Preprints.org

Article

Not peer-reviewed version

Survey and Benchmarking of Large Language Models for RTL Code Generation: Techniques and Open Challenges

Arun Ravindran *, Aditya Patra, Vahid Babaey, Suresh Purini

Posted Date: 19 September 2025

doi: 10.20944/preprints202509.1681.v1

Keywords: large language models; RTL generation; electronic design automation; specification-to-RTL; high-level synthesis; multi-agent systems; benchmarking; natural-language-to-SoC; reinforcement learning; debugging

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Survey and Benchmarking of Large Language Models for RTL Code Generation: Techniques and Open Challenges

Arun Ravindran [1,*], Aditya Patra [2], Vahid Babaey [1] and Suresh Purini [3]

[1] University of North Carolina at Charlotte, Charlotte, USA
[2] California High School, San Ramon, USA
[3] International Institute of Information Technology at Hyderabad, Hyderabad, India
* Correspondence: arun.ravindran@charlotte.edu

Abstract

Large language models (LLMs) are emerging as powerful tools for hardware design, with recent work exploring their ability to generate register-transfer level (RTL) code directly from natural-language specifications. This paper provides a survey and evaluation of LLM-based RTL generation. We review twenty-six published efforts, covering techniques such as fine-tuning, reinforcement learning, retrieval-augmented prompting, and multi-agent orchestration, and we analyze their contributions across eight methodological dimensions including debugging support, post-RTL metrics, and benchmark development. Building on this review, we experimentally evaluate frontier commercial models—GPT-4.1, GPT-4.1-mini, and Claude Sonnet 4—on the VerilogEval and RTLLM benchmarks under both single-shot and lightweight agentic settings. Results show that these models achieve up to 89.74% on VerilogEval and 96.08% on RTLLM, matching or exceeding prior domain-specific pipelines without specialized fine-tuning. Detailed failure analysis reveals systematic error modes, including FSM mis-sequencing, handshake drift, blocking vs. non-blocking misuse, and state-space oversimplification. Finally, we outline a forward-looking research roadmap toward natural-language-to-SoC design, emphasizing controlled specification schemas, open benchmarks and flows, PPA-in-the-loop feedback, and modular assurance frameworks. Together, this work provides both a critical synthesis of recent advances and a baseline evaluation of frontier LLMs, highlighting opportunities and challenges in moving toward AI-native electronic design automation.

Keywords: large language models; RTL generation; electronic design automation; specification-to-RTL; high-level synthesis; multi-agent systems; benchmarking; natural-language-to-SoC; reinforcement learning; debugging

1. Introduction

Designing modern system-on-chips (SoCs) remains one of the most challenging tasks in computing systems engineering. An SoC integrates heterogeneous accelerators for cryptography, signal and image processing, and machine learning, each with deep pipelines, hierarchical reuse, and stringent power, performance, and area (PPA) requirements [1] [2]. The design flow involves multiple stages—capturing specifications, planning microarchitectures, implementing register-transfer level (RTL) modules, synthesizing netlists, performing placement and routing, and verifying correctness at every level [3]. While commercial [4] [5] [6] and open source [7] electronic design automation (EDA) tools have matured in the later stages of RTL-to-GDSII, the earlier task of translating natural-language or informal specifications into correct RTL remains largely manual, error-prone, and iterative.

The emergence of large language models (LLMs) has opened new opportunities to address this gap [8]. LLMs trained on vast amounts of code and technical text demonstrate strong capabilities in program synthesis and reasoning, suggesting their potential for hardware description language

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

(HDL) generation. Over the past two years, research in this space has accelerated rapidly. We critically analyze this evolving body of work to identify the techniques explored—ranging from prompt engineering [9], retrieval augmented generation [10], multi-agentic LLM [11] [12], fine-tuning [13], tool use [14] and reinforcement learning [15]—and to chart how the field has progressed. Benchmarks such as VerilogEval [16] and RTLLM [17] have played a central role in this evolution, enabling standardized evaluation of these techniques. Yet, despite notable successes, most efforts remain point solutions that optimize benchmark performance without deeply examining the root causes of failures.

Two recent surveys have attempted to cover the role of LLMs in EDA, but our work differs in both scope and methodology. Pan et al. [18] provide a broad overview spanning system-level, RTL, synthesis, physical design, and analog domains, but offer limited critical analysis and largely catalog techniques such as fine-tuning, prompt engineering, agentic execution, and retrieval. Similarly, Xu et al. [19] review work across the EDA stack and outline a roadmap of challenges ranging from semantic gaps to tool integration and security, though their literature coverage is uneven and prescriptive guidance sparse. By contrast, our review focuses narrowly on the **specification-to-RTL problem**, which is both more open-ended and less studied, enabling a deeper critical analysis of agentic methods, evaluation strategies, and emerging benchmarks. Crucially, we go beyond prior surveys by conducting an *experimental evaluation of state-of-the-art frontier LLMs* on VerilogEval and RTLLM, and by analyzing failure cases in detail. This allows us to identify not only where models succeed, but also systematic error patterns that current benchmarks expose—an aspect largely absent from earlier surveys. Automated flows such as OpenRoad [7] already exist for the RTL-to-GDSII phase (synthesis and physical design), and recent efforts have explored augmenting this stage with AI support, including backend-aware optimization [20], runtime debugging agents [21], autonomous OpenROAD flows [22], and microservice-based orchestration [23]. These works lie outside the scope of this review, which centers on spec-to-RTL, but are noted here for completeness and to guide readers interested in downstream phases.

This paper makes three contributions:

1. **Critical analysis of recent work and benchmarks:** We review and synthesize techniques proposed in the literature, showing how approaches have evolved over the past three years and how they align with benchmark design.
2. **Experimental evaluation of state-of-the-art LLMs:** We show that baseline techniques, such as single-shot generation and simple ReAct-style [24] reflection, achieve strong performance on current benchmarks, indicating that these benchmarks may no longer stress the limits of today's models. This highlights the need for richer, more complex benchmarks to drive further progress.
3. **Identification of recurring failure modes:** By analyzing failed designs in depth, we surface systematic issues such as finite state machine mis-sequencing, handshake protocol drift, and misuse of blocking vs. non-blocking semantics, offering insight into why LLMs struggle with hardware concurrency and temporal reasoning.

Recently released benchmarks such as ArchXBench [25], which introduce deeper pipelines, hierarchical composition, and application-level accelerators, illustrate the kind of complexity needed to better evaluate emerging methods. Together, these contributions provide the reader with both a comprehensive map of existing approaches and a grounded assessment of how well current LLMs perform. More importantly, they highlight where progress has been made, where gaps remain, and what research directions are most promising. Readers will come away with a clear understanding of the state of the art in LLM-driven RTL generation, the limitations of existing benchmarks, and a forward-looking perspective on how AI-native tools might ultimately enable intent-centric, natural-language-to-SoC design.

The remainder of the paper is organized as follows. Section 2 surveys prior work on LLM-driven RTL generation. Section 3 presents our evaluation setup and benchmark results. Section 4 analyzes systematic failure modes. Section 5 outlines a research roadmap for natural-language-to-SoC design. Section 6 concludes.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

2. Prior Work on LLM Based RTL Generation

We surveyed the literature on RTL generation using large language models (LLMs), focusing on work published from the introduction of publicly accessible LLMs in late 2022 through mid-2025. This period spans the release of foundational models such as ChatGPT [26], Code Llama [27], DeepSeek [28], Gemini [29], and Qwen [30], which made LLM-based code generation broadly feasible for hardware applications.

We selected 26 works that address RTL synthesis, verification, or refinement using generative models. Collectively, they span the state of the art in LLM-based hardware design, employing techniques such as supervised fine-tuning [13], reinforcement learning, [15] retrieval-augmented prompting [10], and multi-agent coordination [11].

2.1. Survey Methodology

Each project is evaluated along eight methodological dimensions that capture the breadth of techniques employed in LLM-based RTL generation. The *fine-tuning (FT)* dimension reflects whether the model is further trained on RTL-specific corpora, typically drawn from GitHub or curated repositories.

The *reinforcement learning (RL)* dimension identifies models that incorporate feedback-driven optimization, such as policy-gradient or reward-weighted fine-tuning, to improve correctness under evaluation constraints.

Agentic execution (Ag) denotes the use of multi-agent workflows, where distinct agents (e.g., planner, generator, verifier) coordinate task decomposition, code generation, and evaluation.

The *retrieval or in-context learning (RAG)* dimension captures systems that enhance generation with example-based conditioning or retrieval-augmented prompts, often improving robustness on long-tail designs.

Debugging support (Dbg) refers to mechanisms that help localize and correct faults, such as Abstract Syntax Tree(AST)-level backtracing, simulation waveform checkpoints, or compiler error alignment.

Post-RTL metrics (Beyond) assesses whether the approach accounts for downstream qualities of the generated RTL—such as gate count, synthesis efficiency, Boolean Satisfiability (SAT) solving time, or resource utilization.

The *benchmark contribution (BM)* dimension marks whether a paper introduces a new benchmark suite for evaluating RTL generation systems.

Finally, *prompt engineering or chain-of-thought (PE)* identifies the deliberate design of structured prompts to guide model reasoning or decompose complex specifications. Together, these dimensions provide a structured basis for comparison across systems and inform the taxonomy presented in Table 1.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

Table 1. Technique coverage. Abbreviations: FT = fine tuning; RL = reinforcement learning; Ag = agentic execution; RAG = retrieval / in-context learning; Dbg = explicit debugging support; Beyond = post-RTL quality metrics; BM = benchmark contribution; PE = prompt or Chain-of-Thought engineering. A tick (✓) means the feature is present, a cross (×) means it is absent.

