CorrectRTL: Boost RTL Generation Quality Through Automated Self-Correction with Instruction Decomposition

Jing Wang, Shang Liu, Hangan Zhou, Wenji Fang, Yuchao Wu, Yugao Zhu, Zhiyao Xie*
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{jwangjw,sliudx,hzhoubu,wfang838,ywu092,yzhuel}@connect.ust.hk, eezhiyao@ust.hk

ABSTRACT

Large language models (LLMs) have shown strong promise in specification-to-RTL generation. Among the explorations, the *self-correction paradigm* has significantly advanced LLM-based RTL generation. However, two challenges limit further progress. **Challenge on benchmarks.** Existing benchmarks are saturated by advanced LLMs, leaving little room to evaluate the agentic self-correction framework. **Challenge on methodology.** The *monolithic self-correction* paradigm rewrites the entire design at once, where LLMs are susceptible to error confusion and corruption of originally correct code. To address the benchmark challenge, we contribute **ScaleBench**, a more challenging benchmark of 200 complex RTL designs. To address the challenge on methodology, we propose **CorrectRTL**, an agentic framework that realizes a new **decompose-then-correct** paradigm, in which the LLM agent explicitly decouples scaffold construction from detailed implementation. *CorrectRTL* contributes three core techniques: (1) *Structure-oriented decomposition*, (2) *Snippet-based RAG*, and (3) *SVA-based self-correction*. On ScaleBench, CorrectRTL improves Pass@1 from 20.5% to 49.0% (+28.5%). On VerilogEval, CorrectRTL achieves up to 96.1% Pass@1, demonstrating consistent improvements across both simple and complex designs.

ACM Reference Format:
Jing Wang, Shang Liu, Hangan Zhou, Wenji Fang, Yuchao Wu, Yugao Zhu, Zhiyao Xie*. 2026. CorrectRTL: Boost RTL Generation Quality Through Automated Self-Correction with Instruction Decomposition. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '26), November 08–12, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831252.3833989

1 INTRODUCTION

Large language models (LLMs) are rapidly transforming hardware design automation, from RTL generation [4, 14, 15, 17, 22] to verification and debugging [6, 11, 23, 24]. Among the various stages of the design flow, specification-to-RTL generation has attracted intensive attention [9, 13, 16, 18, 26]. LLMs can understand hardware specifications and produce synthesizable designs. Such LLM-assisted design flows are emerging as a compelling paradigm that promises to fundamentally reshape hardware development.

Recently, a new paradigm [9, 26] achieves promising results on VerilogEval [13], one of the most commonly used RTL generation benchmarks. We refer to this paradigm as the *self-correction* paradigm, in which the model first generates RTL code and then corrects it based on verification feedback. Specifically, MAGE [26] and VerilogCoder [9] achieve pass rates exceeding 94%, approaching near-perfect functional correctness. These two SOTA works show a more promising direction for automated RTL generation than other specification-based prompt engineering strategies. One important reason is that simple prompt engineering on specifications for LLMs is inherently hard to address detailed signal-level bugs. However, the existing *self-correction* works [9, 26] rely on manually written golden testbenches [9, 26]. In real applications, manually crafting golden testbenches for every module is impractical. To further advance the self-correction paradigm toward greater automation, some works begin to explore auto-generated testbenches [10, 19, 21, 23] or Python reference models [21]. However, two important challenges significantly limit the effectiveness and further advancement of the research direction.

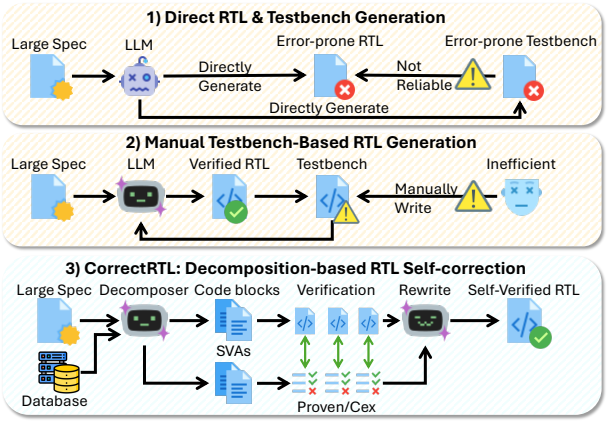


Figure 1: Overview of CorrectRTL. (1) Existing approaches rely on testbenches or reference code for self-correction, which are unreliable or impractical. (2) CorrectRTL combines structure-oriented decomposition, snippet-based RAG, and SVA-based self-correction to improve both generation quality and error identification.

Challenge on benchmarks. Existing benchmarks such as VerilogEval [13] and RTLLM [16] consist predominantly of simple designs. Direct LLM generation already achieves over 80% pass rate on these benchmarks, and the remaining failures are dominated by a small set of recurring error patterns. This leaves little room to differentiate methods or to evaluate whether *self-correction* techniques genuinely generalize, rather than merely fixing a narrow class of easy errors. A more challenging benchmark with more complex designs is needed to meaningfully assess the effectiveness of self-correction approaches.

Challenge on methodology. We refer to the current approach as the *monolithic self-correction* paradigm, in which LLMs generate

*Corresponding author.

Please use nonacm option or ACM Engage class to enable CC licenses
This work is licensed under a Creative Commons Attribution 4.0 International License.
ICCAD '26, November 08–12, 2026, San Jose, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3833989

a full verification reference (e.g., a testbench or a Python reference model) and apply corrections to the entire design as a whole. This paradigm faces two fundamental difficulties. **Difficulty 1: reference difficulty**. The correctness of the verification reference itself is inherently hard to guarantee. Auto-generated testbenches or reference code are not inherently more reliable than the RTL they are meant to verify. Generating a complete testbench or reference model becomes increasingly challenging as design scale increases. **Difficulty 2: correction effectiveness**. Monolithic correction forces the LLM to simultaneously reason about the global structure and local behavioral logic. This entanglement of the whole design leads to two common failure reasons: (1) The model becomes *confused* by the redundant context and fails to identify the actual errors. (2)Even if the error is correctly localized, the model *corrupts originally correct code* while attempting to fix the error, because it inadvertently modifies the surrounding code. As design complexity grows, these failure modes compound, making monolithic self-correction increasingly unreliable.

