STITCH: Assertion-Guided Patching of On-Chip Protocol Implementations using LLMs

Melisande Zonta-Roudes
ETH Zurich
Switzerland

Nora Hinderling
ETH Zurich
Switzerland

Shweta Shinde
ETH Zurich
Switzerland

Abstract

Verification flows use Verification IPs (VIPs) to identify assertion violations. This is well-suited for on-chip protocols (e.g., AXI, AHB) with standard specifications (e.g., AMBA) for developers to write assertions that detect violations. Patching these violations, however, still requires manual effort. A good patch must not only fix the violation but also preserve functionality (e.g., pass the testsuite) without causing new violations. We propose STITCH, based on the intuition that given an implementation, a violated assertion, and a counterexample an LLM can try to synthesize patches. To evaluate STITCH, we develop a new dataset that comprises 100 violations in 11 implementations of 5 protocols (AXI, AHB, APB, Wishbone, and TileLink). Our first experiment reports 19% success rate across 4 LLMs (GPT4, GPT5, Gemini, and Claude). By analyzing the failed cases, we devise 3 improvement strategies: patch localization, violation specific context (cone of influence, counterexample), iterative feedback from model checker outputs over candidate patches. Our experiments show these strategies help STITCH achieve 61% patch success, with GPT5 dominating with 56%. We validate STITCH patches through simulation and on real hardware. We compare STITCH to 3 state-of-the-art non-LLM tools and existing patches.

ACM Reference Format:
Melisande Zonta-Roudes, Nora Hinderling, and Shweta Shinde. 2026. STITCH: Assertion-Guided Patching of On-Chip Protocol Implementations using LLMs. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3804341

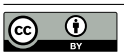
1 Introduction

On-chip communication protocols (e.g., ARM AMBA AXI [14], APB [16], AHB [15], Wishbone [61], TileLink [52]) share fundamental structural and behavioral similarities, as they are designed to ensure reliable data transfer between IP blocks. Post-verification validates if an implementation satisfies functional and performance requirements stated in protocol specifications (e.g., ensuring that data transfers between IP blocks occur without delay or data loss) or developer-defined constraints (e.g., enforcing timeouts after a given number of cycles). If there are assertion violations, a developer writes a patch that satisfies the verification.

In this paper, we propose STITCH that aims to automatically generate patches for implementations that violate assertions. More importantly, our definition of a good patch not only ensures that the assertion is proven but also checks if it is functional and stays compliant with previously proven assertions. LLMs have shown promise in RTL code generation tasks (e.g., test-driven implementation, DFT compatibility, fixing syntax errors) [48, 55, 59]. Thus, for our problem, we aim to assess the effectiveness of LLMs. As a first step, we curate a dataset of 100 protocol violations collected from 11 implementations for 5 protocols. Experimentally, for our dataset, when we provide the violation and implementation to 7 LLMs with a prompt to generate a patch, all of them fail to produce any good patches.

Next, we test the effectiveness of providing counterexamples, with the intuition that it will help the LLM to understand the root cause. This approach yields 19% good patches across GPT5, Gemini, Claude, GPT4. Based on our analysis of these two experiments, we present novel insights to improve the patch success rate. We identify the patch point using assertion, cone of influence and implementation analysis, which helps the LLM in reducing the search space for patches and improves the patch success rate to 40%. Next, even in cases where the patch fails, we guide the LLM to learn from the failure. For each failed patch, we use the model checker to generate counterexamples and ask the LLM to summarize the changes that caused the failure. We then feed this summary back to guide the next patch generation attempt, helping the LLM avoid making the same mistakes. We can iteratively repeat this process for each failed patch. In doing so, we notice that the summary must be informative yet concise to avoid stagnating patch generation. Lastly, we experimented with providing traces of failed testbenches, but instead of guiding the LLM to generate a good patch, we observed that the LLM tends to overfit, which results in proving the assertion but failure of other testbenches and assertions. In our experiments, STITCH achieved 61% successful patches after 5 iterations.

Our above outlined improvement provides an ablation study systematically capturing the impact of our design choices. Of the 4 LLMs, GPT5 outperforms in terms of patch success rate (56%) while GPT4 is least effective (6%). Claude performs the slowest but results in smaller patches (9 line changes); whereas GPT5 patches are larger in size (22 line changes). When synthesized on hardware, Claude patches tend to be more resource intensive, where as Gemini patches have lowest impact. We do not observe any correlation between patch sizes and hardware impact. Next, we evaluate STITCH with respect to ground truth. It has a success rate of 90% and while the code patch sizes are larger than ground truth, the hardware impact is comparable. Lastly, we compare STITCH with three non-LLM approaches by using a standard dataset [3]. STITCH achieves 92% success rate to not only out-compete all three tools individually but also collectively. Our experimental data is available [53].



DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804341

Table 1: (Left side) Buggy Code. (Right side) Patched code. red: exact patch

```
1  assign ARREADY = arready;
2  always @(posedge ACLK) begin
3    if (ARESETN == 1'b0)
4    begin
5      arready <= 1'b0;
6      araddr  <= 7'b0;
7    end
8    else begin
9      if (~arready && ARVALID)
10
11     begin
12       arready <= 1'b1;
13       araddr  <= ARADDR;
14     end
15     else
16       arready <= 1'b0;
17   end
18 end
```

```
1  assign ARREADY = arready;
2  always @(posedge ACLK) begin
3    if (ARESETN == 1'b0)
4    begin
5      arready <= 1'b0;
6      araddr  <= 7'b0;
7    end
8    else begin
9      if (~arready && ARVALID &&
10        (!RVALID|| RREADY))
11     begin
12       arready <= 1'b1;
13       araddr  <= ARADDR;
14     end
15     else
16       arready <= 1'b0;
17   end
18 end
```

```
assert property (@(posedge ACLK) disable iff (!ARESETN)
(~ARREADY && ARVALID && (RVALID && !RREADY)) |=> !ARREADY);
```

2 Challenges & Insights

Example. Consider the bug [65] reported by Ma et al. [37] in a widely used ARM AMBA AXI protocol implementation [14]. Table 1 shows a part of the read address latching in the read transaction. The specific intent of this logic is the ARREADY generation which is to assert it only when the subordinate is ready to accept a new read address. In the buggy version (left), ARREADY could still be raised while a previous read response (RVALID high and RREADY low) was pending, allowing overlapping transactions thus clearly violating the assertions shown at the bottom of Table 1. The patch adds a condition ensuring a new read address is accepted only when no response is outstanding, matching the assertion by preventing the subordinate from signaling readiness during an active transaction.

2.1 Challenges & Insights

Given a syntactically correct and synthesizeable implementation along with a violated assertion, our goal is to automatically synthesize a patch to satisfy the assertion. Below, we summarize the key insights gained while developing Stitch to achieve this objective.

Insight #1: Patch Validation. Assuming we generate a candidate patch, the first challenge is deciding whether it is actually good. Even syntactically correct patches may break functionality, overfit to specific cases, or introduce new violations. Testbenches, and the larger test suites built from them, provide useful coverage, but they do not guarantee correctness. A patch may satisfy the violation by altering unrelated logic rather than fixing the underlying issue. Thus, we combine test-suite compliance with formal verification over the full assertion set, allowing us to catch interactions where satisfying one assertion causes another to fail. This ensures that each patch preserves functionality and remains consistent with the specification. In summary, we consider a candidate patch to be good if it satisfies all of the following: (i) preserves syntactic correctness; (ii) proves the violated assertion; (iii) maintains the original functionality captured by a testsuite; and (iv) proves a strict superset of assertions compared to the original implementation.

Table 2: Patchable violations and good patches per protocol & implementation.

	AXI					AHB	APB			Wishbone	TileLink	Total
Impl.	AL	AG	AS	AP	AF	AH	IP	CH	MH	WB	TL	
# Patchable	12	19	12	15	16	6	4	1	3	5	7	100
# Patches	2	7	3	3	2	0	0	1	1	0	0	19

Insight #2: Impact of Counterexample. We curate a dataset of violated assertions from implementations of the 5 protocols (AXI, AHB, APB, Wishbone, and TileLink). Table 2 shows the 100 violations. Our first experiment was to give implementation and the violated assertions to the 4 LLMs (GPT4/5, Gemini, Claude), but none of the produced patches satisfied the good patch criteria. A closer analysis of failed cases showed that the LLMs lacked guidance; it did not have enough context of what went wrong. In cases such as our example in Table 1, the faulty read-address behavior becomes visible only when examining the precise temporal evolution of signals such as ARREADY, RREADY, and RVALID. Their high–low transitions reveal the violation, information that the assertion alone does not make explicit. While we could rely on mining a large database of hardware bugs and their corresponding patches to learn reusable patterns or templates, such datasets are scarce, making direct fine-tuning of LLMs impractical. Instead, we adopt a context-driven strategy in which the LLM is guided by concrete structural and behavioral information derived from verification outputs. As shown in Table 2, this guidance proved particularly effective: in 19/100 cases, counterexamples directly exposed the failing scenario, allowing the generated patch to correctly handle the missing case.

Insight #3: Accurate Patch Point Identification. Patch localization identifies the precise region of the implementation that must be modified to resolve a detected violation [27]. Because RTL implementations are typically complex, assertions alone rarely reveal the exact fault location. In our study of 100 violations, the 19 successful repairs all correctly localized the underlying bug. RTL code is particularly well suited for localization because its functionality is organized into defined structural blocks (`always` blocks). Assertions include the specific signals involved in the violated behavior, allowing the search space to often be narrowed to the blocks operating on those signals. In our example in Table 1, the signals appear in only two *always* blocks out of 732 LoC. While patch point identification is effective for simple properties, some cases (e.g., reset) may still require involved changes. To handle such cases, we rely on the cone of influence (COI), which captures both directly and indirectly related signals (e.g., through control or data dependencies). In our motivating example, the COI points to the `always` block and the corresponding branch shown in Table 1.