Paper	FT	RL	Ag	RAG	Dbg	Beyond	BM	PE
MAGE (1)	×	×	✓	×	✓	×	×	×
VerilogCoder (2)	×	×	✓	×	✓	×	×	×
Aivril (3)	×	×	✓	×	×	×	×	×
RTLFixer (4)	×	×	✓	✓	×	×	×	×
AutoVCoder (5)	✓	×	×	✓	×	×	×	×
PyHDLEval (6)	×	×	×	✓	×	×	×	×
VeriReason (7)	✓	✓	×	×	×	×	×	×
Verigen (8)	✓	×	×	×	×	×	×	×
Autochip (9)	×	×	✓	×	×	×	×	×
BetterV (10)	✓	×	×	×	×	✓	×	×
VerilogEval (11)	×	×	×	✓	×	×	✓	×
RTLLM (12)	×	×	×	×	×	×	✓	✓
RTLCoder (13)	✓	✓	×	×	×	×	×	×
RTL_repo (14)	×	×	×	×	×	×	✓	×
VeriAssist (15)	×	×	✓	×	×	✓	×	✓
ChipNeMo (16)	✓	×	×	×	×	×	×	×
Paradigm-based (17)	×	×	✓	×	×	×	×	✓
AssertLLM (18)	×	×	✓	✓	×	×	✓	✓
AutoBench (19)	×	×	×	×	✓	×	×	×
CorrectBench (20)	×	×	✓	×	✓	×	×	×
C2HLSC (21)	×	×	×	✓	×	×	×	✓
HLSPilot (22)	×	×	×	✓	×	×	×	✓
LHS (23)	×	×	×	✓	×	✓	×	✓
LLM-based Formal Verification (24)	×	×	×	×	×	×	×	✓
VeriOpt (25)	×	×	✓	✓	×	✓	×	✓
Autosilicon (26)	×	×	✓	✓	✓	✓	✓	✓

2.2. Per-Paper Summaries

The following summaries present each research project in alphabetical order, highlighting core methodologies, contributions, and benchmark results. Full citations are provided in the bibliography.

AIVRIL Two-agent ReAct [24] loop (coding and review) co-generates RTL and testbenches, then applies coverage and assertion checks to iteratively fix errors. Limited verfiication depth. Benchmarked on VerilogEval-Human [16] [31].

AssertLLM Uses a multi-agent method for specification extraction, signal mapping, and System Verilog Assertion(SVA) generation. Techniques include in-context learning and prompt engineering, with evaluations performed using Formal Property Verification(FPV) tools (e.g., Jasper [32]) against a golden RTL. [33]

AutoBench Automates testbench creation by generating JSON-based scenarios, Verilog drivers, and Python checkers from RTL. Iterative corrections are applied, with syntactic and functional correctness validated while mutating the RTL under test for coverage. Evaluated on Verilog-HumanEval. [34]

AutoChip Early ReAct framework that feeds compiler and simulation errors back to the LLM for self-correction. [35]

AutoVCoder Pipeline of GitHub mining, GPT-3.5 synthetic instructions, two-round fine-tuning, and dual RAG retrievers for examples and design rules. Evalauted on VerilogEval [16] and RTLLM [17]. [36]

Autosilicon Multi-agent framework with dedicated agents for decomposition, implementation, and debugging, augmented by persistent vector-memory, serialized waveforms, and automated

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

testbench generation. Evaluated on VerilogEval, RTLLM, and large-scale open-source designs (CPUs, I/O, accelerators), with PPA analysis via Synopsys Silicon Compiler on TSMC 65 nm technology. [37]

BetterV Pairs a small "LLM-as-judge" discriminator with CodeLlama, DeepSeek, and CodeQwen LLM backbones to minimise post Yosys [38] synthesis node count and SAT solve time—the first study targeting post-synthesis metrics. [39]

C2HLSC Proposes a flow from C implementations to HLS-compatible C, synthesizable by tools such as Catapult. Stages include function hierarchy analysis, refactoring to HLS-compatible code, and insertion of pragmas via LLM prompts. Benchmarks include cryptographic primitives, randomness tests, and quickSort. Results show correctness and efficiency approaching hand-coded RTL with reduced effort, though evaluation is limited. [40]

ChipNeMo NVIDIA's 70 B-parameter model domain-adapted for chip design via continued pre-training and instruction alignment. Demonstrations include EDA script generation and bug summarisation. [41]

CorrectBench Uses LLMs as both generator and judge of RT. A pool of imperfect RTLs and testbench scenarios are cross-validated via majority voting with safeguards against over-rejection. [42]

HLSPilot Introduces a set of C-to-HLS optimization strategies tailored to patterns such as nested loops and local arrays. These strategies are applied via in-context learning, where exemplary C/C++–to–HLS prompts guide code refactoring. A design space exploration tool (GenHLSOptimizer) tunes pragma parameters, while software profiling identifies bottlenecks for selective hardware acceleration. [43]

LHS Applies LLMs to improve high level synthesis of deep learning workloads. Error logs guide code refactoring into synthesizable forms with loop optimizations, accumulator insertion, and structural changes. Pragmas for pipelining, unrolling, and partitioning are applied based on synthesis feedback and performance-gain metrics. [44]

LLM-based Formal Verification Represents early work on generating System Verilog Assertion (SVAs) directly from RTL. Consistency between RTL and properties is checked with Formal Property Verification(FPV) tools, leaving resolution of conflicts to designers. [45]

MAGE Multi-agent engine that samples high-temperature RTL candidates, uses checkpoint-based waveform checks, and lets a judge agent steer regeneration, reaching 95.7% functional correctness on VerilogEval-Human. [46]

Paradigm-based Prompt-engineering scheme that classifies each task as combinational or sequential, then applies a tailored "paradigm block" of sub-prompts and tool calls, boosting pass@1 without extra data or tuning. [47]

PyHDLEval Benchmark spanning Verilog and five Python-embedded DSLs; emphasises in-context learning and provides a uniform harness to compare LLMs across HDLs. [48]

RTLCoder Publishes an open dataset for finetuning, plus a lightweight RL fine-tuning recipe (policy-gradient with KL regularisation). Mistral- and DeepSeek-based variants beat GPT-3.5 on VerilogEval. [49]

RTLFixer Specialises in debugging: retrieves compiler-error exemplars and feeds them through ReAct prompting to correct syntax faults, repairing more than 98% of build failures in VerilogEval-Human. [50]

RTLLM Earliest open specification-to-RTL benchmark; groups 30 designs into arithmetic, control, memory, and miscellaneous categories and encourages Chain-of-Thought prompting. [17]

RTL-Repo Large-scale corpus of 4 000 multi-file GitHub projects with full context for code-completion evaluation, stressing the long-context reasoning absent from earlier suites. [51]

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

VeriAssist Self-verifying assistant that iteratively simulates, reasons over waveforms, patches code, and reports FPGA resource utilisation as a quality metric. [52]

Verigen Among the early works in supervised fine-tune of CodeGen-16B on curated GitHub and textbook data; shows that domain data alone can outperform GPT-3.5 on HDLBits completion. [53]

VeriOpt Employs a multi-agent workflow with planning, programming, reviewing, and evaluation roles. Its key contribution lies in PPA-aware prompting, incorporating techniques such as clock gating, power gating, resource sharing, FSM encoding, pipelining, and loop unrolling. Multi-modal feedback—synthesis reports, timing diagrams, and hardware metrics—is fed back to the LLM, enabling optimization for power, performance, and area. [54]

VeriReason Adds human-style reasoning traces and high-quality testbenches, then fine-tunes Qwen 2.5 models using Guided Reward Proximal Optimisation; evaluated on VerilogEval. [55]

VerilogCoder Task-Circuit Relation Graph planner with four collaborating agents (planner, coder, verifier, AST-tracer). AST (Abstract Syntex Tree) back-tracing pinpoints faulty signals before regeneration. Evaluated on VerilogEval. [56]

VerilogEval Widely used benchmark offering code-completion and spec-to-RTL tasks plus reference testbenches; later releases add a failure taxonomy and in-context prompts. [16].

2.3. Technique Taxonomy

To compare the methodological contributions of each project, we construct a taxonomy based on eight recurring dimensions observed across the literature. Table 1 summarizes the presence (✓) or absence (×) of each dimension for all 26 reviewed projects. This matrix highlights both the concentration of effort around certain techniques (e.g., fine-tuning and agentic execution) and the relative scarcity of others (e.g., post-RTL analysis or tool-flow integration). It also reveals distinct methodological clusters, which we analyze further through the observations below.