To address the above challenges, we make two contributions. **(1) New benchmark.** We propose **ScaleBench**, a benchmark containing larger and more complex designs that are substantially more challenging than those in existing benchmarks [13, 16, 18]. **(2) New paradigm.** We propose a new self-correction paradigm. For **reference**, we propose using SVA-based verification because SVA is lightweight, thus inherently shorter and easier to generate. Generating SVAs will not become more challenging with the design scaling up. For **correction effectiveness**, we propose to *decompose and correct*. This paradigm decouples structural scaffold construction from behavioral detail implementation. We implement an agentic framework, **CorrectRTL**, to instantiate and validate this paradigm, as depicted in Figure 1. *CorrectRTL* involves three key techniques to realize our paradigm.[1]

(1) **Structure-oriented decomposition.** We observe that LLMs are remarkably capable of constructing the *structural scaffold* of a design, such as declaring modules, defining port interfaces, and organizing sequential and combinational blocks. But LLMs frequently fail at *behavioral details* within individual blocks. This asymmetry motivates us to explicitly decouple scaffold construction from detail implementation. So that LLMs first produce a well-organized scaffold via specification decomposition into block-level instructions, and self-correction is then focused exclusively on detail errors within individual blocks without disrupting the overall architecture.

(2) **Snippet-based RAG.** We further observe that LLMs are often *stubborn* on detail-level errors. General skill injection through prompt engineering or high-level design guidelines fails to correct many recurring mistakes. Because the model lacks the precise, context-specific knowledge needed for each logic block. This motivates us to design a RAG system that provides exact snippet-level examples of correct implementations for the specific behavioral patterns the LLM struggles with. Our decomposition naturally serves this purpose. Specifically, the decomposed instruction can serve as a query to retrieve relevant examples.

(3) **SVA-based self-correction.** We propose using lightweight SystemVerilog Assertions (SVAs) that target a single logic property of an individual block. Each SVA checks one focused property, so it is easier to generate correctly than a full golden testbench. An EDA tool checks the SVA against the block, and any counterexample pinpoints the faulty block and drives a targeted rewrite. This enables block-level error identification and correction without a manually written golden testbench.

[1]The benchmark and dataset will be open-sourced at https://github.com/hkust-zhiyao/CorrectRTL.

Method	Instr. decomp.	SVA-based ident.	Self-corr. w/o golden TB	Snippet RAG	Correct func.
Syntax error correction only					
HDLDebugger [24]					
RTLFixer [23]					
Self-corr. relying on manually written golden testbench					
MAGE [26]					✓
VerilogCoder [9]	✓				✓
Self-corr. without manually written golden testbench					
VeriAssist [10]			✓		✓
AutoVeriFix [21]			✓		✓
CorrectRTL (ours)	✓	✓	✓	✓	✓

Table 1: Comparison of related works. TB = testbench. CorrectRTL is the first to combine decomposition, SVA-based identification, and snippet-level RAG for RTL generation.

Our contributions are summarized as follows:

- **ScaleBench benchmark.** We propose ScaleBench, a challenging benchmark of 200 more complex RTL designs.
- **Decompose-and-correct paradigm.** We propose a new self-correction paradigm that explicitly decouples structural scaffold construction from behavioral detail implementation.
- **CorrectRTL framework.** We implement an agentic framework that instantiates the decompose-and-correct paradigm.
- **Snippet-level RAG dataset.** We construct a large-scale RTL-specific RAG dataset, enabling fine-grained knowledge injection at the block level.

2 PRELIMINARY

2.1 Related Work

In this section, we mainly focus on automated self-correction on RTL generation [10, 10, 20, 21, 24]. Table 1 summarizes the related works in the scope. Among these works, some very early works explored syntax identification [23, 24], which is not our focus. We separate the self-correction works on RTL into two main types: (1) *self-correction* relying on a manually written golden testbench, and (2) *self-correction* without a manually written golden testbench.

Self-correction without golden testbench is the most challenging yet meaningful direction. Existing works explore automated testbench generation [10] and Python reference code [21]. However, both face fundamental reliability issues: auto-generated testbenches are harder to guarantee correct than RTL itself, and Python reference models remain challenging for precise hardware description. Critically, both approaches focus on the *verification* side—they cannot inject domain knowledge that the LLM lacks.

RAG for code generation. Retrieval-augmented generation has been widely applied in software code generation [8], but remains unexplored for RTL. Standard code RAG retrieves whole files or functions, which is too coarse-grained for hardware designs where a single incorrect bit width or timing polarity causes functional failure. CorrectRTL introduces *snippet-level RAG* specifically designed for RTL: decomposed code blocks paired with specification sections, retrieved via a hardware-aware hybrid retriever.

Symbol	Meaning
$\mathcal{S}$	Design specification (natural language)
f_θ	LLM with parameters θ
$C,\ C'$	Generated / corrected RTL code
$C_i,\ C_i'$	RTL code block i (generated / corrected)
$I_i^{\mathrm{R}},\ I_i^{\mathrm{A}}$	RTL / SVA instruction for block i
$\mathcal{A}_i$	SVA generated for block i
$\mathcal{D}$	RAG snippet database
$\textsc{Retrieve}(q, \mathcal{D})$	Dense semantic retriever; returns top-K pairs
$\mathcal{T} = (\mathcal{V}, \mathcal{E})$	Hierarchical design structure tree
$\mathcal{V}_{\mathrm{mod}},\ \mathcal{V}_{\mathrm{blk}}$	Module / block nodes in $\mathcal{T}$
$\textsc{Parse}(C)$	Structural parser; extracts blocks from RTL

Table 2: Notation used in this paper.

CorrectRTL takes a fundamentally different approach: rather than improving verification feedback alone, we *inject design-family knowledge* into the generation process via snippet-based RAG, while using SVA-based self-correction as a complementary error identification mechanism. The structure-oriented decomposition serves as the architectural backbone, enabling both fine-grained generation and RAG database construction.

2.2 Problem Formulation

Table 2 summarizes the notation. LLM-based RTL generation with self-correction follows a *generate-then-correct* paradigm. Given a design specification $\mathcal{S}$, an LLM f_θ produces RTL code C and a verification artifact $\mathcal{A}$:

$$C = f_\theta(\mathcal{S}), \qquad \mathcal{A} = f_\theta(\mathcal{S}) \tag{1}$$

An EDA tool checks C against $\mathcal{A}$, yielding error feedback $\mathcal{F} = \textsc{Verify}(C, \mathcal{A})$, which guides correction: $C' = f_\theta(C, \mathcal{F}, \mathcal{A})$.