Insight #4: Iterative Context-driven Code Generation. Violated assertion, localized code region, and counterexample trace can repair simple violations, but they are not sufficient for complex cases. In these scenarios, we adopt an incremental repair loop: the LLM proposes an initial patch, the model checker evaluates it, and any new counterexample triggers a new repair iteration. By aggregating counterexample traces across iterations, the LLM progressively uncovers additional corner cases that must be addressed, enabling it to improve the patch until the assertion is proven.

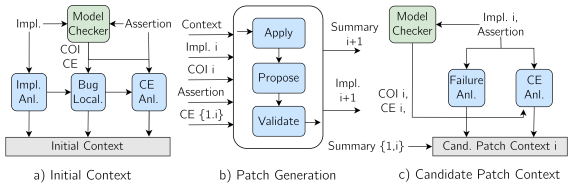


Figure 1: Stitch Overview. COI stands for Cone of Influence, CE for Counterexample Trace, and *i* for the iteration.

Insight #5: Context Summarization. LLMs benefit from extensive contextual guidance. But our experiments showed that propagating too much information (bug-localization outputs, multi-iteration counterexample analyses, and detailed patch summaries) led to stagnation as redundant reasoning accumulated and impeded progress. To address this, we carefully craft context that is carried to the next iteration. We produce compact yet verified reasoning that reduces hallucinations and only preserves the information necessary for the next step. As the only exception, we carry forward the counterexample traces as-is, due to their impact (Insight #2).

Insight #6: Testsuite Feedback. While investigating these failures, especially cases where patches were proven by the model checker but still failed the testsuite, we tried feeding the failing test traces back into the tool. This did not turn any failures into successful repairs: fixing the specific failing test broke other functionality and invalidated assertions that the original patch had satisfied.

3 Stitch Design

Figure 1 presents an overview of the Stitch workflow. Starting from a buggy implementation and its violated assertion, we invoke the model checker to obtain a counterexample trace and the cone of influence. We construct the initial context using this data, combined with the assertion and implementation (Figure 1a). Next, we produce a candidate patch in the repair stage (Figure 1b). We reverify the candidate patch with the model checker. If the assertion remains unproven, we use the candidate patch to build a candidate patch context (Figure 1c). We feedback this new context into the repair stage. This loop continues until the assertion is successfully proven.

Validation. Once the violated assertion is proven, we run the patched implementation through a test suite to assess its correctness, and verifying with the model checker that all previously proven assertions hold. The goal is to ensure that the new implementation not only proves the violating assertion but also a strict superset of assertions that were proven before applying the patch.

3.1 Initial Context

Stitch first analyzes the violated assertion using a model checker, which produces a counterexample (CE) trace and a COI in its formal verification report. We export the trace in VCD format rather than the proprietary WLF format and convert it to CSV, which is both more suitable for LLM processing and easier to interpret.

Bug Localization. Using the assertion, the implementation, and the COI, we ask the LLM to locate the code responsible for the violation. The prompt exploits the block structure of RTL (e.g., `always` blocks) and restricts attention to the blocks that drive the signals in the assertion. In our example from Table 1, this procedure correctly isolates lines 1–18 out of 732 lines of code. The LLM must justify its selection, explaining how the selected region relates to the assertion and the COI. This stage outputs selected line ranges, the corresponding code, and the associated reasoning.

Implementation & Counterexample Analysis. We ask the LLM for a structured description of the implementation. This helps in later reasoning to link the violated property and guide the LLM in its processing of the implementation. To analyze the counterexample trace, we prompt the LLM to reason about how the counterexample trace violates the assertion and to identify the specific signal behaviors involved, (e.g., which signal goes low when it is expected to remain high). We instruct the LLM to think about the erroneous behavior in the implementation based on the observed trace. The LLM reasons about the signal behavior in the counterexample with respect to the code regions previously identified as potential faulty locations, explaining how the existing logic permits the violation.

Aggregation. To aggregate the context of the above 3 analyses, we instruct the LLM to cross-verify and reconcile these three sources of reasoning. This step reduces hallucination and produces a single, coherent explanation of the violation. We prompt the LLM to self-validate the analyses aggregation and to redo it. Once approved, this aggregation represents our initial context.