2.4. Observations

- **Fine-tuning as a starting point.** Early systems such as Verigen, AutoVCoder, BetterV, and ChipNeMo rely primarily on supervised fine-tuning over GitHub-style datasets. More recent work (e.g., VeriReason, RTLCoder) incorporates RL to improve functional correctness on benchmark suites like VerilogEval.
- **Rise of agentic frameworks.** Projects such as MAGE and VerilogCoder demonstrate that multi-agent orchestration with simulation feedback can outperform monolithic prompting. This trend extends to AssertLLM (multi-role pipeline for SVA generation), VeriOpt (PPA-aware agent loop), and CorrectBench (LLM-as-generator-and-judge). A key challenge is enabling agents to reason collectively over a shared context.
- **Retrieval and in-context learning address edge cases.** Systems like AutoVCoder and RTLFixer incorporate RAG to retrieve relevant design examples or compiler-error templates. PyHDLEval extends this to multiple Python based DSLs. Recent HLS-oriented work (C2HLSC, HLSPilot, LHS) relies heavily on in-context strategies and synthesis-feedback prompting to refactor C code, while VeriOpt incorporates exemplar-guided design optimizations.
- **Debugging strategies are becoming more sophisticated.** Early work relied on binary pass/fail simulation outcomes. Recent methods integrate Abstract Syntax Tree (AST) back-tracing (VerilogCoder), checkpoint-based waveform inspection (MAGE), and pattern-matching against compiler diagnostics (RTLFixer). AutoBench and CorrectBench add iterative correction and voting-based self-judgment, while VeriOpt leverages synthesis reports and timing diagrams for reflective corrections. Nevertheless, accurate fault attribution remains an open problem.
- **Post-RTL metrics are underexplored.** BetterV introduces design-time metrics such as Yosys node count and SAT solve time. VeriAssist adds FPGA resource utilization. VeriOpt advances this trend

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

further with explicit PPA-aware prompting, while LHS optimizes for latency/area trade-offs. However, systematic integration of such post-RTL metrics remains limited.

- **Benchmarks shape progress.** VerilogEval and RTLLM remain the dominant evaluation suites, while HDLBits and RTL_Repo expand test coverage to code-completion and long-context inference. AssertLLM contributes a dedicated SVA benchmark. Still, few datasets stress synthesis quality, timing closure, or end-to-end hardware viability.

2.5. Evolution Over Time

LLM-based RTL generation is a recent area of study, emerging only after large language models became widely accessible with the public release of ChatGPT in November 2022. This accessibility catalyzed a wave of research that adapted LLMs to hardware design tasks, initially through supervised fine-tuning and later through more structured, interactive, and feedback-driven approaches. The following timeline outlines key phases of technical progression:

1. **2023.** Initial efforts such as ChipChat and ChipNeMo demonstrated the feasibility of applying large language models to hardware design tasks, including code generation and EDA tool interaction. These systems primarily focused on exploratory use cases and domain adaptation of general-purpose models.
2. **Early 2024.** Projects such as Verigen and BetterV established a baseline for RTL generation using LLMs through domain-specific supervised fine-tuning and basic prompt engineering. These approaches improved performance on benchmark tasks but lacked structured reasoning or interactive workflows.
3. **Mid 2024.** Research shifted toward more structured prompting methods, including instruction prompting, chain-of-thought reasoning, in-context learning, and retrieval-augmented generation. Most systems in this phase remained single-agent, relying on static prompt composition or few-shot exemplars with limited use of external tool feedback. Notable examples include AutoVCoder, RTLFixer, and PyHDLEval.
4. **Late 2024.** Agentic execution emerged as a dominant paradigm, with systems such as MAGE, VerilogCoder, and AIVRIL introducing multi-agent workflows capable of decomposing tasks, coordinating generation and verification, and iteratively refining outputs based on simulation and analysis. AutoBench extended this line of work to automatic testbench generation, integrating iterative correction and evaluation strategies.
5. **2025.** Specialized agentic methods expanded to verification and debugging. AssertLLM applied multi-agent pipelines for specification extraction and assertion generation, while CorrectBench introduced voting-based self-judgment to sidestep the need for a golden DUT. These works marked a shift toward autonomous verification flows beyond RTL generation.
   High-Level Synthesis (HLS) emerged as a parallel research trajectory. C2HLSC and HLSPilot demonstrated how LLMs can refactor C/C++ code into HLS-compatible implementations with pragma insertion and design space exploration. *LHS* further advanced this direction by integrating synthesis feedback to optimize latency/area trade-offs in deep learning accelerators. Reinforcement learning was integrated into the RTL generation workflow, with projects such as VeriReason and RTLCoder applying policy-gradient or reward-based fine-tuning to optimize model outputs for functional correctness and benchmark performance, particularly on suites like VerilogEval.

Overall, while RTL generation remains the central focus, recent work highlights the rise of **parallel tracks**—one emphasizing HLS-driven design flows (C2HLSC, HLSPilot, LHS) and another focusing on autonomous verification and debugging (AssertLLM, AutoBench, CorrectBench)—broadening the scope of LLM integration in hardware design.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

3. Evaluation of Benchmarks with State-of-the-Art LLMs

This section evaluates the capability of state-of-the-art large language models (LLMs) to generate synthesizable Verilog RTL from natural-language specifications. While prior work has introduced techniques aimed at improving benchmark scores, comparatively little attention has been paid to the adequacy of the benchmarks themselves or to the specific failure modes they reveal. A central question is whether current benchmarks are well designed to capture RTL complexity, or whether construction artifacts—e.g., incomplete prompts or flawed testbenches—confound results. In parallel, it remains unclear how effectively contemporary LLMs—*without* task-specific agents, multi-step planning, or external optimizers—can directly produce verifiable RTL from specification.

We therefore establish a baseline along two axes: (i) the generative capabilities and limitations of current models, and (ii) the validity and diagnostic power of existing benchmarks. Our study is guided by the following research questions:

- **RQ1**: How well do current LLMs perform on prompt-to-RTL generation when evaluated without auxiliary agents, planning mechanisms, or external optimization?
- **RQ2**: To what extent do observed failures stem from intrinsic model limitations versus artifacts of benchmark design (e.g., incomplete prompts, incorrect or underspecified testbenches)?
- **RQ3**: Are existing benchmarks sufficiently well designed to provide a reliable and comprehensive measure of RTL generation capability, or do they require refinement to more accurately capture task challenges?

The results below analyze both axes, identifying where failures originate from models and where they arise from benchmark construction.

We focus on two widely used benchmarks: VerilogEval and RTLLM, each providing diverse RTL tasks with ground-truth implementations and testbenches for correctness checking.

To isolate model quality from system-level orchestration, we adopt two inference settings. The first is a *non-agentic baseline*: single prompt with temperature sampling, reporting pass@5 (i.e., success if any of five samples passes the testbench). The second is an *agentic ReAct-style* loop allowing up to 10 iterations of feedback from simulation to guide revisions. This approximates the outer loop of recent agentic frameworks while preserving a controlled evaluation interface.

For our evaluation, we consider commercially available frontier models rather than domain-specific fine-tuned systems. The models selected are **GPT-4.1** and its smaller, lower-cost variant **GPT-4.1-mini** (OpenAI), together with **Claude Sonnet 4** (Anthropic). These systems represent widely used general-purpose LLMs with strong capabilities in reasoning and structured code generation. Their pricing is summarized in Table 2. Other vendors such as Google (Gemini), Meta (LLaMA), and DeepSeek also provide competitive frontier models; however, the three chosen here offer a representative view of current capabilities in commercially accessible LLM-based code generation. Domain-specific RTL models such as VeriReason [55] and RTLCoder-v1.1 [49] are excluded, as our objective is to evaluate how far general-purpose models can be applied to specification-to-RTL tasks without additional fine-tuning.

Table 2. Model pricing (USD per 1M tokens). Prices fluctuate; shown here to contextualize cost at evaluation time.

Model	Input	Output
GPT-4.1	$2.00	$8.00
GPT-4.1-mini	$0.40	$1.60
Claude Sonnet 4	$3.00	$15.00

3.1. Benchmarks

VerilogEval [16]. The spec-to-RTL subset presents a flat list of problems, each pairing a text specification with a reference RTL implementation and a testbench. Testbenches exercise generated designs by simulating waveforms and comparing output signals to golden responses, enabling fine-grained temporal checking.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

RTLLM [17]. Tasks are grouped by circuit category (e.g., counters, multiplexers, adders), providing a structured taxonomy. Each task includes a natural-language specification, a hand-crafted reference RTL implementation, and a testbench with multiple input–output cases. RTLLM's testbenches primarily verify final-value correctness for the test cases rather than waveform-level behavior over time.

3.2. Tool Integration via Model Context Protocol (MCP)

The evaluation framework connects Icarus Verilog [57] to the LLM via MCP, a standardized interface for external tool calls by agents [58]. At the backend, an **MCP server** manages compilation and simulation tasks: it invokes Icarus with configurable timeouts, captures both stdout and stderr streams, and normalizes these into structured JSON responses that report status, outputs, and errors. At the frontend, an **MCP client** embedded in the agent runtime issues requests to this server through well-defined endpoints. This separation allows the agent to remain agnostic to tool invocation details while still receiving actionable, structured feedback. In practice, agentic runs invoke MCP endpoints to (i) compile generated RTL modules, (ii) execute corresponding testbenches, and (iii) collect error traces or assertion violations. These responses are fed back into the reasoning loop, enabling the agent to iteratively design, test, and refine candidate implementations until convergence or a maximum loop count is reached. While the current setup employs Icarus Verilog for simulation, the MCP abstraction makes it straightforward to extend the framework to other tools such as Verilator [59] for simulation, SymbiYosys [60] for formal verification, or Yosys [38] for synthesis, ensuring the architecture can evolve with broader evaluation needs.

3.3. Agentic System Architecture

The SystemVerilog generation framework is implemented as a stateful workflow in LangGraph [61], an open-source library for building multi-step LLM applications as explicit state machines. A directed graph coordinates two core stages: (i) candidate RTL generation and (ii) MCP-backed validation. Control flow routes based on pass/fail, enabling termination when tests succeed or iterative refinement when they fail. Context persists across iterations, combining the original prompt with compiler/simulation feedback. A reflection step proposes targeted edits, which seed the next generation attempt. LangGraph manages state transitions and loop control, enabling robust iterative improvement.