The verification artifact $\mathcal{A}$ differentiates existing self-correction paradigms: golden testbenches [9, 26] (effective but impractical at scale), auto-generated testbenches [10] (harder to guarantee correct than RTL itself), Python reference models [21] (challenging for precise hardware semantics), and *verification SVAs* (ours). Each SVA targets a single logic property, making it easier to generate correctly than a full testbench.

Regardless of the choice of $\mathcal{A}$, generation quality is bounded by f_θ's domain knowledge. CorrectRTL addresses this by injecting verified design-family knowledge from a curated RAG database $\mathcal{D}$ into both RTL and SVA generation:

$$C = f_\theta\big(\mathcal{S},\ \textsc{Retrieve}(\mathcal{S}, \mathcal{D})\big) \tag{2}$$

This improves both phases at once. It produces better RTL in Phase I and better SVAs in Phase II, rather than relying on verification feedback alone. Our paradigm can boost the performance on both generation and correction steps.

3 BENCHMARK AND DATASET

In this section, we introduce the new challenging specification-to-RTL benchmark in 3.1 and the RAG dataset in Section 3.2.

3.1 ScaleBench: A Challenging RTL Benchmark

Existing benchmarks are insufficient for evaluating self-correction methods on complex designs. To address the limitation and evaluate LLMs' ability more thoroughly, we propose **ScaleBench** with 200 larger RTL designs. Table 3 compares ScaleBench with existing benchmarks in terms of scale and difficulty. Table 3 demonstrates the comparison of ScaleBench with existing benchmarks. Our ScaleBench benchmark contains significantly larger (Avg. 120 LoC compared to RTLLM's 48 and VerilogEval's 16 LoC). The lower performance of the baseline (Claude-4.5 Sonnet) also underscores the benchmark's challenging nature. To curate the benchmark, we select 200 designs from the most recent work RTLPP [2] that proposes a high-quality dataset with aligned specifications and RTL pairs. We manually verify the quality of the specifications and make improvements.

Benchmark	#Designs	Avg LoC	Max LoC	Baseline
VerilogEval [13]	156	16	64	80.1%
RTLLM [16]	50	48	175	68.0%
ScaleBench	**200**	**120**	**348**	**20.5%**

Baseline pass rates are from Claude Sonnet 4.5 direct generation.

Table 3: Benchmark comparison. ScaleBench provides larger designs that are more challenging for LLMs.

3.2 RAG Database

General rule-based skill injection, or prompt engineering, is insufficient to augment LLM-based RTL generation for the detailed functional behaviors that LLMs often lack. To address this limitation, we curate a RAG database to provide LLM-based agents with detailed functional knowledge. We deliberately organize this database at the *snippet (block) level* rather than the whole-design level, for two reasons. *Reason 1.* Individual blocks—particularly their RTL bodies—are directly reusable as code exemplars, whereas whole designs mix unrelated logic and cannot be inserted into a partially-generated module. *Reason 2.* A short, behavior-focused spec snippet yields a much tighter semantic signal than a full specification, so top-k retrieval returns exemplars that match the queried sub-behavior rather than a loosely-related design.

Overview. The RAG database has two sub-datasets that share the same retrieval key. The first is the *block-level dataset*, which consists

Code snippets (by structural block type)		
Block type	Snippets	Target logic
Sequential block	6,092	registers, counters, FSM state
Assign group	2,218	control fan-out, datapath
Combinational block	936	muxes, decoders, arithmetic
Total	**9,246**	
SVA templates (by invariant family)		
Invariant family	Templates	Target pattern
Output equation	1,337	combinational output relations
Past-value relation	1,001	X=\$$past(Y)$ style timing
Reset initialisation	1,256	async/sync register reset
Handshake stability	430	valid/ready protocols, AXI-like
Selector exclusion	315	mux selector → output equality
One-hot encoding	261	FSM state-bit invariants
Range / bit-width	260	bounded-signal assertions
Counter bounding	133	counter wrap / maximum

Table 4: Composition of the curated RAG database. Both sub-datasets share the same spec-snippet embedding key.

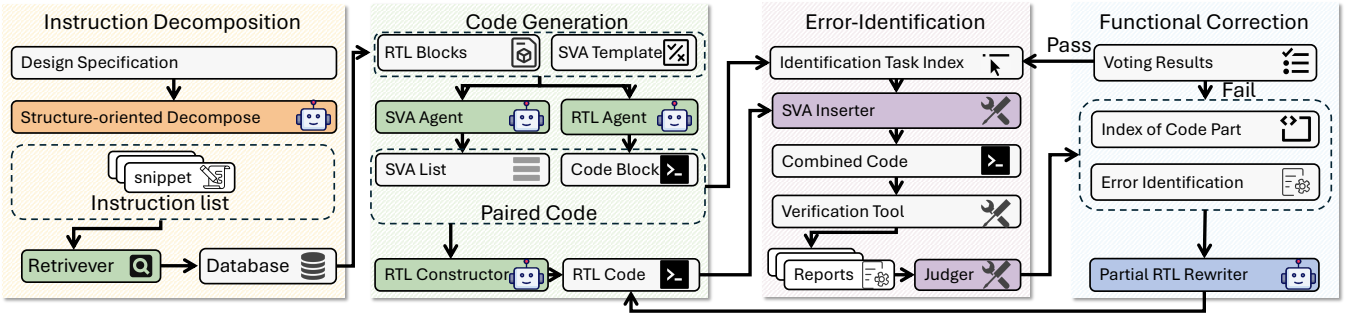


Figure 2: Overview of CorrectRTL framework, orchestrating a multi-agent system with four phases. Phase 2 is augmented with snippet-based RAG from a database constructed via the same decomposition as Phase 1.

of (spec-snippet, RTL-block) pairs: each pair binds a short natural-language description to a single structural block (e.g., `always` block, `assign` group, or memory declaration) extracted from a verified golden design. The second is the *SVA-template-level dataset*, which consists of formally-proven SystemVerilog assertions; each SVA template is a specific assertion that has been *verified* on the golden RTL and is attached to one RTL block from the block-level dataset, so every SVA template inherits a corresponding spec-snippet and code block. Both sub-datasets are curated on top of existing large-scale RTL training corpora (MG-Verilog [25], OriGen [5], AutoV-Coder [7], RTLCoder [14], *etc.*). After whitespace-normalized de-duplication, the block-level dataset contains 9,246 unique (spec-snippet, RTL-block) pairs. The SVA-template-level dataset contains roughly 5k assertions that JasperGold proves on the parent golden, spanning eight invariant families (Table 4).