3.2 Candidate Patch Generation

After establishing the context for patching, we provide the LLM with the inputs shown in Figure 1b: the latest context (initial or candidate), the implementation, the assertion, the COI, and CE(s). Using this, the LLM proposes a patch that satisfies several constraints: it must preserve all signal names and interfaces, use only synthesizable SystemVerilog code, and make the assertion provable while maintaining the design's functionality. In the iterative case, where the previous patch fails model checking, we state that the previous patch is incorrect and forbid superficial rewording or reformatting. To avoid hallucinations and interface or code inconsistencies, we supply both the localized code section and the full implementation. In practice, some LLMs generate a patch limited to the relevant section, while others output a complete corrected implementation; our pipeline accepts both forms. After the LLM proposes a patch, we instruct it to review its own proposed patch and to check if the violated assertion now holds. We ask it to summarize the changes and their intended effects. This enables the LLM to learn from prior attempts without being biased by unnecessary context.

3.3 Candidate Patch context

Given a candidate patch, if the assertion remains unproven, we construct a new context. We feed the counterexample, the COI, and the outputs from candidate-patch generation (Section 3.2) to the LLM, shifting the focus from the original implementation to the observed failure. Bug-localization information, though not provided explicitly, is already reflected in the patch summary. We therefore build the next iteration context by combining output of patch-generation with the model checker feedback, as described below.

- *Failure Analysis.* Concretely, we prompt the LLM to perform failure reasoning, in which it analyzes why the patch failed, identifies the specific logic that still permits the property violation, and compares the new counterexample trace against the patch to pinpoint the failure location.
- *Counterexample Analysis.* We use the model checker to generate a counterexample and COI for the candidate patch. We prompt the LLM to do counterexample trace analysis.
- *Aggregation.* We create a compact context by aggregating the output of above steps for the candidate patch with the patch summary of the fixes and repair reasoning.

Note that we avoid using all previous candidate patches, either explicitly or by including individual summaries. However, the information pertaining to previous failed patches is not all lost; the summaries and failure analyses maintain awareness of past decisions and help prevent the repetition of ineffective repair strategies.

4 Evaluation

We perform an ablation study on 100 violations to show the impact of our design choices in Stitch (Section 4.1). We show the hardware impact of good patches from different LLMs (Section 4.2). We compare Stitch to ground truth and non-LLM tools (Section 4.3). Our full evaluation data is available at [53].

Setup. We implement Stitch in Python using the LangChain library to interface with LLMs [30]. It accepts assertions in SystemVerilog Assertion (SVA) or Property Specification Language (PSL) format and the buggy implementation in Verilog or SystemVerilog as input. All Stitch outputs are produced as structured JSON objects, ensuring consistent formatting and lossless transfer of information between stages. As intermediate reasoning tends to be verbose, we carefully constrain and force each step's output to keep only essential details, enough to preserve traceability and avoid hallucination, while preventing context-window overflow. For debugging and analysis, we save intermediate artifacts per violation, including reasoning, patches, traces, and validation results.

Selecting Implementations. We select the 5 protocols with available VIPs (Table 3) and evaluate the 11 implementations using their protocol-specific cocotb [19] testsuites [4, 5, 13, 22, 58] . Each testsuite exercises core behaviors such as reset, read, and write transactions. We retain only implementations that pass at least 80% of tests, focusing on essential functionality (e.g., correct read-back) while ignoring less critical cases. Of the 15 implementations examined, 11 meet these criteria and form our evaluation set, we discard 4 implementations [35, 38, 50, 60].

Patchable Violations. We run the assertions from the protocol VIPs (Column 3 in Table 3) through Questa 2024 a Siemens model checker [51], and report the number of proven or violated assertions. We report 150 violations (Column 6). After looking into the assertions, 50 of them involve only input signals which make them non patchable. Indeed, such assertions are unprovable, as the implementation has no control over its inputs, which are treated as unconstrained and may take any value during verification. After filtering, 100 patchable violations remain to evaluate Stitch.

Table 3: Implementation, assertions, testsuites details along with number of uncovered violations, patchable cases, and successful Stitch patches.

Impl.	Source	VIP	Testsuite in %	#Ass.	#Viol.	#Patchab.	#Patched.
AL	[8]	[66]	100 [5]	29	16	12	8
AG	[41]	[66]	80[5]	29	23	19	15
AS	[54]	[66]	80 [5]	29	14	12	8
AP	[45]	[66]	90 [5]	29	19	15	9
AF	[7]	[66]	90 [5]	32	21	16	7
AH	[1]	[25]	80 [4]	26	18	6	2
IP	[18]	[10]	100 [22]	32	11	4	4
MH	[42]	[10]	80 [22]	32	8	3	3
CH	[20]	[10]	80 [22]	32	6	1	1
WB	[24]	[21]	85 [58]	17	7	5	2
TL	[12]	[34]	85 [13]	21	7	7	2
Total	–	–	–	**308**	**150**	**100**	**61**

Language Models. We started with 7 LLMs. Based on our counterexample experiment (Insight #2), we eliminated 3 LLMs (Qwen2.5 [49], Llama3.3 [39], and Deepseek-v3 [32]) as that yield 0 good patches. We retained 4 LLMs that yielded non-zero good patches: GPT-4o [43], Gemini 2.5 Pro[26], Claude Sonnet 4 [11], and GPT5 [44].

