# FormalRTL: Verified RTL Synthesis at Scale

Kezhi Li
The Chinese University of Hong Kong
Hong Kong, China

Min Li
South East University
Nanjing, China

Xiangyu Wen
The Chinese University of Hong Kong
Hong Kong, China

Shibo Zhao
South East University
Nanjing, China

Jieying Wu
South East University
Nanjing, China

Junhua Huang
Huawei Technologies Co. Ltd.
Shenzhen, China

Qiang Xu
The Chinese University of Hong Kong
Hong Kong, China

## Abstract

Large language models (LLMs) have demonstrated significant potential in automating hardware synthesis, yet substantial barriers remain for industrial-scale, datapath-centric designs due to ambiguous specifications and a lack of formal correctness guarantees. In this work, we present FormalRTL, a novel end-to-end multi-agent framework that systematically integrates software reference models as formal, executable specifications to guide register-transfer level (RTL) code generation and verification. By tightly coupling planning, synthesis, and formal equivalence checking, FormalRTL achieves scalable and reliable hardware code generation that addresses the critical challenges faced in industrial contexts. The comprehensive evaluation of a new suite of complex industrial-grade benchmarks demonstrates the effectiveness and robustness of our approach. We will open-source the FormalRTL framework and the benchmark suite to facilitate future research in this area.

## 1 Introduction

As AI workloads continue to scale, the design of computation-intensive chips such as GPUs, NPUs and TPUs has become a major engineering bottleneck [8, 16, 24]. This challenge originates from their intricate computational logic and relentless pursuit of extreme power, performance and area (PPA) targets [4, 26], which have significantly prolonged the design cycle and increased the development cost of new chips. Nevertheless, with the advancement of AI, large language models (LLMs) have demonstrated impressive capabilities in natural language understanding and code generation [15, 27], which presents new opportunities for automated and agile hardware synthesis.

Recently, a growing body of studies have investigated the use of LLMs to generate hardware register-transfer level (RTL) code [2, 22, 34, 43]. Approaches based on fine-tuning strategies [11, 12, 30, 42] and multi-agent collaboration [14, 18, 22, 41] have demonstrated encouraging results on simple open source benchmarks [21, 23, 33] (e.g. adders and FIFOs with fewer than 100 lines of code). However, these advances highlight the substantial gap between current research prototypes and industrial-scale applications.

We argue that existing methods for industrial-grade RTL synthesis face two fundamental challenges:

(1) **Large-Scale Design Complexity.** Developing an industrial hardware intellectual property (IP) core typically requires thousands of lines of RTL code. Although planning-based approaches [18, 22] attempt modular decomposition, their effectiveness is limited by incomplete or ambiguous natural language specifications [19] and the inherent difficulty of LLMs in making sound architectural decisions for complex design.

(2) **Intricate Arithmetic and Logical Structures.** Unlike simple open source benchmarks, industrial IPs often feature deep nested datapaths [36] and custom data types (e.g. *bfloat16* [6], *hifloat8* [25]). Simulation-based verification [40] frequently fails to cover corner cases, particularly in datapath-heavy designs [5].

These limitations in automated hardware generation stem from the inherent difficulty of directly synthesizing RTL code from natural language specifications. Specifications with only natural language often fail to capture the complex structural, temporal, and arithmetic requirements necessary for hardware implementation [37]. In conventional industrial practice, hardware design and verification engineers instead rely heavily on software reference models, typically written in C/C++, as executable golden specifications and formal verification artifacts [9, 28]. Commercial hardware formal verification tools such as Synopsys HECTOR [31] and Cadence Jasper C Apps [32] leverage them to perform high-level equivalence checking (EC), mathematically proving that the final RTL implementation is functionally equivalent to the trusted C/C++ reference model. Despite their industry-standard role, prior LLM-driven hardware synthesis approaches have largely overlooked this valuable resource.

Building on these observations, we introduce FormalRTL (see Fig. 1), a novel multi-agent pipeline that systematically addresses the key challenges of industrial-grade hardware synthesis with LLMs. Unlike previous approaches that relied solely on potentially ambiguous natural language specifications, FormalRTL leverages the software reference model as an executable formal specification to guide and anchor the entire synthesis process.

Specifically, our pipeline begins with a **planning agent**, which performs static analysis on the C reference model to reliably decompose the complex design into modular subtasks, thereby mitigating architectural ambiguities often found in purely LLM-based planning. For each subtask, the **initializing agent** generates the RTL code and a verification harness directly anchored to the reference model.

To guarantee correctness, FormalRTL applies rigorous EC (i.e. hwcbmc [28]) between the software and hardware models. Crucially, the **debugging agent** closes the loop utilizing counterexamples from the EC tool to efficiently guide automated code refinement and error resolution. This tightly integrated workflow enables scalable RTL generation with formal correctness guarantees, substantially narrowing the gap between LLM-based prototyping and real-world hardware development.