Step 1: Block-level dataset curation. A Verilog parser decomposes each RTL module in the source corpus into its constituent structural blocks. We treat four block types: module-header / port declarations, sequential `always` blocks, combinational `always` blocks, and `assign` groups. Figure 4 shows an example for a 16-bit adder, where the parser extracts the module header at the top and the per-bit and per-half adders as sequential or combinational leaf blocks below. For each block, we pass both the block code and the module's full design specification to an agent whose task is to produce a 1–3-sentence spec snippet capturing the block's behavioral semantics. Because specifications in public training corpora are often incomplete or loosely worded, the agent is not restricted to pure extraction: when the block's code exposes logic that the surrounding specification underspecifies or omits, the agent may augment the snippet with behavior directly inferred from the RTL. The resulting (spec-snippet, RTL-block) pair forms one entry of the block-level dataset.

Step 2: SVA-template-level dataset curation. Building on the Step 1 output, the agent receives each (spec-snippet, RTL-block) pair and generates a set of candidate SystemVerilog assertions labeled with an invariant family (e.g., reset initialization, past-value relation, handshake stability). Each candidate is then inserted into the original golden RTL and discharged by JasperGold. Only assertions that the tool *proves* on the golden are retained and become SVA templates in the database. A 1,000-SVA calibration across the eight invariant families (125 per category, 855 unique designs) yields a 44% overall proof rate. The per-family rate ranges from 58% on the easiest family (reset initialization) to 32% on the hardest (counter bounding). At the full corpus size this projects to roughly 5k formally-verified templates.

The two sub-datasets are jointly indexed by a single dense embedding over the spec-snippet field. A single retrieval therefore returns both a reference code pattern and its verified assertion oracle for the same block family, eliminating the cold-start problem of prior RAG approaches that index arbitrary unverified code.

4 METHODOLOGY

CorrectRTL assigns agents two roles. **Structural agents** decompose $\mathcal{S}$ into block-level tasks using curated rules such as module boundaries, reset conventions, and port naming. **Detail agents** generate block-level RTL and SVA using snippet-based RAG, because much domain-specific logic cannot be captured by rules alone. Sec 4.1 gives the four-phase overview. Sec 4.2 describes the decomposition strategy. Sec 4.3 presents RAG-augmented RTL and SVA generation. Sec 4.4 details the correction process.

4.1 Framework Overview

Figure 2 depicts the workflow of the CorrectRTL framework, which operates in four phases.

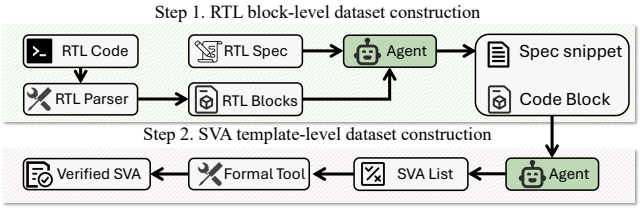


Figure 3: Two-step workflow of RAG dataset construction.

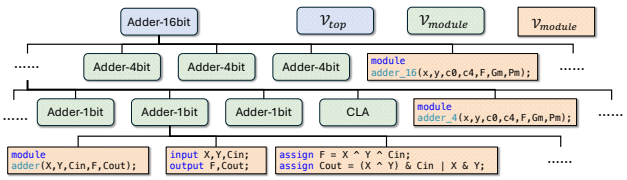


Figure 4: Hierarchical tree $\mathcal{T}$ for a 16-bit adder.

Phase 1: Decomposition (*structural agents*). Depending on design complexity, CorrectRTL applies one of two decomposition strategies. *(1) Code-level decomposition*: the LLM generates the full design from $\mathcal{S}$, and structural parsing extracts blocks from the generated RTL for targeted verification. The following self-correction process is based on the block level. *(2) Spec-level decomposition*: a decomposer agent extracts a hierarchical structure tree $\mathcal{T}$ from $\mathcal{S}$, and paired instructors generate RTL and SVA instruction pairs for each block. Both strategies define the block boundaries shared across generation, retrieval, and verification.

Phase 2: RAG-Augmented RTL and SVA Generation (*detail agents*). For each block, $\textsc{Retrieve}(q_i, \mathcal{D})$ fetches the top-K (spec-snippet, code-block, proven-SVA) entries from the RAG database. The query q_i is the block's spec instruction under spec-level decomposition, or an LLM-authored one-sentence description of the parsed block under code-level decomposition. One agent generates the RTL block C_i using the instruction and the retrieved code blocks. A second agent generates one SVA $\mathcal{A}_i$ using the same retrieval. The two agents share the retrieval so the RTL and the SVA are grounded in the same verified references. Under spec-level decomposition, a constructor assembles the blocks into the full module C.

Phase 3: SVA-Based Verification. For each block C_i, an EDA tool checks $\mathcal{A}_i$ against C. If the assertion produces a counterexample, the block is flagged for rewriting. SVAs target single logic properties derived from the specification, so they serve as the oracle for block-level correctness.

Phase 4: Targeted Rewriting. For each flagged block, a rewriter agent reads the counterexample trace, identifies the root cause, and produces a corrected block C_i'. Only the failing block is replaced: $C' = \textsc{Replace}(C, C_i, C_i')$. Phases 3 and 4 iterate until every block passes its SVA, with a bounded round cap to avoid oscillation.

4.2 Structure-Oriented Decomposition

Phase 1 fixes the block boundaries that all later phases use. We introduce two baselines and then propose our decomposition strategy. The first baseline is *code-level decomposition*, which lets the LLM generate the whole RTL in one pass and parses blocks out of the result. The second baseline is *vanilla spec decomposition*, which splits the specification into block-level instructions first and generates each block independently, with no per-type guidance. On top of vanilla spec decomposition, we propose *structure-oriented decomposition*, which injects a short per-type strategy into the decomposer for five design families. CorrectRTL uses structure-oriented decomposition for its workflow. We evaluate all three as configurations in the ablation study (Sec. 5.3).

Code-Level Decomposition. The LLM generates the complete RTL from the specification in one pass, and a structural parser segments the result into blocks:

$$C = f_\theta(\mathcal{S}), \qquad \{C_1, \ldots, C_n\} = \textsc{Parse}(C) \tag{3}$$

The parser recognizes structural boundaries such as `always` blocks, assign groups, and module headers. The SVA instruction $\mathcal{I}_i^{\mathrm{A}}$ for each block C_i is produced by a single LLM call that reads C_i together with the portion of $\mathcal{S}$ that describes it, and writes down the key behavioral property the block must satisfy. Phase 2 then generates $\mathcal{A}_i$ from $\mathcal{I}_i^{\mathrm{A}}$ and the retrieved SVA templates.