4.1 Generating Patches

We run Stitch on all 100 patchable violations listed in Table 3 using the 4 LLMs. We classify the produced patch into three categories: failure, partial success, and success. A patch fails when the assertion derived from the original implementation is not proven after Stitch completes its pipeline. If the patch proves the assertion, we run it through the testsuite. If it fails the same tests as the original, we mark it as partially successful and show its success rate. When the patched implementation passes the testsuite, we run it against the set of assertions. It must prove at least the same assertions as the original buggy version plus the target violation; otherwise, the patch is declared a failure.

Ablation Study #1. Impact of Counter-examples. As discussed in Section 2, for our first experiment we feed the buggy implementation, the assertion, and the counterexample trace to the LLM and ask it to emit the patched implementation. We validate each candidate patch per our good patch criteria. As shown in Figure 2(a), we produce 19/100 successful patches across the 4 LLMs, of which 15 are from GPT5. Most of these violations are straightforward to address such as data clearance, where the patch ensures that data is cleared once it is no longer valid.

Ablation Study #2. Impact of Initial Context. Figure 2(b) shows the results after generating the first round of candidate patches, following the process outlined in Figure 1. This illustrates the benefit of providing structured initial context and using our staged pipeline: out of 100 violations, the LLMs now repair 40. Most successes overlap with those in Figure 2(a), but an additional set of violations are repaired. Several newly repaired properties, such as AGRR, AGWW, and AGWX, require reasoning over multiple interacting signals, making bug-localization and counterexample analysis particularly valuable. We also observe more patches for stability-related violations in implementations AG and AS, whose logic structures are relatively simple. Interestingly, although WB is a small module where localization is straightforward, violation WBD1 was not fixed by any LLM in Figure 2(a); it is repaired here only after incorporating counterexample-guided and implementation-aware reasoning, which revealed the missing acknowledgment condition.

Ablation Study #3. Impact of More Iterations. Figure 2(c) shows the outcome after the third iteration, which incorporates three counterexamples and their accumulated fix summaries. Iteratively expanding the context improves the repair rate to 58%. While AG and AS stability violations were fixed earlier, more complex cases in AF and AP (e.g., APRA, AFWW) only become patchable here. The three MH violations are also first repaired at this iteration, as MHAT's fine-grained temporal requirement on error signaling requires multiple counterexamples. Some LLMs still fail to produce a functional patch for this assertion. Finally, Figure 2(d) presents the outcome after the fifth candidate-patch iteration, which represents the final performance of STITCH. We get a marginal gain of 3 additional violation repairs at the fifth iteration, which supports our decision to stop after 5 iterations. Among the remaining cases, ALRS stands out: it is an ordering property encoded with advanced temporal constructs such as `eventually`. Because ALRS specifies the entire read-transaction sequence, repairing it requires big restructuring of the implementation, making it one of the most challenging assertions to patch.

STITCH *Patch Size.* We report the number of modified lines across violations using the 4 LLMs. Figure 2 (left) shows the 61 violations that at least one LLM was able to patch. We observe that GPT5 consistently repairs a wider range of violations than the other LLMs, but the patch sizes are larger. In cases such as ALRP and ALWP, these extensive edits are justified: the original designs lacked any logic for the protection signals, and since these signals appear in the interface, STITCH must introduce complete handling, naturally resulting in bigger patches. For violations repaired by multiple LLMs, we observe clear differences in patching behavior. Claude is generally more conservative, producing smaller edits and succeeding mainly on cases where other LLMs also succeed and the required modifications are limited. In contrast, GPT5 and Gemini are able to repair violations whose fixes require more extensive code changes. When looking into the generated patches, we found that the LLMs often propose structurally different repairs, which explains the gap in line-change counts. For the AXI-based implementation AG, several violations are fixed by all LLMs, but because the implementation leaves considerable freedom in how to correct the behavior, the LLMs produce diverse patching strategies.

STITCH *Patch Generation Time.* GPT5 is the slowest, taking 4–32 minutes, about 1.35× slower than Gemini (2–24 min) and 2.42× slower than GPT4 (2.35–7.56 min), while Claude is slightly slower still (5–42 min) due to longer reasoning chains. As expected, later iterations increase latency: GPT5 ranges from 9.05–29.02 min, Gemini 3.95–20.45 min, GPT4 2.67–5.35 min, and Claude 11.23–20.63 min, reflecting the added counterexample analysis required for harder cases such as AGWA. We defer to [53] for detailed data.