In summary, our main contributions are as follows.

- We pioneer a **software reference model-driven** methodology for agile hardware development, ensuring scalability and correctness by planning via static analysis and equivalence checking.
- We develop an **end-to-end, multi-agent engine** that automates the entire workflow, from planning, synthesis, and debugging to formal verification.
- We construct and release a set of **industrial-grade benchmarks**, complete with specifications and software reference models. Extensive experiments on these benchmarks show the effectiveness and robustness of our approach.

## 2 Related Work

### 2.1 LLM for RTL Synthesis

Early work shows that supervised fine-tuning (SFT) can substantially improve the RTL code generation capabilities of LLMs [11, 12, 21, 30, 42]. To this end, significant effort has focused on generating high-quality training data: CodeV [42] proposes multi-level summarization data synthesis, OriGen [11] adopts a self-reflection framework with compiler feedback, and BetterV [30] uses domain-specific instruct-tuning with generative discriminators.

To further enhance reasoning and code generation, recent work has introduced reinforcement learning (RL) mechanisms [7, 44]. ChipSeek-R1 [7], for example, integrates direct feedback from simulators, compilers, and electronic design automation (EDA) tools during the training process. Similarly, CodeV-R1 [44] annotates its training data with pass/fail feedback from an RTL equivalence checking tool.

However, these SFT and RL approaches remain constrained by the quality of dataset and the scale of the LLMs. Moreover, the fundamental method of direct RTL synthesis from natural language specifications is often unstable and fails to scale to industrial-grade cases. In addition, none of these methods can formally guarantee the correctness of the generated code, even those trained with equivalence checking feedback.

### 2.2 Multi-agent Approaches for RTL Synthesis

Multi-agent collaborative systems offer more scalable and robust RTL synthesis compared to single-LLM methods [14, 18, 22, 41, 43]. These frameworks typically adopt a three-agent architecture for the core pipeline: task planning, execution/generation, and debugging/optimization. For example, VerilogCoder [14] employs a graph-based task planning approach using Task and Circuit Relation Graphs (TCRG), while MAGE [43] introduces a Verilog-state checkpoint mechanism to improve its debugging agent.

However, these frameworks face two fundamental limitations that hinder their industrial applicability. First, their planning relies on unstructured natural language specifications, which are often ambiguous and lack the precision to define complex datapath logic. Second, their debugging depends on simulation-based feedback. This approach is inherently constrained by testbench coverage and cannot provide the formal guarantee of functional correctness—an important requirement for industrial hardware development.

### 2.3 Software in Hardware Development

To overcome the limitations mentioned above, we are motivated by the reliance of the software reference model in the hardware industry, a practice now embedded in many established workflows.

High-level synthesis (HLS) requires designers to write C/C++ programs augmented with pragmas that guide the synthesis process, such as loop unrolling, pipelining, and interface specification [29]. This workflow is primarily targeted at FPGA development, enabling rapid prototyping and design-space exploration. Recent advances, including C2HLSC [10] and HLSPilot [39], extend HLS workflows by leveraging LLMs to transform unconstrained C programs into synthesis-friendly C code.

In contrast, ASIC designs, especially those with intensive algorithm and datapath, rely on software models not only as synthesis inputs but also as golden specifications for architectural exploration and RTL verification. Architects typically encode the intended functionality in C++ or SystemC, which verification engineers then use to generate expected outputs for regressions. ARM, for example, provides cycle-accurate SystemC models of its processor IP [3], while tools such as Cadence Jasper C2RTL [32] support formal equivalence between RTL datapaths and high-level C/C++ logic. By grounding verification in these trusted reference models, designers can clarify intent early, expose subtle corner-case bugs, and streamline functional sign-off.

## 3 Proposed Methods

The general workflow of FormalRTL is illustrated in Fig. 1. FormalRTL is designed as a unified multi-agent framework that seamlessly transforms high-level design intent into verified RTL implementations. By explicitly taking both the C reference model and the accompanying specification as inputs, the system orchestrates planning, synthesis, debugging, and formal verification in an automated pipeline. This integrated approach can enable scalable, reliable, and efficient hardware development, ensuring that the generated RTL faithfully adheres to both the intended functionality and the formal specification. In addition, we build a new benchmark that provides both a specification and a software reference model for each case to test our engine.

### 3.1 Planning Agent

The planning agent first partitions the C reference code into subfunctions using static analysis from a C compiler. The compiler analyzes function dependencies via the abstract syntax tree (AST), bundling each C function with its required dependencies (e.g. macros, enum types, and constant variables). Following partitioning, the LLM refactors the main design specification into a targeted specification for each individual submodule based on the C subfunction.