Spec-Level Decomposition. The specification is first parsed into a hierarchical structure tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ whose leaves are individual logic blocks. The block boundaries follow the same definition used in the RAG-database curation (Sec. 3.2): the four block types are module-header / port declarations, sequential `always` blocks, combinational `always` blocks, and `assign` groups. Figure 4 shows an example tree for a 16-bit adder. For each leaf block i, one LLM call produces two instructions in parallel. The RTL instruction $\mathcal{I}_i^{\mathrm{R}}$ describes what the block should compute. The SVA instruction $\mathcal{I}_i^{\mathrm{A}}$ states the key behavioral property the block must satisfy. Both instructions are grounded in $\mathcal{S}$, so the SVA instruction is independent of any later generated RTL. Phase 2 generates C_i and $\mathcal{A}_i$ from these two instructions, and a constructor assembles the blocks into the full module.

Type-oriented decomposition strategies. Asking an LLM to decompose a specification with no extra guidance is brittle. Different design families have distinct coding conventions, and the LLM tends to produce plans that overlap across blocks, request unnecessary testbench scaffolding, or fall back to generic templates that break during assembly. We therefore condition the decomposer on a short per-type strategy. A lightweight classifier assigns each module one of five families, and the decomposer applies the strategy attached to that family before writing the block plan. For *FSM* modules, the strategy splits the design into a clocked state register and a combinational next-state function, and keeps an explicit reset state so the LLM does not fall back to inferred latches. For *sequential datapath* modules, it asks for one non-blocking `always` block per register group and checks that pipeline depth, counter widths, and reset values follow the spec. For *combinational* modules, it restricts the plan to `assign` and `always @(*)` blocks and asks that every output be covered in every branch. For *memory* modules, it pins down array dimensions, per-port read/write timing, and verbatim initialization values. For *structural* modules, it asks the decomposer to list every sub-module instance, use named port connections, and declare each inter-block wire with its exact width. Each strategy targets one failure mode we observed in unconditioned decomposition, and it is relaxed when the spec itself contradicts it.

Vanilla decomposition (baseline). We call direct spec-level decomposition without these skills *vanilla decomposition*. An LLM is asked to break the specification into blocks and write one instruction per block, with no type-specific guidance. In practice the resulting plans are noisy. Block boundaries overlap, so two blocks end up generating the same signal. Some plans include testbench blocks or wrapper modules that the original specification does not require. FSM designs silently collapse into a single always block with inferred latches. Downstream assembly then crashes on signal redefinition or port-width mismatch. We report vanilla decomposition as a baseline in Sec. 5.3 to isolate the contribution of the type-oriented skills above.

4.3 RAG-Augmented RTL and SVA Generation

Generating an RTL block or an SVA from scratch is unreliable. SVA generation is especially error-prone in both syntax (wrong operators, missing sampling edges) and semantics (the property does not match the intended behavior). We therefore retrieve semantically similar examples from the RAG database and feed them to the RTL and SVA agents as references, so the agents adapt existing patterns rather than invent them.

Transformer-based retriever. We use a pretrained transformer-based sentence encoder to embed every spec-snippet in $\mathcal{D}$. At inference we embed the query and return the top-K nearest neighbors

by cosine similarity. Each neighbor brings two things the agents can use: a reference RTL block, and a set of formally-proven SVA templates attached to that block.

Query formation under spec-level decomposition. When Phase 1 uses spec-level decomposition, the per-block RTL instruction $\mathcal{I}_i^{\mathrm{R}}$ is already a natural-language description of the block's intended behavior. We use $\mathcal{I}_i^{\mathrm{R}}$ directly as the retrieval query. The retrieved reference code blocks are passed to the RTL agent, which adapts its coding style to the current block. The retrieved SVA templates are passed to the SVA agent, which adapts them to the block's actual signal names and emits one SVA $\mathcal{A}_i$.

Query formation under code-level decomposition. When Phase 1 uses code-level decomposition, the RTL block C_i has already been generated, so retrieving more reference code is unnecessary. We still need SVA templates. For each parsed block, an LLM reads C_i together with the portion of $\mathcal{S}$ that describes it, and emits a short natural-language description of the block's behavior. This description serves as the retrieval query, and only the proven SVA templates attached to the top-K neighbors are returned. The SVA agent then adapts them to C_i's signal names.

RAG for Generations. Retrieval boosts both sides of the Phase-2 generation, which leads to a stronger first-pass design before any self-correction is applied. *(1) RAG for RTL block generation.* The retrieved reference blocks give the RTL agent concrete patterns for the target behavior. The agent adapts a verified implementation instead of composing one from the spec alone, which reduces structural and signal-level errors in the initial code. *(2) RAG for SVA generation.* Spec descriptions rarely spell out formal assertions, so SVAs generated from scratch often have wrong operators, missing sampling edges, or properties that do not match the intended behavior. The retrieved templates have already been formally discharged on golden RTL. The SVA agent adapts their signal names to the current block, which gives a syntactically well-formed and semantically aligned starting point and removes most of the from-scratch error modes.

4.4 Functional Correction

When Phase 3 flags a block C_i, we invoke a rewriter agent with three well-defined inputs: the current RTL block C_i, the violated SVA $\mathcal{A}_i$, and the counterexample log ξ_i emitted by the verifier. The log contains the signal values at the violation cycle and the relevant history needed to explain how the violation was reached. The agent reasons about these three inputs and produces a corrected block C_i'. No retrieval is performed at this step.

Analyze-plan-rewrite chain. The rewriter is a single LLM call whose structured prompt composes three steps:

$$C_i' = \textsc{Rewrite}\big(C_i,\ \textsc{Plan}\big(\textsc{Analyze}(C_i, \mathcal{A}_i, \xi_i),\ \mathcal{A}_i\big)\big). \tag{4}$$

The Analyze step produces a root-cause diagnosis d_i in natural language. It traces the counterexample values back through the block body to identify the specific line or expression that makes $\mathcal{A}_i$ fail. The Plan step turns d_i into a concrete modification plan p_i that is expressed in terms of the block's signals and aligned with $\mathcal{A}_i$. The Rewrite step edits C_i according to p_i and emits the corrected block C_i'. Only the failing block is changed in the full design:

$$C' = \textsc{Replace}(C,\ C_i,\ C_i'). \tag{5}$$

The block-level decomposition from Phase 1 is what makes this targeted replacement safe, since correct blocks remain untouched.

