FVDebug: An LLM-Driven Debugging Assistant for Automated Root Cause Analysis of Formal Verification Failures

Yunsheng Bai, Ghaith Bany Hamad, Chia-Tung Ho, Syed Suhaib, Haoxing Ren
NVIDIA
{yunshengb, gbanyhamad, chiatungh, ssuhaib, haoxingr}@nvidia.com

***Abstract*—Debugging formal verification (FV) failures represents one of the most time-consuming bottlenecks in modern hardware design workflows. When properties fail, engineers must manually trace through complex counter-examples spanning multiple cycles, analyze waveforms, and cross-reference design specifications to identify root causes—a process that can consume hours or days per bug. Existing solutions are largely limited to manual waveform viewers or simple automated tools that cannot reason about the complex interplay between design intent and implementation logic. We present FVDebug, an intelligent system that automates root-cause analysis by combining multiple data sources—waveforms, RTL code, design specifications—to transform failure traces into actionable insights. Our approach features a novel pipeline: (1) *Causal Graph Synthesis* that structures failure traces into directed acyclic graphs, (2) *Graph Scanner* using batched Large Language Model (LLM) analysis with for-and-against prompting to identify suspicious nodes, and (3) *Insight Rover* leveraging agentic narrative exploration to generate high-level causal explanations. FVDebug further provides concrete RTL fixes through its *Fix Generator*. Evaluated on open benchmarks, FVDebug attains high hypothesis quality and strong Pass@k fix rates. We further report results on two proprietary, production-scale FV counterexamples. These results demonstrate FVDebug's applicability from academic benchmarks to industrial designs.**

I. INTRODUCTION

Formal Verification (FV) is a cornerstone of modern VLSI design, using mathematical methods to prove that hardware designs adhere to their specifications [11, 12]. These specifications are captured through formal properties—often written as SystemVerilog Assertions (SVAs) [30]—that express expected design behaviors, such as "a request must be acknowledged within 3 cycles" or "the FIFO should never overflow." When a design violates such a property, industry-standard model checkers like Jasper [28] and VC Formal [26] generate a Counter-Example (CEX): a concrete execution trace showing cycle-by-cycle signal values that demonstrate the violation. This CEX, typically visualized as a waveform, provides an explicit failure scenario revealing where expected and actual behavior diverge.

However, deciphering a CEX is a notoriously manual and time-consuming task, with debugging consuming nearly 50% of verification engineers' time—their single largest activity [9, 19]. Engineers must trace signal dependencies backward through waveforms, cross-reference RTL logic, consult specifications, and synthesize this information to identify root causes—a process requiring deep understanding of design intent and implementation [17] that can stall development for hours or days. Commercial tools excel at waveform visualization [27, 28] but offer little automated reasoning; the burden of causal analysis remains manual.

Recent LLM-based approaches, while promising, have not yet captured the nuances of this workflow. Many are adapted from software debugging and fail to address hardware-specific concepts like cycle-accurate timing [6, 20, 25], while others rely on simplistic prompting strategies on raw trace files [14, 15]. In our empirical observations, these methods often produce false positives—flagging benign behaviors as suspicious—or miss true root causes by failing to trace multi-cycle causal chains.

We hypothesize that an effective automated debugger must emulate the structured, multi-source reasoning process of a human expert. Instead of treating the CEX as a flat sequence of events, it must first be structured into a representation that captures causality explicitly. We propose building a **Causal Graph** from the failure trace, where nodes represent signal events ("signal@cycle=value") and directed edges represent their immediate causal dependencies. This structured "mental model" of the failure enables systematic tracing of failure chains—mirroring how engineers mentally traverse waveforms backward to identify root causes.

We introduce **FVDebug**, the first *end-to-end* automated system that (i) builds such a causal mental model and (ii) exploits it through an LLM pipeline deliberately mirroring how verification engineers work:

1. **Graph Scanner** acts like an engineer's quick "sanity sweep," scanning every level of the causal graph. Through a context retriever that dynamically fetches relevant RTL code snippets and specification excerpts, the scanner evaluates each signal's behavior against both its implementation and intended functionality. A novel *for-and-against*

prompting scheme compels balanced evaluation—forcing the LLM to weigh evidence on both sides before flagging suspicious behavior.

2. **Insight Rover** plays the role of the engineer's deep dive: using the suspicious nodes from the scanner as initial seeds, it begins an agentic search of the causal graph. At each step, the LLM is presented with candidate neighboring nodes and autonomously selects which paths to pursue based on their relevance to forming coherent failure hypotheses. It generates and iteratively refines multiple competing hypotheses, using the context retriever to back them with cycle-accurate evidence and assigning confidence scores to converge on the most plausible root cause.

3. **Fix Generator & Report** synthesizes concrete RTL patches and produces comprehensive human-readable reports containing ranked hypotheses, causal timelines, and diff-ready code suggestions. This structured output report enables verification engineers and RTL designers to collaborate effectively with a shared, precise understanding of the failure mechanism, replacing the fragmented debugging discussions currently scattered across communication channels or email threads.

The main contributions of this paper are:

- FVDebug is, to our knowledge, the **first realistic debugger** that automates the *entire* FV debug loop—from CEX to validated patch—while closely mimicking industrial engineering practice.
- We introduce novel techniques for FV debugging including causal graph synthesis from counter-examples, for-and-against prompting for balanced signal analysis, and agentic narrative exploration that emulates how human engineers progressively refine hypotheses.
- We develop a complete pipeline from failure trace to human-readable reports containing ranked root cause hypotheses, supporting evidence, causal chain timelines, and concrete RTL fixes—enabling effective collaboration between verification engineers and RTL designers.
- On 38 real hardware failures, FVDebug achieves 95.6 % hypothesis quality for root cause identification, 71.1 % *Pass@1* and 86.8 % *Pass@5* fix rates. We also showcase FVDebug on failures found in real-world larger designs.

II. Related Work

A. *LLM/AI-Driven Debugging* without *Waveforms*

Static repair pipelines such as LLM-HDL leverage retrieval-augmented prompts to locate and patch functional RTL bugs [23]. VeriDebug couples contrastive embeddings with generative edits to unify localisation and fixing [32]. Beyond RTL, HLSDebugger adapts encoder–decoder models to high-level synthesis code, substantially improving logic-bug repair [31]. While effective on code-visible faults, these approaches [10, 13, 24, 34] lack temporal reasoning and cannot analyse failures that only manifest in execution traces.

B. *LLM/AI-Driven Debugging* with *Waveforms*

Trace-centric methods incorporate assertion failures or counter-examples directly into LLM prompts. AssertSolver learns from contrasting "right vs. wrong" traces to diagnose simulation-time assertion failures [36]. GenAI-Induction proposes helper invariants that unblock formal k-induction [14]. The multi-agent framework Saarthi iteratively proves properties and analyses failing CEXs in a closed loop [15]. These techniques typically still treat waveforms as flat text, limiting root-cause fidelity on multi-cycle failures; our work tackles this via explicit causal graphs.

C. *Commercial and Industrial Debug Solutions*

Mainstream EDA platforms—Cadence *Indago*, Synopsys *Verdi*, Cadence *JasperGold*—provide rich waveform viewers and coverage dashboards but leave causal reasoning to engineers [27–29]. Several start-ups now advertise AI-driven RTL debug, while general-purpose coding copilots showcase agentic software-debug workflows [1–3, 8]. Technical details remain scarce, and adapting these systems to waveform-centric hardware failures is non-trivial—highlighting the need for deeper, graph-structured research.

III. Methodology

A. *Problem Setup and System Overview*

When a formal property fails verification, model checkers like Jasper produce counter-examples showing signal values over time that violate the property. However, these traces present signals as flat sequences, obscuring the causal

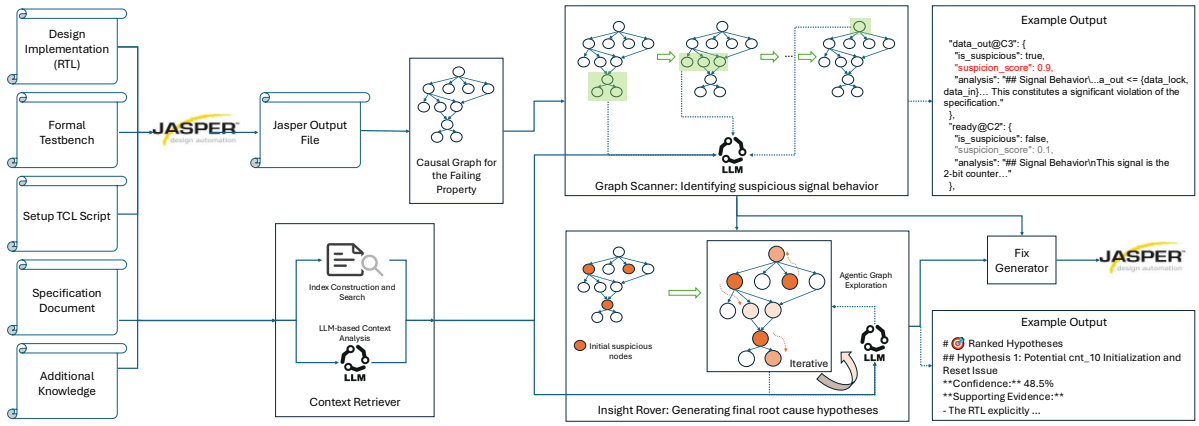


Figure 1: Overview of FVDebug. The system transforms formal verification counter-examples into actionable debugging insights through four stages: (1) Causal Graph Synthesis builds a directed acyclic graph capturing signal dependencies, (2) Graph Scanner performs efficient batched analysis to identify suspicious nodes, (3) Insight Rover explores competing hypotheses through intelligent graph navigation, and (4) Fix Generator produces concrete RTL patches and debugging reports. "C" denotes cycle in the example output of Graph Scanner.

relationships that explain why the failure occurred. Engineers must manually trace through waveforms to identify root causes—a process that can take hours or days per bug.

