VERIREFINE: AN LLM-ENABLED ABSTRACTION REFINEMENT FLOW FOR RTL DESIGN GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) can generate register-transfer-level (RTL) designs from natural-language specifications, yet many generation errors stem from misinterpreting the specification Zhang et al. (2026). Intermediate representations make specification interpretations explicit, but these interpretations still require checks for missing requirements and contradictions with the source specification. This paper presents VeriRefine, an LLM-enabled abstraction-refinement framework that refines informal specifications into schema-constrained Abstract Signal Transition Functions (ASTFs), capturing per-signal logic, clocking, reset, and transition semantics before RTL generation. The resulting ASTF-based IR is audited for specification grounding, coverage, consistency, state transition integrity, and RTL design-rule compliance. Post-generation simulation failures are classified to guide ASTF refinement or RTL repair. Using Claude Sonnet 4.6, VeriRefine achieves 96.0% functional correctness on *RTLLM v2.0* and 98.1% on *VerilogEval-Human v2*, exceeding direct generation by 32.0 and 12.8 percentage points, respectively. Iterative debugging improves upon the audited generation flow on benchmarks.

1 INTRODUCTION

RTL code generation with Large Language Models (LLMs) aims to translate natural-language specifications into synthesizable hardware, accelerating chip design and reducing manual effort. However, generating correct RTL remains challenging because the model must recover cycle-accurate behavior, signal dependencies, reset semantics, state transitions, and hardware concurrency from specifications that are often ambiguous or incomplete. This challenge arises from the fundamental semantic gap between informal, intent-level specifications and precise RTL implementations constrained by timing, concurrency, and implementation objectives.

Most existing approaches directly translate natural-language specifications into Verilog Thakur et al. (2024); Liu et al. (2023; 2024a), making it difficult to distinguish specification-interpretation errors from coding errors. Evidence from VerilogEval and RTLCoder shows substantial performance degradation on human-written specifications compared with machine-generated specifications, highlighting the difficulty of recovering implementation semantics from natural language Liu et al. (2023; 2024a). Recent staged and feedback-driven approaches Nadimi et al. (2025); Tsai et al. (2024); Ho et al. (2025); Zhao et al. (2025b); Yubeaton et al. (2025); Ping et al. (2025b); Wang et al. (2026b); Hong et al. (2024) improve generation through intermediate reasoning, compilation, simulation, or testbench feedback. However, intermediate reasoning does not by itself establish that the represented behavior is consistent with the source specification. Structured intermediate representations have begun to address this gap DeLorenzo et al. (2025), but existing methods typically generate the representation once and use it without validating its coverage, consistency, or faithfulness to the source specification, allowing interpretation errors to propagate into RTL.

To address these limitations, we propose *VeriRefine*, an agentic abstraction-refinement framework that explicitly represents and audits specification interpretations before RTL generation. The framework introduces *Abstract Signal Transition Functions* (ASTFs), a schema-constrained, per-signal representation of declarations, clocking, reset behavior, and guarded updates. We define the ASTF-based intermediate representation (IR) as a JSON document that combines signal-level ASTFs with module-level metadata and constraints. The ASTF-based IR provides a traceable design-intent layer for identifying and revising interpretation errors before they propagate into RTL. Auditing

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

checks specification grounding, coverage, consistency, state-transition integrity, and RTL design-rule compliance. The audited ASTF-based IR then guides RTL generation, and internally generated testbenches provide feedback for simulation-guided debugging. Failures attributed to the ASTF-based IR are classified as ASTF-level issues and routed to representation refinement, RTL-level issues are routed to implementation repair. This separation provides distinct targets for correcting specification-interpretation and implementation errors. The main contributions of this work are:

- We introduce VeriRefine, an abstraction-refinement framework with representation-level auditing and root-cause-directed repair for LLM-enabled RTL generation.
- We propose ASTF, a signal-level representation that makes hardware-relevant semantics explicit and traceable to the specification. The module-level intermediate representation integrates these ASTFs with module metadata and constraints, enabling specification-grounded auditing to identify and correct interpretation errors before RTL generation.
- VeriRefine achieves 96.0% and 98.1% functional correctness on *RTLLM v2.0* and *VerilogEval-Human v2*, improving over direct generation by 32.0 and 12.8 percentage points. Compared with our VerilogCoder reproduction, VeriRefine passes three more designs on *RTLLM v2.0* and two fewer on *VerilogEval-Human v2*. It consumes 13.2% and 26.6% as many tokens under the evaluated configurations.

2 Background and Related Work

LLM-based RTL generation has advanced tremendously in recent years. RTLCoder, CodeV, and ScaleRTL Liu et al. (2024a); Zhao et al. (2025a); Deng et al. (2025) improve generation through specialized modeling and data construction, while VeriReason, VeriRL, and VeriPrefer Wang et al. (2025a); Teng et al. (2025); Wang et al. (2025b) leverage testbench feedback, reward modeling, and preference optimization. Other methods incorporate non-textual specifications, synthesis or formal feedback, retrieval, and evolutionary search Liu et al. (2025); Chen et al. (2026); Ping et al. (2025b); Ibnat et al. (2025); Yubeaton et al. (2025); Hsin et al. (2026). Agentic frameworks such as VerilogCoder, MAGE, VERIMOA, Spec2RTL-Agent, VeriGraphi, and VeriOpt Ho et al. (2025); Zhao et al. (2025b); Ping et al. (2025a); Yu et al. (2025); Islam et al. (2026); Tasnia et al. (2025) further introduce planning, debugging, specialist-agent coordination, structured representations, and PPA optimization. Despite these advances, specification interpretation is generally implicit, with errors exposed only through stream compilation, simulation, synthesis, or formal feedback. Their intermediate states are also typically unstructured or used primarily to guide generation rather than independently verified before RTL emission. VeriRefine instead audits ASTFs before code generation and routes failures to either representation refinement or RTL repair.

Intermediate representations (IRs) have also been explored to bridge the semantic gap between specifications and RTL. Abstraction-of-Thought, AoT-RTL, ComplexVCoder, QiMeng-CRUX, and paradigm based HDL Hong et al. (2024); DeLorenzo et al. (2025); Zuo et al. (2025); Huang et al. (2026); Sun et al. (2025) use explicit abstractions, structured representations, or circuit paradigms, while AutoFSM and LLM-FSM Luo et al. (2025); Wu et al. (2026) target FSM-based synthesis. Other approaches represent existing RTL using CFG/DFG graphs or incorporate security knowledge, pseudocode, and formal representations Akyash et al. (2025); Hu et al. (2025); Wang et al. (2026b); Basu Roy et al. (2025), while NL2Silicon Fu et al. (2026) demonstrates the impact of representation choice on RTL-generation performance. However, many existing IRs are fixed, design-specific, derived from RTL, or primarily prompting scaffolds. In contrast, VeriRefine introduces Abstract Signal Transition Functions (ASTFs) as a general, specification-grounded, per-signal representation that is audited before RTL generation. Simulation-based failure diagnosis then determines whether refinement should target the design-intent representation or the generated RTL.

3 VeriRefine Framework

Fig. 1 presents an overview of the VeriRefine framework. Given a natural-language specification, the Refinement module (Section 3.2) refines its requirements into an ASTF-based intermediate representation (IR). ASTFs (Section 3.1) express per-signal declarations, hardware attributes, and guarded commands, while the JSON schema organizes these records and module-level design information.

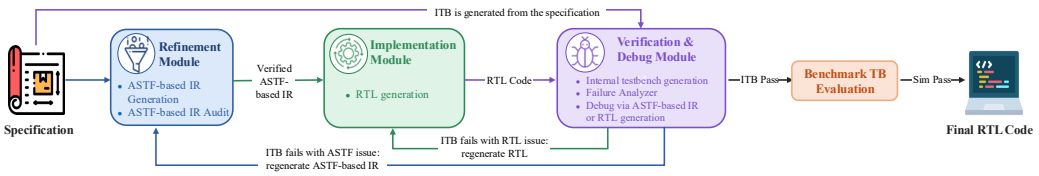

Figure 1: Overview of the VeriRefine framework

The module validates schema conformance and audits the represented behavior against the source specification, using detected violations to guide iterative refinement. The Implementation module (Section 3.3) translates the audited IR into synthesizable Verilog by mapping declarations, hardware types, and guarded commands to corresponding RTL structures and signal assignments. Finally, the Verification & Debug module (Section 3.4) verifies and debugs the generated RTL using internally generated testbenches before producing the final RTL design.

3.1 Abstract Signal Transition Functions (ASTF)

Direct RTL generation forces LLMs to infer signal declarations, hardware structure, and cycle-level behavior while simultaneously producing syntactically valid Verilog. VeriRefine decouples these responsibilities by introducing Abstract Signal Transition Functions (ASTFs), a signal-level representation for module-driven signals. By making implementation semantics explicit before RTL generation, ASTFs allow hardware logic to be checked and validated independently from the final code. Formally, let $\mathcal{S}$ be the set of internal and output signals driven by the target design. For each $s \in \mathcal{S}$, its ASTF is defined as:

$$ASTF_s = \langle D_s, H_s, G_s \rangle.$$

D_s specifies declaration attributes, including signal name, direction, data type, width, and optional memory depth. H_s specifies the hardware implementation class, including logic style, clock domain, reset behavior, and optional initialization or periodic-delay information. G_s specifies the signal behavior as a set of guarded commands

$$G_s = \langle (c_1, a_1, \sigma_1), \ldots, (c_n, a_n, \sigma_n) \rangle.$$

Here, c_i is a Boolean guard determining when action a_i can take place. Action a_i specifies a particular update assigned to signal s. σ_i is a verbatim citation from the input specification. Thus, D_s determines the Verilog declaration, H_s determines the RTL implementation structure, and G_s determines the signal-update logic.

The ordering of G_s is semantic. When multiple guards are satisfied, the first determines the update to s, preserving the priority of the generated `if/else if` structure. For clocked signals, guards and update expressions are evaluated using register values before the triggering edge updates them. This avoids requiring the LLM to rewrite overlapping cases, such as reset and update, into mutually exclusive guards before audit. Parameterized behavior is represented by loop-index fields, such as `index_var`, `range_from`, and `range_to`, which encode repeated bit- or array-level behavior without enumerating each element.

The ASTF-based IR combines signal-level ASTFs with module-level metadata and constraints:

$$\langle M, P, I, \mathcal{F}, B, \Pi \rangle.$$

M is the module name, P is the parameter array, I is the set of input-signal declarations, B is the optional boundary-condition set, and Π is optional port-order metadata. $\mathcal{F}$ contains ASTFs for all internal and output signals of the target design:

$$\mathcal{F} = \{ASTF_s \mid s \in \mathcal{S}\}.$$

The module-level fields record module metadata and explicit constraints associated with the signal-level ASTFs. The parameter array P stores both module-header parameters and internal constants. Entries without `localparam` are emitted as Verilog module parameters, whereas entries with `localparam: true` are emitted as internal `localparam`s. FSM state encodings are represented as such localparam entries; the state register is an ordinary internal sequential signal, and its transitions are encoded in $\mathcal{F}$ as guarded commands on that register.