Analysis of Unpatched Violations. After running STITCH on the 100 violations for 5 iterations across the 4 LLMs, we obtain 61 successful i.e., good patches. Among the remaining 39, STITCH produces 15 partial fixes. Stability violations are largely repaired, but AL-related patches remain non-functional: although they touch the correct region and make the assertion provable, they also alter unrelated logic or overfit to the assertion, causing testsuite failures. The remaining 24 violations fail entirely. Some assertions, such as those governing acknowledgment stability, fail across nearly all implementations, suggesting difficulty inherent to the assertion rather than the design, though the exact cause remains unclear. Other failed or partial cases stem from ordering properties whose repair would require major structural changes to the memory subsystem.

Table 4: FPGA resource (LUTs,FFs), logic levels, and power overheads (in %) relative to the original implementation. Missing patches shown as —.

	GPT5				Gemini				Claude			
Viol.	LUTs	FFs	LLs	P(W)	LUTs	FFs	LLs	P(W)	LUTs	FFs	LLs	P(W)
ALRO	6.5	1.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6	0.0	0.0
ALDC	34.8	0.0	0.0	0.0	2.2	0.0	-50.0	0.0	2.2	0.0	-50.0	0.0
AGRO	47.7	0.0	0.0	0.0	-21.6	0.0	0.0	0.0	2.1	2.4	0.0	0.0
AGRA	1.2	0.0	0.0	0.0	0.0	-41.5	0.0	0.0	—	—	—	—
AGWW	-11.6	6.1	100.0	0.0	—	—	—	—	15.1	2.4	100.0	0.0
AGR1	12.8	1.2	0.0	0.0	-12.4	0.0	100.0	0.0	—	—	—	—
AGR2	-1.2	1.2	100.0	0.0	-8.2	1.2	100.0	0.0	7.2	0.0	100.0	0.0
AGRS	12.8	0.0	100.0	0.0	3.5	0.0	0.0	0.0	—	—	—	—
AGW1	14.0	7.3	100.0	0.0	-3.1	8.5	0.0	0.0	0.0	1.2	100.0	0.0
AGDC	9.3	0.0	100.0	0.0	9.3	0.0	100.0	0.0	20.9	0.0	0.0	0.0
APA1	15.9	0.0	-50.0	50.0	15.9	-5.3	0.0	0.0	—	—	—	—
APDC	15.9	0.0	-50.0	50.0	15.9	-5.3	0.0	0.0	—	—	—	—
ASRD	-68.2	-74.2	100.0	0.0	-73.6	-68.4	0.0	-66.7	—	—	—	—
AFWO	4.5	0.0	-50.0	0.0	-0.9	0.0	50.0	0.0	—	—	—	—
IPSR	0.0	0.9	0.0	0.0	0.0	0.0	-100.0	0.0	—	—	—	—
MHAT	-2.9	0.0	-75.0	0.0	-2.9	-100.0	0.0	0.0	0.0	100.0	-75.0	0.0
MHSR	—	—	—	—	0.0	0.0	-25.0	0.0	26967.6	32800.0	75.0	55.0
CHER	0.0	-0.1	0.0	0.0	0.0	-0.1	0.0	0.0	0.0	-0.1	0.0	0.0
AHID	0.0	0.0	0.0	0.0	—	—	—	—	0.0	0.0	0.0	0.0
WBD1	3.7	0.0	50.0	14.3	-7.4	0.0	-50.0	0.0	—	—	—	—
WBEH	-14.8	0.0	-50.0	0.0	-22.2	0.0	-50.0	14.3	0.0	0.0	0.0	0.0
TLLE	-2.5	0.0	0.0	-2.9	—	—	—	—	-2.5	0.0	0.0	-2.9

Table 5: Comparison between ground truth and STITCH. "—": unavailable.

Impl.	Ground Truth					STITCH patch				
	GT Δ	LUTs	FFs	LLs	Power (W)	STITCH Δ	LUTs	FFs	LLs	Power (W)
S1.B	(+2/-2)	423	1073	1	0.004	(+12/-6)	422	1073	2	0.004
S1.R	(+1/-1)	423	1073	1	0.004	(+4/-3)	423	1073	3	0.005
S2	(+1/0)	12	19	1	0.001	(+13/-5)	14	46	3	0.001
S3	(+33/-13)	59	105	1	0.002	(+52/-27)	57	103	3	0.002
D8	(+3/-3)	78	61	1	0.002	(+13/-5)	—	—	—	—
D9	(+2/-2)	15	20	0	0.002	(+4/-4)	15	20	0	0.002
D11	(+7/-3)	31	27	0	0.002	(+36/-4)	33	27	1	0.002
D12	(+1/-1)	34	54	3	0.002	(+1/-1)	34	54	3	0.002
D13	(+3/-1)	17	18	3	0.001	(+3/0)	18	19	3	0.001