FVDebug automates this debugging process by transforming unstructured counter-example traces into structured causal graphs and systematically analyzing them to identify root causes. Given a failing property, RTL design, and counter-example trace, our goal is to automatically identify the root cause and generate fixes that make the design satisfy the property. Figure 1 illustrates our four-stage pipeline.

B. Causal Graph Synthesis

The foundation of our approach is transforming the counter-example trace into a causal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where nodes $\mathcal{V}$ represent signal events (signal, cycle, value) and edges $\mathcal{E}$ represent causal dependencies. This explicit structure enables systematic analysis that would be intractable on flat waveforms.

B.1 Graph Construction

We construct the causal graph $\mathcal{G}$ through recursive dependency analysis using JasperGold's built-in capabilities. Starting from the failing property at the violation cycle, we recursively query Jasper's `visualize -why` command to identify which signals caused each event. For example, querying why `ready_add@5=0` might reveal that `valid_out@5=0` and `valid_in@5=1` caused this value, establishing edges from these parent events.

We parameterize the construction with a trace depth (default: 20 cycles) that controls how far back we trace dependencies. This balances completeness against computational cost, as industrial designs can have very deep causal chains spanning hundreds of cycles.

B.2 Graph Consolidation

The recursive construction initially produces a tree where reconverging paths create duplicate nodes for the same signal-cycle pair. We consolidate these into a directed acyclic graph (DAG) by merging duplicates, ensuring each unique event appears exactly once. This transformation is critical—it prevents redundant analysis of the same events while preserving all causal paths. The resulting DAG typically contains far fewer nodes than the original tree, making subsequent analysis tractable.

C. Graph Scanner

The Graph Scanner systematically evaluates each node in $\mathcal{G}$ to identify suspicious signal behaviors. Inspired by constitutional AI [4] and self-critique mechanisms [18], we introduce *for-and-against prompting* that enforces balanced reasoning. This requires the LLM to argue both sides before reaching conclusions, preventing confirmation bias where the LLM might flag all behaviors as problematic or miss subtle issues.

Algorithm 1 Graph Scanner with Token-Aware Batching

```
Input: Causal graph 𝒢, scenario description, max_tokens
Output: Suspicious nodes, node analyses
for level in TopologicalLevels(𝒢) do
    remaining ← level
    while remaining not empty do
        batch_size ← BinarySearchMaxBatch(remaining, max_tokens)
        batch ← remaining[1:batch_size]
        prompt ← BuildPromptWithForAgainst(batch, context_cache)
        response ← LLM.Generate(prompt)
        for node in batch do
            analysis ← ParseAnalysis(response, node)
            all_analyses.append(analysis)
            if analysis.is_suspicious then
                suspicious_nodes.append(node)
            end if
        end for
        remaining ← remaining[batch_size+1:]
    end while
end for
return suspicious_nodes, all_analyses
```

C.1 Context Pre-fetching

Before analysis begins, we pre-fetch all necessary context to avoid redundant retrievals. We extract unique signal names from $\mathcal{G}$ and pre-cache their RTL code snippets, specification excerpts, and design documentation. This context cache is reused throughout the analysis.

C.2 For-and-Against Prompting

For each node, our balanced analysis technique requires the LLM to provide arguments both FOR and AGAINST the signal behavior being suspicious. The LLM outputs signal behavior analysis, arguments for/against suspicion, a balanced conclusion with suspicion score (0.0-1.0), classification as root cause vs symptom, and suggested fixes if applicable.

C.3 Token-Aware Batched Analysis

The core challenge is analyzing potentially thousands of nodes efficiently. Algorithm 1 shows our dynamic batching approach that maximizes throughput while respecting token constraints. The scanner processes nodes level-by-level in topological order, using *BinarySearchMaxBatch* to find the maximum batch size that fits within token limits.

D. Insight Rover

While the Graph Scanner identifies individual suspicious signals, the Insight Rover transforms these into coherent failure narratives. Motivated by tree-of-thought reasoning [35] and multi-agent debate [16], we maintain multiple competing hypotheses $\mathcal{H}$ and use the LLM as an autonomous agent to explore the most promising paths through $\mathcal{G}$.

D.1 Hypothesis Management

Each hypothesis $h \in \mathcal{H}$ represents a potential explanation for the failure. We initialize $\mathcal{H}$ from all suspicious nodes identified by the Graph Scanner, ensuring comprehensive coverage of potential root causes. The system dynamically adjusts to accommodate all suspicious nodes. Each h maintains a narrative description, chronological timeline, evidence collections, confidence score, and frontier of unexplored nodes.

This adaptive approach prevents premature pruning of potentially critical failure paths. While we configure a minimum of 3 narratives, the system expands as needed—for instance. During exploration, weak narratives (confidence < 0.2 after three iterations) are marked but retained for final reporting.

Algorithm 2 Insight Rover: Agentic Hypothesis Exploration

Input: Suspicious nodes, causal graph $\mathcal{G}$, LLM
Output: Ranked hypotheses $\mathcal{H}$ with evidence
// Create hypothesis for each suspicious node
for node **in** suspicious_nodes **do**
    theory ← LLM.*GenerateTheory*(node, $\mathcal{G}$)
    $\mathcal{H}$.add(*CreateHypothesis*(theory, node))
end for
for iteration = 1 **to** max_iterations **do**
    active ← [h for h in $\mathcal{H}$ if h.confidence < 0.9]
  if len(active) = 0 **then**
        break
  end if
  for h **in** active **do**
        next ← LLM.*SelectNode*(h.context, h.frontier)
        analysis ← LLM.*Analyze*(next, h)
        h.*UpdateFromAnalysis*(analysis)
        h.frontier.expand(next.neighbors)
  end for
    ManageNarrativePool() // Record weak, spawn new if unexplored
  if *CheckConvergence*() **then**
        break
  end if
end for
scores ← LLM.*EvaluateAll*($\mathcal{H}$)
return *SortByScore*($\mathcal{H}$, scores)

D.2 Intelligent Exploration

Rather than exhaustively exploring all paths, the LLM examines each hypothesis's current state and frontier, then selects which nodes would best validate or refute that theory. Algorithm 2 shows how the LLM uses *SelectNode* to choose frontier nodes, accumulates evidence through *UpdateFromAnalysis*, and manages the narrative pool until convergence. This selective exploration reduces the search space by over 90% compared to breadth-first search.

D.3 Final Ranking

After exploration, we perform holistic ranking using LLM.*EvaluateAll*($\mathcal{H}$) that considers sufficiency, evidence quality, mechanistic clarity, actionability, and narrative coherence. The function *SortByScore* orders $\mathcal{H}$ by these scores, with top-ranked hypotheses becoming the basis for generating fixes.

E. Fix Generator with Ensemble Strategies

The Fix Generator translates root cause hypotheses into concrete RTL patches. Inspired by best-of-N sampling [7] and self-consistency [33],we use an ensemble approach that generates fixes through multiple prompting strategies (full context, suspicious focus, narrative focus, minimal context, bugs-and-suggestions-only), followed by a best-of meta-strategy that reviews all generated fixes to select and refine the most promising solutions. This diversity ensures robustness—if one strategy misinterprets the context, others can compensate.

E.1 Validation and Consensus Ranking

A critical challenge is ensuring generated fixes are applicable to the actual RTL code map $\mathcal{R}$. We implement multi-level validation through *ValidateFix* that checks exact substring matching, handles whitespace variations, and aligns structural patterns. Algorithm 3 shows how each strategy generates fixes independently, validates them against $\mathcal{R}$, and merges duplicates identified by *CreateSignature*. The consensus information from multiple strategies becomes a ranking signal—fixes generated by more strategies receive higher confidence scores.

Algorithm 3 Ensemble Fix Generation

```
Input: Context, RTL code map R
Output: Validated RTL fixes
unique_fixes ← {}
for strategy in strategies do
    prompt ← BuildStrategyPrompt(strategy, context)
    fixes ← ParseFixes(LLM.Generate(prompt))
  for fix in fixes do
    if ValidateFix(fix, R).is_valid then
        sig ← CreateSignature(fix)
        unique_fixes[sig] ← Merge(unique_fixes[sig], fix)
    end if
  end for
end for
return RankByConsensus(unique_fixes.values())
```

E.2 Report Generation

Beyond code fixes, FVDebug produces structured debugging reports containing a report with the most likely root cause, ranked hypotheses from $\mathcal{H}$ with supporting evidence, unified causal timeline, concrete RTL fixes with validation status, and specification cross-references. This comprehensive output enables effective collaboration between verification engineers and RTL designers.

IV. EXPERIMENTS

We evaluate FVDebug on the SVA-Eval-Human benchmark [36], a curated collection of 38 real hardware debugging challenges with human-verified ground truth fixes. Our implementation interfaces Python with Cadence Jasper 2023.12 through TCL commands. We compare FVDebug against several baselines and ablated versions to demonstrate the effectiveness of our structured approach and understand each component's contribution. In addition, we evaluate FVDebug on two challenging CVA6 RISC-V processor failures originally discovered in AutoSVA [22], and we also report results on proprietary, production-scale FV counterexamples (design details masked).

