

Recursive Language Models for RTL Code Generation

Eduard Anton
McGill University
Montreal, Canada

Abstract

Large Language Models (LLMs) show promise for RTL code generation but are limited by hallucinations and context rot in multi-step reasoning. We present a staged, self-correcting framework based on a Recursive Language Model (RLM) workbench that isolates context and decomposes generation into structured phases (Plan $\rightarrow$ Contract $\rightarrow$ Code $\rightarrow$ Verify). Our approach combines zero-footprint orchestration with uncertainty-driven gating and spec-aware verification to trigger targeted correction only when needed. Evaluated on Verilog-Eval [7], our method achieves 83.2 pass@1 and 88.6 pass@5 with Claude Haiku 4.5, outperforming monolithic and prior agentic approaches. These results highlight the importance of controlled context and structured verification for reliable RTL generation.

1 Introduction

The automatic generation of Register Transfer Level (RTL) code using Large Language Models (LLMs) is an increasingly active research area, driven by the promise of accelerating hardware design workflows. However, generating correct RTL remains fundamentally challenging. Hardware description languages such as Verilog encode concurrency, timing semantics, and structural constraints that differ significantly from sequential programming paradigms [14]. Correct synthesis, therefore, requires globally consistent decisions across long specifications, where small local errors can produce functionally incorrect circuits despite syntactic validity.

A central limitation of current LLM-based approaches is hallucination, which leads to both syntactic errors and subtle logical design flaws. Two key factors exacerbate these failures. First, the availability of high-quality, human-written RTL datasets is limited, forcing many approaches to rely on synthetic or AI-generated benchmarks for training and evaluation [12]. Second, modern generation strategies, including retrieval-augmented generation (RAG) [6], chain-of-thought (CoT) [15], and agentic pipelines, depend on increasingly long and complex contexts. While these methods improve reasoning, they introduce context rot, where irrelevant, redundant, or noisy intermediate information accumulates and degrades generation quality over time, intensifying hallucinations [16].

Existing mitigation strategies, such as context pruning or summarization, are inherently lossy and do not fully address the underlying issue with context rot [17]. As a result,

current systems struggle to maintain consistency and correctness across multi-step reasoning processes, particularly in long-horizon tasks such as RTL synthesis. This creates a fundamental reliability limitation to the effectiveness of LLMs in hardware design.

In this work, we present a staged, self-correcting agentic generation framework for Verilog synthesis. Our approach separates planning, specification formalization, code generation, and verification into distinct phases, while triggering recursive refinement only when the model detects high uncertainty. We introduce a zero-footprint orchestration design that prevents context accumulation at the root level, a calibrated uncertainty mechanism that triggers correction, and a spec-aware self-verifier that detects semantic inconsistencies in the generation phases. We evaluate our system on Verilog synthesis benchmarks using a lightweight worker model, demonstrating that structured verification and controlled context usage outperform monolithic generation.

Contributions. Our main contributions are as follows:

- **Zero-Footprint Workbench Architecture.** We design a Recursive Language Model (RLM) [17] generation framework in which the root agent never observes raw Verilog code or full specifications. Instead, all detailed artifacts are confined to a persistent REPL environment accessed only by worker LLM calls, while the orchestrator operates on compact metadata (e.g., interface summaries, confidence scores, uncertainty signals). This mitigates context rot.
- **Self-Reflective Retrieval-Augmented Correction.** We combine self-reflection [2] with RAG [6] and spec-aware verification to detect and correct hallucinations and inconsistencies in planning and code generation.
- **Staged Pipeline with Gated Recursive Correction.** We implement a structured pipeline (Plan $\rightarrow$ Contract $\rightarrow$ Code $\rightarrow$ Verify) with conditioned self-verification activated only under uncertainty escalation. This combines the efficiency of staged generation with the flexibility of recursive refinement, avoiding uncontrolled context growth.