The generation process then proceeds strictly following the topological order of the function dependencies derived from the AST.

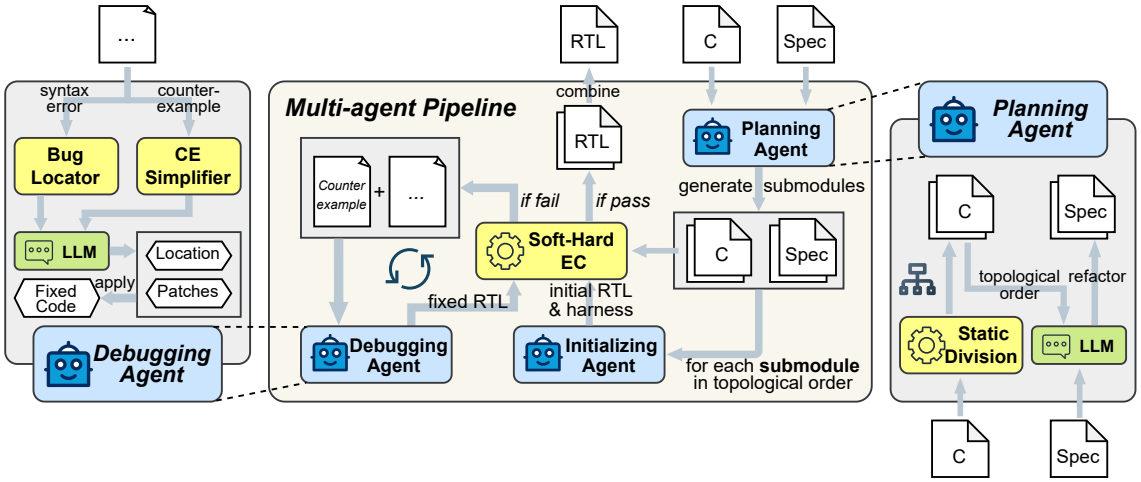


**Figure 1: The general workflow of FormalRTL. The central pipeline (middle) takes a C model and specification, then iteratively generates, verifies, and debugs RTL code on a per-submodule basis. The *planning agent* (right) uses C static analysis (AST) to intelligently partition the main task. The *debugging agent* (left) leverages feedback from equivalence checking, using counterexamples to automatically guide an LLM in generating code patches.**

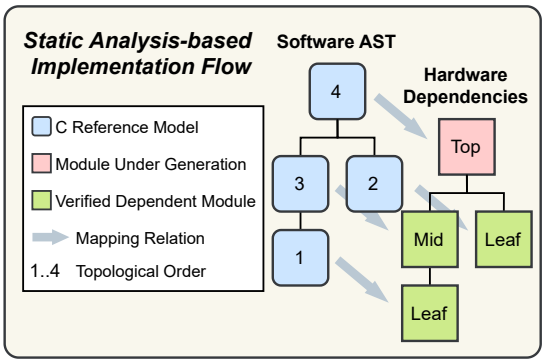


**Figure 2: Submodule implementation flow guided by C function dependencies.**

As illustrated in Fig. 2, when implementing a given submodule, the agent receives its C reference and all previously verified RTL modules on which it depends.

This partitioning strategy offers three key benefits. First, by explicitly leveraging the functional hierarchy of the C reference model, our planning is more reliable than planning based on the specification alone. Second, the decomposition limits the scope of EC to an acceptable scale, ensuring that the verification tool can efficiently process the task and provide rapid feedback. Third, it enables a bottom-up verification philosophy: each RTL submodule is formally checked for equivalence against its corresponding C function. This incremental verification ensures the correctness of the dependent components, which in turn reduces the debugging burden when building the top modules.

## 3.2 Initializing Agent

The initializing agent takes each submodule's reference model and specification as input and generates both an initial RTL file and a corresponding **verification harness**. The harness establishes the verification environment for the RTL and C codes by assuming identical inputs and asserting output equivalence [28]. To ensure reliability, we adopt a few-shot prompting strategy that guides the agent in producing the necessary components for a structurally correct harness.

Subsequently, the agent examines whether the submodule requires sequential logic as specified. If so, it determines the appropriate timeframe of the harness, i.e., the exact number of clock cycles required for the hardware to yield final results. This parameter plays a dual role: it governs the activation of transactional equivalence checking [9, 31, 32], while simultaneously serving as a design knob to regulate the depth and granularity of the RTL pipelining. By explicitly encoding the timeframe, pipeline stages are aligned with latency requirements, enabling systematic verification of sequential behavior and offering designers precise control over the generated timing behavior.