A. Experimental Setup

The SVA-Eval-Human benchmark comprises diverse hardware designs with formal verification failures, ranging from simple arithmetic units to complex system-level modules. Each design includes a counter-example trace, the buggy RTL code, and the ground truth fix verified by human experts. We refer readers to [36] for detailed design descriptions and failure characteristics.

A.1 Baselines and Ablations

We evaluate the following methods, using o3-mini [21] as the underlying language model for all approaches:

Unstructured Baselines: (1) **Direct LLM**: Provides the counter-example trace, RTL code, and failing property directly to the LLM, asking it to identify the root cause and generate fixes. This represents the naive approach without any structural analysis or specialized prompting. (2) **Flat Trace Analysis**: Parses the counter-example into a structured chronological format, presenting signals and their values over time, but without constructing the causal graph. The LLM analyzes this temporal sequence to identify issues. This baseline isolates the value of causal structure versus mere temporal organization.

Component Ablations: (1) **FVDebug w/o For-Against**: Replaces our balanced evaluation prompting with standard bug-finding prompts that only ask for suspicious behaviors without requiring counterarguments. This ablation measures the impact of enforced analytical balance on false positive reduction. (2) **FVDebug w/o Rover**: Uses the Graph Scanner to identify suspicious nodes but skips the Insight Rover's narrative construction phase. Fixes are generated directly from the scanner's suspicious node analysis without exploring causal narratives. This tests whether sophisticated narrative exploration is necessary beyond initial suspicion identification. (3) **FVDebug w/o Ensemble**: Leverages only a single fix generation strategy (specifically, the full_context strategy) rather than our ensemble approach with multiple prompting perspectives. This ablation quantifies the benefit of diverse fix generation strategies.

Table 1: Comparison of FVDebug against baselines and ablations on the SVA-Eval-Human benchmark. Best results in **bold**.

	Root Cause Hypothesis Quality				Fix Generation	
Method	**Quality@Best**	**NDCG@5**	**MRR**	**Kendall's τ**	**Pass@1**	**Pass@5**
Unstructured Baselines						
Direct LLM	0.783	0.981	0.806	0.526	0.605	0.658
Flat Trace Analysis	0.474	0.948	0.804	0.511	0.632	0.816
Component Ablations						
FVDebug w/o For-Against	0.953	0.969	0.817	0.634	0.632	**0.868**
FVDebug w/o Rover	0.795	0.844	0.567	-0.176	0.643	0.821
FVDebug w/o Ensemble	**0.956**	**0.983**	**0.858**	**0.708**	0.684	0.763
FVDebug (Full)	**0.956**	**0.983**	**0.858**	**0.708**	**0.711**	**0.868**

A.2 Evaluation Metrics

Our evaluation uses two complementary assessment strategies. First, we assess the quality of generated root cause hypotheses through LLM-based evaluation metrics that compare hypotheses against ground truth: (1) **Quality@Best**: Evaluates the absolute quality of the best hypothesis generated, regardless of ranking. (2) **NDCG@5**: Measures ranking quality by comparing hypothesis scores against ground truth relevance, with higher-ranked correct hypotheses contributing more. (3) **MRR**: Captures the average reciprocal rank at which the first relevant hypothesis appears. (4) **Kendall's τ**: Quantifies correlation between predicted rankings and ground truth quality.

Second, we measure functional correctness through Pass@k metrics, where proposed fixes are validated by applying them to the RTL and re-running formal verification in Jasper: (1) **Pass@1**: Measures first-attempt success rate for generated fixes. (2) **Pass@5**: Captures whether any of the top five fix suggestions resolves the failure.

These complementary metrics address different aspects of debugging effectiveness—hypothesis quality metrics capture whether the system correctly identifies and explains root causes, while Pass@k metrics verify that this understanding translates into working fixes. Notably, high hypothesis quality often correlates with fix success, as accurate root cause identification is essential for generating correct patches.

B. Main Results

Table 1 presents comprehensive evaluation results across all methods. FVDebug achieves the highest Quality@Best score of 0.956, demonstrating superior root cause identification, while achieving the best fix generation performance with 71.1% Pass@1 and 86.8% Pass@5 rates.

C. Analysis of Results

The results reveal three critical insights. First, **causal structure is essential**: Flat Trace Analysis achieves only 0.474 Quality@Best despite temporal organization, while causal graph-based approaches achieve 0.795-0.956, demonstrating that explicit causal relationships are fundamental to accurate root cause identification. Second, **narrative construction drives performance**: removing the Insight Rover causes dramatic degradation, confirming that connecting suspicious nodes into coherent causal chains is crucial. Third, **ensemble strategies improve robustness**: while single-strategy generation maintains hypothesis quality (0.956 Quality@Best), it achieves lower Pass@5 (0.763 vs 0.868).

Notably, the for-against prompting ablation achieves high individual hypothesis quality (0.953) but lower ranking correlation (Kendall's τ: 0.634 vs 0.708) and Pass@1 (0.632 vs 0.711), suggesting that balanced evaluation improves consistency and translates to better fix generation. Overall, FVDebug's combination of causal graphs, narrative exploration, and ensemble strategies produces both accurate understanding and practical fixes.

D. Evaluation on Complex Processor Designs

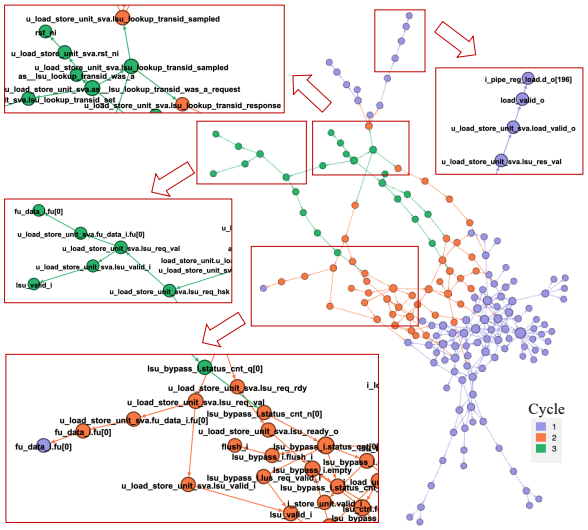
To further assess FVDebug's capabilities on realistic hardware complexity, we evaluate on two challenging FV failures from the CVA6 RISC-V processor [22]. These failures, originally discovered in the AutoSVA project[1], represent real verification challenges encountered in production-grade processor designs:

- **MMU Ghost Response**: A memory management unit (MMU) issue where misaligned requests trigger duplicate

[1] https://github.com/PrincetonUniversity/AutoSVA

```
as__lsu_lookup_transid_was_a_request (C:3, V:FAIL)
|-- lsu_lookup_transid_response (C:3, V:1'b1)
|   |-- lsu_res_hsk (C:3, V:1'b1)
|   |   `-- load_valid_o (C:3, V:1'b1)
|   |       `-- i_pipe_reg_load.d_i[196] (C:2, V:1'b1)
|   |           |-- i_load_unit.ex_i.valid (C:2, V:1'b1)
|   |           |   |-- i_mmu.misaligned_ex_q.valid (C:2, V:1'b1)
|   |           |   |   `-- i_mmu.misaligned_ex_n.valid (C:1, V:1'b1)
|   |           |   |       |-- i_mmu.lsu_req_i (C:1, V:1'b1)
|   |           |   |       |   |-- ld_translation_req (C:1, V:1'b1)
|   |           |   |       |   `-- lsu_ctrl.fu (C:1, V:LOAD)
|   |           |   |       `-- i_mmu.misaligned_ex_i.valid (C:1, V:1'b1)
|   |           |   |           |-- data_misaligned (C:1, V:1'b1)
|   |           |   |           |   |-- lsu_ctrl.operator (C:1, V:SD)
|   |           |   |           |   `-- lsu_ctrl.vaddr[2] (C:1, V:1'b1)
|   |           |   |           `-- lsu_ctrl.overflow (C:1, V:1'b0)
|   |           |   `-- i_mmu.pmp_data_allow (C:2, V:1'b1)
|   |           `-- i_load_unit.state_q[1:0] (C:2, V:2'b00)
|   `-- lsu_res_transid (C:3, V:3'b000)
|       `-- load_trans_id_o (C:3, V:3'b000)
|           `-- i_load_unit.load_data_q.trans_id (C:2, V:3'b000)
|-- lsu_lookup_transid_sampled (C:3, V:4'b0000)
|   |-- lsu_lookup_transid_response (C:2, V:1'b0)
|   |-- lsu_lookup_transid_sampled (C:2, V:4'b0000)
|   `-- lsu_lookup_transid_set (C:2, V:1'b0)
|-- lsu_lookup_transid_set (C:3, V:1'b0)
|   `-- lsu_req_hsk (C:3, V:1'b0)
|       |-- lsu_req_rdy (C:3, V:1'b0)
|       `-- lsu_req_val (C:3, V:1'b0)
|       |-- u_load_store_unit_sva.fu_data_i.fu[0] (C:3, V:1'b0)
|       |   `-- fu_data_i.fu[0] (C:3, V:1'b0)
|       `-- u_load_store_unit_sva.lsu_valid_i (C:3, V:1'b0)
|           `-- lsu_valid_i (C:3, V:1'b0)
`-- u_load_store_unit_sva.rst_ni (C:3, V:1'b1)
    `-- rst_ni (C:3, V:1'b1)
```

(a) Partial causal graph from the CVA6 LSU failure. "C" and "V" refer to cycle and value, respectively.