2 Related Work

2.1 Fine-Tuning

Fine-tuning adapts LLMs to RTL generation by training on paired natural language specifications and ground-truth Verilog code, enabling the model to learn hardware-specific syntax and structural reasoning. Prior work explores various

strategies, including instruction tuning and reinforcement learning. For example, BetterV [10] introduces a generative discriminator to guide optimization objectives, RTLCoder [8] incorporates syntax-aware rewards through compiler feedback, and CodeV-R1 [8] leverages reinforcement learning with verifiable rewards and synthetic round-trip data generation. Despite these advances, fine-tuning approaches remain constrained by the limited availability of high-quality human-written RTL datasets [12] and the inability to fine-tune closed-source frontier models. Moreover, these models still exhibit hallucinations and lack reliable mechanisms for enforcing semantic correctness during inference. In contrast, our approach avoids reliance on fine-tuning altogether by introducing a verification-driven generation framework that detects and corrects uncertainties at inference time.

2.2 Retrieval-Augmented Generation (RAG)

RAG enhances RTL synthesis by incorporating external knowledge. Typically, this involves retrieving similar designs or reference implementations to guide generation. Systems such as VeriRAG [11] use structure-aware retrieval by encoding Verilog designs into intermediate embedding representations to capture hardware topology, while DeepV [5] uses dynamic sampling strategies to select relevant examples. These methods improve performance on specialized tasks such as design-for-test (DFT) by grounding generation in prior validated designs. However, RAG pipelines introduce significant overhead, requiring large curated databases and high-quality retrieval mechanisms, while often inflating the input context with lengthy code examples. This leads to token budget constraints, context truncation, and context rot. Our approach differs by using targeted retrieval only within a self-reflective correction loop, avoiding context bloat while still leveraging retrieval for precise error correction.

2.3 Agentic Frameworks

Agentic approaches decompose RTL generation into multiple interacting components, typically assigning specialized roles to different LLM instances. For example, VeriMind [9] uses a multi-agent architecture with dedicated agents for prompting, code generation, testbench generation, and verification, coordinated by a supervisor through iterative refinement cycles. Similarly, AutoChip [13] integrates LLMs with simulation tools, using compiler and simulation feedback to iteratively correct errors. While these systems demonstrate some of the best improvements to correctness through feedback loops, they often incur high computational cost due to repeated iterations and rely heavily on external verification tools, which can limit scalability and responsiveness. Additionally, feedback is typically reactive, addressing errors only after they manifest in generated code. In contrast, our framework proactively uses estimated uncertainty and spec-aware verification to trigger targeted correction, reducing unnecessary iterations and improving efficiency.

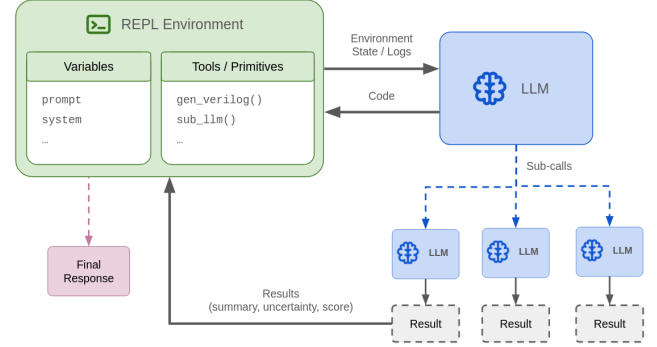


Figure 1. Overview of the RLM [17] architecture.

2.4 Pipeline Frameworks

Pipelined approaches using Chain-of-Thought (CoT) improve RTL generation by decomposing tasks into intermediate reasoning steps that mirror hardware design workflows. Paradigm-based methods [12] classify problems into design categories such as sequential (SEQU), combinational (COMB), and behavioral (BEHAV), guiding the model to generate structured representations such as state machines or truth tables, depending on the design, before synthesizing any code. Extensions such as Chain-of-Descriptions (CoDes) [14] further improve alignment by requiring natural language descriptions prior to code generation, reducing syntax errors and improving synthesizability. While these methods enhance reasoning and structure, they significantly increase prompt length and introduce verbose intermediate representations, which contribute to context growth which leads to context rot in long-horizon tasks. Our approach differs by preserving structured reasoning within isolated execution contexts, preventing intermediate representations from polluting the global context.

3 Methodology

3.1 Recursive Language Model Workbench

We build our system on a RLM [17] workbench architecture shown in fig. 1 that prevents context rot by isolating generation artifacts from the orchestrator. The orchestrator agent never directly observes raw Verilog code or full specifications; instead, all artifacts are confined to a persistent execution environment. At the core is a persistent REPL workbench, implemented as a Python process that maintains a shared state dictionary across calls. This workbench stores all intermediate artifacts (e.g., plans, code, outputs, parsing) and is the only component that invokes worker LLM agents. The orchestrator interacts with the workbench through controlled execution calls and receives only compact metadata summaries in addition to self-reflective confidence scores and uncertainty signals [2]. These signals are used to enable reliable and dynamic control flow to produce a robust

gates that determine whether outputs at any given phase are accepted or to be refined. In this way, the orchestrator never accumulates intermediate reasoning or code, eliminating context growth and mitigating context rot at the root level.