## 3.3 Debugging Agent

The EC tool provides two types of feedback: syntax errors and counterexamples. For syntax errors, we use a **bug locator** to first identify the error's position. We then extract the code lines surrounding the error to explicitly show the agent the erroneous code. This reduces the LLM's burden of locating the error solely from a line number, which is often difficult without the full code structure. For counterexamples, we use a **simplifier** that simplifies the report to show only the information about the problematic function and the mismatching signals. This helps the agent quickly identify the source of the logic mismatch. A qualitative demonstration of these two tools is shown in Fig. 3. Finally, the LLM generates the location

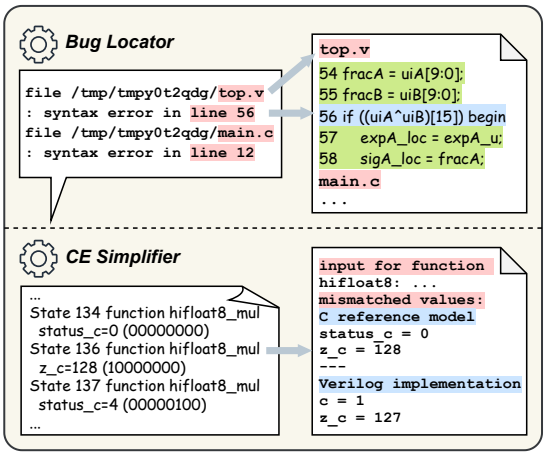


**Figure 3: Qualitatively demonstrate bug localization and counterexample simplification.**

of the erroneous code and a corresponding patch, which is then applied to the original file to produce the fixed code.

In this agent, the EC tool plays an important role as both a syntax checker and a counterexample provider. Counterexample-guided debugging is significantly more efficient than binary "pass/fail" feedback. Furthermore, the tool guarantees a counterexample upon any equivalence failure. This automated approach effectively replaces the laborious process of manual testbench creation, thereby avoiding the common pitfalls of insufficient coverage and high engineering effort.

## 3.4 Benchmark Curation

Current open source benchmarks for RTL generation often lack industrial-grade specifications and corresponding reference C models. Therefore, a contribution of this work is the collection and curation of a benchmark suite derived from both academic and industrial scenarios. For this initial work, we primarily focus on datapath-intensive modules, deferring the integration of more complex control-logic components to future research.

**Academic Cases.** Our academic cases are 16-bit floating-point (FP16) operators (e.g. FP16 adder and FP16 mulitiplier) derived from the Berkeley "softfloat" repository [13], which provides C implementations of IEEE 754 computation units [17]. To prepare inputs for FormalRTL, we first use the static analysis of a C compiler, i.e., Clang, to merge all scattered subfunctions for a given top-level module into a single file. Then, the file is manually cleaned to ensure that it could be compiled independently. Finally, we author an industrial-style specification for each module, detailing necessary information such as special case handling and sequential requirements to simulate a real-world development process. Each top-level module and its specification constitute a single test case.

**Industrial Cases.** For industrial cases, we manually implement C reference models for the multiplication and addition functions of Hifloat8 [25], an 8-bit floating-point datatype featured in the recent Ascend AI Chip [20]. We also write a detailed specification for this datatype and define its computational behavior with reference to the IEEE 754 standard [17]. These cases serve to demonstrate our pipeline's applicability to the agile development of state-of-the-art chip designs.

# 4 Experiments

## 4.1 Experimental Setup

We build FormalRTL using Langchain [35]. After considering factors such as cost-effectiveness and task complexity, we select different base LLMs for different agents. Candidate LLMs are GPT-4.1 and GPT-5. Their application is agent-specific:

(1) The planning agent utilizes GPT-4.1, as this stage primarily involves refining the initial natural language specifications according to the C subfunctions—a task that requires only moderate reasoning ability.
(2) The initializing agent employs GPT-5, which is better suited to generate correct, reasonable, and high-quality RTL code. This step requires advanced reasoning capabilities of the model.
(3) The debugging agent initially uses GPT-4.1 for rapid code fixes. If persistent issues remain after five iterations, the process escalates to GPT-5 to take advantage of its stronger reasoning for more complex debugging tasks.

We conducted all experiments using LLMs with default configurations.

Furthermore, we use Clang as the compiler choice for static analysis. For verification of software/hardware equivalence, we use an academic C-RTL EC tool, i.e., hw-cbmc [28].

## 4.2 Metrics

We define several key metrics to evaluate FormalRTL.