Iterative correction. After each rewrite, Phases 3 and 4 run again on the updated design C'. Any block that is still flagged is corrected again using the same analyze-plan-rewrite chain on its latest $(C_i, \mathcal{A}_i, \xi_i)$ triple. The loop stops early once all blocks pass their SVAs, and is capped by a fixed number of rounds to avoid oscillation between rewrites. Syntax correction [23] is applied as a preprocessing step and is not discussed further here.

5 EXPERIMENT

5.1 Experimental Setup

Evaluation benchmarks. We evaluate on four benchmarks spanning different complexity levels: (1) VerilogEval Human v1/v2 [13] and VerilogEval Machine v1 (156/156/143 designs, 10–50 LoC), (2) RTLLM v1.1/v2 [16] (50 designs each, medium complexity), and (3) **ScaleBench** (200 designs, 30–348 LoC), a more challenging benchmark with balanced lines of code (LoC).

Metrics. Functional *Pass@1* is the primary metric. To isolate the effect of initial code generation and self-correction step, we compare each design's Pass@1 before and after correction. A design whose Pass@1 goes up contributes to *Fix*. A design whose Pass@1 goes down contributes to *Regression*. Both are reported as averages over the benchmark. The net effect of correction is Δ = Fix − Regression.

Auxiliary metrics. Pass@1 is strict at the design level. Many complex designs are hard enough that no method fully passes them, but a corrected design can still be much closer to the golden than its initial code. We therefore report two partial-correctness metrics. *SER* (signal equivalence rate) measures the fraction of output signals that are formally equivalent to the golden. We run JasperGold's sequential equivalence check between the generated RTL and the golden, and parse the log to obtain the per-output proven rate. *SSim* (structural similarity) measures how closely the generated RTL's block structure matches the golden's, as a blend of set overlap and longest-common-subsequence over parsed blocks. A design that misses Pass@1 but lifts SER from, say, 30% to 80% is still a meaningful partial fix. Together, SER and SSim let us quantify progress on designs that remain too hard to pass outright.

We formalize the two metrics as follows. Given a generated RTL C and its golden reference $C^\star$:

$$\mathrm{SER}(C) = \tfrac{1}{|O|}\,\#\{s \in O : \mathrm{SEC}(s, C, C^\star) = \textsc{proven}\},$$
$$\mathrm{SSim}(C) = \alpha\,\mathrm{Jaccard}\big(B(C), B(C^\star)\big) + (1-\alpha)\,\mathrm{LCS}\big(B(C), B(C^\star)\big),$$

where O is the set of output signals, $B(\cdot)$ is the multiset of parsed structural blocks, and α=0.5.

Baselines. We compare against three recent self-correction methods: RTLFixer [23], VeriAssist [10], and AutoVeriFix [21]. All baselines run on the same model and temperature as CorrectRTL. Formal verification uses JasperGold 2024.06.

Implementation details. We use the *Microsoft Agentic Framework* to develop our methods and reproduce the baseline methods. We evaluate three backbones: Claude Sonnet 4.5 [3], GPT-4o [1], and DeepSeek-V3 [12]. Generation uses temperature 0.2 with a single sample. The RAG retriever uses a pretrained transformer-based sentence encoder (BGE-class, e.g., `bge-small-en-v1.5`) and a FAISS flat cosine index over the spec-snippet field of $\mathcal{D}$. At inference, we return the top-K neighbors with K=8. The correction loop caps at $T_{\max}$=3 rounds with early stop once all blocks pass.

Base LLM	Method	VerilogEval Human v1 (156 designs)	VerilogEval Machine v1 (143 designs)	VerilogEval Human v2 (156 designs)	RTLLM v1.1 (50 designs)	RTLLM v2 (50 designs)	ScaleBench (200 designs)
GPT-4o [1]	Direct generation	59.0%	69.2%	68.6%	69.1%	67.2%	9.5%
	RTLFixer [23]	61.5%	73.4%	69.2%	69.1%	67.8%	10.5%
	VeriAssist [10]	65.3%	74.8%	70.5%	71.2%	67.1%	12.0%
	AutoVeriFix [21]	76.4%	83.3%	79.8%	74.3%	69.3%	14.5%
	CorrectRTL	**85.2%** (+26.2)	**87.1%** (+17.9)	**85.3%** (+16.7)	**77.4%** (+8.3)	**73.2%** (+6.0)	**22.0%** (+12.5)
Claude Sonnet 4.5 [3]	Direct generation	80.1%	88.8%	79.5%	68.0%	74.0%	20.5%
	RTLFixer [23]	81.4%	89.5%	80.1%	68.0%	74.0%	22.0%
	VeriAssist [10]	83.3%	90.2%	82.1%	70.0%	74.0%	24.0%
	AutoVeriFix [21]	85.3%	91.6%	85.3%	71.0%	75.0%	27.0%
	CorrectRTL	**91.3%** (+11.2)	**96.1%** (+7.3)	**90.4%** (+10.9)	**85.0%** (+17.0)	**82.0%** (+8.0)	**49.0%** (+28.5)
DeepSeek-V3 [12]	Direct generation	65.5%	73.1%	63.4%	56.0%	60.0%	13.5%
	RTLFixer [23]	67.2%	75.8%	64.1%	56.0%	60.0%	14.5%
	VeriAssist [10]	70.1%	77.6%	67.3%	58.0%	60.0%	16.0%
	AutoVeriFix [21]	73.5%	80.4%	71.2%	60.0%	62.0%	18.5%
	CorrectRTL	**76.8%** (+11.3)	**82.3%** (+9.2)	**75.1%** (+11.7)	**66.0%** (+10.0)	**68.0%** (+8.0)	**30.0%** (+16.5)

VerilogEval and RTLLM use simulation-based verification; ScaleBench uses JasperGold SEC against golden RTL.
Numbers in parentheses denote improvement over direct generation with the same LLM.

Table 5: Overall functional correctness (Pass@1) across backbones and self-correction methods.

Decomposition configuration	Step 1	Step 2	Identification (Step 3)		Correction (Step 3)		Final Pass@1
	Gen Pass@1	+RAG Pass@1	Recall	FN	Fix	Regression	Pass@1
Vanilla RTL generation	20.5%	—	—	—	0.0%	0.0%	20.5%
Code-level decomposition	20.5%	20.5%	42%	10%	3.5%	0.3%	23.7%
Vanilla spec decomposition	20.5%	28.0%	50%	8%	5.2%	0.4%	32.8%
CorrectRTL (Structure-oriented decomposition)	**28.0%**	**40.0%**	**68%**	**5%**	**9.2%**	**0.2%**	**49.0%**

Table 6: Ablation of the four decomposition configurations on ScaleBench (200 designs, Claude Sonnet 4.5).