The optional fields B and Π address two common sources of specification-to-RTL mismatch. B records boundary behavior at counter, accumulator, or waveform limits, such as wraparound, saturation, hold, or terminal-cycle updates. This makes boundary semantics explicit before RTL generation and gives the audit concrete targets for off-by-one behavior. Π records positional port order when required by the specification or benchmark interface, preventing modules with correct port names from failing under positional testbench binding.

To illustrate how ASTFs guide RTL generation, consider an FSM-based pulse detector whose specification requires `data_out` to assert in the final cycle of each input pulse. Direct generation incorrectly registers this output, delaying its assertion by one cycle. VeriRefine instead assigns the state register a sequential hardware type and assigns `data_out` a combinational hardware type with guards over the current state and input. The resulting RTL implements state updates in clocked logic and output conditions in combinational logic, preserving the required same-cycle assertion.

3.2 Refinement Module

The Refinement Module constructs and audits an ASTF-based IR from the natural-language specification before RTL generation. By integrating per-signal ASTFs with module-level properties, the IR makes the model's interpretation of design intent explicit, enabling specification-grounded auditing to identify and correct interpretation errors before code generation.

ASTF-based IR Generation. The module first analyzes the specification to select refinement rules for IR generation. The IR supports heterogeneous specification evidence, including waveforms, Karnaugh maps, examples, and other specification forms, without requiring form-specific IR fields. VeriRefine therefore constructs the IR-generation prompt from a shared base prompt and a set of selected rule blocks. The base prompt defines the common output structure: module-level fields are emitted once, each driven signal is represented by an ASTF, guarded commands include source citations, and unsupported fields are rejected. Rule blocks impose additional constraints for recurring RTL structures, such as FSMs, counters, FIFOs, dividers, and table-defined combinational logic.

For table-defined combinational logic, VeriRefine deterministically preprocesses supported machine-readable formats, including Karnaugh maps, mux-vector tables, and decimal truth tables. The extracted output rows and don't-care entries are parsed and checked by the same audit parser, then inserted into the IR-generation prompt as constraints and reused during auditing. This reduces the LLM's need to reinterpret deterministic table content while preserving end-to-end automation. If no supported table is detected, VeriRefine uses the general refinement prompt.

ASTF-based IR Auditing. The IR is validated in two stages. First, JSON-schema validation checks structural conformance, including required fields, declared fields, and signal records. If validation fails, VeriRefine rejects the IR and returns the validation errors to the Refinement Module. Only schema-valid IRs proceed to semantic auditing, which evaluates the IR across five layers summarized in Table 1: specification grounding, coverage, consistency, state-transition integrity, and RTL design-rule compliance. These layers verify that guarded commands are supported by the specification, required ports and behaviors are captured, declarations and guards are consistent, symbolic state encodings and transitions are complete, and no pre-RTL design-rule violations are present.

When schema validation or semantic auditing identifies a violation, VeriRefine converts it into targeted feedback for the Refinement Module. The repair prompt identifies the affected field, signal, or audit condition and instructs the model to revise only the affected content while preserving valid IR components. This localized refinement enables errors such as missing guards, incorrect hardware types, inconsistent state encodings, and omitted boundary cases to be corrected at the representation level before Verilog generation.

3.3 Implementation Module

The Implementation Module translates the audited ASTF-based IR into Verilog code. Within a single LLM invocation, a structured prompt guides the model to build the module interface, parameters, and internal signal declarations, applying initialization statements per target implementation semantics. The model generates combinational or sequential logic according to each signal's `hardware_type`, handling clock and reset logic while translating guarded commands into conditional assignments that preserve priority ordering.

Table 1: Semantic audit layers for ASTF refinement.

Layer	Check Objective
Spec grounding	Check that guarded-command citations are traceable to the source specification, flagging excerpts that cannot be matched to the input text.
Coverage	Check required ports, interface attributes, and specification-derived behavioral requirements, including missing signal-update cases where these can be extracted.
Consistency	Check agreement among declarations, referenced identifiers, guard conditions, parsed table entries, and boundary records. Flag contradictions between constraints and signal updates.
STI	Check that symbolic state and reset values resolve to declared encodings and fit the state-register width. Check outgoing transitions and supported input-case coverage.
RDRC	Check IR-level RTL rules, including assignment-style consistency, combinational default coverage, dependency cycles, loop-index bounds, and arithmetic result sizing.

STI: state-transition integrity. RDRC: RTL design-rule compliance.

The prompt integrates static Verilog guidelines and reference examples with dynamic IR definitions and repair feedback. To reduce latency and computational cost across iterations, static instructions and examples are configured for prompt caching. Generated RTL is evaluated through automated structural and syntactic checks:

1. *Interface Consistency:* Verifies module names and port declarations against IR metadata.
2. *Language Rules:* Scans for non-synthesizable or disallowed Verilog constructs.
3. *Compilation:* Compiles the code to catch syntax and typing errors.

When errors occur, the system returns diagnostic feedback and candidate RTL to the LLM for targeted repair, repeating generation until the design compiles cleanly or the retry budget is exceeded.

3.4 Verification & Debug Module

This module verifies the generated RTL against an internal testbench (ITB) with specification-derived stimuli, and repairs the design when the ITB exposes a mismatch. It performs two tasks: ITB generation and simulation-guided debugging.

3.4.1 Internal Testbench (ITB) Generation

ITB generation is a two-stage LLM pipeline driven only by the natural-language specification. It does not access the ASTF-based IR or RTL generated by VeriRefine, so its tests are not biased toward the implementation.

Verification plan. The LLM first produces a schema-constrained JSON plan containing the interface (port names, directions, and widths), clock and reset semantics (polarity, synchronous or asynchronous, reset values), a short list of assumptions for underspecified behavior, the behaviors to verify with their observable outputs, and concrete test scenarios. Each scenario records its triggering condition, stimulus, expected outputs, and covered behaviors. Every plan contains directed functional and corner-case scenarios, one handcrafted multi-cycle sequence, and one pseudo-random stress scenario of 80–100 cycles for combinational designs or 200 cycles for sequential designs.

Stimulus and output checking. From the plan and specification, the LLM generates a SystemVerilog module, `itb_stimulus`, that drives the DUT inputs, samples the outputs after they settle. For reference-assisted VerilogEval runs, the harness compares DUT outputs with those of the provided reference RTL under the same inputs. In the absence of a reference model, the generated stimulus compares outputs with expected values derived from the plan, which serve as the ITB oracle. Each mismatch is logged with its scenario, step, input vector, expected value, and observed value, and simulation continues through all scenarios. A deterministic script assembles the harness, which instantiates the DUT and stimulus module, includes the reference module when applicable, and supplies the clock and a timeout watchdog.

Infrastructure checks. Generated stimulus is linted and compiled before use. Errors attributed to ITB-owned files trigger regeneration of the plan and stimulus within a separate budget of two attempts, without changing the RTL, and are never passed to the failure analyzer. If no valid ITB is obtained, debugging stops and the current RTL is submitted for final evaluation. The simulation log is condensed into a digest reporting the verdict, mismatch count, first mismatch time, completed and failing scenarios, grouped mismatch patterns, and representative failing samples. This digest is the feedback for simulation-guided debugging.

3.4.2 Simulation-Guided Debugging

The internal testbench provides feedback for debugging. The LLM-based failure analyzer receives a diagnostic prompt, the specification, a compact ASTF-based summary, the generated RTL, and the ITB simulation digest or compilation log. Using these inputs, it identifies the root cause and provides repair hints for ASTF-level or RTL-level errors. If the analyzer identifies an ASTF-level error, the ASTF-based IR does not correctly represent the behavior required by the specification. The framework then returns to the Refinement Module to correct the IR. If it identifies an RTL-level error, the RTL does not correctly implement the behavior described by the IR. The framework then returns to the Implementation Module to repair the RTL. The benchmark testbench evaluates the final RTL and provides no feedback to IR refinement or RTL repair.

4 Experiments and Results

4.1 Experimental Setup

Benchmarks. We evaluate VeriRefine on *RTLLM v2.0* Liu et al. (2024b) (50 designs) and *VerilogEval-Human v2* Liu et al. (2023) (156 designs). These benchmarks provide complementary specification styles for evaluating specification interpretation and RTL generation. *RTLLM v2.0* provides design descriptions with implementation guidance, whereas *VerilogEval-Human v2* emphasizes behavior through interface descriptions and input–output requirements. Together, they evaluate whether VeriRefine can preserve implementation requirements and infer RTL behavior from function-oriented specifications.

Models. We evaluate direct generation and VeriRefine-Direct using GPT-4 Turbo and Sonnet 4.6. The VeriRefine framework and the VerilogCoder reproduction are evaluated using Sonnet 4.6. Both models are accessed through their official APIs. For direct generation and the VeriRefine configurations, the sampling temperature is zero to reduce generation variability. The VerilogCoder reproduction retains the prompts and per-agent temperature settings of the original implementation.

Evaluation Conditions. We evaluate four conditions:

1. *Direct generation* serves as the baseline and generates Verilog from the specification.
2. *VeriRefine-Direct* generates an audited ASTF-based IR via the Refinement Module, then translates it into Verilog via the Implementation Module without simulation-guided repair.
3. *VeriRefine* extends VeriRefine-Direct with the Verification & Debug Module, allowing up to two repair iterations per design.
4. *VerilogCoder* Ho et al. (2025) serves as the SOTA agentic baseline. We reproduce the released implementation using Sonnet 4.6, preserving the original agents, prompts, and per-agent temperature settings for comparison with VeriRefine.

Metrics. We report functional correctness, synthesizability, and token cost.

1. *Functional correctness.* A task passes if the generated RTL compiles and passes the benchmark's testbench. The functional pass rate is

$$\text{Functional pass rate} = \frac{N_{\text{passed}}}{N_{\text{total}}} \times 100\%, \tag{1}$$

where N_{total} includes all benchmark tasks. All four evaluation conditions, including VerilogCoder, use one complete run per task, with internal repair attempts where applicable.

Table 2: Main results on RTLLM v2.0 and VerilogEval-Human v2.

Method	Func. Correctness	Synthesizability	Tokens(M)
RTLLM v2.0			
Direct generation (GPT-4 Turbo)	25/50 (50.0%)	25/25 (100.0%)	0.05
Direct generation (Sonnet 4.6)	32/50 (64.0%)	32/32 (100.0%)	0.06
VeriRefine-Direct (GPT-4 Turbo)	34/50 (68.0%)	34/34 (100.0%)	1.32
VeriRefine-Direct (Sonnet 4.6)	42/50 (84.0%)	41/42 (97.6%)	1.53
VerilogCoder (Sonnet 4.6)	45/50 (90.0%)	45/45 (100.0%)	22.7
VeriRefine (Sonnet 4.6)	48/50 (96.0%)	47/48 (97.9%)	3.00
VerilogEval-Human v2			
Direct generation (GPT-4 Turbo)	98/156 (62.8%)	98/98 (100.0%)	0.07
Direct generation (Sonnet 4.6)	133/156 (85.3%)	133/133 (100.0%)	0.09
VeriRefine-Direct (GPT-4 Turbo)	114/156 (73.1%)	113/114 (99.1%)	5.53
VeriRefine-Direct (Sonnet 4.6)	143/156 (91.7%)	143/143 (100.0%)	5.91
VerilogCoder (Sonnet 4.6)	155/156 (99.4%)	155/155 (100.0%)	30.5
VeriRefine (Sonnet 4.6)	153/156 (98.1%)	153/153 (100.0%)	8.10

Pass status is determined by the final RTL's benchmark testbench verdict. This metric measures success per complete run, rather than pass@k over multiple independently sampled candidates. Intermediate repair candidates are not counted as independent samples.