**Initial Success Rate** (ISR, Eq. 1): This is equivalent to the pass@1 metric in code generation benchmarks, which measures the percentage of runs that pass equivalence checking on the first attempt (i.e. without any debugging iterations).

$$\mathrm{ISR} = \frac{N_{\mathrm{pass}}}{N_{\mathrm{total}}} \times 100\%, \tag{1}$$

where $N_{\mathrm{pass}}$ is the number of runs that pass after the initial generation and $N_{\mathrm{total}}$ is the total number of runs, which is 20 in the following experiment.

**# of Iterations:** For the debugging phase, we set a maximum iteration limit ($I_{\mathrm{limit}}$) of 20. If the agent fails to fix the error within 20 iterations, the run is marked as a failure. To penalize failure cases, we treat the iteration count for any failed run as double the limit.

We record the Average Fixing Iterations ($I_{\mathrm{avg}}$, Eq. 2) and the Standard Deviation ($I_{\mathrm{std}}$, Eq. 3).

$$I_{\mathrm{avg}} = \frac{1}{N_{\mathrm{total}}} \sum_{j=1}^{N_{\mathrm{total}}} I'_j, \tag{2}$$

$$I_{\mathrm{std}} = \sqrt{\frac{1}{N_{\mathrm{total}}} \sum_{j=1}^{N_{\mathrm{total}}} (I'_j - I_{\mathrm{avg}})^2}, \tag{3}$$

where $I'_j$ is the adjusted iteration count for the run $j$ as shown in Eq. 4:

$$I'_j = \begin{cases} I_j & \text{if run } j \text{ passes } (I_j \leq I_{\text{limit}}) \\ 2 \times I_{\text{limit}} & \text{if run } j \text{ fails} \end{cases}. \quad (4)$$

**Final Success Rate** (FSR): Finally, we record the FSR, which measures the total percentage of designs that were successfully fixed by our agent within the iteration limit.

## 4.3 Performance on Module-level Cases

Table 1 presents the module-level results. Although FormalRTL processes a complete design, including multiple submodules and a specification, as a single input, we report metrics for each individual submodule for better demonstration. The modules labeled "top" represent the top-level module of their respective design. To avoid redundancy, the submodules shared between different designs are tested only once. The last column records the RTL code lines of a submodule and the total lines, including all its dependencies. Based on the results, we draw the following conclusions.

**Our benchmark cases are challenging, reflecting industrial-level complexity,** and this is demonstrated in two ways. First, the total length of the RTL code shows that the scale of the task is significant, with the hardest case requiring generating more than 1000 lines of code. Second, the ISR for many modules is low, underscoring that current commercial LLMs struggle to produce a correct module on the first attempt.

**The debugging agent is effective in fixing errors.** All submodules achieve a high FSR within iteration limits. This highlights the efficiency of our counterexample-guided debugging approach.

**The planning agent effectively distributes implementation complexity,** and this is demonstrated in two ways. First, the agent balances the implementation load. The partitioned submodules are kept within a reasonable and evenly distributed code length, avoiding any single monolithic bottleneck. Second, the bottom-up verification strategy is highly effective. Counter-intuitively, the average ISR for the top-level modules is higher than that of the leaf or mid-level modules. This is because, when generating the top module, the agent has already implemented and formally verified all its dependencies, reducing the implementation burden.

**FormalRTL is effective in handling sequential logic.** To demonstrate compatibility with sequential circuits, we introduce specific sequential requirements into the specifications of some designs. Generating sequential logic is inherently more difficult and requires correct identification of sequential requirements and the use of transactional equivalence checking. Despite these challenges, the engine still achieved a high FSR for these sequential circuits.

## 4.4 Ablation Study of Planning Methods

To demonstrate the effectiveness of our reference-code-based planning, we compare it against a baseline that represents the current paradigm of specification-only RTL generation [14, 41, 43]. Although these contemporary approaches do not support C reference models as a structured input, we design this baseline to strictly emulate their methodology. We first treated the C reference model as an unstructured text within the specification, rather than using it for static analysis. Second, to ensure a fair comparison and control for backend differences, this baseline uses our formal verification engine (i.e. the same initializing and debugging agents) instead of the testbench-based verification or different EDA tools found in their pipelines. This configuration ensures that any observed performance difference is attributable only to the planning method itself.