3.4. Results

We evaluate VerilogEval (spec-to-RTL) and RTLLM using their provided testbenches under the two settings above. Unless noted, temperature is fixed at 0.7. Models tested are GPT-4.1, GPT-4.1-mini, and Claude Sonnet 4. Tables 3–4 summarize VerilogEval results, and Table 5 summarizes RTLLM. Failures are highlighted in red. The *baseline* column reports the number of independent generations required to produce a functionally correct RTL design (i.e., one that passes all tests). The *agentic* column reports the number of feedback-driven refinement iterations needed to achieve a correct design.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

Table 3. Evaluation results for VerilogEval spec-to-rtl benchmark (Problem 001-075) with GPT 4.1, GPT 4.1-mini, and Sonnet 4 LLM models for both baseline and agentic approaches. Red indicates where the generated Verilog failed the tests. The *baseline* column reports the number of independent generations (pass @5) required to produce a functionally correct RTL design (i.e., one that passes all tests). The *agentic* column reports the number of feedback-driven refinement iterations (max 10) needed to achieve a correct design.

Name	GPT 4.1		GPT 4.1mini		Sonnet 4	
	Baseline	Agentic	Baseline	Agentic	Baseline	Agentic
Prob001_zero	1	1	1	1	1	1
Prob002_m2014_q4i	1	1	1	1	1	1
Prob003_step_one	1	1	1	1	1	1
Prob004_vector2	1	1	1	1	1	1
Prob005_notgate	1	1	1	1	1	1
Prob006_vectorr	1	1	3	2	1	1
Prob007_wire	1	1	1	1	1	1
Prob008_m2014_q4h	1	1	1	1	1	1
Prob009_popcount3	1	1	1	1	1	1
Prob010_mt2015_q4a	1	1	1	1	1	1
Prob011_norgate	1	1	1	1	1	1
Prob012_xnorgate	1	1	1	1	1	1
Prob013_m2014_q4e	1	1	1	1	1	1
Prob014_andgate	1	1	1	1	1	1
Prob015_vector1	1	1	1	1	1	1
Prob016_m2014_q4j	1	1	1	1	1	1
Prob017_mux2to1v	1	1	1	1	1	1
Prob018_mux256to1	1	1	1	1	1	1
Prob019_m2014_q4f	1	1	1	1	1	1
Prob020_mt2015_eq2	1	1	1	1	1	1
Prob021_mux256to1v	1	1	1	1	1	1
Prob022_mux2to1	1	1	1	1	1	1
Prob023_vector100r	1	1	5	2	1	1
Prob024_hadd	1	1	1	1	1	1
Prob025_reduction	1	1	1	1	1	1
Prob026_alwaysblock1	1	1	1	1	1	1
Prob027_fadd	1	1	1	1	1	1
Prob028_m2014_q4a	1	1	1	1	1	1
Prob029_m2014_q4g	1	1	1	1	1	1
Prob030_popcount255	1	1	1	1	1	1
Prob031_dff	4	1	1	1	1	1
Prob032_vector0	1	1	1	1	1	1
Prob033_ece241_2014_q1c	1	1	1	1	1	1
Prob034_dff8	5	2	5	2	5	2
Prob035_count1to10	1	1	1	1	1	1
Prob036_ringer	1	1	1	1	1	1
Prob037_review2015_count1k	1	1	1	1	1	1
Prob038_count15	1	1	1	1	1	1
Prob039_always_if	1	1	1	1	1	1
Prob040_count10	1	1	1	1	1	1
Prob041_dff8r	1	1	1	1	1	1
Prob042_vector4	1	1	1	1	1	1
Prob043_vector5	2	2	5	5	1	1
Prob044_vectorgates	1	1	1	1	1	1
Prob045_edgedetect2	2	1	1	2	1	1
Prob046_dff8p	1	1	1	1	1	1
Prob047_dff8ar	1	1	1	1	1	1
Prob048_m2014_q4c	1	1	1	1	1	1
Prob049_m2014_q4b	1	1	1	1	1	1
Prob050_kmap1	1	1	1	1	1	1
Prob051_gates4	1	1	1	1	1	1
Prob052_gates100	1	1	1	1	1	1
Prob053_m2014_q4d	5	2	5	2	5	2
Prob054_edgedetect	5	2	1	2	1	1
Prob055_conditional	1	1	1	1	1	1
Prob056_ece241_2013_q7	1	1	1	1	1	1
Prob057_kmap2	1	4	1	1	5	5
Prob058_alwaysblock2	1	1	1	1	1	1
Prob059_wire4	1	1	1	1	1	1
Prob060_m2014_q4k	1	1	1	1	1	1
Prob061_2014_q4a	1	1	1	1	1	1
Prob062_bugs_mux2	5	10	5	10	5	3
Prob063_review2015_shiftcount	1	3	5	10	1	1
Prob064_vector3	1	1	1	1	1	1
Prob065_7420	1	1	1	1	1	1
Prob066_edgecapture	5	2	5	3	5	5
Prob067_countslow	1	1	1	1	1	1
Prob068_countbcd	1	1	1	1	1	1
Prob069_truthtable1	1	1	1	1	1	1
Prob070_ece241_2013_q2	5	10	5	10	5	10
Prob071_always_casez	1	1	1	1	1	1
Prob072_thermostat	1	1	1	1	1	1
Prob073_dff16e	1	1	1	1	1	1
Prob074_ece241_2014_q4	5	3	1	1	5	2
Prob075_counter_2bc	1	1	1	1	1	1

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

Table 4. Evaluation results for VerilogEval spec-to-rtl benchmark (Problem 075-156)

Name	GPT 4.1		GPT 4.1mini		Sonnet 4	
	Baseline	Agentic	Baseline	Agentic	Baseline	Agentic
Prob076_always_case	1	1	1	1	1	1
Prob077_wire_decl	1	1	1	1	1	1
Prob078_dualedge	1	1	1	10	1	1
Prob079_fsm3onehot	1	1	1	1	1	1
Prob080_timer	1	2	1	1	1	1
Prob081_7458	1	1	1	1	1	1
Prob082_lfsr32	4	10	5	10	5	10
Prob083_mt2015_q4b	1	1	1	1	1	1
Prob084_ece241_2013_q12	1	3	1	1	5	2
Prob085_shift4	1	1	1	1	1	1
Prob086_lfsr5	5	3	5	10	5	3
Prob087_gates	1	1	1	1	1	1
Prob088_ece241_2014_q5b	1	1	1	1	1	1
Prob089_ece241_2014_q5a	5	5	5	10	5	10
Prob090_circuit1	1	1	1	1	1	1
Prob091_2012_q2b	1	1	2	1	1	1
Prob092_gatesv100	1	1	2	10	1	1
Prob093_ece241_2014_q3	5	10	5	10	5	10
Prob094_gatesv	1	2	1	1	3	2
Prob095_review2015_fsmshift	1	1	1	1	1	1
Prob096_review2015_fsmseq	5	4	4	1	1	1
Prob097_mux9to1v	1	1	1	1	1	1
Prob098_circuit7	1	1	1	1	1	1
Prob099_m2014_q6c	5	10	5	10	5	10
Prob100_fsm3comb	1	1	1	1	1	1
Prob101_circuit4	1	1	1	1	5	10
Prob102_circuit3	1	1	1	1	2	2
Prob103_circuit2	1	1	1	1	1	1
Prob104_mt2015_muxdff	5	2	5	2	5	2
Prob105_rotate100	1	1	1	1	1	1
Prob106_always_nolatches	1	1	1	1	1	1
Prob107_fsm1s	5	3	5	5	1	1
Prob108_rule90	1	1	1	1	1	1
Prob109_fsm1	5	3	5	5	1	1
Prob110_fsm2	1	3	1	4	1	1
Prob111_fsm2s	1	1	3	4	1	1
Prob112_always_case2	1	1	1	1	5	3
Prob113_2012_q1g	2	8	2	2	5	10
Prob114_bugs_case	1	1	1	1	1	1
Prob115_shift18	1	1	1	1	5	2
Prob116_m2014_q3	5	10	5	10	5	10
Prob117_circuit9	2	10	5	10	5	2
Prob118_history_shift	3	3	5	2	1	1
Prob119_fsm3	5	4	5	3	5	2
Prob120_fsm3s	5	4	5	5	5	2
Prob121_2014_q3bfsm	1	1	5	3	1	1
Prob122_kmap4	2	1	1	1	1	1
Prob123_bugs_addsubz	1	1	1	1	1	1
Prob124_rule110	5	1	1	1	5	2
Prob125_kmap3	5	10	1	4	3	7
Prob126_circuit6	1	1	1	1	1	1
Prob127_lemmings1	5	2	5	10	1	1
Prob128_fsm_ps2	3	10	5	10	5	5
Prob129_ece241_2013_q8	5	3	1	1	1	1
Prob130_circuit5	1	1	1	1	1	1
Prob131_mt2015_q4	1	1	1	1	5	2
Prob132_always_if2	1	1	1	1	1	1
Prob133_2014_q3fsm	5	10	1	10	5	4
Prob134_2014_q3c	1	1	1	1	1	1
Prob135_m2014_q6b	5	1	5	9	5	5
Prob136_m2014_q6	5	10	1	10	5	2
Prob137_fsm_serial	5	2	5	2	5	2
Prob138_2012_q2fsm	1	4	5	10	5	3
Prob139_2013_q2bfsm	5	10	5	10	5	3
Prob140_fsm_hdlc	5	10	5	10	5	3
Prob141_count_clock	5	10	5	10	5	10
Prob142_lemmings2	5	10	5	10	1	1
Prob143_fsm_onehot	1	1	4	1	1	1
Prob144_conwaylife	1	6	1	10	1	1
Prob145_circuit8	5	10	5	10	5	10
Prob146_fsm_serialdata	5	10	5	10	5	10
Prob147_circuit10	5	10	5	10	5	10
Prob148_2013_q2afsm	1	10	1	2	1	1
Prob149_ece241_2013_q4	5	10	5	10	5	10
Prob150_review2015_fsmonehot	1	1	5	2	1	1
Prob151_review2015_fsm	5	10	5	10	5	10
Prob152_lemmings3	5	10	5	10	5	3
Prob153_gshare	1	1	5	10	1	1
Prob154_fsm_ps2data	5	10	5	10	5	10
Prob155_lemmings4	5	10	5	3	5	10
Prob156_review2015_fancytimer	5	10	5	10	5	10