(b) Visualization of the same failure's causal graph. The image is generated via Gephi [5].

Figure 2: CVA6 LSU failure: textual subgraph (left) and full-graph visualization (right).

exception responses, violating transaction ID uniqueness properties.

- **LSU Transaction ID Mismatch**: A load-store unit (LSU) failure where simultaneous exceptions and cache responses cause responses with untracked transaction IDs.

Table 2 illustrates the substantial scale of these industrial designs. The LSU failure alone involves analyzing 1,918 lines of RTL across 7 files, generating a causal graph with 169 nodes and 1,145 edges tracking 122 unique signals. Through our graph consolidation technique (Section B.2), duplicate nodes from reconverging paths are merged into a DAG structure, making the analysis computationally tractable.

Table 2: Scale and complexity metrics for CVA6 processor designs

Design	RTL Files	Lines of Code	Graph Nodes	Graph Edges	Unique Signals
CVA6 MMU	3	695	172	235	76
CVA6 LSU	7	1,918	170	239	122

Figure 2a shows a representative subgraph from the LSU failure's causal analysis, illustrating the complex signal dependencies FVDebug must navigate. This snippet traces backwards from the failing assertion through 5 levels of causality—a fraction of the full 20-level, 170-node graph.

Table 3 presents evaluation results. FVDebug achieves highest Quality@Best (0.713) despite the complexity, though absolute scores are lower than simpler benchmarks, reflecting the inherent difficulty of multi-module processor debugging. While automated fix generation for such multi-line issues remains future work, FVDebug's ability to navigate these massive causal graphs and identify root causes with 0.713 quality represents an advance for industrial verification workflows.

E. Industrial FV Case Studies (Proprietary Designs)

We evaluate FVDebug on two proprietary, production-scale FV counterexamples (CEX), with design details masked per policy. Signal names have been replaced with descriptive placeholders in angle brackets (e.g., `<signal_name>`) to protect intellectual property. For each case, an expert provides the ground truth root cause, which we use to verify the FVDebug's outputs.

CEX-1 (FV Testbench Error)

This high-difficulty case involves an incorrect internal token counter. FVDebug correctly identifies the root cause as a

Table 3: Comparison of FVDebug against baselines and ablations on CVA6 processor failures. Best results in **bold**.

	Hypothesis Quality Metrics			
Method	**Quality@Best**	**NDCG@5**	**MRR**	**Kendall's τ**
Unstructured Baselines				
Direct LLM	0.575	0.801	0.500	0.333
Flat Trace Analysis	0.475	0.800	0.531	0.100
Component Ablations				
FVDebug w/o For-Against	0.644	0.791	0.531	0.306
FVDebug w/o Rover	0.300	0.742	0.276	0.294
FVDebug (Full)	**0.713**	**0.875**	**0.667**	**0.371**

functional bug in the testbench logic.

> *FVDebug Root Cause:* The <internal_token_count> calculation is incomplete—it only sums 4 specific FIFO token counts but does NOT include the mesh input path (<mesh_in_signal>) where tokens are actually entering the design.

Expert evaluation confirms an "Excellent Match," noting that FVDebug correctly identifies the missing signal in the counter logic and understands the architectural flaw. The system pinpoints the specific functional error and recommends the correct fix, aligning perfectly with the expert's independent analysis.

CEX-2 (Missing Constraint)

This case involves a FIFO overflow caused by a missing input constraint. FVDebug diagnoses the issue as a failure in the backpressure mechanism.

> *FVDebug Root Cause:* Backpressure Mechanism Failure: The FIFO continues accepting write requests (<fifo_write_enable>=1) even when the FIFO is full (<fifo_full_status>=1). This is caused by an unconstrained input signal (<req_valid_signal>) driving write logic without proper backpressure handling.

This analysis is rated as a "Strong Match" by the expert. Although the terminology differs slightly (FVDebug's "backpressure failure" vs. the expert's "credit protocol violation"), the core diagnosis is identical. FVDebug successfully identifies the problematic input signal and recommends the correct solution: adding a constraint to the formal testbench.

Table 4: Industrial CEX complexity and FVDebug performance metrics (masked design details).

Case	RTL Files	Lines of Code	Unique Signals	Graph Nodes	Graph Edges	Graph Depth	Jasper Calls	LLM Calls	Total Runtime
CEX-1 (Testbench Error)	265	560,202	123	189	227	15	163	25	4m 12s
CEX-2 (Missing Constraint)	257	554,983	41	80	86	11	67	21	6m 07s

Table 4 summarizes the complexity and performance metrics for these case studies. Despite the scale of the designs—over half a million lines of code—FVDebug automatically constructs and analyzes deep causal graphs, tracing the chain of dependencies back up to 15 levels from the point of failure for CEX-1. With a modest number of calls to Jasper and the LLM, FVDebug pinpoints the root causes in minutes, demonstrating its capability to accelerate the debugging of complex failures in production-scale industrial designs.

V. CONCLUSION AND FUTURE WORK

We presented FVDebug, an automated FV debugging system that transforms counter-examples into *causal graphs* and applies a multi-stage LLM pipeline—balanced scanning and agentic narrative exploration—to generate high-quality root-cause explanations and practical fixes. On open benchmarks, FVDebug achieves strong hypothesis quality and Pass@k rates. On proprietary, production-scale counterexamples, we demonstrate applicability by reporting graph- and code-scale metrics with expert-verified root causes.

Looking forward, the causal graph approach naturally extends to simulation-based verification, where graphs could be constructed from signals whose values mismatch golden references rather than from failed properties. This would unify debugging workflows across verification methodologies, providing consistent root cause analysis regardless of failure detection method.

REFERENCES

[1] Alpha Design AI. Chipagents: Agentic AI for RTL design and debug. https://chipagents.ai, 2025. Accessed Jul. 2025.

[2] ChipStack AI. Chipstack – chip design reimagined. https://www.chipstack.ai, 2025. Accessed Jul. 2025.

[3] Anthropic. Claude code. https://www.anthropic.com/claude-code, 2025. Accessed Aug. 2025.

[4] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

[5] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks, 2009.

[6] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[8] Cursor. Cursor – the AI code editor. https://cursor.com, 2025. Accessed Aug. 2025.

[9] Harry D Foster. 2020 wilson research group functional verification study. Technical report, Siemens EDA, 2020.

[10] Weimin Fu, Kaichen Yang, Raj Gautam Dutta, Xiaolong Guo, and Gang Qu. Llm4sechw: Leveraging domain-specific large language model for hardware debugging. In *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6. IEEE, 2023.

[11] Orna Grumberg, EM Clarke, and D Peled. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science; Springer: Berlin/Heidelberg, Germany*, 1999.

[12] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2):151–238, 1992.

[13] Muhammad Hassan, Mohamed Nadeem, Khushboo Qayyum, Chandan Kumar Jha, and Rolf Drechsler. Prompt. verify. repeat. llms in the hardware verification cycle.

[14] Aman Kumar and Deepak Narayan Gadde. Generative ai augmented induction-based formal verification. In *2024 IEEE 37th International System-on-Chip Conference (SOCC)*, pages 1–2. IEEE, 2024.

[15] Aman Kumar, Deepak Narayan Gadde, Keerthan Kopparam Radhakrishna, and Djones Lettnin. Saarthi: The first ai formal verification engineer. *arXiv preprint arXiv:2502.16662*, 2025.

[16] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*, 2023.

[17] Minh Luu, Surya Jasper, Khoi Le, Evan Pan, Michael Quinn, Aakash Tyagi, and Jiang Hu. Vcdiag: Classifying erroneous waveforms for failure triage acceleration. *arXiv preprint arXiv:2506.03590*, 2025.

[18] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

[19] Ann Mutschler. Debug tops verification tasks, Dec 2018. Semiconductor Engineering.

[20] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR, 2023.

[21] OpenAI. OpenAI o3-mini. `https://openai.com/index/openai-o3-mini/`, January 2025. Accessed: 2025-08-26.

[22] Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. Autosva: Democratizing formal verification of rtl module interactions. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 535–540. IEEE, 2021.

[23] Khushboo Qayyum, Chandan Kumar Jha, Sallar Ahmadi-Pour, Muhammad Hassan, and Rolf Drechsler. Llm-assisted bug identification and correction for verilog hdl. *ACM Transactions on Design Automation of Electronic Systems*, 2025.

[24] Dipayan Saha, Shams Tarek, Hasan Al Shaikh, Khan Thamid Hasan, Pavan Sai Nalluri, Md Ajoad Hasan, Nashmin Alam, Jingbo Zhou, Sujan Kumar Saha, Mark Tehranipoor, et al. Sv-llm: An agentic approach for soc security verification using large language models. *arXiv preprint arXiv:2506.20415*, 2025.

[25] Atefeh Sohrabizadeh, Jialin Song, Mingjie Liu, Rajarshi Roy, Chankyu Lee, Jonathan Raiman, and Bryan Catanzaro. Nemotron-cortexa: Enhancing llm agents for software engineering tasks via improved localization and solution diversity. In *Forty-second International Conference on Machine Learning*.

[26] Synopsys Inc. Vc formal. `https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html`, 2023.

[27] Synopsys, Inc. *Verdi Automated Debug System*, 2024. Online product page, accessed Jun 2025.

[28] Cadence Design Systems. *Cadence JasperGold Formal Verification Platform*, 2023. Version 2023.12.