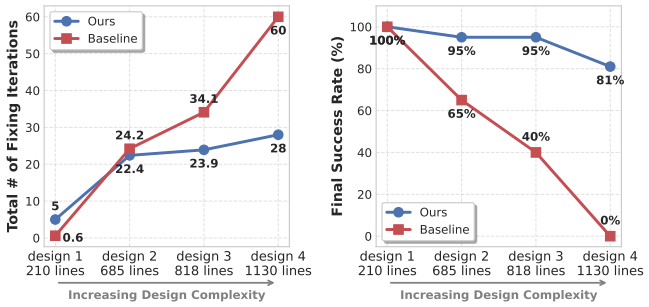


**Figure 4: Comparison of planning methods on fixing iterations and FSR.**

We quantify the total implementation effort ($I_{\text{total}}$, Eq. 5) for FormalRTL by summing the average fixing iterations of the top module and all its dependent submodules, as follows:

$$I^{\text{module}}_{\text{total}} = I^{\text{module}}_{\text{avg}} + \sum_{i \in \text{dependencies}} I^{i}_{\text{avg}}. \quad (5)$$

The FSR of FormalRTL for a given design is defined as the product of the FSR of its top-level module and the FSRs of all its dependent submodules.

The baseline approach must debug the entire design as a single, monolithic task, making it impossible to formally verify individual sub-components against a C reference. Consequently, the iteration budget for the baseline must be compared with our total summed effort ($I_{\text{total}}$). For a fair comparison, we set the maximum fixing iteration limit for each baseline task to be close to the calculated $I_{\text{total}}$ from Table 1. Specifically, we choose four designs with varying complexity and set the maximum iteration limits as follows: "f16_roundToInt" (Design 1, limit: 5), "hifloat8_mul" (Design 2, limit: 25), "hifloat8_add" (Design 3, limit 25) and "f16_add" (Design 4, limit: 30).

As shown in Fig. 4, the performance changes significantly with the design scale. For the small-scale Design 1 ("f16_roundToInt"), the baseline method requires fewer fixing iterations, as it benefits from generating fixes for all submodules in a single iteration. However, this advantage rapidly diminishes as the complexity increases. For larger designs, the burden of managing the entire design state overwhelms the LLM. In Design 4 ("f16_add"), the baseline method fails to generate a single correct output with 0% success rate. In contrast, our reference-code-based planning pipeline maintains a high success rate across all designs, demonstrating superior stability and scalability in handling complex industrial-level RTL generation.

## 4.5 Ablation Study of Debugging Tools

To evaluate the components of the debugging agent, we perform an ablation study on the leaf module "normalize". We compare FormalRTL with three baseline configurations: (1) without the bug

**Table 1: Performance of our pipeline on module-level cases.**

| Design | Module Name | Type | Initial Success Rate (pass@1) | # of Iterations | | Final Success Rate | Module / Total RTL Length (Average # of Lines) |
|---|---|---|---|---|---|---|---|
| | | | | $I_{avg}$ | $I_{std}$ | | |
| Softfloat IEEE754 FP16 | countLeadingZeros16 | leaf | 90% | 0.25 | 0.77 | 100% | 48.55 / 48.55 |
| | packToF16 | leaf | 40% | 2.50 | 2.40 | 100% | 27.45 / 27.45 |
| | shiftRightJam32 | leaf | 50% | 0.90 | 1.26 | 100% | 38.45 / 38.45 |
| | propagateNaNF16 | leaf | 45% | 1.75 | 1.89 | 100% | 57.50 / 57.50 |
| | normSubnormalF16 | mid | 30% | 5.00 | 9.05 | 95% | 37.70 / 86.25 |
| | roundPackToF16 | mid | 50% | 3.45 | 4.60 | 100% | 146.30 / 173.70 |
| | addMagsF16 | mid | 35% | 8.85 | 14.04 | 85% | 345.15 / 576.35 |
| | subMagsF16 | mid | 5% | 9.25 | 8.69 | 95% | 370.00 / 700.25 |
| | f16_le | top | 50% | 3.15 | 8.63 | 95% | 62.05 / 62.05 |
| | f16_roundToInt | top | 80% | 0.70 | 1.58 | 100% | 124.70 / 209.60 |
| | f16_add | top | 65% | 1.00 | 1.84 | 100% | 46.15 / 1129.95 |
| | f16_sub | top | 90% | 0.30 | 1.10 | 100% | 43.75 / 1127.60 |
| | f16_mul | top | 15% | 7.30 | 8.45 | 95% | 209.85 / 565.75 |
| HiFloat8 | calculate_dot_and_m | leaf | 75% | 0.55 | 1.12 | 100% | 48.25 / 48.25 |
| | normalize | leaf | 15% | 6.40 | 8.88 | 95% | 67.50 / 67.50 |
| | lzc | leaf | 35% | 1.45 | 1.63 | 100% | 24.10 / 24.10 |
| | decode_hifloat8 | leaf | 50% | 3.30 | 4.83 | 100% | 134.40 / 134.40 |
| | encode_hifloat8 | mid | 25% | 4.20 | 5.14 | 100% | 153.40 / 201.65 |
| | round_half_to_away | mid | 45% | 4.75 | 6.25 | 100% | 140.75 / 189.00 |
| | hifloat8_add | top | 70% | 0.45 | 0.80 | 100% | 257.00 / 818.35 |
| | hifloat8_add (sequential) | top | 20% | 16.00 | 16.25 | 70% | 420.64 / 981.99 |
| | hifloat8_mul | top | 95% | 0.05 | 0.22 | 100% | 147.05 / 684.35 |
| | hifloat8_mul (sequential) | top | 60% | 9.65 | 15.67 | 80% | 220.10 / 757.35 |