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

Table 5. Evaluation results for RTLLM benchmark with GPT 4.1, GPT 4.1-mini, and Sonnet 4 LLM models for both baseline and agentic approaches. Red indicates where the generated Verilog failed the tests. The *baseline* column reports the number of independent generations (pass @5) required to produce a functionally correct RTL design (i.e., one that passes all tests). The *agentic* column reports the number of feedback-driven refinement iterations (max 10) needed to achieve a correct design.

Name	GPT 4.1		GPT 4.1mini		Sonnet 4	
	Baseline	Agentic	Baseline	Agentic	Baseline	Agentic
Arithmetic						
comparator_4bit	2	1	1	1	5	3
comparator_3bit	1	1	1	1	1	1
accumulator	1	1	1	1	1	1
adder_pipe_64bit	3	1	1	1	1	1
adder_bcd	1	1	1	1	1	1
adder_16bit	1	1	1	1	1	1
adder_32bit	1	1	1	1	1	1
adder_8bit	1	1	1	1	1	1
sub_64bit	1	1	1	1	1	1
multi_8bit	1	1	1	1	1	1
multi_booth_8bit	1	1	1	1	1	1
multi_pipe_8bit	2	4	5	9	3	2
multi_pipe_4bit	1	1	1	10	1	1
multi_16bit	1	1	1	1	1	1
fixed_point_sub	2	1	1	1	1	1
fixed_point_adder	2	1	1	1	1	1
float_multi	1	1	5	10	4	1
div_16bit	1	1	1	1	1	1
radix2_div	5	10	5	4	5	10
Control						
sequence_detector	5	2	5	2	5	2
fsm	1	1	3	10	1	1
ring_counter	1	1	1	1	1	1
JC_counter	1	1	1	1	1	1
up_down_counter	1	1	1	1	1	1
counter_12	1	1	1	1	1	1
Memory						
asyn_fifo	2	10	4	10	1	1
LIFObuffer	1	1	3	2	1	1
LFSR	1	1	1	1	1	1
barrel_shifter	1	1	1	1	2	1
right_shifter	1	1	1	1	1	1
Miscellaneous						
freq_div	1	1	1	1	1	1
freq_divbyeven	1	1	1	1	1	1
freq_divbyfraq	1	1	1	1	1	1
freq_divbyodd	1	1	1	1	1	1
signal_generator	5	2	5	3	5	10
square_wave	1	1	1	1	1	
edge_detect	5	2	1	3	1	
calendar	1	1	1	1	1	
serial2parallel	5	1	1	1	2	1
synchronizer	1	1	1	1	1	1
pulse_detect	1	1	1	10	1	1
width_8to16	1	1	1	1	1	1
traffic_light	5	2	5	10	5	2
parallel2serial	1	1	5	10	1	1
RISC-V/ROM	1	1	1	1	2	1
RISC-V/clkgen	1	1	1	1	1	1
RISC-V/alu	1	1	1	10	1	1
RISC-V/instr_reg	1	1	1	1	1	1
RISC-V/RAM	1	1	1	1	1	1
RISC-V/pe	1	1	2	10	1	1

On VerilogEval, GPT-4.1 improved from 75.00% (baseline) to 85.26% (agentic). GPT-4.1-mini rose from 73.08% to 80.77%. Claude Sonnet 4 showed the largest gain, from 73.72% to 89.74%.

On RTLLM, GPT-4.1 improved from 88.24% to 96.08%. Claude Sonnet 4 matched this improvement (90.20% $\rightarrow$ 96.08%). GPT-4.1-mini exhibited strong baseline performance at 90.20%, though its agentic run decreased to 82.35%.

Overall, agentic refinement substantially improves reliability for most model–benchmark pairs, with particularly pronounced gains on VerilogEval. RTLLM results indicate that one-shot generation is already competitive for many tasks, narrowing the headroom for iterative loops.

To contextualize our findings, we compare with recent reports. Pinkney et al. [16] a 62.6% pass rate on VerilogEval spec-to-RTL using GPT-4o with in-context learning alone. Autosilicon [37] reports 73.8% for GPT-4 on VerilogEval spec-to-RTL and similar RTLLM results. VeriReason reports higher scores—

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

83% on VerilogEval spec-to-RTL with GPT-4-Turbo and 83.1% with VeriReason-Qwen2.5-7B—via supervised fine-tuning and reinforcement learning.

In contrast, our results show that newer foundation models (GPT-4.1, Claude Sonnet 4), coupled with a lightweight reflection loop, can meet or exceed these numbers on VerilogEval (up to 89.74%) and RTLLM (up to 96.08%) *without* multi-agent orchestration, persistent memory, or specialized testbench generation. This suggests a shift in the evaluation landscape: stronger base models plus minimal iterative refinement can match more elaborate pipelines, raising the bar for benchmark design and analysis going forward.

4. Analysis and Discussion

The quantitative results in the previous section show that newer foundation models, when paired with a lightweight agentic refinement loop, achieve strong performance on both VerilogEval and RTLLM. However, aggregate pass rates do not explain *why* failures persist or which mistakes dominate. A closer analysis of failures provides insight into current LLM limitations for RTL generation, the kinds of corrections needed in benchmark prompts/testbenches, and systematic error patterns across designs.

We first examine RTLLM, where several prompt/testbench adjustments were required and where the `radix-2 divider` remained unsolved despite repeated attempts. We then turn to VerilogEval, focusing on `Prob154_fsm_ps2data` as a representative failure case in FSM reasoning. Building on these, we synthesize recurring error patterns and close with recommendations toward reliable spec-to-RTL generation.

4.1. RTLLM

During RTLLM evaluation, we made targeted modifications to align prompts, reference designs, and the simulation harness. These addressed inconsistencies in text descriptions, missing parameters, and testbench behavior. Examples include clarifying parameterization in the 64-bit pipelined adder; adding the missing `res_ready` signal in the radix-2 divider prompt; toggling display statements in FIFO modules to aid reflection; fixing module-name mismatches in frequency dividers; adjusting the ring-counter testbench loop bounds; clarifying shift vs. rotate semantics in the barrel shifter; and correcting instantiation order in the LFSR testbench. These adjustments improved benchmark robustness and enabled systematic reflection across iterations, with an overall success rate of **96.08%**

Despite these changes, `radix2_div` failed across models and settings. Dominant error sources included:

Specification ambiguity: Prompts leave degrees of freedom (restoring vs. non-restoring, tie-breaking, don't-care handling), and models often pick a valid but incompatible variant.
Cycle accuracy: Precise per-cycle sequencing (shift–subtract–restore, final commit) is required; off-by-one or misordered steps corrupt results.
Bit-width arithmetic: Restoring division needs a 9-bit remainder path and borrow detection; models frequently implement 8-bit compares/subtracts.
Finalization: The last quotient bit and final remainder must be explicitly committed; this step is often omitted.
Sign rules: Signed division requires $|a|/|b|$ internally, quotient sign via $a \oplus b$, and remainder sign matching the dividend; frequently mishandled.
Handshake logic: Misuse of `res_ready/res_valid` and level vs. edge confusion produce deadlocks or stale outputs.
State control: Incorrect sequencing of `start/active/done` leads to premature termination or extra cycles.
Edge cases: Divide-by-zero, $0/N$, N/N are often unspecified or incorrectly handled.
Syntax vs. semantics: Syntactically valid code with semantic errors (e.g., blocking in sequential logic, replacing borrow-detect with simple compare).

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

Lack of internal verification: No internal symbolic trace or truth-table reasoning to catch subtle algorithmic mistakes during generation.

Overall, division circuits stress algorithmic rigor, cycle-level timing, and careful state/handshake control—areas where current models most often stumble.

4.2. VerilogEval

On VerilogEval spec-to-RTL, the overall success rate reached **89.74%**. A representative failure is `Prob154_fsm_ps2data`, which requires detecting message boundaries in a byte stream and emitting valid 3-byte frames with a one-cycle `done` pulse after the third byte.

The reference solution uses a four-state FSM (BYTE1 $\rightarrow$ BYTE2 $\rightarrow$ BYTE3 $\rightarrow$ DONE), a shift register to collect bytes, and asserts `done` only in the DONE state. Model outputs consistently deviated by asserting `done` combinationally whenever `in[3]==1`, and by outputting the sliding window {`byte2`, `byte1`, `in`} without state tracking. This caused:

- **Protocol violation:** `done` became level-sensitive, not a one-cycle pulse after three valid bytes.
- **Frame misalignment:** Without explicit states, frame boundaries drifted, yielding frequent cycle-shifted outputs at valid times.