4.2 Hardware Impact of Good Patches

While line changes provide a rough indication of patch size, they do not reflect hardware behavior. Because resource usage is critical, we synthesize all patched implementations on a VCU118 FPGA [9] using Vivado [6] and measure LUT, FF, power, and logic-level (LL) differences. Table 4 reports hardware impact for violations where at least two of GPT5, Gemini, and Claude produced a successful patch; GPT4 is omitted due to low effectiveness (Figure 2). Power differences are generally negligible. A notable exception is MHSR (Claude), where a small (+5/−1) patch introduces logic clearing a 1024-wide array, causing large hardware overhead. This highlights that line-change counts alone cannot predict hardware cost and warrant synthesis-based evaluation. Aside from this case, LUT/FF/LL increases remain bounded (max 47.7%, 8.5%, 100%), with averages of 0.34%, −3.14%, and 12.7%. Claude tends to add resources, Gemini often reduces them, and GPT5 typically leaves utilization nearly unchanged (1–3%). Reductions also appear in ASRD, where FF and LUT usage decrease by 68.4% and 66.7% due to logic restructuring.

4.3 Comparison to Prior Work

Ground-truth. To compare against an established ground truth, we use the open-source testbed of reproducible hardware bugs from

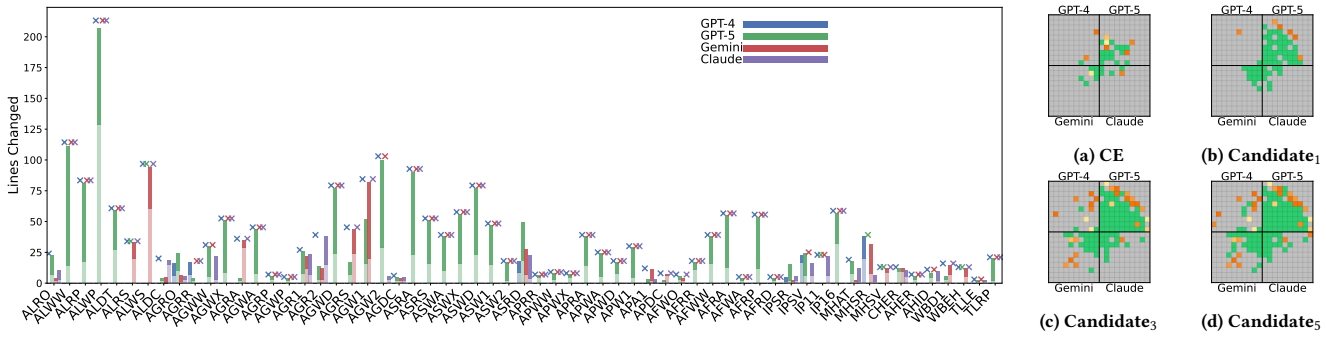


Figure 2: (Left) Line-change breakdown for all successfully patched violations; darker bars indicate insertions, lighter bars deletions, and × mark missing patches. (Right) Patch outcomes across ablations: grey = fail, orange = partial (gradient to dark orange = 0 passed testbenches), green = success.

Table 6: Line changes between the original and patched implementations (Δ Lines) across SoTA repair tools and Stitch. ○ marks a patch failure, × a failed testbench, and numeric line changes indicate a passing testbench.

Impl.	Δ Lines				Impl.	Δ Lines			
	CirFix	RTLRepair	SRepair	Stitch		CirFix	RTLRepair	SRepair	Stitch
decoder_w1	✗	2	35	2	shift_k1	1	✗	○	4
decoder_w2	○	✗	38	9	mux_w2	✗	2	11	11
counter_w1	1	○	○	8	mux_w1	✗	9	12	11
counter_k1	5	1	○	3	i2c_w1	1	○	○	5
counter_w2	1	2	○	13	i2c_w2	✗	○	○	1
fsm_s2	✗	15	○	○	i2c_k1	1	1	○	1
fsm_w2	✗	3	○	28	sha3_w1	1	○	○	○
fsm_s1	✗	2	○	7	sha3_s1	✗	1	○	1
flop_w1	1	0	○	1	reed_o1	✗	○	○	1
flop_w2	2	0	○	5	sdram_w2	○	2	135	1
shift_w1	✗	4	○	4	sdram_k2	○	2	140	19
shift_w2	1	0	○	6	sdram_w1	○	○	○	3

[37], focusing on the on-chip protocol cases in Table 5. For each of the 9 bugs, we derive an assertion from the diff and testbench traces between the buggy and patched versions, and supply this assertion with the buggy implementation to Stitch using GPT5. Cases where no reliable assertion could be constructed are excluded. Stitch patches 9/9 bugs thus passing corresponding testbenches. Table 5 shows line-change counts and hardware metrics. axis-switch-d8 fails synthesis despite passing the testbench. For the remaining 8 cases, although Stitch often modifies more lines, hardware impact remains comparable to or slightly better than ground truth.