2. *Synthesizability.* Among functionally correct designs, we report the percentage for which logic synthesis produces a gate-level netlist without errors.
3. *Token cost.* We report total token consumption for each evaluation condition over the entire benchmark, in millions. The total includes input tokens (including cached input), output tokens, and all retry and repair attempts across both successful and failed tasks.

Implementation. VeriRefine is implemented in Python. The `jsonschema` library validates the IR against the JSON Schema Draft 2020-12 specification, and Icarus Verilog 13.0 performs functional simulation. Synthesizability is assessed with Yosys 0.59+110 using its generic synthesis flow (`hierarchy -auto-top` followed by `synth`).

4.2 Main Results

Table 2 summarizes the results for functional correctness, synthesizability, and token consumption.

Functional correctness. VeriRefine-Direct improves functional correctness over direct generation for both models on both benchmarks. On *RTLLM v2.0*, the pass rate increases from 50.0% to 68.0% with GPT-4 Turbo and from 64.0% to 84.0% with Sonnet 4.6. The gains are 18.0 and 20.0 percentage points. On *VerilogEval-Human v2*, the respectiv gains are 10.3 and 6.4 percentage points. These gains suggest that the audited ASTF-based flow improves RTL functional correctness even without simulation-guided repair. With Sonnet 4.6, iterative debugging increases the number of passing designs from 42 to 48 on *RTLLM v2.0* and from 143 to 153 on *VerilogEval-Human v2*. VeriRefine achieves pass rates of 96.0% and 98.1%, exceeding direct generation by 32.0 and 12.8 percentage points. Using the same model, VeriRefine passes three more designs than VerilogCoder on *RTLLM v2.0* (48 vs. 45) but two fewer on *VerilogEval-Human v2* (153 vs. 155). These results reflect the verification configurations used by each framework, including reference-assisted checking in VeriRefine's VerilogEval ITB runs. The remaining failures are analyzed in Section 4.4.

Synthesizability. Synthesis is evaluated only for functionally correct designs. VeriRefine achieves synthesis pass rates of 97.9% (47/48) on *RTLLM v2.0* and 100% (153/153) on *VerilogEval-Human v2*, compared with 100% on both benchmarks for VerilogCoder (45/45 and 155/155). The sole synthesis failure in VeriRefine is `clkgenerator`, whose specified free-running clock is implemented using simulation-only # delays. The corresponding VerilogCoder implementation fails functional

verification and is excluded from synthesis evaluation. These rates therefore reflect different sets of functionally correct designs. On *RTLLM v2.0*, VeriRefine yields 47 designs satisfying both functional correctness and synthesizability, compared with 45 for VerilogCoder.

Token cost. Token efficiency is a practical consideration for deploying LLM-based RTL generation under limited inference budgets. Prior work emphasizes joint evaluation of agent accuracy and cost Kapoor et al. (2024). We therefore compare token consumption and functional correctness across the Sonnet 4.6 configurations.

On *RTLLM v2.0*, direct generation, VeriRefine-Direct, and VeriRefine consume 0.06M, 1.53M, and 3.00M tokens, achieving pass rates of 64.0%, 84.0%, and 96.0%, respectively. On *VerilogEval-Human v2*, the corresponding token totals are 0.09M, 5.91M, and 8.10M, with pass rates of 85.3%, 91.7%, and 98.1%. The audited ASTF-based flow improves correctness at additional token cost. Enabling iterative debugging adds 1.47M tokens for six additional passing designs on *RTLLM v2.0* and 2.19M tokens for ten on *VerilogEval-Human v2*.

Compared with VerilogCoder, VeriRefine uses only 13.2% of the tokens on *RTLLM v2.0* while achieving higher functional correctness (96.0% vs. 90.0%). On *VerilogEval-Human v2*, it uses only 26.6% of the tokens while achieving comparable correctness (98.1% vs. 99.4%). These results demonstrate that VeriRefine achieves competitive RTL functional correctness with substantially lower token consumption than the SOTA baseline.

4.3 Ablation Study

We evaluate two integrated stages, audited ASTF-based generation and iterative debugging, by comparing direct generation, VeriRefine-Direct, and VeriRefine using Claude Sonnet 4.6. VeriRefine-Direct corresponds to iteration 0 and generates RTL through the audited ASTF-based flow without simulation-guided repair. Iterations 1 and 2 enable the Verification & Debug Module, which classifies internal-testbench failures as ASTF-level or RTL-level issues and directs repair to the corresponding module. Each iteration uses feedback from the latest candidate. The benchmark testbench determines the reported functional correctness. Comparing direct generation with iteration 0 evaluates the combined effect of ASTF-based refinement and auditing. Comparing iteration 0 with iterations 1 and 2 measures the additional benefit of simulation-guided diagnosis and repair.

Fig. 2 reports functional correctness across repair iterations using Sonnet 4.6. Without simulation-guided repair, VeriRefine-Direct achieves 84.0% on *RTLLM v2.0* and 91.7% on VerilogEval-Human v2. These pass rates exceed those of direct generation by 20.0 and 6.4 percentage points, respectively, supporting the effectiveness of the audited ASTF-based generation flow before simulation-guided debugging.

Iterative debugging further improves functional correctness on both benchmarks. On *RTLLM v2.0*, each repair iteration adds three passing designs, increasing the total from 42 to 45 and then 48 of 50 designs. The corresponding pass rates are 90.0% and 96.0%. On *VerilogEval-Human v2*, the two iterations add seven and three passing designs, increasing the total from 143 to 150 and then 153 of 156 designs. The corresponding pass rates are 96.2% and 98.1%

These results support the effectiveness of audited ASTF-based generation as an integrated flow and demonstrate further gains from simulation-guided debugging. The comparisons evaluate the two stages as complete components. They do not separately quantify the contributions of ASTF construction and auditing or establish the advantage of diagnostic routing over code-only repair.

4.4 Discussion on Failure Cases

We analyze the five remaining functional correctness failures of VeriRefine and related defects in both benchmarks, and examine synthesis failures.

Functional correctness failure analysis. The five remaining failures under VeriRefine comprise four cases with specification defects and one with a benchmark-testbench defect. These observations are consistent with prior findings on RTL benchmark defects Wang et al. (2026a). In *radix2_div*, the specification omits the `res_ready` input used by the benchmark and leaves several timing and encoding facts undefined, including the width/sign extension of `NEG_DIVISOR`, the valid cycle of `result`, the mutual exclusion between initiation and execution, and the quotient location in

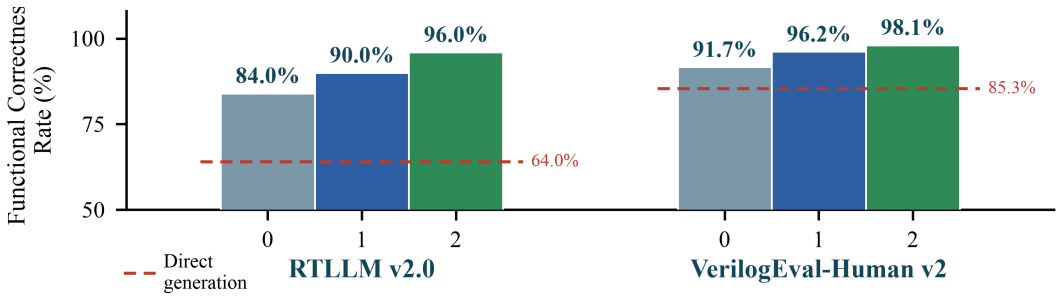

Figure 2: Functional correctness across bug repair iterations.

SR. In *freq_divbyeven*, the specification gives the module name as `freq_diveven`, but the benchmark instantiates `freq_divbyeven`. It states only that `NUM_DIV` is even and compatible with a four-bit counter, while the benchmark assumes `NUM_DIV`=6. These facts are benchmark-specific and cannot be inferred from the text. In *Prob153_gshare*, the specification defines a table of two-bit saturating counters but omits the reset state and prediction bit. Multiple reset encodings are specification-consistent but produce different behavior after updates. Stating that each entry resets to the weakly-not-taken state and that the upper bit is the prediction resolved the failure. In *Prob149*, the specification gives the supplemental-flow rule for `dfr` in the wrong direction and omits history needed to distinguish identical water levels reached from different directions. It omits the one-level-per-cycle transition constraint. Correcting these facts allowed VeriRefine to generate passing RTL without changing the method. The remaining case, `Prob099`, contains a testbench interface mismatch. The specification and reference module expose ports `Y1` and `Y3`, but the testbench binds ports `Y2` and `Y4`. Because the incorrect bindings also affect the reference instance, modifying the generated RTL alone cannot resolve the elaboration failure. The correction updates both instances to use the declared ports. After correcting the affected specifications or testbench, the corresponding designs passed the benchmark tests in separate reruns. These cases identify benchmark defects that impede agreement between specification-derived RTL and external evaluation.

Synthesis failure analysis. `clkgenerator` in RTLLM v2.0 generated by VeriRefine (Sonnet 4.6) is functionally correct, however the target design of this specification is not synthesizable as a delay $\#n$ is specified for the design which is not supported by the synthesis flow. For *VerilogEval-Human v2*, VeriRefine achieves 100% synthesizability even though it generates two fewer functionally correct designs compared with VerilogCoder as explained above. For the above five failure cases, all of them pass synthesis after the corresponding corrections are made.

5 Conclusion

We presented *VeriRefine*, an LLM-enabled abstraction-refinement framework that refines specifications into an audited ASTF-based intermediate representation before RTL generation. VeriRefine achieves functional pass rates of 96.0% on *RTLLM v2.0* and 98.1% on *VerilogEval-Human v2*, using 13.2% and 26.6% of the tokens consumed by our VerilogCoder reproduction, respectively. The ablation study shows that audited ASTF-based generation improves functional correctness over direct generation, while simulation-guided debugging provides further gains. These results support making specification interpretations explicit and auditable before RTL implementation. Analysis of the five remaining functional failures identifies four specification defects and one testbench defect associated with these failures. Future work will address ambiguous and incomplete specifications and improve the reliability of generated internal testbenches. We plan to release the framework code and prompts on Github to support reproducibility.

Acknowledgement This work is supported by the National Science Foundation under Grant No. ——. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

REFERENCES

Mohammadreza Akyash, Kimia Azar, and Hadi Kamali. Rtl++: Graph-enhanced llm for rtl code generation. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, 2025.

Prithwish Basu Roy et al. Veritas: Deterministic verilog code synthesis from llm-generated conjunctive normal form. *arXiv preprint arXiv:2506.00005*, 2025.