**Table 2: Ablation study of the Debugging Agent.**

| Methods | Average # of Iterations | FSR |
|---|---|---|
| w/o bug locator | 14.15 | 80% |
| w/o CE simplifier | 9.55 | 90% |
| w/o CE-guided debugging | 21.30 | 55% |
| **FormalRTL** | **6.40** | **95%** |

**Table 3: QoR Comparison with Manual Designs.**

| | hifloat8_mul | | f16_mul | |
|---|---|---|---|---|
| | Area ($\mu m^2$) | Delay ($ns$) | Area ($\mu m^2$) | Delay ($ns$) |
| FormalRTL | 1015 | 3.29 | 1629 | 3.54 |
| Engineer | 743 | 1.78 | 1343 | 2.50 |

locator, (2) without the counterexample simplifier, and (3) without counterexample-guided debugging.

As shown in Table 2, all baseline methods require more fixing iterations and achieve a lower Final Success Rate (FSR) than our complete agent, where the absence of counterexample-guided debugging causes the most significant performance degradation.

## 4.6 Quality of Result (QoR) Comparison

We compare the designs generated by FormalRTL with their manual, engineer-optimized counterparts. All designs are evaluated using Yosys [38] for synthesis with the Nangate45 library and OpenROAD [1] for placement and routing.

Table 3 reports two representative cases: "hifloat8_mul" and "f16_mul". Similar trends are observed in the evaluation of other modules. The LLM-generated RTL typically trails manual designs in area and delay, which is consistent with our emphasis on functional correctness over immediate PPA optimization. Nevertheless, verified RTL provides tangible benefits for engineers: a correct, executable baseline facilitates agile development and offers a reliable foundation for RTL design. Furthermore, a stable golden reference enables systematic PPA iteration and ensures the validity of optimizations.

## 5 Conclusion and Future Work

This paper presents FormalRTL, a scalable multi-agent engine for LLM-based RTL synthesis. Our pipeline utilizes software reference models to guide the generation of formally verified hardware. The system employs static analysis for task planning and formal EC counterexamples to guide automated debugging. Experiments on a new suite of industrial-grade benchmarks indicate that FormalRTL

can generate and verify complex and datapath-heavy designs. Despite these advances, several limitations remain and point toward avenues for future work.

**Scalability to Industrial-Scale RTL.** Our pipeline has been validated on RTL generation tasks exceeding 1000 lines, significantly narrowing the gap between academic prototypes and industrial needs. Although a gap remains for full industrial-scale designs, our effective decomposition strategy enables the construction of verified large-scale systems from manageable subtasks.

**Specialized LLMs for Debugging.** While our pipeline is designed to be model-agnostic, its effectiveness heavily relies on the model's ability to understand debugging feedback (e.g. counterexamples). Future work could focus on training small and specialized models specifically for this counterexample-guided fixing task to reduce the reliance on large-scale, proprietary commercial models.

**Robust Open Source Equivalence Checking.** We observe a scarcity of actively maintained open source tools for C-RTL EC. The community would greatly benefit from the development of new open source EC tools that support modern RTL syntax and offer more efficient performance.

## References


[1] Tutu Ajayi and David Blaauw. 2019. OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain. In *Proceedings of Government Microcircuit Applications and Critical Technology Conference*.

[2] Ahmed Allam, Youssef Mansour, and Mohamed Shalan. 2025. ASIC-Agent: An Autonomous Multi-Agent System for ASIC Design with Benchmark Evaluation. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. IEEE, 23–29.

[3] Arm. 2021. Introduction to Cycle Model reference platforms. https://developer.arm.com/documentation/101497/1105/Introduction-to-Cycle-Model-reference-platforms/Introduction-to-Cycle-Model-reference-platforms?lang=en. SystemC Cycle Models Reference Platform Getting Started Guide, Version 11.5.

[4] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 320–332.

[5] Manuel Blum and Hal Wasserman. 2002. Reflections on the Pentium division bug. *IEEE Trans. Comput.* 45, 4 (2002), 385–393.

[6] Neil Burgess, Jelena Milanovic, Nigel Stephens, Konstantinos Monachopoulos, and David Mansell. 2019. Bfloat16 processing for neural networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, 88–91.