In the testbench, this produced widespread mismatches (thousands of incorrect `done` cycles; hundreds of output miscompares at valid times). The case highlights a broader trend: models often succeed on combinational logic but falter on temporal protocols requiring explicit FSM reasoning and gated validity.

4.3. Observations on Errors Made by LLMs

We observe recurring error classes spanning protocol handling, control sequencing, and RTL semantics:

Transition priority mistakes: Missing explicit precedence for mutually exclusive conditions (e.g., read vs. write) yields nondeterministic behavior.
Moore vs. Mealy confusion: Outputs meant to depend on registered state are derived combinationally from inputs, causing glitches.
Over/under-registering signals: Poor register placement adds latency or instability (e.g., exposing next-state directly as output).
Missing hysteresis/invariants: Lack of memory of past events (e.g., debouncing) leads to flapping outputs.
Boundary/off-by-one errors: Incorrect thresholding ($>$ vs. $\geq$) or counter endpoints violate protocols.
Handshake timing drift: Valid/ready asserted one cycle early/late desynchronizes modules; overloading a single signal (e.g., `done`) conflates phases.
Event vs. level detection: Edge detection used where level is required (or vice versa) causes missed/repeated triggers.
Asymmetry violations: Treating asymmetric inputs as symmetric breaks intended control policies.
Algebraic incompleteness: Dropping terms in SOP/POS simplifications passes some tests but fails corners.
Output-validity contract ignored: Driving outputs outside validity windows risks latching garbage downstream.
Error-path underspecification: No clear recovery/rollback sequence leads to repeated faults.
Bit-ordering/shift-direction mistakes: MSB/LSB swaps and left/right confusion misalign data.
Reset semantics drift: Wrong polarity or missing init causes invalid post-reset states.
Blocking vs. non-blocking misuse: Using = in sequential logic (or mixing with <=) induces races and sim/synth mismatches.
State-space oversimplification: Collapsing required states skips timing gaps and induces premature transitions.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

These patterns show that while models produce syntactically valid RTL, they often miss temporal correctness, state consistency, and interface contracts—precisely the aspects that matter for reliable hardware. Table 6 summarizes these observations.

Table 6. Common error patterns observed in LLM-generated RTL designs.

Error Type	Description	Consequence
Transition priority	Ambiguous handling of mutually exclusive conditions	Nondeterminism, conflicts
Moore/Mealy confusion	Outputs derived from inputs instead of registered state	Glitches, unstable signals
Over/under-registering	Misplaced pipeline/register boundaries	Extra latency or instability
Missing hysteresis	No memory of past events (e.g., debounce)	Flapping outputs
Boundary/off-by-one	Incorrect counter/timer comparisons	Protocol violations by ± 1 cycle
Handshake drift	Misaligned valid/ready	Desynchronization, data loss
Event vs. level	Edge used where level needed (or vice versa)	Missed/repeated triggers
Asymmetry violation	Treating asymmetric inputs as symmetric	Wrong transition gating
Algebraic incompleteness	Incorrect Boolean simplification	Corner-case failures
Output-validity contract	Driving outputs outside valid window	Downstream latches invalid data
Error-path underspecified	No recovery after fault	Re-triggered errors
Bit-ordering/shifts	MSB/LSB swaps, wrong shift direction	Misaligned/corrupted data
Reset semantics drift	Wrong polarity/init	Invalid post-reset state
Blocking/non-blocking	= in sequential or mixing with <=	Races, sim/synth mismatch
State-space oversimplified	Missing required FSM states	Premature transitions

4.4. Future Work: Toward Reliable Spec-to-RTL Generation

The recurring errors observed in current LLM-generated RTL suggest several avenues for advancing research:

Contract-based synthesis: Embedding explicit temporal and structural contracts into the design process can ensure that generated control logic respects precedence rules, handshake timing, and reset semantics by construction.
Schema-driven specifications: Capturing I/O, signal roles, invariants, and protocol semantics in machine-checkable schemas can reduce ambiguity and provide stronger guardrails against misinterpretation during generation.
Counterexample-guided refinement: Leveraging simulation and formal verification traces to drive iterative repair can help address subtle corner cases such as off-by-one errors, algebraic incompleteness, and state-space underspecification.
Automated protocol and invariant checkers: Integrating lightweight protocol monitors and invariant analyzers into the workflow can catch timing drift, invalid outputs, and missing recovery sequences early in the design loop.
Persistent error memory: Maintaining a record of past errors and their fixes in a structured knowledge base can help prevent recurrence and enable models to learn corrective patterns over time.
Symbolic and structural reasoning aids: Combining language models with symbolic execution, AST-based debugging, and circuit-graph reasoning may improve robustness in handling concurrency, bit-level manipulations, and complex state machines.

Taken together, these directions point toward a systematic methodology in which specification clarity, automated checks, and iterative refinement loops converge to reduce the brittleness of current approaches and move closer to trustworthy specification-to-RTL automation.

5. Future Work: Toward Natural-Language-to-SoC Design

The long-term vision of AI-native EDA is to specify entire systems-on-chip (SoCs) in natural language and generate correct, optimized, and verifiable hardware-software platforms. While mature tol flows exist from register-transfer level (RTL) to GDSII [3,7], the NL-to-RTL frontier remains fragmented and far from full automation. Bridging this gap requires tight integration of language models, specification formalisms, verification frameworks, and physical design flows, with AI embedded across all stages.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

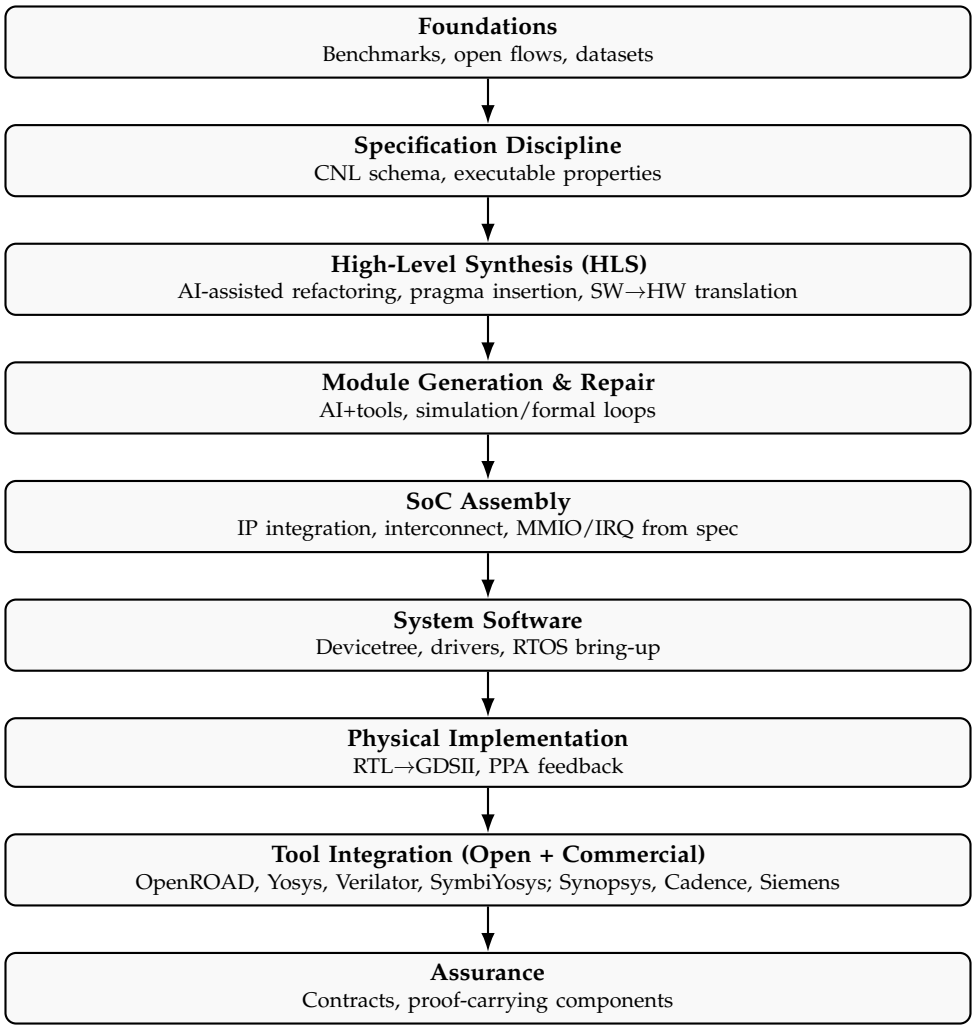

Figure 1. Research roadmap for natural-language-to-SoC design, spanning benchmarks, specification, HLS, RTL generation, SoC assembly, system software, physical implementation, tool integration, and assurance.

5.1. Current Landscape and Gaps

EDA has steadily raised the level of abstraction—from RTL to transaction-level modeling and high-level synthesis (HLS). Emerging AI capabilities extend this trajectory by generating RTL from textual descriptions, repairing errors via compile–simulate loops, and exploring early power, performance, and area (PPA) trade-offs. Key gaps persist: (i) ambiguity in natural-language specs (missing corner cases, performance constraints, protocol definitions), (ii) datasets that rarely link spec→RTL→verification→PPA, (iii) verification bottlenecks—functional correctness on small modules does not scale without modular, contract-based methods, (iv) limited automation of IP integration, clocks/resets, and interconnect from a single authoritative spec, (v) weak feedback from PPA closure to earlier architectural choices, and (vi) manual derivation of memory maps, interrupt configurations, and driver stubs even when RTL generation succeeds.