Zhirong Chen, Kaiyan Chang, Zhuolin Li, Cangyuan Li, Xinyang He, Chujie Chen, Mengdi Wang, Haobo Xu, Yinhe Han, Huawei Li, et al. Chipseek: Optimizing verilog generation via eda-integrated reinforcement learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 25180–25201, 2026.

Matthew DeLorenzo, Kevin Tieu, Prithwish Jana, Piyush Jha, Dileep Kalathil, Vijay Ganesh, and Jeyavijayan Rajendran. Abstractions-of-thought: Intermediate representations for llm reasoning in hardware design. *arXiv preprint arXiv:2505.15873*, 2025.

Chenhui Deng, Yun-Da Tsai, Guan-Ting Liu, Zhongming Yu, and Haoxing Ren. Scalertl: Scaling llms with reasoning data and test-time compute for accurate rtl code generation. *arXiv preprint arXiv:2506.05566*, 2025.

Wei Fu et al. From natural language to silicon: The representation bottleneck in llm hardware design. *arXiv preprint arXiv:2604.17097*, 2026.

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

Ruixin Hong, Hongming Zhang, Xinyu Pan, Dong Yu, and Changshui Zhang. Abstraction-of-thought makes language models better reasoners. *arXiv preprint arXiv:2406.12442*, 2024.

Wei-Po Hsin, Ren-Hao Deng, Yao-Ting Hsieh, En-Ming Huang, and Shih-Hao Hung. Evolve: Evolutionary search for llm-based verilog generation and optimization. *arXiv preprint arXiv:2601.18067*, 2026.

Ziteng Hu, Yiwen Xia, Xiang Chen, and Liang Kuang. Secfsm: Knowledge graph-guided verilog code generation for secure finite state machines in soc. *arXiv preprint arXiv:2508.12910*, 2025.

Lei Huang et al. Qimeng-crux: Narrowing the gap between natural language and verilog via core refined understanding expression for circuit design. *arXiv preprint arXiv:2511.20099*, 2026.

Zahin Ibnat, Paul E Calzada, Rasin Mohammad Ihtemam, Sujan Kumar Saha, Jingbo Zhou, Farimah Farahmandi, and Mark Tehranipoor. Deepv: A model-agnostic retrieval-augmented framework for verilog code generation with a high-quality knowledge base. *arXiv preprint arXiv:2510.05327*, 2025.

Sazzadul Islam, Tasnim Tabassum, and Hao Zheng. Verigraphi: A multi-agent framework of hierarchical rtl generation for large hardware designs. *arXiv preprint arXiv:2604.14550*, 2026.

Sayash Kapoor, Benedikt Stroebl, Zachary S Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter. *arXiv preprint arXiv:2407.01502*, 2024.

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

Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. Craftrtl: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair. *arXiv preprint arXiv:2409.12993*, 2025.

Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 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)*, pp. 1–5. IEEE, 2024a.

Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 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*, pp. 1–9, 2024b.

Qing Luo et al. Autofsm: A multi-agent framework for fsm code generation with intermediate representation and systemc-based testing. *arXiv preprint arXiv:2512.11398*, 2025.

Bardia Nadimi, Ghali Omar Boutaib, and Hao Zheng. Verimind: Agentic llm for automated verilog generation with a novel evaluation metric. *arXiv preprint arXiv:2503.16514*, 2025.

Haoyuan Ping et al. Verimoa: A mixture-of-agents framework for spec-to-hdl generation. *arXiv preprint arXiv:2510.27617*, 2025a.

Heng Ping, Shixuan Li, Peiyu Zhang, Anzhe Cheng, Shukai Duan, Nikos Kanakaris, Xiongye Xiao, Wei Yang, Shahin Nazarian, Andrei Irimia, et al. Hdlcore: A training-free framework for mitigating hallucinations in llm-generated hdl. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 108–116. IEEE, 2025b.

Wenhao Sun, Bohan Li, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, and Ulf Schlichtmann. Paradigm-based automatic hdl code generation using large language models. *arXiv preprint arXiv:2501.12702*, 2025.

Kimia Tasnia, Alexander Garcia, Tasnuva Farheen, and Sazadur Rahman. Veriopt: Ppa-aware high-quality verilog generation via multi-role llms. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9. IEEE, 2025.

Fu Teng et al. Verirl: Boosting llm-based verilog code generation via reinforcement learning. *arXiv preprint arXiv:2508.18462*, 2025.

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

YunDa Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language model. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pp. 1–6, 2024.

Jing Wang, Shang Liu, Hangan Zhou, and Zhiyao Xie. Rtl-benchmt: Dynamic maintenance of rtl generation benchmark through agent-assisted analysis and revision. *arXiv preprint arXiv:2605.15537*, 2026a.

Nan Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Nan Guan, and Zhe Jiang. Verireason: Reinforcement learning with testbench feedback for reasoning-enhanced verilog generation. *arXiv preprint arXiv:2505.11849*, 2025a.

Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Nan Guan, and Zhe Jiang. Insights from verification: Training a verilog generation llm with reinforcement learning with testbench feedback. *arXiv preprint arXiv:2504.15804*, 2025b.

Runzhi Wang, Prianka Sengupta, Cristhian Roman-Vicharra, Yiran Chen, and Jiang Hu. Lorecast: Layout-aware performance and power forecasting from natural language. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 304–310. IEEE, 2026b.

Yuhao Wu et al. Llm-fsm: Scaling large language models for finite-state reasoning in rtl code generation. *arXiv preprint arXiv:2602.07032*, 2026.

Zhongkai Yu, Mingjie Liu, Max Zimmer, Yi-Chun Lin, Yang Liu, and Haoxing Ren. Spec2rtl-agent: Automated hardware code generation from complex specifications using llm agent systems. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, 2025.

Patrick Yubeaton, Andre Nakkab, Wenjie Xiao, Luca Collini, Ramesh Karri, Chinmay Hegde, and Siddharth Garg. Verithoughts: Enabling automated verilog code generation using reasoning and formal verification. *arXiv preprint arXiv:2505.20302*, 2025.

Jiazheng Zhang, Cheng Liu, Long Cheng, Xiaowei Li, and Huawei Li. Understanding and mitigating errors of llm-generated rtl code. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2026.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Ziyuan Nan, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, et al. Codev: Empowering llms for verilog generation through multi-level summarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025a.

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

Jian Zuo, Junzhe Liu, Xianyong Wang, Yicheng Liu, Navya Goli, Tong Xu, Hao Zhang, Umamaheswara Rao Tida, Zhenge Jia, and Mengying Zhao. Complexvcoder: An llm-driven framework for systematic generation of complex verilog code. *arXiv e-prints*, pp. arXiv–2504, 2025.

A VeriRefine RTL Generation Framework

This appendix presents the generation and verification of the RTLLM design `freq_divbyodd` using VeriRefine. The example documents specification refinement, ASTF auditing, RTL generation, and internal and benchmark simulation. The listings contain saved execution artifacts and condensed prompt templates. Section A.6 presents the simulation-guided debugging procedure, which was not invoked in this execution.

A.1 Design Specification

This example combines rising- and falling-edge state updates, counter boundaries, and a combinational output in one module. The odd-frequency divider divides the clock frequency by an odd integer specified by `NUM_DIV`, with a default of five. The specification assigns two counters to the rising and falling clock edges. Each counter controls an intermediate divider signal, and the logical OR of these signals produces `clk_div`. An active-low reset initializes the counters and divider signals. These requirements define the interface, both clocked paths, and the combinational output.

Input Specification

```
Please act as a professional verilog designer.

A frequency divider that divides the input clock frequency by odd numbers. The module
     generates a divided clock output by an odd divisor value provided as a parameter.

Module name:
    freq_divbyodd

Input ports:
    clk: Input clock signal.
    rst_n: Active low reset signal that initializes the divider.

Output ports:
    clk_div: Divided clock output.

Implementation:

The module divides the input clock frequency by an odd number defined by the parameter
     NUM_DIV, which defaults to 5.
Two counters, cnt1 and cnt2, are used for tracking the rising and falling edges of the
     clock. Each counter counts up to NUM_DIV - 1.
Two separate clock dividers, clk_div1 and clk_div2, are used for positive and negative
     edges of the clock, respectively. These are toggled when the counters reach half of
     NUM_DIV.
The final divided clock output, clk_div, is derived by logically OR-ing clk_div1 and
     clk_div2 to account for both clock edges.
The active low reset signal rst_n initializes the counters and the clock divider outputs.


Give me the complete code.
```

Listing 1: Complete benchmark specification for the RTLLM odd-frequency divider.

A.2 ASTF-Based Specification Refinement and Auditing

The Refinement Module extracts signal declarations, hardware attributes, and update rules from the specification according to Sections 3.1 and 3.2. The resulting ASTFs encode update conditions, command priority, and boundary behavior. The ASTF-based IR combines these signal records with module metadata for validation and semantic auditing.

The IR-generation prompt combines common instructions, applicable design rules, reference examples, and the input specification. For this design, the rules address sequential updates, clock edges, and counter boundaries. Each guarded command includes a verbatim specification excerpt. Listing 2 summarizes the generation instructions.

IR-Generation Prompt

```
Refine the natural-language specification into signal-level IR.
1. Preserve the specified module, parameters, and port contract.
2. Identify each output and internal driven signal. Classify
   its logic style before defining its update behavior.
3. Construct one ASTF per driven signal with specs,
   hardware_type, and guarded_commands. Inputs use specs only.
4. Make clock edges, reset behavior, update conditions, and
   command priority explicit. Preserve intermediate signals
   required by the specified implementation structure.
5. Attach an exact specification excerpt to each command
   using spec_citation. Do not invent supporting quotations.
6. Record counter and switching limits in boundary_conditions.
   Keep these records consistent with the guarded commands.
7. Document design-choice conventions where the specification
   is silent, rather than presenting them as quoted facts.
8. Return only the required JSON object, without RTL or prose.

<applicable design-specific rules and reference examples>
<complete input specification>
```

Listing 2: Condensed specification-to-ASTF refinement instructions. The complete request also includes applicable design-specific rules and reference examples.

The resulting document contains module and parameter declarations, two input declarations, and five ASTFs for `cnt1`, `cnt2`, `clk_div1`, `clk_div2`, and `clk_div`. Inputs contain declaration records only. For driven signals, `specs`, `hardware_type`, and `guarded_commands` instantiate D_s, H_s, and G_s, respectively. The four internal ASTFs each contain three commands, and the output ASTF contains one OR command, giving thirteen commands in total.

Listing 3 presents the complete audited document. Shaded keys distinguish its document structure, declarations, hardware attributes, and guarded commands.