5.2 Overall Results

Table 5 presents the main results across six benchmarks and three backbone LLMs. CorrectRTL achieves consistent improvements over all baselines on every (model, benchmark) pair.

Performance on existing benchmarks in Table 5. CorrectRTL delivers consistent improvements across the existing benchmarks. With Claude Sonnet 4.5 as the base model, CorrectRTL reaches **96.1%** on VerilogEval Machine v1 and **91.3%** on Human v1, compared to 91.6% and 85.3% for the strongest prior baseline (AutoVeriFix). On RTLLM v1.1, all baseline methods stay consistently below 71%, while CorrectRTL lifts Claude to **85.0%**. This is an improvement of +17.0% over direct generation and +14.0% over AutoVeriFix. These gains show that CorrectRTL remains the strongest method even on benchmarks where prior work is already near saturation.

Performance on ScaleBench in Table 5. The gain from CorrectRTL is larger on the more complex ScaleBench designs. With Claude Sonnet 4.5 as the base model, CorrectRTL improves Pass@1 from 20.5% to **49.0%**, an improvement of +28.5%. The same trend holds for DeepSeek-V3, which CorrectRTL improves from 13.5% to 30.0% (+16.5%), and for GPT-4o, which CorrectRTL improves from 9.5% to 22.0% (+12.5%). The relative ordering across baselines is preserved on every backbone.

5.3 Ablation Study

We ablate CorrectRTL along four decomposition configurations that cover the full design space of our method: (1) *Vanilla RTL generation*, a direct one-shot generation with no decomposition; (2) *Code-level decomposition*, which generates RTL one-shot and then parses blocks for block-level verification; (3) *Vanilla spec decomposition*, which splits the specification before generation but without the type-oriented strategies from Sec. 4.2; and (4) *Structure-oriented decomposition* (ours), which adds the type-oriented strategies on top of spec-level decomposition. Each configuration is evaluated at three pipeline steps: generation (Step 1), RAG-augmented generation (Step 2), and SVA-feedback correction (Step 3).

Ablation results in Table 6. Step 1 Pass@1 stays at 20.5% for the first three configurations. Code-level decomposition only parses RTL after generation, and vanilla spec decomposition splits the spec without per-type guidance. Neither directly improves the initial RTL. Only the structure-oriented configuration lifts Step 1 to 28.0%, a +7.5% gain from the type-oriented strategies. Step 2 shows the contribution of RAG. Vanilla RTL generation has no block boundaries, so RAG is not applicable. Code-level decomposition can retrieve SVA templates only, since the RTL is already generated, so Step 2 stays at 20.5%. Vanilla spec decomposition can retrieve both reference blocks and SVA templates, which lifts Step 2 to 28.0%. The same RAG step lifts structure-oriented decomposition from 28.0% to 40.0%. Vanilla spec decomposition therefore shows a relatively minor effect overall: its Final Pass@1 (32.8%) is well

Variant	Compile on Step 2	Correct on Golden	Precision	Recall on fail
Vanilla RTL Gen	—	—	—	—
Code-level Decomp	55%	52%	63%	42%
Vanilla spec Decomp	78%	70%	74%	50%
CorrectRTL	**92%**	**85%**	**88%**	**68%**

Table 7: Ablation study on SVA quality on ScaleBench.

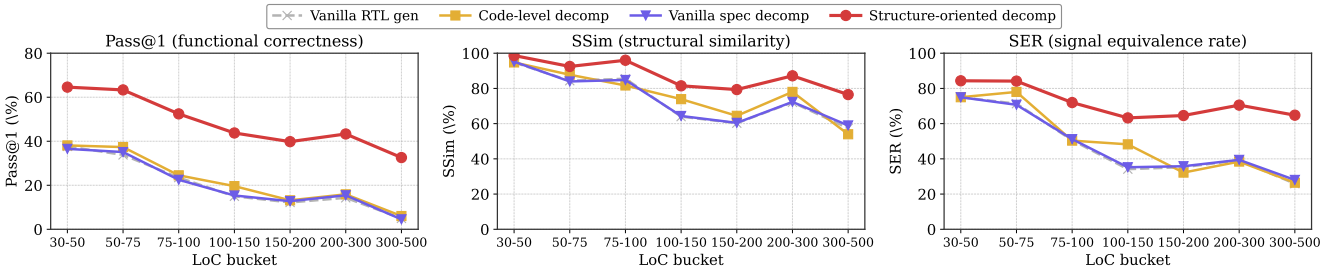


Figure 5: Pass@1, structural similarity (SSim), and SER by ScaleBench LoC bucket for the four ablation configurations based on the Sonnet 4.5.

below structure-oriented decomposition (49.0%). Step 3 adds SVA-feedback correction. The Final Pass@1 ranges from 20.5% (vanilla RTL) to 49.0% (structure-oriented).

SVA recall and false-negative rate in Table 6. SVA recall on Step-2-failing designs is 42% for code-level, 50% for vanilla spec, and 68% for structure-oriented. Vanilla RTL has no block-level SVA, so identification metrics are not applicable. Two factors explain the trend. Code-level SVAs reference internal signal names that do not port across RTL versions, so many fail to compile. Vanilla spec SVAs use spec-mentioned signals but miss domain-specific patterns. Structure-oriented decomposition combines spec-grounded signals with RAG-retrieved proven templates and reaches the highest recall and the lowest false-negative rate. The regression rate stays $\leq$0.4% across all configurations, so the correction loop rarely overwrites a correct block.

SVA quality metrics. We evaluate each generated SVA set $\mathcal{A}_i$ on four metrics. *Compile on Step 2* is the fraction of SVAs that parse and elaborate against the Step-2 RTL. *Correct on Golden* is the fraction that formally prove on the golden RTL $C_i^\star$, measuring spec-consistency. *Precision* is the fraction of fired SVAs that point to a real bug, computed only over SVAs that fail on a buggy design. *Recall on fail* is the fraction of failing designs in which at least one SVA fires and exposes the bug.

SVA quality in Table 7. Code-level SVAs reference code-internal signal names and often fail to compile or prove on the golden. Vanilla spec SVAs use only spec-mentioned signals, so their compile and correctness rates improve, but their recall plateaus without domain-specific patterns. Structure-oriented decomposition paired with SVA-RAG reaches 92% compile rate, 85% correctness on the golden, 88% precision, and 68% recall on failing designs.