[7] Zhirong Chen, Kaiyan Chang, Zhuolin Li, Xinyang He, Chujie Chen, Cangyuan Li, Mengdi Wang, Haobo Xu, Yinhe Han, and Ying Wang. 2025. ChipSeek-R1: Generating Human-Surpassing RTL with LLM via Hierarchical Reward-Driven Reinforcement Learning. *arXiv preprint arXiv:2507.04736* (2025).

[8] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[9] Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*. 368–371.

[10] Luca Collini, Siddharth Garg, and Ramesh Karri. 2024. C2HLSC: Leveraging large language models to bridge the software-to-hardware design gap. *ACM Transactions on Design Automation of Electronic Systems* (2024).

[11] Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, et al. 2024. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[12] Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. 2024. Autovcoder: A systematic framework for automated verilog code generation using llms. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. IEEE, 162–169.

[13] John R. Hauser. 2018. Berkeley SoftFloat Release 3e: Library Interface. https://www.jhauser.us/arithmetic/SoftFloat-3/doc/SoftFloat.html.

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

[15] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[16] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*. 1–14.

[17] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996), 11.

[18] Cangyuan Li, Chujie Chen, Yudong Pan, Wenjun Xu, Yiqi Liu, Kaiyan Chang, Yujie Wang, Mengdi Wang, Huawei Li, Yinhe Han, et al. 2025. Autosilicon: Scaling up rtl design generation capability of large language models. *ACM Transactions on Design Automation of Electronic Systems* 30, 6 (2025), 1–21.

[19] Mengming Li, Wenji Fang, Qijun Zhang, and Zhiyao Xie. 2025. Specllm: Exploring generation and review of vlsi design specification with large language model. In *2025 International Symposium of Electronics Design Automation (ISEDA)*. IEEE, 749–755.

[20] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 789–801.

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

[22] Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[23] Yao Liu, Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 2024. Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[24] Tao Luo, Shaoli Liu, Ling Li, Yuqing Wang, Shijin Zhang, Tianshi Chen, Zhiwei Xu, Olivier Temam, and Yunji Chen. 2016. DaDianNao: A neural network supercomputer. *IEEE Trans. Comput.* 66, 1 (2016), 73–88.

[25] Yuanyong Luo, Zhongxing Zhang, Richard Wu, Hu Liu, Ying Jin, Kai Zheng, Minmin Wang, Zhanying He, Guipeng Hu, Luyao Chen, et al. 2024. Ascend hifloat8 format for deep learning. *arXiv preprint arXiv:2409.16626* (2024).

[26] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 522–531.

[27] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the robustness of code generation techniques: An empirical study on github copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2149–2160.

[28] Rajdeep Mukherjee, Mitra Purandare, Raphael Polig, and Daniel Kroening. 2017. Formal techniques for effective co-verification of hardware/software co-designs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[29] Declan O'Loughlin, Aedan Coffey, Frank Callaly, Darren Lyons, and Fearghal Morgan. 2014. Xilinx vivado high level synthesis: Case studies. In *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014)*. IET, 352–356.

[30] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. 2024. BetterV: controlled verilog generation with discriminative guidance. In *Proceedings of the 41st International Conference on Machine Learning*. 40145–40153.

[31] Synopsys. 2025. VC Formal Datapath Validation. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal/vc-formal-datapath-validation.html.

[32] Cadence Design Systems. 2025. Jasper C Formal Verification. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-c-formal-verification.html.

[33] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[34] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 1–31.

[35] Oguzhan Topsakal and Tahir Cetin Akinci. 2023. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. In *International conference on applied engineering and natural sciences*, Vol. 1. 1050–1056.

[36] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, Ankur Agrawal, Monodeep Kar, Shubham Jain, Alberto Mannari, Hoang Tran, et al. 2021. RaPiD: AI accelerator for ultra-low precision training and inference. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 153–166.

[37] Yangbo Wei, Zhen Huang, Huang Li, Wei W Xing, Ting-Jung Lin, and Lei He. 2025. Vflow: Discovering optimal agentic workflows for verilog generation. *arXiv preprint arXiv:2504.03723* (2025).

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

[39] Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. 2024. Hlspilot: Llm-based high-level synthesis. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[40] Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. 2025. Assertllm: Generating hardware verification assertions from design specifications via multi-llms. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*. 614–621.

[41] Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine, Yong Liu, and Haoxing Ren. 2025. Spec2RTL-Agent: Automated Hardware Code Generation from Complex Specifications Using LLM Agent Systems. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. IEEE, 37–43.

[42] Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, et al. 2025. Codev: Empowering llms with hdl generation through multi-level summarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025).

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

[44] Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, et al. 2025. CodeV-R1: Reasoning-Enhanced Verilog Generation. *arXiv preprint arXiv:2505.24183* (2025).