[29] Cadence Design Systems. Indago debug platform – product brief. `https://dvcon-proceedings.org/wp-content/uploads/Indago%E2%84%A2-Debug-Platform-Overview.pdf`, 2025. Accessed Jul. 2025.

[30] Srikanth Vijayaraghavan and Meyyappan Ramanathan. *A practical guide for SystemVerilog assertions*. Springer, 2005.

[31] Jing Wang, Shang Liu, Yao Lu, and Zhiyao Xie. Hlsdebugger: Identification and correction of logic bugs in HLS code with LLM solutions. In *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, 2025.

[32] Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Nan Guan, and Zhe Jiang. Veridebug: A unified llm for verilog debugging via contrastive embedding and guided correction. *ICLAD*, 2025.

[33] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[34] Ke Xu, Jialin Sun, Yuchen Hu, Xinwei Fang, Weiwei Shan, Xi Wang, and Zhe Jiang. Meic: Re-thinking rtl debug automation using llms. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2024.

[35] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

[36] Jie Zhou, Youshu Ji, Ning Wang, Yuchen Hu, Xinyao Jiao, Bingkun Yao, Xinwei Fang, Shuai Zhao, Nan Guan, and Zhe Jiang. Insights from rights and wrongs: A large language model for solving assertion failures in rtl design. *DAC*, 2025.

A. EXAMPLE OUTPUT REPORT

This appendix presents a complete example output from FVDEBUG analyzing an accumulator design failure from the SVA-EVAL-HUMANbenchmark. The report demonstrates the system's ability to identify root causes, rank competing hypotheses, and generate concrete fixes.

A. Ranked Hypotheses

Hypothesis 1: Upstream Control Logic Issue

Confidence: 48.5%

Hypothesis Statement

The accumulation control chain (likely the `ready_add`/`end_cnt` logic) is not properly asserting `end_cnt` when the count reaches 3, which prevents `valid_out` from being driven high as expected.

Supporting Evidence

- The RTL code `assign end_cnt = ready_add && (count == 'd3);` shows that even if count equals 3, `end_cnt` will remain low unless `ready_add` is high.
- The always block for `valid_out` (lines 55-62) shows: `else if (end_cnt) valid_out <= 1'b1;`, yet at cycle 3, `valid_out` remains 0 even though the assertion expected it to be high.
- The failing assertion (`accu.valid_out_check_2_assertion` at cycle 3) directly indicates that `valid_out` is not asserted as required when `count==3` and `valid_in` is high.

Contradicting Evidence

- The design intentionally gates `end_cnt` on `ready_add`, meaning that if `ready_add` is low then a low `end_cnt` is valid as per design.
- The definition of `ready_add` as `(valid_out | !valid_in)` may be intentionally crafted for specific input conditions.

Hypothesis 2: Wrong Condition in ready_add Signal

Confidence: 46.9%

Hypothesis Statement

The use of the OR operator in the definition of `ready_add = valid_out | !valid_in` might be causing the `end_cnt` condition to fail when `valid_in` is expected to be high, thereby preventing `valid_out` from being asserted.

Supporting Evidence

- The RTL shows `assign ready_add = valid_out | !valid_in;`, indicating that the state of `valid_out` and `valid_in` directly affect the `ready_add` signal.
- At cycle 3, count is `2'b11` (i.e., 3) which is exactly the threshold condition, but with `valid_in` being `1'b0`, the OR operator prevents proper accumulation triggering.

Analysis

- The `ready_add` signal definition appears nonstandard and may inadvertently disable accumulation triggering when `valid_in` is high.
- This could be either a design intent problem or an under-constrained formal property where the assumption about `valid_in` being high is missing.
- There is a possibility that the duplicated accumulation logic is introducing subtle timing issues.

B. Causal Chain Timeline

Causal Chain Timeline

Cycle 1 ⚠ The `count` node is observed at `2'b11` (value of 3), which is the threshold for asserting `end_cnt`. However, the expected behavior (resetting counter and driving `valid_out` high) does not occur.

Cycle 1 The node `end_cnt`, which should signal the end of accumulation when count equals 3, is observed as `1'b0`. Its value is computed as `ready_add && (count == 'd3)` making it dependent on `ready_add`.

Cycle 3 ⚠ The assertion `accu.valid_out_check_2_assertion` fails because `valid_out` is 0 when expected to be 1.

Cycle 3 The `count` signal is `2'b11`, indicating the counter reached value 3 but did not trigger the `end_cnt` condition.

Cycle 3 `valid_out = 1'b0` (suspicious)

C. Suggested Fix

RTL Fix (Confidence: 90%)

```
// Buggy Code:
assign ready_add = valid_out | !valid_in;

// Fixed Code:
assign ready_add = valid_in & !valid_out;

// Explanation: The original logic makes the module ready to add
// when there is NO valid input (!valid_in), which is backwards.
// The fix ensures ready_add is high only when:
// - New valid data is available (valid_in = 1)
// - Previous data has been processed (valid_out = 0)
```

D. Causal Graph Visualization

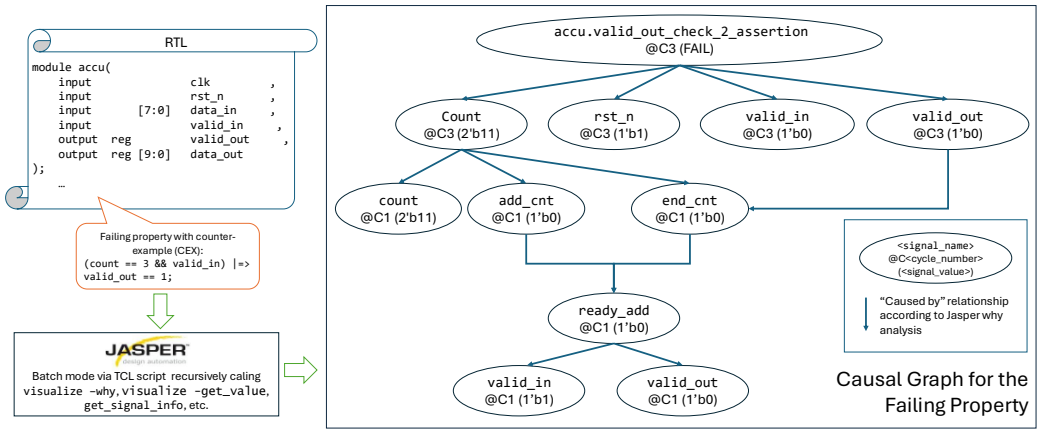


Figure 3: Causal graph automatically generated by FVDebug for the accumulator verification failure.

B. CASE STUDY: IMPACT OF FOR-AND-AGAINST PROMPTING

This section demonstrates the critical importance of balanced evaluation through a real debugging example from the accumulator design. We compare the analysis quality with and without for-and-against prompting on the same suspicious signal.

A. The Problem: `ready_add` Signal Analysis

Consider the `ready_add` signal in an accumulator module at cycle 1, where the signal controls when data should be added to the accumulator. The RTL implementation is:

Listing 1: RTL implementation of ready_add signal

```
assign ready_add = valid_out | !valid_in;
```

At cycle 1, we observe:

- `ready_add = 1'b0` (not ready to add)
- `valid_in = 1'b1` (valid input available)
- `valid_out = 1'b0` (no valid output yet)
- `count = 2'b11` (already at maximum count)

B. Without For-and-Against Prompting

When prompted with standard bug-finding instructions ("Analyze if this signal behavior indicates a bug"), the LLM produces:

Standard Prompting Analysis

Analysis of ready_add@C1:
The signal `ready_add` is low when `valid_in` is high and `valid_out` is low. According to the RTL logic `valid_out | !valid_in`, this evaluates to `0 | !1 = 0 | 0 = 0`, which matches the observed value.
Conclusion: The signal behavior is correct and follows the RTL implementation. No bug detected. The logic is working as designed.
Confidence: Normal behavior (no issue found)

This analysis completely misses the fundamental design flaw—the LLM focuses solely on whether the implementation matches the RTL, without questioning whether the RTL logic itself makes sense.

C. With For-and-Against Prompting

When prompted with mandatory balanced evaluation, requiring both FOR and AGAINST arguments, the same LLM produces:

For-and-Against Prompting Analysis

Analysis of ready_add@C1:
Arguments FOR Being Suspicious (REQUIRED - MIN 2):

- The logic `valid_out | !valid_in` seems counterintuitive because `ready_add` should ideally be high when `valid_in` is high (indicating new data is ready), not when it's low.
- The dependency on `valid_out` being high to set `ready_add` might create a circular dependency or delay in processing new inputs when `valid_out` is low.

Arguments AGAINST Being Suspicious (REQUIRED - MIN 2):

- The design might intend for `ready_add` to be high in scenarios other than just new data arrival, such as during certain states of output validity.
- The inversion of `valid_in` might be a design choice to handle specific edge cases or reset conditions not detailed in the provided context.

Balanced Conclusion: The behavior of `ready_add` is suspicious due to its reliance on the inverted `valid_in`, which contradicts typical ready signal behavior. This is likely a root cause of potential data handling issues.
Classification: ROOT CAUSE (Suspicion score: 0.70)

D. The Actual Bug and Fix

The for-and-against analysis correctly identified the root cause. The original logic makes the module ready to add data when there is *no* valid input (!valid_in), which is backwards. The correct implementation should be:

Listing 2: Corrected ready_add implementation

```
assign ready_add = valid_in & !valid_out;
```

This fix ensures the module is ready to add data only when:

- New valid data is available (`valid_in = 1`)
- Previous data has been processed (`valid_out = 0`)

E. Why For-and-Against Prompting Works

The effectiveness of this approach stems from several cognitive mechanisms:

E.1 Overcoming Confirmation Bias

Standard prompting allows the LLM to follow the path of least resistance. When it sees that signal values match the RTL implementation, it concludes "working as designed" without questioning the design itself. The forced dual perspective breaks this pattern.