The system maintains two distinct contexts that are loaded once into the workbench. The system context contains role-level instructions for RTL generation and is prepended to all worker calls, while the prompt context contains the hardware specification and is injected only when required during code generation. The specification itself is lazily loaded inside the REPL and is never directly accessible to the orchestrator agent.

All interactions with the worker models occur through two primitive operations provided by the workbench. The first, `sub_llm`, executes a single model call and stores both the generated artifact and structured diagnostic metadata. The second, `generate_rtl`, performs Verilog synthesis using the specification context. Each call produces a standardized output consisting of a structured summary (confidence score and uncertainties) and the generated artifact.

4 Staged Generation Pipeline

We decompose RTL synthesis into five sequential phases: *Plan*, *Contract*, *Code Generation*, *Static Verification*, and *Port Verification*. Each phase in fig. 2 transforms the representation of the specification while preserving explicit uncertainty signals that control progression and correction.

Planning. The *Plan* phase produces a high-level structural interpretation of the specification without committing to RTL. It outputs a structured summary containing the circuit paradigm (e.g., combinational, sequential, FSM), interface definition, and key behavioral properties, alongside a verbose plan retained in the workbench. A key feature of this phase is an internal adversarial ambiguity check, where the model evaluates whether any specification clause within the plan is ambiguous, inconsistent or logically incorrect. Identified uncertainties are explicitly recorded and passed downstream for correction.

Contract Construction. The *Contract* phase converts the high-level plan into a structured behavioral specification suitable for the synthesis of one module. This includes explicit interface definitions, Boolean equations, transition tables, and design constraints (e.g., clocking, reset behavior). The contract serves as a bridge between natural language reasoning and formal RTL representation. By separating this phase from planning, we decompose the design into simpler individual modules. Similar to the planning phase, the contract comes with an uncertainty score and a list of new uncertainties.

Code Generation. The *Code Generation* phase synthesizes Verilog from the contract while maintaining consistency

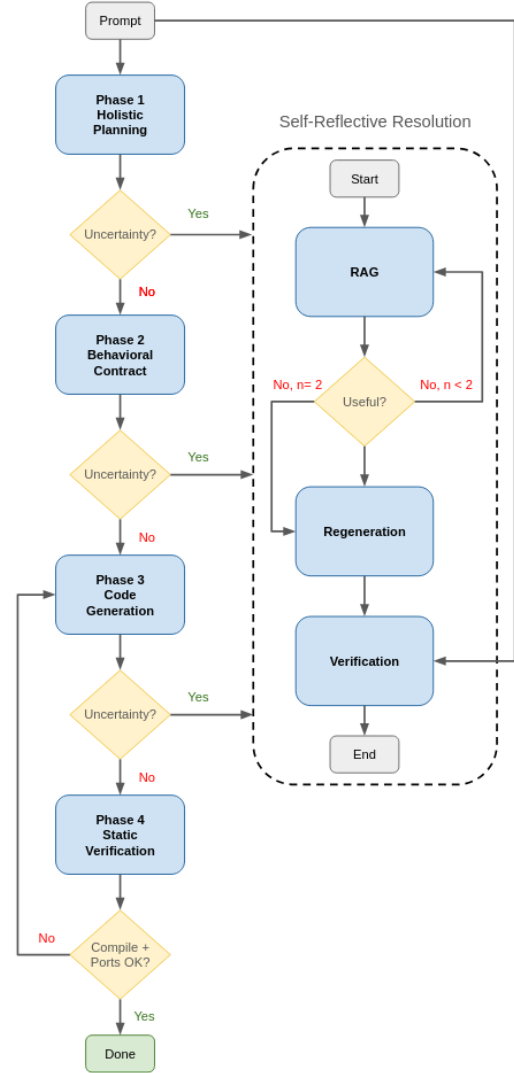


Figure 2. Structured pipeline (Plan → Contract → Code → Verify) with conditioned self-verification resolution activated only under uncertainty escalation.

with the original specification. The specification remains the authoritative source, and any detected inconsistency between the contract and the specification is surfaced as an uncertainty rather than implicitly resolved. The generated module follows strict conventions (e.g., deterministic module naming, explicit handling of state and timing semantics) and is required to implement all specified equations and constraints from the contract without simplification or omission. Once again, the code generation phase provides the workbench with an uncertainty score and a list of new uncertainties.