Audited ASTF-Based IR

```
{
  "module_name": "freq_divbyodd",
  "parameters": [{"name": "NUM_DIV", "default": 5, "description": "Odd divisor value for
     frequency division"}],
  "input_signals": [{"specs": {"signal_name": "clk", "direction": "input", "data_type": "
     wire", "width": 1}}, {"specs": {"signal_name": "rst_n", "direction": "input", "
     data_type": "wire", "width": 1}}],
  "output_signals": [
    {
      "specs": {"signal_name": "clk_div", "direction": "output", "data_type": "wire", "
     width": 1},
      "hardware_type": {"logic_style": "combinational", "clock_domain": null, "
     reset_behavior": null},
      "guarded_commands": [
        {"condition": "true", "action": "clk_div = clk_div1 | clk_div2", "spec_citation": "
     The final divided clock output, clk_div, is derived by logically OR-ing clk_div1 and
     clk_div2 to account for both clock edges."}
      ]
    }
  ],
  "internal_signals": [
    {
      "specs": {"signal_name": "cnt1", "direction": "internal", "data_type": "reg", "width
     ": 32},
```

```
      "hardware_type": {"logic_style": "sequential", "clock_domain": {"clock": "clk", "edge": "rising"}, "reset_behavior": {"reset_signal": "rst_n", "reset_value": 0, "type": "asynchronous"}},
      "guarded_commands": [
        {"condition": "!rst_n", "action": "cnt1 <= 0", "spec_citation": "The active low reset signal rst_n initializes the counters and the clock divider outputs."},
        {"condition": "cnt1 == NUM_DIV - 1", "action": "cnt1 <= 0", "spec_citation": "Two counters, cnt1 and cnt2, are used for tracking the rising and falling edges of the clock. Each counter counts up to NUM_DIV - 1."},
        {"condition": "cnt1 != NUM_DIV - 1", "action": "cnt1 <= cnt1 + 1", "spec_citation": "Two counters, cnt1 and cnt2, are used for tracking the rising and falling edges of the clock. Each counter counts up to NUM_DIV - 1."}
      ]
    },
    {
      "specs": {"signal_name": "cnt2", "direction": "internal", "data_type": "reg", "width": 32},
      "hardware_type": {"logic_style": "sequential", "clock_domain": {"clock": "clk", "edge": "falling"}, "reset_behavior": {"reset_signal": "rst_n", "reset_value": 0, "type": "asynchronous"}},
      "guarded_commands": [
        {"condition": "!rst_n", "action": "cnt2 <= 0", "spec_citation": "The active low reset signal rst_n initializes the counters and the clock divider outputs."},
        {"condition": "cnt2 == NUM_DIV - 1", "action": "cnt2 <= 0", "spec_citation": "Two counters, cnt1 and cnt2, are used for tracking the rising and falling edges of the clock. Each counter counts up to NUM_DIV - 1."},
        {"condition": "cnt2 != NUM_DIV - 1", "action": "cnt2 <= cnt2 + 1", "spec_citation": "Two counters, cnt1 and cnt2, are used for tracking the rising and falling edges of the clock. Each counter counts up to NUM_DIV - 1."}
      ]
    },
    {
      "specs": {"signal_name": "clk_div1", "direction": "internal", "data_type": "reg", "width": 1},
      "hardware_type": {"logic_style": "sequential", "clock_domain": {"clock": "clk", "edge": "rising"}, "reset_behavior": {"reset_signal": "rst_n", "reset_value": 0, "type": "asynchronous"}},
      "guarded_commands": [
        {"condition": "!rst_n", "action": "clk_div1 <= 0", "spec_citation": "The active low reset signal rst_n initializes the counters and the clock divider outputs."},
        {"condition": "cnt1 < NUM_DIV / 2", "action": "clk_div1 <= 1", "spec_citation": "Two separate clock dividers, clk_div1 and clk_div2, are used for positive and negative edges of the clock, respectively. These are toggled when the counters reach half of NUM_DIV."},
        {"condition": "cnt1 >= NUM_DIV / 2", "action": "clk_div1 <= 0", "spec_citation": "Two separate clock dividers, clk_div1 and clk_div2, are used for positive and negative edges of the clock, respectively. These are toggled when the counters reach half of NUM_DIV."}
      ]
    },
    {
      "specs": {"signal_name": "clk_div2", "direction": "internal", "data_type": "reg", "width": 1},
      "hardware_type": {"logic_style": "sequential", "clock_domain": {"clock": "clk", "edge": "falling"}, "reset_behavior": {"reset_signal": "rst_n", "reset_value": 0, "type": "asynchronous"}},
      "guarded_commands": [
        {"condition": "!rst_n", "action": "clk_div2 <= 0", "spec_citation": "The active low reset signal rst_n initializes the counters and the clock divider outputs."},
        {"condition": "cnt2 < NUM_DIV / 2", "action": "clk_div2 <= 1", "spec_citation": "Two separate clock dividers, clk_div1 and clk_div2, are used for positive and negative edges of the clock, respectively. These are toggled when the counters reach half of NUM_DIV."},
        {"condition": "cnt2 >= NUM_DIV / 2", "action": "clk_div2 <= 0", "spec_citation": "Two separate clock dividers, clk_div1 and clk_div2, are used for positive and negative edges of the clock, respectively. These are toggled when the counters reach half of NUM_DIV."}
      ]
    }
  ],
  "boundary_conditions": [{"signal_name": "cnt1", "limit_value": "NUM_DIV - 1", "behavior": "When cnt1 reaches NUM_DIV-1, it wraps back to 0 on the next rising edge. clk_div1 is LOW at this point (cnt1 >= NUM_DIV/2)."}, {"signal_name": "cnt2", "limit_value": "NUM_DIV - 1", "behavior": "When cnt2 reaches NUM_DIV-1, it wraps back to 0 on the next falling edge. clk_div2 is LOW at this point (cnt2 >= NUM_DIV/2)."}, {"signal_name": "clk_div1", "limit_value": "NUM_DIV / 2", "behavior": "clk_div1 transitions from HIGH to LOW when cnt1 reaches NUM_DIV/2 (integer division). HIGH for first floor(NUM_DIV/2) counts, LOW for remaining ceil(NUM_DIV/2) counts."}, {"signal_name": "clk_div2", "limit_value": "NUM_DIV / 2", "behavior": "clk_div2 transitions from HIGH to LOW when cnt2 reaches NUM_DIV/2 (integer division). HIGH for first floor(NUM_DIV/2) counts, LOW
```

```
      for remaining ceil(NUM_DIV/2) counts. Half-period shifted from clk_div1 due to negedge
      clocking."}]
}
```

Listing 3: Complete audited ASTF-based IR. Shaded keys identify document structure (blue), declarations (green), hardware attributes (purple), and guarded commands (amber).

The ASTF for `cnt1` implements counting through `NUM_DIV-1` using a terminal-count guard and an increment guard. The reset command precedes both guards and therefore has priority. The hardware record selects the rising edge of `clk`. The boundary record specifies wraparound at the terminal count. Each guarded command retains the associated specification excerpt for auditing.

The IR records implementation choices where the specification is silent. Both counters have 32-bit declarations, and the four registers use active-low asynchronous reset. These attributes are explicit choices rather than facts established by the cited text. The two divider registers use their associated counters to select high or low values at the integer threshold `NUM_DIV/2`. The falling-edge path mirrors the rising-edge path, and the combinational output ORs the two divider signals. Four boundary records capture the counter and switching limits. Optional `port_order` metadata is absent.

Schema validation checks the candidate's required fields, types, and nesting. The five audit layers in Table 1 apply to the specification evidence and IR structures. In this example, specification grounding concerns citation traceability, coverage concerns extractable interface and behavioral requirements, and consistency concerns referenced identifiers and agreement between boundary records and guarded commands. RTL design-rule compliance checks applicable pre-RTL constraints. For `cnt1`, checks relate the counter citation, declared signal, terminal-count guard, and wraparound record. The symbolic state-encoding checks within state-transition integrity are inapplicable because this IR contains no symbolic FSM states. The first candidate passed schema validation and semantic auditing with zero reported errors and warnings. No audit-driven revision was required.

Audit findings are produced by programmatic checks rather than an LLM auditing prompt. When a candidate fails validation or auditing, the Refinement Module retains the original specification and generation instructions, appends the failed candidate, and supplies the diagnostic feedback in Listing 4. Each audit error identifies the check and its diagnostic message, with supporting details and check-specific repair guidance when available. The revised IR undergoes the same validation and auditing procedure within the retry budget.

Audit-Guided IR Repair Prompt

```
Attempt <n> failed. Fix ONLY the issues listed below.
Preserve all other signals, commands, and field values.
Return the complete corrected JSON.

## Schema validation errors
<validation errors, when present>

## Audit errors (must fix)
[<check identifier>] <diagnostic message>
  <reported citation, signal, state, reference, or guard>
  FIX: <check-specific repair guidance, when available>

## Audit warnings (fix if possible)
[<check identifier>] <warning message>

Return ONLY the corrected JSON.
Do not include prose or Markdown fences.
```

Listing 4: Condensed diagnostic-feedback template for IR repair. Placeholders denote programmatically generated findings. This request was not triggered in the worked execution.

The repair template requests a complete corrected document while limiting revisions to affected content, as described in Section 3.2. This conditional repair branch was not invoked for the divider.

A.3 RTL GENERATION

The Implementation Module translates the audited IR using the rules in Section 3.3 and examples of sequential and combinational logic. The request specifies the module name, parameter values, and port sequence `clk`, `rst_n`, and `clk_div`. Listing 5 gives the condensed prompt, with the complete IR represented by a placeholder. Static instructions and reference examples form the cacheable input.

The first RTL generation produced a complete candidate that passed the interface, language, and compilation checks described in Section 3.3. No RTL repair was required.

RTL-Generation Prompt

```
[Reusable translation instructions]
Translate the audited signal-level IR into Verilog.
1. Preserve the module name, port contract, and
   parameter names and values supplied by the IR.
2. Declare every output and internal signal using
   its width and, when present, memory depth.
3. Select clock edges and reset behavior from each
   signal's hardware_type. Apply initialization
   according to the target implementation policy.
4. Use nonblocking assignments for clocked state;
   use continuous or procedural combinational
   assignments for combinational signals.
5. Translate guarded commands in their given order.
   Preserve conditions, actions, and priority.
6. Preserve encoded boundary behavior. Do not
   reconstruct the algorithm from the task name.
7. Return only the complete Verilog source, without
   Markdown fences, testbench code, or prose.

[Design-specific request]
Generate the Verilog module for this IR
(type: sequential).
REQUIRED -- module name: "freq_divbyodd".
Port names must be exactly (in order):
clk, rst_n, clk_div.
Return ONLY the Verilog -- no Markdown, no prose.

<complete audited IR JSON>
```

Listing 5: Condensed RTL-generation prompt combining reusable translation instructions and the design-specific request. The IR placeholder denotes the complete document in Listing 3.

For `cnt1`, D_s produces the declaration `reg [31:0] cnt1`, and H_s specifies rising-edge clocking with active-low asynchronous reset. The ordered commands in G_s produce the reset, terminal-count, and increment branches. This mapping preserves the declaration, event control, and update priority encoded in the ASTF.

Listing 6 reproduces the saved RTL, including both 32-bit counter declarations and the four separate `initial` statements. These statements set the internal registers to zero at simulation initialization, in addition to the asynchronous reset behavior. They are implementation-policy choices, not initialization requirements established by the cited specification. This example does not establish their portability across synthesis targets.

The implementation contains four clocked processes and one continuous assignment. Each clocked process uses the edge specified by its hardware record and includes the active-low asynchronous reset in its event control. Reset is evaluated before normal updates, preserving guarded-command priority. The counter processes implement the terminal-count comparison against `NUM_DIV-1`, while the divider-register processes implement the threshold comparison against `NUM_DIV/2`. The output command maps directly to `assign clk_div = clk_div1 | clk_div2`.