E.2 Encouraging Critical Thinking

By mandating arguments for both sides, the prompt forces the model to actively search for potential issues even when the initial impression suggests normalcy. This mirrors how expert engineers debug—they question assumptions and consider alternative explanations.

E.3 Revealing Hidden Assumptions

The AGAINST arguments often reveal the LLM's default assumptions (e.g., "the RTL must be correct"). By making these explicit, the FOR arguments can then challenge them directly, leading to more thorough analysis.

C. IMPLEMENTATION DETAILS

This section provides detailed implementation insights for the key components of FVDebug.

A. Causal Graph Construction and Consolidation

The recursive dependency analysis for constructing $\mathcal{G}$ begins at the failing property node and traverses backwards through the counter-example trace. At each node, the system issues a `visualize -why` query to JasperGold, which returns the immediate causal parents. The default trace depth of 20 cycles was empirically determined to capture most relevant causal chains while maintaining tractability—deeper traces showed diminishing returns in root cause identification accuracy.

The consolidation phase transforms the initial tree structure into a DAG through node deduplication. Each node is uniquely identified by the tuple (signal_name, cycle, value). When multiple tree paths converge to the same signal event, we merge these into a single DAG node while preserving all incoming edges. This consolidation typically reduces node count in designs with significant reconvergent fanout, substantially improving analysis efficiency.

B. Context Retrieval Architecture

The context retrieval system pre-processes design documentation into searchable chunks before analysis begins. For each unique signal in $\mathcal{G}$, we generate three types of queries: RTL queries that search for signal definitions and assignments, specification queries that look for functional descriptions, and cross-reference queries that identify related signals and modules. The retrieval system maintains a two-level cache—a global cache for design-wide information shared across all nodes, and a node-specific cache for local context.

The mapping between signals and documentation uses fuzzy matching to handle naming variations. For instance, a signal `valid_out` might be documented as "output valid signal" or "valid output flag." The system computes similarity scores using edit distance and semantic embeddings, accepting matches above a threshold of 0.7.

C. Token-Aware Batch Optimization

The binary search algorithm for determining optimal batch sizes operates within the token budget of 50,000 tokens per prompt (as configured in our experiments). The search begins by estimating the token count for a single node's analysis context, typically consuming 800-1200 tokens. The algorithm then performs binary search between 1 and min(|remaining nodes|, $\lfloor$50000/single_node_tokens$\rfloor$).

At each iteration, the system constructs a test prompt with the candidate batch size and uses a token counter to determine exact usage. If the prompt exceeds the budget, the algorithm reduces the upper bound; otherwise, it increases the lower bound. The search terminates when the bounds converge, typically requiring 4-6 iterations. Our experiments with batch_size=5 as the target showed that actual achieved batch sizes varied from 3 to 8 depending on node context complexity.

D. Hypothesis Initialization and Management

The Insight Rover initializes hypotheses from suspicious nodes using a scoring-based selection. FVDebug maintains exactly three active narratives throughout exploration. When the scanner identifies more than three suspicious nodes, we select the top three based on their suspicion scores, ensuring diverse initial hypotheses by requiring a minimum score difference of 0.1 between selected nodes.

Each hypothesis maintains a frontier of unexplored nodes, initially populated with all parents and children of the seed node. The frontier grows as exploration progresses—when a node is analyzed, its neighbors are added to the frontier if not already explored. To prevent explosive frontier growth in highly connected regions of $\mathcal{G}$, we limit frontier size to 20 nodes, prioritizing those with highest suspicion scores from their initial analysis.

The narrative pool management occurs at each iteration's end. Hypotheses with confidence below 0.2 after three iterations are marked as weak but retained for final reporting. If any narrative slot becomes vacant due to convergence, the system attempts to spawn a new hypothesis from unexplored suspicious nodes, ensuring maximum utilization of the narrative budget.

E. Intelligent Node Selection for Exploration

The LLM-guided node selection mechanism evaluates frontier nodes based on three criteria: relevance to the current hypothesis, potential information gain, and structural importance in $\mathcal{G}$. The selection process provides the LLM with the current narrative state, including accumulated evidence and confidence score, along with a summary of up to 10 frontier nodes showing their signal names, values, and graph-theoretic properties such as in-degree and out-degree.

To ensure comprehensive analysis, the system dynamically adjusts the number of narratives based on the number of suspicious nodes identified. While configured with insight_rover_max_narratives=3 as a minimum, the implementation automatically expands to accommodate all suspicious nodes—if 5 nodes are flagged as suspicious, 5 initial narratives are created. This design choice ensures no potentially critical failure path is overlooked during the initial hypothesis generation phase.

The system tracks exploration efficiency through a path coverage metric. Across our experiments, the intelligent selection explored an average of 15-20 nodes per hypothesis, compared to over 100 nodes that would be explored by breadth-first search to the same depth. This selective exploration maintains comparable root cause identification accuracy while reducing computational cost.

F. Hypothesis Ranking with LLM Evaluation

The final hypothesis ranking uses a pairwise comparison approach rather than absolute scoring. For n hypotheses, the system performs $O(n \log n)$ comparisons using a tournament-style evaluation. Each comparison presents two hypotheses to the LLM along with their evidence chains, asking which better explains the observed failure.

The ranking criteria are weighted as follows: causal sufficiency (0.3), evidence quality (0.25), mechanistic clarity (0.25), actionability (0.15), and narrative coherence (0.05). The LLM assigns scores for each criterion, which are combined using the weighted sum to produce a final ranking score.

G. Fix Generation Strategy Diversification

The ensemble fix generator implements five distinct strategies that vary in their context emphasis. The **full context strategy** includes all available information but risks overwhelming the LLM with irrelevant details. The **suspicious focus strategy** filters context to only highly suspicious signals (score > 0.7), reducing noise but potentially missing systemic issues. The **narrative focus strategy** uses only the top-ranked hypothesis from $\mathcal{H}$, providing strong causal reasoning but limited coverage. The **minimal context strategy** extracts just the identified root cause, generating focused fixes but lacking broader understanding. The **best-of strategy** reviews outputs from other strategies and selects the most promising fixes.

Each strategy operates with a retry mechanism, attempting up to twice if initial generation fails validation.

H. Fix Signature Generation and Deduplication

The *CreateSignature* function generates a unique identifier for each fix by normalizing and hashing the buggy code and corrected code pair. The normalization process removes all whitespace variations, converts tabs to spaces, and

eliminates comments. It then applies a deterministic ordering to commutative operations (e.g., `a & b` becomes `a & b` regardless of original ordering).

I. Multi-Level Fix Validation

The validation pipeline for generated fixes operates in three stages. First, exact substring matching attempts to locate the buggy code directly in the RTL codebase $\mathcal{R}$. For the remaining fixes, whitespace normalization handles formatting variations. Fixes that fail all the validation stages are discarded.

J. Consensus Ranking and Confidence Scoring

The consensus ranking mechanism assigns higher confidence to fixes generated by multiple strategies. For a fix generated by k strategies out of 5 total, the consensus boost is calculated as $\min(0.2 \cdot k, 0.6)$, capping the maximum boost at 0.6 to prevent over-reliance on consensus alone. The final confidence score combines the original confidence from fix generation and consensus boost.

D. PROMPT DETAILS

This appendix provides the detailed prompts used by FVDebug's three main components: Graph Scanner, Insight Rover, and Fix Generator.

A. Graph Scanner Prompts

The Graph Scanner analyzes each node in the causal graph using a structured prompt that enforces balanced evaluation with mandatory FOR and AGAINST arguments.

B. Insight Rover Prompts

The Insight Rover uses three distinct prompts for hypothesis generation, exploration planning, and node analysis within narrative context.

C. Fix Generator Prompts

The Fix Generator uses an ensemble approach with multiple prompting strategies. All strategies share a common base prompt and fix generation instructions.

D. Narrative Ranking Prompts

FVDebug uses LLM-based ranking at two critical points: ranking competing narratives during exploration and evaluating hypothesis quality against ground truth during benchmarking.

Graph Scanner Full Prompt Template

SCENARIO
{scenario_description}

GLOBAL CONTEXT AND INSIGHTS
The following high-level insights provide important context about the design:

Design Overview
{design_overview_content}

Specification Requirements
{specification_content}

NODES TO ANALYZE
node_id	signal	cycle	value
{node_table_rows}

SUBGRAPH EDGE LIST
{edge_list}

NODE-SPECIFIC CONTEXT (RTL & SPEC)
CONTEXT 1
{rtl_and_spec_context}

Based on the parent analysis and context, analyze this signal transition.