Static Verification. The *Static Verification* phase performs a deterministic compilation and elaboration check using a

lightweight toolchain (i.e., `iverilog`). This step detects syntactic and structural errors such as type mismatches, missing constructs, or invalid sensitivity lists. Simple errors may be corrected through minimal inline transformations by the root agent, while more complex failures trigger a new code generation cycle. This phase provides a fast and reliable correctness filter without requiring full simulation.

Port Verification. The *Port Verification* phase is a final guardrail that ensures consistency between the generated module interface and the planned specification. It extracts declared ports from the generated code and compares them against the expected interface defined in the planning phase. Any mismatch triggers a correction step, ensuring that interface-level inconsistencies are resolved before final acceptance.

Self-Verification and Correction. Across all gated phases (*Plan*, *Contract* and *Code Generation*), we apply a unified self-verification mechanism based on calibrated uncertainty. Outputs from phases are accepted only when confidence is high, and no unresolved uncertainties remain; otherwise, the system triggers a bounded recursive correction loop. This loop combines targeted retrieval with RAG, regeneration, and spec-aware verification to resolve ambiguities and errors. Retrievals are included in the regeneration only if they are considered relevant and useful. Crucially, verification is performed directly against the original specification rather than prior outputs, enabling detection of systematic semantic errors that persist across iterations. This design provides a lightweight alternative to simulation-based feedback while maintaining strong guarantees of consistency and correctness.

5 Experiments

In this paper, we evaluate our framework against a zero-shot baseline. The orchestrator is implemented using Claude Sonnet 4.6, while all worker calls use a lightweight model, Claude Haiku 4.5. As a baseline, we implement a CoT pipeline using only the Haiku model and a single prompt where it is asked to plan and generate the code within the same context.

Benchmark. We evaluate on the human subset of *Verilog-Eval* [7], consisting of 156 problems. Each problem provides a natural language hardware specification, a reference Verilog implementation, and an executable testbench. The benchmark includes a diverse set of RTL tasks, including combinational logic (e.g., K-maps), arithmetic modules, filters, and both Mealy and Moore state machines.

During the evaluation, we perform $k = 5$ runs for each problem and report both pass@1 and pass@5 metrics. All generated designs are validated using the provided testbenches, and a solution is considered correct only if it passes all test cases.

Table 1. Performance comparison on Verilog-Eval [7] (human subset).

Method	Model	Pass@1	Pass@5
CoT (Ours)	Haiku 4.5	69.9	74.6
RLM (Ours)	Haiku 4.5	83.2	88.6
VeriMind [9]	Claude 3.5 Sonnet	56.4	63.5
VeriMind [9]	PyraNet-DeepSeek	58.3	67.9
RTLCoder [8]	DeepSeek	41.6	50.1
CodeV-R1-7B [18]	Qwen2.5-Coder-7B	68.8	78.2
Paradigm [12]	GPT-4o-mini	68.2	72.8
DeepV [5]	GPT-5 Chat	76.9	81.3
DeepV [5]	Claude Sonnet 4	75.6	80.6

Retrieval Dataset. For retrieval-augmented correction, we use *VerilogDB* [3], a dataset containing over 20k Verilog modules paired with natural language descriptions. Retrieved examples are used selectively within the self-verification loop, based on the description field only, to resolve uncertainties and improve generation quality, rather than being injected into the main generation context.

Metrics. We use the standard pass@k metric [4] to evaluate functional correctness. Let n be the number of generated samples and c the number of correct solutions among them. The pass@k metric estimates the probability that at least one of k sampled solutions is correct, and is defined as:

$$\text{pass@k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$$

Intuitively, pass@1 measures the accuracy rate of a single generation attempt, while pass@5 captures the model’s ability to produce at least one correct solution across multiple attempts.