5.4 Analysis on Design Scale

The aggregate Pass@1 numbers above hide how each method behaves as design size grows. ScaleBench spans 30–500 LoC, so we group designs by LoC bucket and report Pass@1, SER, and SSim per bucket. We use this view to compare CorrectRTL with prior baselines and ablation configurations of CorrectRTL.

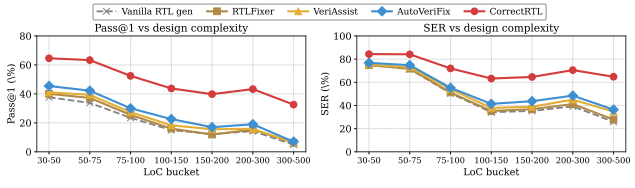


Figure 6: Pass@1 (left) and SER (right) versus design complexity (LoC bucket) on ScaleBench for Claude Sonnet 4.5.

Ablation across design scale in Figure 5. The figure plots Pass@1, SSim, and SER against the ScaleBench LoC bucket for the four ablation configurations. Three observations follow. (i) SSim stays high and roughly flat (74%–96%) across all configurations and all LoC buckets, which suggests that the structural skeleton is recovered well even by weaker configurations. (ii) Code-level decomposition drops the fastest with LoC. Its Pass@1 falls below 10% by 150 LoC. Vanilla spec decomposition also drops, though more slowly. (iii) Structure-oriented decomposition is the only configuration whose Pass@1 stays close to its SSim at large LoC, indicating it preserves both structure and signal-level correctness as designs grow. The SER curves sit between Pass@1 and SSim, consistent with the gap being at the signal level.

Comparison with baselines in Figure 6. This figure plots Pass@1 and SER against the ScaleBench LoC bucket for the three baselines and CorrectRTL (Claude Sonnet 4.5). All three baselines drop sharply past 200 LoC. Their Pass@1 falls below 10% on 200–300 LoC and below 5% on 300–500 LoC. CorrectRTL drops more slowly with design size: it holds $\geq$30% Pass@1 through 150–200 LoC and retains 12% at the largest bucket. The SER curves on the right show the same trend at the signal level. CorrectRTL keeps $\geq$50% signal-level equivalence on 200–300 LoC designs, where the baselines fall below 35%. Overall, the gap between CorrectRTL and the baselines widens as designs grow larger.

6 CONCLUSION

We presented **CorrectRTL**, a four-phase framework that improves both the generation and the self-correction of RTL designs. We also introduced **ScaleBench**, a 200-design benchmark of complex RTL with formal equivalence checking against golden references. On ScaleBench, CorrectRTL improves Pass@1 from 20.5% to 49.0% with Claude Sonnet 4.5, and consistent gains hold across GPT-4o, DeepSeek-V3, and the small-to-medium VerilogEval and RTLLM benchmarks. We hope this work motivates further research on block-level retrieval and assertion-driven self-correction for large-scale RTL generation.

7 ACKNOWLEDGEMENT

This work is supported by Hong Kong Research Grants Council (RGC) CRF-YCRG C6003-24Y, 16216825, and T46-415/25-R. It is also supported by RGC JLFS-YSF/E-605/26. It was partially conducted by ACCESS – AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Ahmed Allam et al. 2025. RTLPP: A Parallel Processing RTL Code Generation Framework Using LLMs. In *Workshop on LLM-Aided Design (LAD)*.

[3] Anthropic. 2024. The Claude Model Family: Claude 3.5 Sonnet, Haiku, and Opus. *Anthropic Technical Report* (2024).

[4] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019* (2023).

[5] Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Demin Song, Dahua Lin, Xingcheng Zhang, et al. 2024. OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection. *arXiv preprint arXiv:2407.16237* (2024).

[6] Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. 2024. Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms. *arXiv preprint arXiv:2402.00386* (2024).

[7] 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.

[8] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. *arXiv preprint arXiv:2312.10997* (2024).

[9] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2024. VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and Abstract Syntax Tree (AST)-based Waveform Tracing Tool. *arXiv preprint arXiv:2408.08927* (2024).

[10] Hanxian Huang, Zhenghan Lin, Zixuan Wang, Xin Chen, Ke Ding, and Jishen Zhao. 2024. Towards llm-powered verilog rtl assistant: Self-verification and self-correction. *arXiv preprint arXiv:2406.00115* (2024).

[11] Minwoo Kang, Mingjie Liu, Ghaith Bany Hamad, Syed Suhaib, and Haoxing Ren. 2024. FVEval: Understanding Language Model Capabilities in Formal Verification of Digital Hardware. *arXiv preprint arXiv:2410.23299* (2024).

[12] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).

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

[14] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. In *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 1–5.

[15] Yuxiao Liu, Chenxing Xu, Yanrui Zhou, Zhen Li, and Qiang Xu. 2025. DeepRTL: Bridging Verilog Understanding and Generation with a Unified Representation Model. *arXiv preprint arXiv:2502.15832* (2025).

[16] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. Rtllm: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 722–727.

[17] Hammond Pearce, Benjamin Tan, and Ramesh Karri. 2020. Dave: Deriving automatically verilog from english. In *Workshop on Machine Learning for CAD (MLCAD)*.

[18] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2024. Revisiting VerilogEval: Newer LLMs, In-Context Learning, and Specification-to-RTL Tasks. *arXiv preprint arXiv:2408.11053* (2024).

[19] Humza Sami, Pierre-Emmanuel Gaillardon, Valerio Tenace, et al. 2024. AIvril: AI-Driven RTL Generation With Verification In-The-Loop. *arXiv preprint arXiv:2409.11411* (2024).

[20] Humza Sami, Pierre-Emmanuel Gaillardon, Valerio Tenace, et al. 2024. Eda-aware rtl generation with large language models. *arXiv preprint arXiv:2412.04485* (2024).

[21] Yan Tan, Xiangchen Meng, Zijun Jiang, and Yangdi Lyu. 2025. AutoVeriFix: Automatically Correcting Errors and Enhancing Functional Correctness in LLM-Generated Verilog Code. *arXiv preprint arXiv:2509.08416* (2025).

[22] 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.

[23] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2024. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Model. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[24] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024. Hdldebugger: Streamlining hdl debugging with large language models. *arXiv preprint arXiv:2403.11671* (2024).

[25] Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. 2024. Mg-verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 1–5.

[26] Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. 2024. MAGE: A Multi-Agent Engine for Automated RTL Code Generation. *arXiv preprint arXiv:2412.07822* (2024).