CRITICAL REQUIREMENT - EXACT RTL REFERENCES:
When analyzing any signal behavior or identifying issues, you MUST:

1. Provide exact file:line references from the RTL context
Example: "store_unit.sv:145-147"

2. Quote the specific RTL code that shows the behavior
Example:
```verilog
assign valid_out = end_cnt ? 1'b1 : 1'b0;
```

3. NEVER make claims without showing the supporting RTL code
4. If RTL context is missing, explicitly state: "RTL context not available for this signal"

MANDATORY BALANCED ANALYSIS APPROACH:
YOU MUST PROVIDE BOTH SECTIONS - NEVER SKIP EITHER ONE:

1. **Arguments FOR Being Suspicious (REQUIRED - MINIMUM 2 POINTS):**
- Even if the signal seems normal, you MUST identify at least 2 potential concerns
- Consider: timing issues, edge cases, specification mismatches, unusual patterns
- Think critically: What COULD go wrong? What assumptions might be invalid?

2. **Arguments AGAINST Being Suspicious (REQUIRED - MINIMUM 2 POINTS):**
- Even if the signal seems problematic, you MUST provide at least 2 counterarguments
- Consider: valid design patterns, expected behavior, specification compliance
- Think: Why might this actually be correct behavior?

Figure 4: Graph Scanner prompt template (part 1 of 3).

Graph Scanner Prompt Template (continued)

IMPORTANT: The LLM tends to assume signals are normal. To counteract this bias:
- Be MORE critical in the FOR section - look harder for potential issues
- Challenge assumptions - just because a signal follows RTL doesn't mean RTL is correct
- Consider specification violations, race conditions, and edge cases

Your analysis should examine:
- **RTL Logic Correctness**: Does the combinational logic make sense? Are there any logical inversions or operations that seem incorrect?
- **Signal Dependencies**: Are the boolean operations (AND, OR, NOT) used correctly? Could there be a logic error?
- **Specification Alignment**: Does the RTL match the intended behavior described in the specification?
- **Common RTL Bugs**:
* Incorrect polarity (should a signal be inverted?)
* Wrong logical operators (OR vs AND, etc.)
* Missing or extra conditions in assignments
* Circular dependencies or combinational loops
* Reset/initialization issues
- **Design Intent**: Based on signal names, does the implementation make semantic sense?
- **Edge Cases**: Could the current logic fail under certain conditions?

IMPORTANT: Prioritize ROOT CAUSES over SYMPTOMS:
- A root cause is the earliest point where incorrect behavior originates
- A symptom is a downstream effect of a root cause
- Only mark signals as highly suspicious (greater than 0.7) if they are likely root causes
- For symptoms, use lower scores (0.3-0.5) and explicitly state which root cause they derive from

Use this scoring rubric for suspicion_score:
0.9-1.0: Direct RTL bug found (e.g., wrong operator, missing condition)
0.7-0.8: Likely logic error (e.g., incorrect state transition)
0.5-0.6: Suspicious pattern that might indicate issue
0.3-0.4: Downstream symptom of another issue
0.0-0.2: Normal behavior or insufficient evidence

INSTRUCTIONS
Return a single JSON object where keys are node_id and values follow this schema:

{
"node_id": {
"is_suspicious": bool,
"is_key_event": bool,
"suspicion_score": float (0.0-1.0),
"importance_score": float (0.0-1.0),
"causal_validity": {"parent_id": bool, ...},
"analysis": "Structured markdown analysis following this EXACT template:

Signal Behavior
[Description of what the signal does and its current value]

RTL Evidence
- File: [filename:line_numbers]
```verilog
[relevant RTL code]
```

Figure 5: Graph Scanner prompt template (part 2 of 3) with JSON response schema.

Graph Scanner Prompt Template (continued)

Arguments FOR Being Suspicious (REQUIRED - MIN 2)
- [First potential issue/concern]
- [Second potential issue/concern]

Arguments AGAINST Being Suspicious (REQUIRED - MIN 2)
- [First reason this might be normal]
- [Second reason this might be normal]

Balanced Conclusion
[Weigh both sides and conclude whether this is suspicious or not]

Root Cause vs Symptom
[If suspicious: Is this a root cause or symptom? If symptom, what's the root?]

Fix Required
[If issue found: Specific code change needed. If not: 'No fix required']"
}
}

Figure 6: Graph Scanner prompt template (part 3 of 3).

Insight Rover: Initial Hypothesis Generation Prompt

Given a suspicious node in a hardware failure analysis, generate an initial hypothesis.

Node: {node.signal_name} at cycle {node.cycle}
Value: {node.value}
RTL Context: {context.rtl}
Spec Context: {context.spec}

 # Prior Analysis From GraphScanner
{prior_analysis_raw}

Generate a hypothesis about what might be wrong. Consider these possibilities:
- RTL bug (incorrect logic, missing conditions, wrong operators)
- Under-constrained inputs (missing assumptions in formal verification)
- Assertion/property issue (the checker itself might be wrong)
- Design intent mismatch (RTL correct but doesn't match specification)

Generate a hypothesis in JSON format:
{
"title": "Brief title for the hypothesis",
"hypothesis": "One-line hypothesis about what might be wrong (be specific about the type of issue)",
"initial_insights": ["insight1", "insight2", ...]
}

Figure 7: Insight Rover prompt for generating initial hypotheses from suspicious nodes.

Insight Rover: Exploration Target Selection Prompt

Given a narrative hypothesis and exploration frontier, select the most promising nodes to explore next.

Narrative: {narrative.hypothesis}
Current confidence: {narrative.confidence_score:.2f}
Events found: {len(narrative.events)}

Exploration frontier:
- {node_id1}: {signal1} = {value1}
- {node_id2}: {signal2} = {value2}
- {node_id3}: {signal3} = {value3}
[... up to 10 frontier nodes shown ...]

Select up to 3 nodes that would best help validate or refute this hypothesis.
Return as JSON: {"targets": ["node_id1", "node_id2", ...]}

Figure 8: Insight Rover prompt for selecting which frontier nodes to explore next.

Insight Rover: Node Analysis in Narrative Context

Analyze this node in the context of the narrative hypothesis.

Narrative: {narrative.hypothesis}
Current timeline:
C{cycle1}: {signal1} = {value1}
C{cycle2}: {signal2} = {value2}
[... last 5 events shown ...]

Node to analyze: {node.signal_name} at cycle {node.cycle}
Value: {node.value}
RTL Context: {context.rtl}

 # Prior Analysis From GraphScanner
{prior_analysis_raw}

IMPORTANT: When providing evidence, directly quote or reference specific facts from the RTL context above.
For example: "The RTL shows 'assign ready = valid && !busy' which indicates..."

Determine:
1. Is this node relevant to the narrative?
2. Does it support or contradict the hypothesis?
3. Is it part of the critical path?
4. Extract specific evidence from the provided context

Return analysis as JSON with fields:
- is_relevant: boolean
- is_critical: boolean
- event_description: string (if relevant)
- importance: float (0-1)
- evidence_strength: float (0-1)
- evidence_for: [list of SPECIFIC facts/quotes from the RTL context that support the hypothesis]
- evidence_against: [list of SPECIFIC facts/quotes from the RTL context that contradict the hypothesis]
- new_insights: [list of analytical insights based on the evidence]

Figure 9: Insight Rover prompt for analyzing nodes within the context of a specific narrative hypothesis.

Fix Generator: Core Instructions

IMPORTANT: Please analyze this formal verification issue carefully and provide your response ONLY in the following JSON format:

```
{
"category": "RTL Bug" or "Under-Constraint" or "Over-Constraint",
"analysis": "Your detailed analysis of the issue including root cause and evidence",
"fixes": [
{
"buggy_code": "The EXACT problematic code snippet that needs to be fixed
(must be an exact substring from the original code)",
"code": "Your proposed fixed code that should replace the buggy code",
"description": "Explanation of what this fix does and why it addresses the root cause",
"confidence": 0.9,  // A value between 0 and 1 indicating your confidence in this fix
"location": {
"module": "Target module name",
"signal": "Target signal name",
"file": "Target file path",
"line": 42  // Target line number where fix should be applied
}
},
// Please try to include at least 5 alternative fixes in RANKED ORDER
// The first fix should be your best solution (highest confidence)
// Each additional fix should be an alternative approach with decreasing confidence
{
"buggy_code": "The same EXACT problematic code that needs to be fixed",
"code": "An alternative fixed code that should replace the buggy code",
"description": "Explanation of this alternative approach",
"confidence": 0.8,  // Lower confidence than your top solution
"location": { ... }
}
// Continue with more alternative approaches (3-5 total fixes)
]
}
```

Figure 10: Fix Generator JSON format and basic instructions (part 1 of 2).

Fix Generator: Critical Requirements

CRITICAL REQUIREMENTS:
1. The "buggy_code" field MUST contain EXACT code that exists in the RTL
2. The "code" field must contain ACTUAL CORRECTED RTL CODE - not empty, not placeholders
3. Both fields must contain valid Verilog/SystemVerilog syntax
4. Generate AT LEAST 3-5 different fixes if possible
5. Focus on FUNCTIONAL bugs that affect behavior - NOT style issues
6. Do NOT add testbench/verification signals (like _assert)
7. Do NOT use angle brackets or placeholder text like "TODO", "TBD", etc.
8. Make sure the "buggy_code" is an exact substring that exists in the original code
9. The "line" number should point to the approximate location of the buggy code

WHITESPACE HANDLING:
- IMPORTANT: Pay careful attention to whitespace in the "buggy_code" field
- Use SPACES instead of TABS in your code - avoid using
t characters
- Try to match the whitespace pattern from the original RTL code
- When in doubt, use single spaces between tokens (e.g., "assign signal = value;")
- The validation will try to match with flexible whitespace, but exact matches work best

IMPORTANT NOTES:
- Ensure your analysis thoroughly explains the root cause of the issue
- Provide multiple alternative fixes ranked from highest to lowest confidence
- Each fix should be a complete, working solution - no placeholders
- Consider different types of fixes:
* RTL bug (incorrect logic, missing conditions, wrong operators)
* Under-constrained inputs (missing assumptions in formal verification)
* Assertion/property issue (the checker itself might be wrong)
* Design intent mismatch (RTL correct but doesn't match specification)

STRICT SPAN RULES:
- Use a SINGLE, MINIMAL SPAN for both fields (one assignment or contiguous lines only)
- Do NOT include case labels, begin/end, or asserts in buggy_code or code
- We perform literal text replacement of buggy_code with code; include only the exact text to replace

Return either a JSON array of fixes or an object with a 'fixes' array.
Do NOT include any markdown code blocks or additional text outside the JSON.