Compilation Setup. We use Icarus Verilog (`iverilog`) version 11 for static compilation of generated designs. All code is compiled with the `-g2012` flag to enable SystemVerilog (IEEE 1800-2012) support, ensuring compatibility with modern constructs such as `always_ff`, `always_comb`, and `logic`.

6 Results

As shown in Table 1, our approach achieves the strongest performance among methods using comparable or smaller models. The CoT baseline using Haiku 4.5 achieves 69.9 pass@1, but by incorporating our framework (RLM + RAG), there is significant performance improvement to 83.2 pass@1 and 88.6 pass@5, outperforming all compared methods, including those using larger frontier models. This corresponds to an improvement of +13.3 pass@1 and +14.0 pass@5 over our baseline.

A breakdown of failures at pass@1 shows that 12.9% are due to faulty logic and 1.9% are compiler errors, with the remaining cases caused by simulation timeouts or failure to generate valid code. This indicates that most remaining errors are semantic rather than syntactic, suggesting that structural correctness is largely enforced by the framework.

These results indicate that structured generation combined with uncertainty-driven correction can outperform larger monolithic models. Furthermore, the division of tasks into specialized sub-agents, combined with strict separation of context across phases, reduces the risk of context rot while maintaining strong performance. In contrast, existing approaches, including CoT, RAG, and agentic frameworks, rely on accumulating context across reasoning steps, making them susceptible to context rot as intermediate representations grow. Our design demonstrates the effectiveness of RLM frameworks for RTL code generation.

7 Discussion

Despite strong performance gains, our approach has several limitations. First, the framework relies on model-reported uncertainty for gating decisions, which in many instances was shown to be overconfident. In practice, we observe that some hallucinations persist due to ambiguous or underspecified prompts, where the model incorrectly assumes a valid interpretation and reports high confidence. While the framework improves structural consistency and reduces paradigm shifts compared to the CoT baseline, it does not fully eliminate semantic errors arising from ambiguous specifications.

Second, the performance of the framework is not fully model-agnostic. Results depend on the underlying LLM, and behaviors such as uncertainty calibration and reasoning quality vary across models. In particular, we observe that the Haiku model used in our experiments tends to "overthink" intermediate steps, which can introduce unnecessary complexity and affect generation stability [1]. This suggests that part of the observed performance is tied to model-specific characteristics rather than solely to the framework design. We expect that stronger frontier models could further improve results.

Third, the retrieval component remains inefficient. The current RAG mechanism relies on keyword-based queries derived from uncertainties and structural descriptors, which do not fully capture the semantic behavior of hardware designs. As a result, retrieved examples may be only loosely relevant, limiting their effectiveness in resolving deeper logical errors. Finally, the framework introduces additional computational overhead. The staged pipeline, combined with gated correction and retrieval, increases the number of model calls compared to monolithic generation, leading to higher inference cost.

Future work should focus on improving semantic correctness beyond retrieval-based augmentation. One promising

direction is the integration of testbench generation and execution, similar to VeriMind’s feedback system [9], enabling direct functional verification of generated designs. This could provide stronger feedback signals than uncertainty estimation alone and help resolve subtle logical errors. Additionally, more expressive retrieval mechanisms that capture structural and behavioral semantics of RTL code could improve the effectiveness of the correction loop. However, our results suggest diminishing returns from naive RAG, and further ablation studies are needed to isolate its true contribution within the framework.

8 Artifact Availability

The artifacts for this paper, including source code, evaluation scripts, and experimental configurations, are available at: <https://github.com/PersoSirEduard/rtl-code-gen-with-rlm>

9 Conclusions

We present a staged, self-correcting framework for RTL code generation that addresses two key related limitations of LLM-based systems: hallucinations and context rot. By combining zero-footprint orchestration, structured pipeline decomposition, and uncertainty-driven correction, our approach reduces error propagation while maintaining efficient context usage. Experiments on Verilog-Eval [7] show significant improvements over a CoT baseline and prior work, even when using a lightweight model. These results primarily highlight the importance of controlled context and structured verification and generation for reliable hardware synthesis with LLMs.