5.2. Open Foundations

Progress hinges on shared foundations. Recent spec-to-RTL suites (e.g., ArchXBench [25] [62]) introduce realistic datapaths and scalable harnesses; future suites should extend to subsystems (processor pipelines, crypto accelerators, interconnect fabrics) and eventually full SoC templates. Benchmarks should link specifications, RTL, verification artifacts, and physical-design metrics for end-to-end assessment. Open flows integrating front-end spec, RTL, synthesis, and place-and-route are critical for reproducibility and access. Integration with Yosys [38], Verilator [59], SymbiYosys [60], and OpenRoad [7] provides a baseline; commercial tools remain important for realism and industry transfer.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

5.3. Specification Discipline

Free-form prose is insufficient for automation. Controlled natural language (CNL) schemas should capture I/O protocols, timing, invariants, memory maps, arbitration policies, and IRQ assignments. Specs should be executable, embedding assertions, protocol traces, and partial testbenches. Well-structured CNLs can serve as the single source of truth for both hardware and software.

5.4. AI-Assisted Generation and Repair

At module scale, LLMs must pair with simulation and formal feedback to iteratively repair designs. At subsystem/SoC scale, generation must be hierarchical, decomposing specs into interacting modules and integrating them into larger fabrics. Synthesis and physical-design feedback should inform early architectural decisions (e.g., pipelining depth, parallelism, memory organization). Promising directions include multi-agent orchestration, retrieval-augmented design memories, and reinforcement learning on PPA signals.

5.5. SoC Assembly and Software Interfaces

Automating SoC assembly requires generators that derive interconnects, memory maps, and interrupt tables directly from spec. Correct-by-construction glue logic and parameterized IP instantiation should be synthesized alongside interface metadata for software. The same spec should yield device-tree entries, driver skeletons, and OS/RTOS bring-up collateral, enabling early HW–SW co-validation (e.g., Zephyr RTOS or embedded Linux).

5.6. Physical Design and PPA-Informed Feedback

RTL-to-GDSII flows are mature but rarely feed back upstream. Future work should explore PPA-in-the-loop design, where synthesis/place-and-route metrics guide architectural choices. Lightweight ML surrogates for backend flows can enable interactive, constraint-aware exploration. Open backends (OpenLane [3]/OpenRoad [7]) offer research-friendly baselines; commercial flows help validate transfer.

5.7. Assurance and Proof-Carrying Design

Assurance at SoC scale demands modular contracts and compositional reasoning. Priorities include expanding property libraries for standard protocols, defining reusable contracts for IP, and exploring proof-carrying hardware where designs ship with machine-checkable certificates. Trust in AI-generated artifacts also requires provenance tracking, semantic differencing, and automated equivalence checking between generated and golden models.

5.8. Outlook

We envision a flow where designers specify intent in precise yet accessible forms, and AI-native tools generate, refine, and assure full SoC designs—including system software—under realistic PPA constraints. Achieving this requires community investment in benchmarks, open flows, and modular tooling, progressively scaling from modules to subsystems to full SoCs. The outcome would shift the bottleneck from manual RTL authoring to higher-level architectural and application-driven decisions.

6. Conclusion

This paper presented a survey and evaluation of large language models (LLMs) for specification-to-RTL (spec-to-RTL) generation, synthesizing findings from twenty-six recent works and benchmarking frontier models on VerilogEval and RTLLM. Our review highlights the rapid evolution of the field: early efforts relied on supervised fine-tuning over GitHub-style datasets, while more recent approaches embrace multi-agent orchestration, retrieval-augmented prompting, reinforcement learning, and post-RTL metrics. Alongside these methods, new benchmarks such as ArchXBench are beginning to push beyond toy problems toward realistic datapaths and subsystem-level complexity.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

Our experimental evaluation shows that frontier commercial models such as GPT-4.1 and Claude Sonnet 4, when paired with lightweight reflection loops, achieve strong performance on VerilogEval (up to 89.74%) and RTLLM (up to 96.08%). These results approach or surpass those of domain-specific fine-tuned pipelines, highlighting both the strength of today's frontier models and the need for richer benchmarks. Case studies on the radix-2 divider (RTLLM) and FSM detection task (VerilogEval) reveal recurring LLM failure modes in algorithmic sequencing, temporal reasoning, and FSM design. Systematic error patterns—including transition-priority mistakes, handshake drift, blocking vs. non-blocking misuse, and state-space oversimplification—illustrate the challenges of applying natural-language-driven generation to hardware with tight concurrency and protocol constraints.

From this analysis, several lessons emerge. First, benchmark design must evolve: underspecified prompts and brittle testbenches obscure true capability and hinder progress. Second, structured specification methods, combined with systematic debugging aids such as AST- or graph-based reasoning, offer practical ways to address recurring classes of design errors. Third, evaluation must expand beyond functional correctness to include post-RTL metrics such as power, performance, and area (PPA), bridging the gap between correctness and design quality. Finally, moving from module-level generation to SoC-scale automation will require advances in formal specification languages, hierarchical composition techniques, and scalable assurance frameworks.

Looking ahead, we outlined a research roadmap toward natural-language-to-SoC design. Key priorities include developing controlled natural language schemas for specifications, building open and reproducible end-to-end flows, incorporating PPA feedback into early design stages, and advancing modular contracts for assurance. Bridging RTL generation with SoC assembly, system software interfaces, and physical implementation will require both open-source and commercial tool integration, as well as new benchmarks that capture subsystem- and system-level complexity.

References

1. Flynn, M.J.; Luk, W. *Computer system design: system-on-chip*; John Wiley & Sons, 2011.
2. Dally, W.J.; Turakhia, Y.; Han, S. Domain-specific hardware accelerators. *Communications of the ACM* **2020**, *63*, 48–57.
3. The OpenROAD Project and contributors. OpenLane. GitHub repository, 2025. An automated RTL to GDSII flow.
4. Synopsys, Inc.. Fusion Compiler. Synopsys, Inc., 2024. Accessed: September 13, 2025.
5. Abdelazeem, A. Cadence-RTL-to-GDSII-Flow. GitHub, 2023.
6. Siemens EDA. Aprisa. Siemens EDA, 2025. Accessed: September 13, 2025.
7. The OpenROAD Project and contributors. OpenROAD: Open-Source RTL-to-GDSII Flow. GitHub repository, 2025. An open-source RTL-to-GDSII flow for semiconductor digital design.
8. Matarazzo, A.; Torlone, R. A survey on large language models with some insights on their capabilities and limitations. *arXiv preprint arXiv:2501.04040* **2025**.
9. Schulhoff, S.; Ilie, M.; Balepur, N.; Kahadze, K.; Liu, A.; Si, C.; Li, Y.; Gupta, A.; Han, H.; Schulhoff, S.; et al. The prompt report: a systematic survey of prompt engineering techniques. *arXiv preprint arXiv:2406.06608* **2024**.
10. Schulhoff, S.; Ilie, M.; Balepur, N.; Kahadze, K.; Liu, A.; Si, C.; Li, Y.; Gupta, A.; Han, H.; Schulhoff, S.; et al. The prompt report: a systematic survey of prompt engineering techniques. *arXiv preprint arXiv:2406.06608* **2024**.
11. He, J.; Treude, C.; Lo, D. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Transactions on Software Engineering and Methodology* **2025**, *34*, 1–30.
12. Bandi, A.; Kongari, B.; Naguru, R.; Pasnoor, S.; Vilipala, S.V. The Rise of Agentic AI: A Review of Definitions, Frameworks, Architectures, Applications, Evaluation Metrics, and Challenges. *Future Internet* **2025**, *17*, 404.
13. Yang, H.; Zhang, Y.; Xu, J.; Lu, H.; Heng, P.A.; Lam, W. Unveiling the generalization power of fine-tuned large language models. *arXiv preprint arXiv:2403.09162* **2024**.
14. Qu, C.; Dai, S.; Wei, X.; Cai, H.; Wang, S.; Yin, D.; Xu, J.; Wen, J.R. Tool learning with large language models: A survey. *Frontiers of Computer Science* **2025**, *19*, 198343.
15. Srivastava, S.S.; Aggarwal, V. A Technical Survey of Reinforcement Learning Techniques for Large Language Models. *arXiv preprint arXiv:2507.04136* **2025**.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