Figure 11: Fix Generator critical requirements and guidelines (part 2 of 2).

Fix Generator: Strategy-Specific Context Additions

Strategy: full_context
Key Insights from Analysis:
- {insight1}
- {insight2}
[... up to 10 insights ...]

Most Suspicious Signals:
- Signal '{signal}' at cycle {cycle}: suspicion score {score:.2f}
Insights: {insight1}; {insight2}
[... up to 5 suspicious signals ...]

Causal Analysis Narratives:
1. {narrative1}
2. {narrative2}
[... up to 3 narratives ...]

Strategy: suspicious_focus
CRITICAL: Focus on these suspicious signals:
- Signal '{signal}' (cycle {cycle}): HIGH SUSPICION ({score:.2f})
→ {insight1}
→ {insight2}
→ {insight3}
[... up to 7 suspicious signals with detailed insights ...]
Prioritize fixes for these highly suspicious signals!

Strategy: causal_narratives_focus
Root Cause Narratives (FOCUS ON THESE):
Narrative 1:
{full_narrative1}
Narrative 2:
{full_narrative2}
[... up to 5 narratives ...]
Generate fixes that directly address the root causes identified in these narratives.

Strategy: minimal_context
Critical Issue:
ROOT CAUSE: {root_cause_bug}
Generate 3-5 surgical fixes for this specific issue.

Strategy: bugs_and_suggestions_only
Bugs and Fix Suggestions:
Signal '{signal}' bugs:
- {bug1}
- {bug2}
Suggested fixes:
- {suggestion1}
- {suggestion2}
[... up to 5 signals with bugs and suggestions ...]

Figure 12: Examples of strategy-specific context additions for the Fix Generator ensemble approach.

Narrative Ranking: Intrinsic Quality Assessment

You are an expert hardware verification engineer evaluating competing hypotheses for a formal verification failure.

PROBLEM DESCRIPTION:
{problem_description}

Your task is to rank these hypotheses based on their intrinsic qualities that correlate with correctness, WITHOUT knowing the ground truth.

HYPOTHESES TO EVALUATE:
—
HYPOTHESIS #1 (ID: {narrative_id1})
{narrative_text1}
—
HYPOTHESIS #2 (ID: {narrative_id2})
{narrative_text2}
[... additional hypotheses ...]

EVALUATION CRITERIA:

Score each hypothesis on these dimensions (0.0 to 1.0):

1. **Sufficiency** (0.0-1.0): Does the hypothesis provide a complete explanation of the failure?
- High (0.8-1.0): Fully explains the failure mechanism with clear causal chain
- Medium (0.4-0.7): Partial explanation with some gaps
- Low (0.0-0.3): Incomplete or superficial explanation

2. **Evidence** (0.0-1.0): Quality and quantity of supporting evidence from the causal analysis
- High (0.8-1.0): Strong evidence with specific signal values, RTL snippets, and clear causal links
- Medium (0.4-0.7): Some evidence but lacks specificity or completeness
- Low (0.0-0.3): Weak or contradictory evidence

3. **Mechanistic Insight** (0.0-1.0): Clarity of the failure mechanism explanation
- High (0.8-1.0): Clear explanation of HOW the bug manifests in hardware behavior
- Medium (0.4-0.7): Some mechanistic understanding but unclear details
- Low (0.0-0.3): Vague or incorrect understanding of hardware behavior

4. **Actionability** (0.0-1.0): Does it provide clear guidance on what to fix?
- High (0.8-1.0): Specific fix location and clear correction needed
- Medium (0.4-0.7): General area identified but unclear exact fix
- Low (0.0-0.3): No clear fix guidance

5. **Coherence** (0.0-1.0): Internal consistency and logical flow
- High (0.8-1.0): Logically consistent with no contradictions
- Medium (0.4-0.7): Mostly consistent with minor issues
- Low (0.0-0.3): Contains contradictions or illogical jumps

IMPORTANT CONSIDERATIONS:
- Favor hypotheses that identify specific RTL bugs over vague constraint issues
- Value concrete evidence (specific signal values, code snippets) over speculation
- Prefer hypotheses with clear causal chains showing propagation of errors
- Penalize hypotheses that blame tools/extraction without strong justification
- Reward specificity about the exact issue and its location

Figure 13: LLM prompt for ranking narratives based on intrinsic quality metrics without ground truth.

Narrative Ranking: Output Format

OUTPUT FORMAT:
Provide your evaluation as a JSON array where each element corresponds to a hypothesis:

```
[
{
"hypothesis_id": "ID of the hypothesis being evaluated",
"sufficiency": 0.85,
"evidence": 0.90,
"mechanistic_insight": 0.80,
"actionability": 0.75,
"coherence": 0.95,
"overall_score": 0.85, // Average of the five scores
"reasoning": "Brief explanation of why these scores were assigned",
"rank_suggestion": 1 // Your suggested rank (1 = best)
},
{
"hypothesis_id": "ID of the second hypothesis",
"sufficiency": 0.60,
"evidence": 0.55,
"mechanistic_insight": 0.50,
"actionability": 0.45,
"coherence": 0.70,
"overall_score": 0.56,
"reasoning": "Explanation for this hypothesis",
"rank_suggestion": 2
}
]
```

CRITICAL: Evaluate ALL hypotheses and return them in your suggested rank order (best first).

Figure 14: JSON output format for narrative ranking results.

Ground Truth Evaluation Prompt

You are an expert hardware verification engineer evaluating hypothesis quality for debugging formal verification failures.

PROBLEM DESCRIPTION:
{problem_description}

GOLDEN ANSWER (Ground Truth):
{golden_answer}

HYPOTHESIS TO EVALUATE (Rank #{hypothesis_rank}):
{hypothesis}

ADDITIONAL CONTEXT:
- This hypothesis was ranked #{hypothesis_rank} in a list of hypotheses
- The golden answer above shows the correct root cause identification (could be RTL bug, constraint issue, property issue, etc.)

EVALUATION TASK:
Score this hypothesis on the following dimensions (0.0 to 1.0):

1. **Relevance** (0.0-1.0): Does it address the actual issue described in the golden answer?
2. **Preciseness** (0.0-1.0): Is it specific about the root cause? Does it correctly identify the exact issue?
3. **Causal_Timeline** (0.0-1.0): Does it include a causal timeline or temporal analysis showing how the bug manifests over time? Higher scores for detailed cycle-by-cycle analysis.
4. **Correctness** (0.0-1.0): Does it correctly identify the root cause as shown in the golden answer?

SCORING GUIDELINES:
- High relevance (0.8-1.0): Directly mentions the specific issue from the golden answer
- Medium relevance (0.4-0.7): Mentions related issues but not the specific root cause
- Low relevance (0.0-0.3): Vague or mentions unrelated issues

- High preciseness (0.8-1.0): Specifically identifies the exact issue as in the golden answer
- Medium preciseness (0.4-0.7): Mentions the general area of the issue but lacks specifics
- Low preciseness (0.0-0.3): Vague statements without specific identification

- High causal_timeline (0.8-1.0): Includes detailed cycle-by-cycle timeline showing bug progression
- Medium causal_timeline (0.4-0.7): Includes some temporal analysis or partial timeline
- Low causal_timeline (0.0-0.3): No timeline or temporal analysis provided

- High correctness (0.8-1.0): Correctly identifies the root cause as shown in golden answer
- Medium correctness (0.4-0.7): Partially correct but includes incorrect elements
- Low correctness (0.0-0.3): Incorrect or focuses on non-existent issues

Figure 15: Evaluation prompt for comparing hypotheses against ground truth (part 1 of 2).

Ground Truth Evaluation Prompt (continued)

IMPORTANT CONSIDERATIONS:
- Score based on alignment with the golden answer, regardless of whether it's an RTL bug, constraint issue, or property issue
- Hypotheses that identify a different type of issue than the golden answer should receive lower scores
- Value specificity: hypotheses that identify the exact issue (e.g., specific condition, signal, or constraint) should score higher
- REWARD detailed causal timelines that show the bug's progression through cycles - this demonstrates thorough analysis
- Do NOT penalize verbosity if it provides valuable temporal analysis or causal chain information

OUTPUT FORMAT:
Provide your evaluation in the following JSON format:
{
"relevance": <float between 0.0 and 1.0>,
"preciseness": <float between 0.0 and 1.0>,
"causal_timeline": <float between 0.0 and 1.0>,
"correctness": <float between 0.0 and 1.0>,
"overall": <float between 0.0 and 1.0 (average of the four scores)>,
"reasoning": "<brief explanation of scores>"
}

Analyze the hypothesis carefully and provide your JSON evaluation:

Figure 16: Evaluation prompt for comparing hypotheses against ground truth (part 2 of 2).