Nonblocking assignments ensure that the clocked processes evaluate their conditions using the register values preceding the update. With `NUM_DIV`=5, integer division yields a switching threshold of two. If `cnt1`=1 immediately before a rising edge and reset is inactive, that edge schedules `cnt1`=2 and `clk_div1`=1. The following rising edge evaluates a count of two and assigns zero to `clk_div1`. The falling-edge path applies the same predicates to its own counter. Preserving both the clock edge and the counter predicate is therefore necessary to retain the timing encoded

by the ASTFs. The boundary descriptions in Listing 3 retain their original wording. The guarded commands specify the update behavior at each clock edge. In particular, reaching the threshold after an edge does not immediately change the divider register. Its next update uses the counter value observed before the corresponding rising or falling edge.

Generated Verilog

```
`timescale 1ns/1ns
module freq_divbyodd #(
    parameter NUM_DIV = 5
)(
    input  wire clk,
    input  wire rst_n,
    output wire clk_div
);
    reg [31:0] cnt1;
    reg [31:0] cnt2;
    reg        clk_div1;
    reg        clk_div2;
    initial cnt1     = 0;
    initial cnt2     = 0;
    initial clk_div1 = 0;
    initial clk_div2 = 0;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            cnt1 <= 32'd0;
        end else begin
            if (cnt1 == NUM_DIV - 1)
                cnt1 <= 32'd0;
            else
                cnt1 <= cnt1 + 32'd1;
        end
    end
    always @(negedge clk or negedge rst_n) begin
        if (!rst_n) begin
            cnt2 <= 32'd0;
        end else begin
            if (cnt2 == NUM_DIV - 1)
                cnt2 <= 32'd0;
            else
                cnt2 <= cnt2 + 32'd1;
        end
    end
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            clk_div1 <= 1'b0;
        end else begin
            if (cnt1 < NUM_DIV / 2)
                clk_div1 <= 1'b1;
            else
                clk_div1 <= 1'b0;
        end
    end
    always @(negedge clk or negedge rst_n) begin
        if (!rst_n) begin
            clk_div2 <= 1'b0;
        end else begin
            if (cnt2 < NUM_DIV / 2)
                clk_div2 <= 1'b1;
            else
                clk_div2 <= 1'b0;
        end
    end
    assign clk_div = clk_div1 | clk_div2;
endmodule
```

Listing 6: Saved generated Verilog with two 32-bit counters, four initialization statements, four clocked processes, and one continuous assignment.

A.4 Internal Testbench Generation

The example proceeds from specification-based planning to stimulus generation and harness assembly. The following listings illustrate this workflow through selected instructions from the saved generation requests and excerpts from their outputs.

Verification Plan Generation. The specification is supplied to the planning stage. Listing 7 reproduces selected planning instructions, and Listing 8 shows the resulting design description, interface, and one complete scenario record. Other scenario records and plan fields are omitted.

Verification-Plan Prompt

```
Selected planning instructions from the saved request.
Other instructions and the specification are omitted.

- Treat the specification as the **source of truth** for intended **correct** behavior.
- If the spec embeds a **buggy example** `TopModule`, plan tests for the **correct**
  behavior described by the task.
- Do **not** assume access to `RefModule`, `ref.sv`, or any reference RTL source.
- Do **not** assume access to an external reference testbench.
- Module under test is always **`TopModule`**.
- When an **RTL port declaration header** is provided below the spec, treat it as the
  authoritative source for **port names, directions, and bit widths**. Behavioral
  intent and test scenarios still come from the specification.
- When no RTL header is provided, derive all ports from the specification.
```

Listing 7: Selected planning instructions from the saved request.

Generated Verification Plan


```
{
  "design": {
    "name": "TopModule",
    "purpose": "Frequency divider that divides input clock by an odd number (default
     NUM_DIV=5) using dual-edge counters and OR-combined sub-clocks",
    "design_type": "sequential"
  },
  "interface": {
    "inputs": [
      {
        "name": "clk",
        "width": "1",
        "description": "Input clock signal"
      },
      {
        "name": "rst_n",
        "width": "1",
        "description": "Active-low synchronous/asynchronous reset"
      }
    ],
    "outputs": [
      {
        "name": "clk_div",
        "width": "1",
        "description": "Divided clock output (input frequency / NUM_DIV)"
      }
    ]
  },
  "test_scenarios": [
    {
      "id": "S02_division_ratio_measurement",
      "category": "deterministic",
      "condition": "rst_n released; DUT runs freely for at least 8 full output periods",
      "stimulus": "Release rst_n=1 after S01. Run clk for 80 input clock cycles (8 expected
       output periods of 10 cycles each). Record every rising and falling edge of clk_div
      using $display probes on every input clock edge.",
      "expected": "Each full period of clk_div (rising-to-rising) must span exactly 10
      input clock cycles (NUM_DIV=5, period=2*5=10). At least 6 complete periods must be
      observed within 80 cycles. No period deviation allowed.",
      "checks": [
        "$display clk_div on every input clock edge; post-process rising-edge timestamps to
       verify consecutive period = 10 input cycles.",
        "Report MISMATCH if any measured period differs from 10.",
        "Report MISMATCH if fewer than 6 complete periods are detected in 80 cycles."
      ],
      "linked_behaviors": [
        "B02_division_ratio",
        "B04_continuous_operation"
      ]
    }
  ]
}
```

Listing 8: Unmodified fields from the saved plan, including the complete division-ratio scenario.

The generated plan is not a correctness specification. In the displayed scenario, the expected period is incorrectly stated as ten input clock cycles for `NUM_DIV`=5. The stated division ratio instead requires five input clock cycles per output period. This inconsistency is retained to distinguish the saved model output from the intended requirement. The plan alone does not establish verification correctness or coverage.

Stimulus Generation. The specification and verification plan are supplied to the stimulus-generation stage. Listing 9 reproduces selected instructions governing the interface and scenario implementation. Listing 10 shows the generated interface and an observation-loop excerpt. The stimulus drives `rst_n` and observes `clk_div` under the harness clock. Seven scenarios cover reset sequences, steady operation, directed activity, and randomized reset stress. Scenario names identify generated test activity and do not establish assertion coverage.

Stimulus-Generation Prompt

```
Condensed property-mode generation instructions
1. Generate itb_stimulus from the specification and plan.
2. Receive clk and observed outputs. Drive DUT inputs.
3. Implement scenarios using tasks and loops. Preserve IDs.
4. Expose itb_accept_reset and itb_accept_ready.
5. Do not generate a waveform oracle or verdict logic.
6. Call itb_finish once after completing all scenarios.
7. The runner inserts the fixed-period checker.

<complete specification>
<complete verification plan>
```

Listing 9: Selected stimulus-generation instructions from the saved request.

Generated Stimulus

```
// Excerpts from the accepted stimulus. Intervening code omitted.
module itb_stimulus (
    input  wire clk,
    input  wire clk_div,
    output reg  rst_n
);

localparam integer NUM_DIV        = 5;
localparam integer CLK_DIV_RATIO  = 5;
localparam integer CLK_DIV_PERIOD = 10;

wire itb_accept_reset = (rst_n === 1'b0);
wire itb_accept_ready = 1'b1;


    for (i = 0; i < 80; i = i + 1) begin
        @(posedge clk);
        itb_trace_step = i;
        $display("S02 cycle=%0d rst_n=%b clk_div=%b", i, rst_n, clk_div);
        // Diagnostic-only edge logging omitted.
    end
    $display("SCENARIO_DONE:S02_division_ratio_measurement");
// Remaining scenarios omitted. After all scenarios:
    itb_finish;
```

Listing 10: Saved stimulus interface and observation-loop excerpts. Intervening code is omitted.

Harness Assembly and Simulation. The framework assembles the harness programmatically. This step does not require another LLM prompt. Listing 11 shows the saved connections between the stimulus and DUT. The harness instantiates `TopModule` without a reference module. These connections illustrate assembly rather than verdict generation. The benchmark simulation result is reported separately in Section A.5.

Assembled Harness

```
// Excerpt from the saved assembled harness.
itb_stimulus stim1 (
    .clk(clk),
    .rst_n(rst_n),
    .clk_div(clk_div)
  );
  TopModule dut1 (
    .clk(clk),
    .rst_n(rst_n),
    .clk_div(clk_div)
  );
```

Listing 11: Instance connections from the saved harness. Other harness code is omitted.

A.5 Simulation Results

The saved execution used one ASTF-based IR generation and one RTL generation, with the default parameter `NUM_DIV`=5. The IR passed schema validation and semantic auditing with zero errors and warnings, and the first RTL candidate passed the implementation checks. ITB generation required two repair requests before the final candidate was used for simulation. The rejected candidates contained syntax or compatibility defects, including unmatched block delimiters and an unsupported cast. These repairs concerned the ITB and did not constitute IR or RTL refinement iterations.

The generated RTL passed the separate benchmark-testbench simulation, which supplies the functional correctness verdict for this example. No simulation-guided IR or RTL repair was performed in the saved execution. This result applies to the benchmark test stimuli and the default parameter configuration and does not establish exhaustive functional equivalence or correctness across other parameter values.

A.6 Pseudocode for Simulation-Guided Debugging

Listing 12 presents the simulation-guided debugging procedure in Section 3.4.2 with the iteration limit in Section 4.1. The procedure permits at most two repair iterations after the initial candidate. The divider passed its first ITB simulation, so the listing describes the method without reporting an additional repair experiment.

The procedure takes the specification `spec`, audited `IR`, generated RTL, and validated testbench `ITB`. The IR includes the module metadata and signal-level ASTFs defined in Section 3.1. The ITB is generated and validated as described in Section 3.4.1.

The classifier receives the specification, compact ASTF summary, generated RTL, and compilation or simulation log. It identifies an ASTF-level or RTL-level issue and returns targeted repair hints. An ASTF-level issue returns to the Refinement Module for IR revision and auditing, followed by RTL generation. An RTL-level issue returns to the Implementation Module for repair under the current IR. Each repair iteration uses feedback from the latest candidate, as described in Section 4.3.

Simulation-Guided Debugging

```
Input: spec, audited IR, generated RTL, validated internal testbench ITB
       K = 2  // maximum simulation-guided repair iterations (Section 4.1)
Output: final RTL and benchmark verdict

for iteration = 0, ..., K:
    passed, log = CompileAndSimulate(RTL, ITB)
    if passed:
        break
    if iteration == K:
        break                          // repair budget exhausted

    issue, hints = ClassifyFailure(
        spec, CompactASTFSummary(IR), RTL, log)
    if issue == ASTF_LEVEL:
        IR = RefineAndAudit(spec, IR, hints)
        RTL = GenerateAndCheckRTL(IR)
    else if issue == RTL_LEVEL:
```

```
        RTL = RepairAndCheckRTL(IR, RTL, log, hints)
    // The next iteration evaluates the latest repaired candidate.

verdict = EvaluateBenchmark(RTL)       // final evaluation only
return RTL, verdict                   // verdict never guides repair
```

Listing 12: Schematic debugging procedure aligned with Sections 3.4.2 and 4.1. At most two simulation-guided repair iterations follow the initial candidate. Benchmark feedback is excluded from repair.

