;
43 if (cnt == 3)
44 state <= s2_green;
45 else
46 state <= s1_red;
47 end
......
71 always @(posedge clk or
negedge rst_n)
72 if(!rst_n)
73 cnt <= 7'd10;
76 else if (!green&&p_green)
77 cnt <= 7'd60;
......
85 always @(posedge clk or
negedge rst_n)
86 if(!rst_n)
87 begin
88 yellow <= 1'd0;
......
99 endmodule
```

Arithmetic Bug (24)

```
1 module TopModule(
2 input clk,
3 input rst_n,
4 output reg red,
......
10 );
......
38 s1_red:
39 begin:
40 p_red <= 1'b1;
41 p_green <= 1'b1;
42 p_yellow <= 1'b0;
43 if (cnt == 3)
44 state <= s2_green;
45 else
46 state <= s1_red;
47 end
......
71 always @(posedge clk or
negedge rst_n)
72 if(!rst_n)
73 cnt <= 7'd10;
76 else if (green&&p_green)
77 cnt <= 7'd60;
......
85 always @(posedge clk or
negedge rst_n)
86 if(!rst_n)
87 begin
88 yellow <= 1'd0;
......
99 endmodule
```

Assignment Bug (30)

```
1 module TopModule(
2 input clk,
3 input rst_n,
4 output reg red,
......
10 );
......
38 s1_red:
39 begin:
40 p_red <= 1'b1;
41 p_green <= 1'b1;
42 p_yellow <= 1'b0;
43 if (cnt == 3)
44 state <= s2_green;
45 else
46 state <= s1_red;
47 end
......
71 always @(posedge clk or
negedge rst_n)
72 if(!rst_n)
73 cnt <= 7'd10;
76 else if (!green&&p_green)
77 cnt <= 7'd60;
......
85 always @(posedge clk or
negedge rst_n)
86 if(!rst_n)
87 begin
88 yellow <= 1'd0;
......
99 endmodule
```

Timing Bug (29)

```
1 module TopModule(
2 input clk,
3 input rst_n,
4 output reg red,
......
10 );
......
38 s1_red:
39 begin:
40 p_red <= 1'b1;
41 p_green <= 1'b0;
42 p_yellow <= 1'b0;
43 if (cnt == 3)
44 state <= s2_green;
45 else
46 state <= s1_red;
47 end
......
71 always @(posedge clk or
negedge rst_n)
72 if(!rst_n)
73 cnt <= 7'd10;
76 else if (!green&&p_green)
77 cnt <= 7'd60;
......
85 always @(*)
86 if(!rst_n)
87 begin
88 yellow = 1'd0;
......
99 endmodule
```

State Machine Bug (6)

```
1 module TopModule(
2 input clk,
3 input rst_n,
4 output reg red,
......
10 );
......
38 s1_red:
39 begin:
40 p_red <= 1'b1;
41 p_green <= 1'b0;
42 p_yellow <= 1'b0;
43 if (cnt == 3)
44 state <= s3_yellow;
45 else
46 state <= s1_red;
47 end
......
71 always @(posedge clk or
negedge rst_n)
72 if(!rst_n)
73 cnt <= 7'd10;
76 else if (!green&&p_green)
77 cnt <= 7'd60;
......
85 always @(posedge clk or
negedge rst_n)
86 if(!rst_n)
87 begin
88 yellow <= 1'd0;
......
99 endmodule
```

Figure 6: Demonstration of four types of bugs injected into original golden modules.