References

- [1] Milad Aghajohari, Kamran Chitsaz, Amirhossein Kazemnejad, Sarath Chandar, Alessandro Sordoni, Aaron Courville, and Siva Reddy. 2025. The Markovian Thinker: Architecture-Agnostic Linear Scaling of Reasoning. arXiv:2510.06557 [cs.LG] <https://arxiv.org/abs/2510.06557>
- [2] Keivan Alizadeh, Parshin Shojaee, Minsik Cho, and Mehrdad Farajtabar. 2026. Recursive Language Models Meet Uncertainty: The Surprising Effectiveness of Self-Reflective Program Search for Long Context. arXiv:2603.15653 [cs.CL] <https://arxiv.org/abs/2603.15653>
- [3] Paul E. Calzada, Zahin Ibnat, Tanvir Rahman, Kamal Kandula, Danyu Lu, Sujun Kumar Saha, Farimah Farahmandi, and Mark Tehranipoor. 2025. VerilogDB: The Largest, Highest-Quality Dataset with a Preprocessing Framework for LLM-based RTL Generation. arXiv:2507.13369 [cs.AR] <https://arxiv.org/abs/2507.13369>
- [4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] <https://arxiv.org/abs/2107.03374>
- [5] Zahin Ibnat, Paul E. Calzada, Rasin Mohammed Ihtemam, Sujun Kumar Saha, Jingbo Zhou, Farimah Farahmandi, and Mark Tehranipoor. 2025. DeepV: A Model-Agnostic Retrieval-Augmented Framework for Verilog Code Generation with a High-Quality Knowledge Base. arXiv:2510.05327 [cs.AR] <https://arxiv.org/abs/2510.05327>
- [6] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL] <https://arxiv.org/abs/2005.11401>
- [7] Mingjie Liu, Nathaniel Pinckney, Bruce Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. arXiv:2309.07544 [cs.LG] <https://arxiv.org/abs/2309.07544>
- [8] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2025. RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. arXiv:2312.08617 [cs.PL] doi:10.1109/LAD62341.2024.10691788
- [9] Bardia Nadimi, Ghali Omar Boutaleb, and Hao Zheng. 2025. VeriMind: Agentic LLM for Automated Verilog Generation with a Novel Evaluation Metric. arXiv:2503.16514 [cs.AR] <https://arxiv.org/abs/2503.16514>
- [10] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. 2024. BetterV: Controlled Verilog Generation with Discriminative Guidance. arXiv:2402.03375 [cs.AI] <https://arxiv.org/abs/2402.03375>
- [11] Haomin Qi, Yuyang Du, Lihao Zhang, Soung Chang Liew, Kexin Chen, and Yining Du. 2026. VeriRAG: A Retrieval-Augmented Framework for Automated RTL Testability Repair. arXiv:2507.15664 [cs.AR] <https://arxiv.org/abs/2507.15664>
- [12] Wenhao Sun, Bing Li, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, and Ulf Schlichtmann. 2025. Paradigm-Based Automatic HDL Code Generation Using LLMs. arXiv:2501.12702 [cs.PL] <https://arxiv.org/abs/2501.12702>
- [13] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2024. AutoChip: Automating HDL Generation Using LLM Feedback. arXiv:2311.04887 [cs.PL] <https://arxiv.org/abs/2311.04887>
- [14] Prashanth Vijayaraghavan, Apoorva Nitsure, Charles Mackin, Luyao Shi, Stefano Ambrogio, Arvind Haran, Viresh Paruthi, Ali Elzein, Dan Coops, David Beymer, Tyler Baldwin, and Ehsan Degan. 2024. Chain-of-Descriptions: Improving Code LLMs for VHDL Code Generation and Summarization. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD '24)*. ACM, 1–10. doi:10.1145/3670474.3685966
- [15] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] <https://arxiv.org/abs/2201.11903>
- [16] Howard Yen, Ashwin Paranjape, Mengzhou Xia, Thejas Venkatesh, Jack Hessel, Danqi Chen, and Yuhao Zhang. 2025. Lost in the Maze: Overcoming Context Limitations in Long-Horizon Agentic Search. arXiv:2510.18939 [cs.CL] <https://arxiv.org/abs/2510.18939>
- [17] Alex L. Zhang, Tim Kraska, and Omar Khattab. 2026. Recursive Language Models. arXiv:2512.24601 [cs.AI] <https://arxiv.org/abs/2512.24601>
- [18] Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, and Xing Hu. 2025. CodeV: Empowering LLMs with HDL Generation through Multi-Level Summarization. arXiv:2407.10424 [cs.PL] <https://arxiv.org/abs/2407.10424>