The refinement helper includes schema validation and the five semantic audit layers from Section 3.2. The implementation helpers include interface, language-rule, and compilation checks from Section 3.3. Their local retries, and ITB generation or validation retries, are separate from the simulation-guided repair iterations shown here. The listing abstracts local failure handling and assumes that each helper returns a candidate that passes its applicable checks.

Debugging terminates when the ITB passes or the repair budget is exhausted. The benchmark testbench then evaluates the final RTL according to the functional correctness criterion in Section 4.1. Its verdict is not used for ASTF refinement or RTL repair.

B Specification and Testbench Corrections for Failure Cases

This appendix supplements the failure-case analysis in Section 4.4 with the source-level corrections for five affected benchmark designs. We examine four specification defects and one testbench interface defect, describing the original input, the applied correction, and its effect on RTL generation or evaluation. Benchmark defects have also been documented by RTL-BenchMT, which reports manually confirmed flawed cases accounting for 4.3%–10.0% of tasks across its evaluated VerilogEval, RTLLM benchmarks.

Figures 3–7 show the original and revised source excerpts for the corrections underlying the reruns in Section 4.4. The figures preserve the source wording while normalizing whitespace and omitting unaffected context. For `Prob149`, both complete source tables are included to show the added output constraints in context. The following subsections explain why each edit changes the interface contract, the required behavior, or the validity of the benchmark evaluation.

B.1 Specification Modifications

The specification defines the information available for ASTF construction and specification-derived testing. Missing interface facts or conflicting behavioral requirements can therefore affect both generated RTL and its internal testbench. The following corrections make the benchmark's intended interface and behavior explicit. They do not change the VeriRefine generation procedure.

B.1.1 `RADIX2_DIV` Transaction Control and Arithmetic Encoding

The original specification describes an iterative divider but leaves result consumption and several implementation-dependent timing and encoding rules incomplete. Figure 3 organizes the corrections into five groups.

The correction adds the missing `res_ready` input and defines consumption as a clock edge with both `res_valid` and `res_ready` asserted. This establishes the handshake needed to distinguish an available result from an accepted result. The revised specification requires `result` to expose the completed quotient and remainder in the same cycle that `res_valid` is asserted. The value must remain stable while validity remains asserted. These requirements exclude an additional output-register delay that would present stale data during the valid cycle.

Operation initiation is restricted to `opn_valid` high, `res_valid` low, and `start_cnt` low. The added condition prevents an active division from being reinitialized by a continuing request. The revised specification defines a 17-bit `SR` and a 9-bit two's-complement `NEG_DIVISOR`. For a signed divisor of `8'hF6` (-10), the latter must be `9'h1F6`. Zero extension would instead represent a positive value and alter the subtraction.

The correction also specifies the final quotient assembly. Because each iteration shifts SR left and leaves its least significant bit zero, the accumulated quotient bits occupy SR[7:1]. The final quotient is therefore {SR[7:1], carry_out}, rather than an expression based on SR[6:0]. For result validity, reset clears res_valid, completion at cnt=8 sets it, and the result-consumption handshake clears it. Together, these corrections specify when an operation may start, how its arithmetic state is interpreted, and when its result can be accepted.

B.1.2 FREQ_DIVBYEVEN MODULE IDENTITY AND DEFAULT CONFIGURATION

This case contains two independent mismatches between the written specification and the benchmark assumptions, as shown in Fig. 4. The specification names the module freq_diveven, whereas the benchmark instantiates freq_divbyeven. The corrected identifier ensures that a module generated from the specification can be bound to the testbench instance.

The original text also requires an even division factor without specifying its default. Several parameter values satisfy that requirement, but they produce different output periods. The revised specification sets the default NUM_DIV to six, matching the configuration expected by the benchmark. The two edits resolve different failure mechanisms. The identifier determines elaboration compatibility, while the parameter default determines the waveform under default instantiation.

B.1.3 PROB149 DIRECTION-DEPENDENT FLOW CONTROL

The original supplemental-flow description associates opening dfr with a rise in water level. Its accompanying table specifies the nominal flow outputs but does not specify dfr. The revised specification removes this conflicting prose and adds an explicit supplemental-flow column, as shown in Fig. 5(a)–(c).

At the highest water level, dfr=0. Below the lowest sensor, dfr=1. At either intermediate level, dfr=1 when that level was reached by falling and zero when reached by rising. Consequently, the current sensor pattern alone does not determine the supplemental output. The controller must retain sufficient state to distinguish the direction from which it entered an intermediate level. The nominal-flow columns remain unchanged, making the added history-dependent requirement directly visible in the complete table comparison.

The revised source additionally limits state transitions to one adjacent water level per clock cycle, as shown in Fig. 5(d). The controller considers the sensors bordering its current level instead of jumping directly to a level decoded from all current sensor values. If multiple sensors change simultaneously, several cycles may therefore be required to reach the corresponding level. This constraint affects the transient output sequence as well as the final state, and is distinct from specifying the dfr values.

B.1.4 PROB153_GSHARE PREDICTOR INITIALIZATION

The original description defines 128 two-bit saturating counters but does not identify their reset encoding or explicitly select the prediction bit. Figure 6 shows the added requirement. Each counter resets to the weakly-not-taken state, encoded as one, and its upper bit supplies the predicted direction.

These details affect observable behavior even when the table indexing and update mechanism are otherwise implemented correctly. For example, counters initialized to zero and one both initially predict not taken when their upper bit is used. After one taken update, however, their values become one and two, respectively, yielding different predictions. Specifying the reset encoding therefore fixes the training response, rather than merely naming an initial output. The correction leaves the original table size and history-based indexing description intact.

B.2 TESTBENCH MODIFICATION

B.2.1 PROB099 OUTPUT-PORT BINDINGS

The specification and reference module expose outputs Y1 and Y3, but the original testbench connects to named ports Y2 and Y4. This mismatch prevents correct elaboration of a module that

radix2_div

RTLLM v2.0
COMPLETE SPECIFICATION

Original

```
1  Please act as a professional verilog designer.
2
3  Implement a simplified radix-2 divider on 8-bit signed or
   unsigned integers. and the inputs are two 8-bit operands.
   The module accepts a dividend and a divisor as inputs and
   provides a 16-bit result containing both the quotient and
   the remainder. The design supports both signed and
   unsigned division operations.
4
5  Module name:
6      radix2_div
7
8  Input ports:
9      clk: Clock signal used for synchronous operation.
10     rst: The reset signal to initialize or reset the
   module.
11     sign: 1-bit indicates if the operation is signed (1)
   or unsigned (0).
12     dividend: 8-bit input signal representing the dividend
   for division.
13     divisor: 8-bit input signal representing the divisor
   for division.
14     opn_valid: 1-bit indicates that a valid operation
   request is present.
15 Output ports:
16     res_valid: 1-bit output signal indicating the result
   is valid and ready.
17     result: 16-bit the output containing the remainder in
   the upper 8 bits and the quotient in the lower 8 bits.
18
19 Implementation:
20
21 Operation Start:
22 When opn_valid is high and res_valid is low, the module
   saves the inputs dividend and divisor.
23 Initializes the shift register SR with the absolute value
   of the dividend shifted left by one bit.
24 Sets NEG_DIVISOR to the negated absolute value of the
   divisor.
25 Sets the counter cnt to 1 and start_cnt to 1 to begin the
   division process.
26
27 Division Process(If start_cnt is high, the module performs
   the following steps):
28 If the counter cnt reaches 8 (most significant bit of cnt
   is set), the division is complete:
29 cnt and start_cnt are cleared.
30 Updates the shift register SR with the final remainder and
   quotient.
31 Otherwise, the counter cnt is incremented, and the shift
   register SR is updated based on the subtraction result:
32 Computes the subtraction of NEG_DIVISOR.
33 Uses a multiplexer to select the appropriate result based
   on the carry-out.
34 Updates SR by shifting left and inserting the carry-out.
35
36 Result Validity:
37 res_valid is managed based on the reset signal, the
   counter, and whether the result has been consumed.
38
39 Give me the complete code.
```

Modified

```
1  Please act as a professional verilog designer.
2
3  Implement a simplified radix-2 divider on 8-bit signed or
   unsigned integers. and the inputs are two 8-bit operands.
   The module accepts a dividend and a divisor as inputs and
   provides a 16-bit result containing both the quotient and
   the remainder. The design supports both signed and
   unsigned division operations.
4
5  Module name:
6      radix2_div
7
8  Input ports:
9      clk: Clock signal used for synchronous operation.
10     rst: The reset signal to initialize or reset the
   module.
11     sign: 1-bit indicates if the operation is signed (1)
   or unsigned (0).
12     dividend: 8-bit input signal representing the dividend
   for division.
13     divisor: 8-bit input signal representing the divisor
   for division.
14     opn_valid: 1-bit indicates that a valid operation
   request is present.
15     res_ready: 1-bit indicates that the receiver is ready
   to take the result. The result counts as consumed on a
   clock edge where res_valid and res_ready are both high.
16 Output ports:
17     res_valid: 1-bit output signal indicating the result
   is valid and ready.
18     result: 16-bit the output containing the remainder in
   the upper 8 bits and the quotient in the lower 8 bits. The
   final remainder and quotient appear on result in the same
   cycle in which SR is updated with them, that is, in the
   cycle res_valid is asserted; result must not lag SR by a
   cycle. The value is then held unchanged while res_valid
   stays high.
19
20 Implementation:
21
22 Operation Start:
23 When opn_valid is high, res_valid is low, and start_cnt is
   low, the module saves the inputs dividend and divisor.
24 Initializes the 17-bit shift register SR with the absolute
   value of the dividend shifted left by one bit.
25 Sets NEG_DIVISOR to the negated absolute value of the
   divisor, held as a 9-bit two's-complement value. For a
   signed negative divisor the negated absolute value is the
   divisor itself, so it must be SIGN-extended to 9 bits,
   never zero-extended: divisor = 8'hF6, which is -10, gives
   NEG_DIVISOR = 9'h1F6.
26 Sets the counter cnt to 1 and start_cnt to 1 to begin the
   division process.
27
28 Division Process(If start_cnt is high, the module performs
   the following steps):
29 If the counter cnt reaches 8 (most significant bit of cnt
   is set), the division is complete:
30 cnt and start_cnt are cleared.
31 Updates the shift register SR with the final remainder and
   quotient. Because every step shifts SR left and leaves a
   zero in SR[0], the quotient bits accumulate in SR[7:1],
   never in SR[6:0]; the final quotient is {SR[7:1],
   carry_out} and the final remainder is the selected
   subtraction result.
32 Otherwise, the counter cnt is incremented, and the shift
   register SR is updated based on the subtraction result:
33 Computes the subtraction of NEG_DIVISOR.
34 Uses a multiplexer to select the appropriate result based
   on the carry-out.
35 Updates SR by shifting left and inserting the carry-out.
36
37 Result Validity:
38 res_valid is cleared by the reset signal, is set when the
   counter cnt reaches 8, and is cleared again on a clock
   edge where res_valid and res_ready are both high, which is
   how the result is consumed. While res_valid remains high
   no new operation is started.
39
40 Give me the complete code.
```

Highlights: removed or replaced (left), added or replaced (right). Numbers denote source lines.