16. Pinckney, N.; Batten, C.; Liu, M.; Ren, H.; Khailany, B. Revisiting verilogeval: A year of improvements in large-language models for hardware code generation. *ACM Transactions on Design Automation of Electronic Systems* **2025**.
17. Lu, Y.; Liu, S.; Zhang, Q.; Xie, Z. Rtllm: An open-source benchmark for design rtl generation with large language model. In Proceedings of the 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2024, pp. 722–727.
18. Pan, J.; Zhou, G.; Chang, C.C.; Jacobson, I.; Hu, J.; Chen, Y. A survey of research in large language models for electronic design automation. *ACM Transactions on Design Automation of Electronic Systems* **2025**, *30*, 1–21.
19. Xu, K.; Schwachhofer, D.; Blocklove, J.; Polian, I.; Domanski, P.; Pflüger, D.; Garg, S.; Karri, R.; Sinanoglu, O.; Knechtel, J.; et al. Large Language Models (LLMs) for Electronic Design Automation (EDA). *arXiv preprint arXiv:2508.20030* **2025**.
20. Wang, Y.; Ye, W.; He, Y.; Chen, Y.; Qu, G.; Li, A. MCP4EDA: LLM-Powered Model Context Protocol RTL-to-GDSII Automation with Backend Aware Synthesis Optimization. *arXiv preprint arXiv:2507.19570* **2025**.
21. Li, J.; Wong, S.Z.; Wan, G.W.; Wang, X.; Yang, J. EDA-Debugger: An LLM-Based Framework for Automated EDA Runtime Issue Resolution. In Proceedings of the 2025 26th International Symposium on Quality Electronic Design (ISQED). IEEE, 2025, pp. 1–7.
22. Wu, H.; He, Z.; Zhang, X.; Yao, X.; Zheng, S.; Zheng, H.; Yu, B. Chateda: A large language model powered autonomous agent for eda. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **2024**, *43*, 3184–3197.
23. Lu, Y.; Au, H.I.; Zhang, J.; Pan, J.; Wang, Y.; Li, A.; Zhang, J.; Chen, Y. AutoEDA: Enabling EDA Flow Automation through Microservice-Based LLM Agents. *arXiv preprint arXiv:2508.01012* **2025**.
24. Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; Cao, Y. ReAct: Synergizing Reasoning and Acting in Language Models. In Proceedings of the International Conference on Learning Representations (ICLR), 2023.
25. Purini, S.; Garg, S.; Gaur, M.; Bhat, S.; Ravindran, A. ArchXBench: A Complex Digital Systems Benchmark Suite for LLM Driven RTL Synthesis. In Proceedings of the Proceedings of the 7th ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD), Santa Cruz, California, USA, 2025.
26. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* **2022**, *35*, 27730–27744.
27. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.L.; Lample, G.; Lavril, T.; Izacard, G.; Kossakowski, K.; et al. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* **2023**.
28. Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* **2024**.
29. Team, G.; et al. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805* **2023**.
30. Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; et al. Qwen Technical Report. *arXiv preprint arXiv:2309.16609* **2023**.
31. Sami, H.; Gaillardon, P.E.; Tenace, V.; et al. Aivril: Ai-driven rtl generation with verification in-the-loop. *arXiv preprint arXiv:2409.11411* **2024**.
32. Cadence Design Systems. Jasper Formal Property Verification App. Website, 2024. Accessed: 2025-09-15.
33. Yan, Z.; Fang, W.; Li, M.; Li, M.; Liu, S.; Xie, Z.; Zhang, H. Assertllm: Generating hardware verification assertions from design specifications via multi-llms. In Proceedings of the Proceedings of the 30th Asia and South Pacific Design Automation Conference, 2025, pp. 614–621.
34. Qiu, R.; Zhang, G.L.; Drechsler, R.; Schlichtmann, U.; Li, B. Autobench: Automatic testbench generation and evaluation using llms for hdl design. In Proceedings of the Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD, 2024, pp. 1–10.
35. Thakur, S.; Blocklove, J.; Pearce, H.; Tan, B.; Garg, S.; Karri, R. Autochip: Automating hdl generation using llm feedback. *arXiv preprint arXiv:2311.04887* **2023**.
36. Gao, M.; Zhao, J.; Lin, Z.; Ding, W.; Hou, X.; Feng, Y.; Li, C.; Guo, M. Autovcoder: A systematic framework for automated verilog code generation using llms. In Proceedings of the 2024 IEEE 42nd International Conference on Computer Design (ICCD). IEEE, 2024, pp. 162–169.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

37. Li, C.; Chen, C.; Pan, Y.; Xu, W.; Liu, Y.; Chang, K.; Wang, Y.; Wang, M.; Wang, Y.; Li, H.; et al. Autosilicon: Scaling up rtl design generation capability of large language models. *ACM Transactions on Design Automation of Electronic Systems* **2025**.
38. Wolf, C.; the Yosys Community. Yosys Open Synthesis Suite (Yosys). GitHub repository, 2025. Framework for Verilog RTL synthesis. Accessed: 2025-09-15.
39. Pei, Z.; Zhen, H.L.; Yuan, M.; Huang, Y.; Yu, B. Betterv: Controlled verilog generation with discriminative guidance. *arXiv preprint arXiv:2402.03375* **2024**.
40. Collini, L.; Garg, S.; Karri, R. C2HLSC: Leveraging large language models to bridge the software-to-hardware design gap. *ACM Transactions on Design Automation of Electronic Systems* **2024**.
41. Liu, M.; Ene, T.D.; Kirby, R.; Cheng, C.; Pinckney, N.; Liang, R.; Alben, J.; Anand, H.; Banerjee, S.; Bayraktaroglu, I.; et al. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176* **2023**.
42. Qiu, R.; Zhang, G.L.; Drechsler, R.; Schlichtmann, U.; Li, B. Correctbench: Automatic testbench generation with functional self-correction using llms for hdl design. In Proceedings of the 2025 Design, Automation & Test in Europe Conference (DATE). IEEE, 2025, pp. 1–7.
43. Xiong, C.; Liu, C.; Li, H.; Li, X. Hlspilot: Llm-based high-level synthesis. In Proceedings of the Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, 2024, pp. 1–9.
44. Reddy, E.B.E.; Bhattacharyya, S.; Sarmah, A.; Nongpoh, F.; Maddala, K.; Karfa, C. Lhs: Llm assisted efficient high-level synthesis of deep learning tasks. *ACM Transactions on Design Automation of Electronic Systems* **2025**.
45. Orenes-Vera, M.; Martonosi, M.; Wentzlaff, D. Using llms to facilitate formal verification of rtl. *arXiv preprint arXiv:2309.09437* **2023**.
46. Zhao, Y.; Zhang, H.; Huang, H.; Yu, Z.; Zhao, J. Mage: A multi-agent engine for automated rtl code generation. *arXiv preprint arXiv:2412.07822* **2024**.
47. Sun, W.; Li, B.; Zhang, G.L.; Yin, X.; Zhuo, C.; Schlichtmann, U. Paradigm-based automatic hdl code generation using llms. In Proceedings of the 2025 26th International Symposium on Quality Electronic Design (ISQED). IEEE, 2025, pp. 1–8.
48. Batten, C.; Pinckney, N.; Liu, M.; Ren, H.; Khailany, B. Pyhdl-eval: An llm evaluation framework for hardware design using python-embedded dsls. In Proceedings of the Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD, 2024, pp. 1–17.
49. Liu, S.; Fang, W.; Lu, Y.; Wang, J.; Zhang, Q.; Zhang, H.; Xie, Z. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **2024**.
50. Tsai, Y.; Liu, M.; Ren, H. Rtlfixer: Automatically fixing rtl syntax errors with large language model. In Proceedings of the Proceedings of the 61st ACM/IEEE Design Automation Conference, 2024, pp. 1–6.
51. Allam, A.; Shalan, M. Rtl-repo: A benchmark for evaluating llms on large-scale rtl design projects. In Proceedings of the 2024 IEEE LLM Aided Design Workshop (LAD). IEEE, 2024, pp. 1–5.
52. Huang, H.; Lin, Z.; Wang, Z.; Chen, X.; Ding, K.; Zhao, J. Towards llm-powered verilog rtl assistant: Self-verification and self-correction. *arXiv preprint arXiv:2406.00115* **2024**.
53. Thakur, S.; Ahmad, B.; Pearce, H.; Tan, B.; Dolan-Gavitt, B.; Karri, R.; Garg, S. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems* **2024**, *29*, 1–31.
54. Tasnia, K.; Garcia, A.; Farheen, T.; Rahman, S. VeriOpt: PPA-Aware High-Quality Verilog Generation via Multi-Role LLMs. *arXiv preprint arXiv:2507.14776* **2025**.
55. Wang, Y.; Sun, G.; Ye, W.; Qu, G.; Li, A. VeriReason: Reinforcement Learning with Testbench Feedback for Reasoning-Enhanced Verilog Generation. *arXiv preprint arXiv:2505.11849* **2025**.
56. Ho, C.T.; Ren, H.; Khailany, B. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. In Proceedings of the Proceedings of the AAAI Conference on Artificial Intelligence, 2025, Vol. 39, pp. 300–307.
57. Williams, S. Icarus Verilog. GitHub repository, 2023. Open-source Verilog compiler. Accessed: 2025-09-15.
58. Anthropic Team. Model Context Protocol (MCP), 2024. Initial release in November 2024. Accessed on 2025-09-09.
59. Snyder, W.; the Verilator Community. Verilator: Fast Verilog/SystemVerilog Simulator. GitHub repository, 2025. Open-source SystemVerilog simulator and lint system. Accessed: 2025-09-15.
60. YosysHQ.; contributors. SymbiYosys (sby). GitHub repository, 2025. Front-end for Yosys-based formal verification flows. Accessed: 2025-09-15.
61. LangChain-ai. LangGraph: A library for building robust and stateful multi-actor applications with LangChain. GitHub repository, 2024. Accessed: 2025-09-15.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 September 2025 doi:10.20944/preprints202509.1681.v1

62. Purini, S.; Garg, S.; Gaur, M.; Bhat, S.; Ravindran, A. ArchXBench: A Complex Digital Systems Benchmark Suite for LLM Driven RTL Synthesis, 2025.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.