Figure 3: Specification corrections for `radix2_div` covering the result-consumption handshake, output timing and retention, operation-start gating and signed arithmetic, quotient assembly, and result-valid control.

freq_divbyeven

RTLLM v2.0
COMPLETE SPECIFICATION

Original

```
1  Please act as a professional verilog designer.
2
3  Frequency divider that divides the input clock frequency
   by even numbers. This module generates a divided clock
   signal by toggling its output every specified number of
   input clock cycles.
4
5  Module name:
6      freq_diveven
7
8  Input ports:
9      clk: Input clock signal that will be divided.
10     rst_n: Active-low reset signal to initialize the
   module.
11
12 Output ports:
13     clk_div: Divided clock output signal.
14
15 Implementation:
16     The frequency divider uses a counter (`cnt`) to count
   the number of clock cycles. The `NUM_DIV` parameter
   specifies the division factor, which must be an even
   number.
17     - When the reset signal (`rst_n`) is low, the counter
   and the divided clock signal (`clk_div`) are initialized
   to zero.
18     - On each positive edge of the input clock (`clk`), if
   the counter is less than half of `NUM_DIV - 1`, the
   counter increments without changing the divided clock
   output.
19     - When the counter reaches the specified limit, it
   resets to zero and toggles the `clk_div` output signal,
   effectively dividing the frequency of the input clock by
   the even number specified by `NUM_DIV`.
20
21 Counter:
22     - The counter is a 4-bit register (`cnt`) that tracks
   the number of clock cycles.
23
24 Give me the complete code.
```

Modified

```
1  Please act as a professional verilog designer.
2
3  Frequency divider that divides the input clock frequency
   by even numbers. This module generates a divided clock
   signal by toggling its output every specified number of
   input clock cycles.
4
5  Module name:
6      freq_divbyeven
7
8  Input ports:
9      clk: Input clock signal that will be divided.
10     rst_n: Active-low reset signal to initialize the
   module.
11
12 Output ports:
13     clk_div: Divided clock output signal.
14
15 Implementation:
16     The frequency divider uses a counter (`cnt`) to count
   the number of clock cycles. The `NUM_DIV` parameter
   specifies the division factor, which must be an even
   number and defaults to 6.
17     - When the reset signal (`rst_n`) is low, the counter
   and the divided clock signal (`clk_div`) are initialized
   to zero.
18     - On each positive edge of the input clock (`clk`), if
   the counter is less than half of `NUM_DIV - 1`, the
   counter increments without changing the divided clock
   output.
19     - When the counter reaches the specified limit, it
   resets to zero and toggles the `clk_div` output signal,
   effectively dividing the frequency of the input clock by
   the even number specified by `NUM_DIV`.
20
21 Counter:
22     - The counter is a 4-bit register (`cnt`) that tracks
   the number of clock cycles.
23
24 Give me the complete code.
```

Highlights: removed or replaced (left), added or replaced (right). Numbers denote source lines.

Figure 4: Specification corrections for `freq_divbyeven`. The module identifier is aligned with the benchmark interface, and the default division factor is specified as six.

Prob149_ece241_2013_q4

VerilogEval-Human v2
COMPLETE SPECIFICATION

Original

I would like you to implement a module named TopModule with the following interface. All input and output ports are one bit unless otherwise specified.

- input clk
- input reset
- input s (3 bits)
- output fr2
- output fr1
- output fr0
- output dfr

A large reservior of water serves several users. In order to keep the level of water succificently high, three sensors are placed vertically at 5-inch intervals. When the water level is above the highest sensor s[2], the input flow rate should be zero. When the level is below the lowest sensor s[0], the flow rate should be at maximum (both Nominal flow valve and Supplemental flow valve opened). The flow rate when the level is between the upper and lower sensors is determined by two factors: the water level and the level previous to the last sensor change. Each water level has a nominal flow rate associated with it as show in the table below. If the sensor change indicates that the previous level was lower than the current level, the flow rate should be increased by opening the Supplemental flow valve (controlled by dfr).

Water Level	Sensors Asserted	Nominal Flow Rate Inputs to be Asserted
Above s[2]	s[0], s[1], s[2]	None
Between s[2] and s[1]	s[0], s[1]	fr0
Between s[1] and s[0]	s[0]	fr0, fr1
Below s[0]	None	fr0, fr1, fr2

Also include an active-high synchronous reset that resets the state machine to a state equivalent to if the water level had been low for a long time (no sensors asserted, and all four outputs asserted).

Modified

I would like you to implement a module named TopModule with the following interface. All input and output ports are one bit unless otherwise specified.

- input clk
- input reset
- input s (3 bits)
- output fr2
- output fr1
- output fr0
- output dfr

A large reservior of water serves several users. In order to keep the level of water succificently high, three sensors are placed vertically at 5-inch intervals. When the water level is above the highest sensor s[2], the input flow rate should be zero. When the level is below the lowest sensor s[0], the flow rate should be at maximum (both Nominal flow valve and Supplemental flow valve opened). The flow rate when the level is between the upper and lower sensors is determined by two factors: the water level and the level previous to the last sensor change. Each water level has a nominal flow rate associated with it, and a value for the Supplemental flow valve (controlled by dfr), as show in the table below.

Water Level	Sensors Asserted	Nominal Flow Rate Inputs to be Asserted	dfr
Above s[2]	s[0], s[1], s[2]	None	0
Between s[2] and s[1]	s[0], s[1]	fr0	1 if this level was reached by falling, 0 if by rising
Between s[1] and s[0]	s[0]	fr0, fr1	1 if this level was reached by falling, 0 if by rising
Below s[0]	None	fr0, fr1, fr2	1

The state machine moves at most one level per clock cycle: from its current level it looks only at the sensor bordering the level above, rising one level when that sensor is asserted, and at the sensor bordering the level below, falling one level when that sensor is deasserted. Levels are never skipped, so when more than one sensor changes at once it takes several cycles to settle.

Also include an active-high synchronous reset that resets the state machine to a state equivalent to if the water level had been low for a long time (no sensors asserted, and all four outputs asserted).

Highlights: removed or replaced (left), added or replaced (right). Numbers denote source lines.

Figure 5: Specification corrections for `Prob149`. The supplemental-flow description is replaced with explicit `dfr` values. Panels (b) and (c) reproduce the complete original and revised tables, respectively. The added `dfr` column is highlighted. Panel (d) shows the additional one-level-per-cycle constraint in the revised source.

Prob153_gshare

VerilogEval-Human v2
SPECIFICATION EXCERPT

Original

```
1
2  I would like you to implement a module named TopModule
   with the following
3  interface. All input and output ports are one bit unless
   otherwise
4  specified.
5
6   - input  clk
7   - input  areset
8
9   - input  predict_valid,
10  - input  predict_pc (7 bits)
11  - output predict_taken
12  - output predict_history (7 bits)
13
14  - input  train_valid
15  - input  train_taken
16  - input  train_mispredicted
17  - input  train_history (7 bits)
18  - input  train_pc (7 bits)
19
20 The module should implement a gshare branch predictor with
   7-bit pc and
21 7-bit global history, hashed (using xor) into a 7-bit
   index. This index
22 accesses a 128-entry table of two-bit saturating counters.
   The branch
23 predictor should contain a 7-bit global branch history
   register. The
24 branch predictor has two sets of interfaces: One for doing
   predictions
25 and one for doing training. The prediction interface is
   used in the
26 processor's Fetch stage to ask the branch predictor for
   branch direction
27 predictions for the instructions being fetched. Once these
   branches
28 proceed down the pipeline and are executed, the true
   outcomes of the
29 branches become known. The branch predictor is then
   trained using the
30 actual branch direction outcomes.
```

Modified

```
1
2  I would like you to implement a module named TopModule
   with the following
3  interface. All input and output ports are one bit unless
   otherwise
4  specified.
5
6   - input  clk
7   - input  areset
8
9   - input  predict_valid,
10  - input  predict_pc (7 bits)
11  - output predict_taken
12  - output predict_history (7 bits)
13
14  - input  train_valid
15  - input  train_taken
16  - input  train_mispredicted
17  - input  train_history (7 bits)
18  - input  train_pc (7 bits)
19
20 The module should implement a gshare branch predictor with
   7-bit pc and
21 7-bit global history, hashed (using xor) into a 7-bit
   index. This index
22 accesses a 128-entry table of two-bit saturating counters,
   each reset to the
23 weakly-not-taken state, which is counter value 1; the
   counter's upper bit is
24 the predicted direction. The branch
25 predictor should contain a 7-bit global branch history
   register. The
26 branch predictor has two sets of interfaces: One for doing
   predictions
27 and one for doing training. The prediction interface is
   used in the
28 processor's Fetch stage to ask the branch predictor for
   branch direction
29 predictions for the instructions being fetched. Once these
   branches
30 proceed down the pipeline and are executed, the true
   outcomes of the
31 branches become known. The branch predictor is then
   trained using the
32 actual branch direction outcomes.
```

Highlights mark changes. Unchanged later sections omitted. Numbers denote source lines.

Figure 6: Specification correction for `Prob153_gshare`. Each two-bit counter resets to the weakly-not-taken encoding, and its upper bit determines the prediction.

follows the stated interface. It arises in the testbench connections before output behavior can be compared.

As shown in Fig. 7, we change the reference-instance bindings to `.Y1(Y2_ref)` and `.Y3(Y4_ref)`. We apply the corresponding changes to the DUT instance, using `.Y1(Y2_dut)` and `.Y3(Y4_dut)`. In each named association, the identifier before the parentheses is the module port. The identifier inside is a testbench signal. Retaining the existing signal names therefore preserves the downstream comparison logic while connecting it to the intended outputs. The stimulus, statistics collection, and timeout mechanism remain unchanged.

Prob099_m2014_q6c

VerilogEval-Human v2
TESTBENCH CONNECTIONS

Original

```
63    wire tb_match;      // Verification
64    wire tb_mismatch = ~tb_match;
65
66    stimulus_gen stim1 (
67        .clk,
68        .* ,
69        .y,
70        .w );
71    RefModule good1 (
72        .y,
73        .w,
74        .Y2(Y2_ref),
75        .Y4(Y4_ref) );
76
77    TopModule top_module1 (
78        .y,
79        .w,
80        .Y2(Y2_dut),
81        .Y4(Y4_dut) );
```

Modified

```
63    wire tb_match;      // Verification
64    wire tb_mismatch = ~tb_match;
65
66    stimulus_gen stim1 (
67        .clk,
68        .* ,
69        .y,
70        .w );
71    RefModule good1 (
72        .y,
73        .w,
74        .Y1(Y2_ref),
75        .Y3(Y4_ref) );
76
77    TopModule top_module1 (
78        .y,
79        .w,
80        .Y1(Y2_dut),
81        .Y3(Y4_dut) );
```

Highlights mark changes. Only the complete connection block is shown. Numbers denote source lines.

Figure 7: Testbench correction for `Prob099`. Both instances bind the comparison wires to ports `Y1` and `Y3` instead of `Y2` and `Y4`. The comparison-wire names are retained. The remaining testbench code is unchanged.