Non-LLM Repair. We compare Stitch to SoTA non-LLM repair tools: RTL-Repair [29], SRepair [33], and CirFix [3]. We use the CirFix [3] testbed for comparison, as all three tools use it. We reproduce RTL-Repair [29] and SRepair [33], but report their published metrics when available and use SRepair's artifact to fill in missing LoC and coverage data. As in the ground-truth experiment, we craft assertions for each bug and supply them, with the buggy implementation, to Stitch. Stitch repairs 22/24 benchmarks, and all patches pass their testbenches; the remaining 2 cases fail to compile in the model checker, so we cannot evaluate them. Stitch changes more lines than CirFix or RTL-Repair (but fewer than SRepair, which may add hundreds of registers), yet for 10/22 benchmarks where at least one tool succeeds, it produces equal or fewer modifications. Manual inspection shows that in other cases, Stitch performs the same logical fix with different verbosity. In terms of correctness, CirFix fails 9 testbenches, RTL-Repair fails 2, SRepair fails several, whereas Stitch passes all tests for its patches. Finally, we aim to evaluate CirFix, RTL-Repair, SRepair on our dataset of 100 violations. Since these tools require hand-crafted templates and traces we limit the comparison to 5 violations (AGDC, ASRA, AGRO, IPSR, CHER) with crafted behavioral traces. None of the tools could patch 2/5 violations (IPSR or CHER). For the remaining 3/5 cases, the patches failed the testsuites due to unintended logic changes.

5 Related Work

Automated program repair is studied in software engineering. Recent SoKs [27, 31] outline three typical stages: fault localization, patch generation, and validation via tests or formal specifications. LLMs outperform classical techniques for software implementations [62]. Inspired by software engineering, Stitch targets hardware implementations which has its unique challenges.

There are three noteworthy non-LLM based for repairing RTL [3, 29, 33]. All of them use template-driven repair strategies: CirFix applies genetic programming over mutation templates evaluated with testbench feedback; SRepair uses symbolic regression to search expression templates; and RTL-Repair instantiates repair templates using SMT solving to synthesize locally correct patches. Stitch, an LLM-based approach, does not rely on hand-crafted templates, and outperforms them (Section 4.3).

LLMs can do RTL generation and debugging (e.g., syntax correction, compatibility with testing frameworks using retrieval augmentation) [40, 48, 55, 56, 59]. LLMs finetuned for hardware design, along with evolutionary or optimization-driven generation frameworks, improve RTL synthesis quality, via domain adaptation and specialized training. Concurrent assertion-driven repair works either require training data or causal graphs [17, 64]; bug-driven works neither use violated specification assertions nor enforce strong test-suite validation [2, 23, 36, 46, 47]; and syntax error and UVM-based works address orthogonal problems [28, 57, 63]. Stitch uses LLMs, but is the first work that patches assertion violations.

6 Conclusion

Stitch uses assertions to automatically generate patches for on-chip protocol implementations using LLMs. Across 100 real violations, our counterexample-guided and iterative design yields 61% good patches that preserving functionality and hardware efficiency.

7 Content Generated by AI

We used ChatGPT and Copilot for editing the text and Latex of this manuscript, authors inspected all outputs to ensure accuracy.

Acknowledgments

We thank reviewers and Daniel Moghimi for feedback. This work was partially funded by an unrestricted gift from Google's GARA.

References

[1] Adki. 2019. *AHB subordinate implementation with decoder.* https://github.com/adki/AMBA_AXI_AHB_APB/blob/master/codes/ahb_dma/bench/verilog/ahb_lite_s3.v
[2] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2024. On Hardware Security Bug Code Fixes by Prompting Large Language Models. *IEEE Transactions on Information Forensics and Security* (2024).
[3] Hammad Ahmad, Yu Huang, and Westley Weimer. 2022. CirFix: Automatically repairing defects in hardware design code. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.*
[4] Aignaccio. 2025. *AHB testsuite.* https://github.com/aignacio/cocotbext-ahb
[5] Alex Forencich. 2025. *AXI4 test.* https://github.com/alexforencich/cocotbext-axi
[6] AMD Xilinx. 2023. Vivado design suite. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html
[7] AMD Xilinx. 2025. *AXI-Full Xilinx secondary implementation.* https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/axi.html
[8] AMD Xilinx. 2025. *AXI-Lite secondary implementation.* https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/axi.html
[9] AMD Xilinx. 2025. *VCU118 evaluation kit.* https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/vcu118.html
[10] AMIQ. 2015. APB verification IP. https://github.com/amiq-consulting/amiq_apb/blob/master/sv/amiq_apb_if.sv
[11] Anthropic. 2025. *Introducing Claude.* https://www.anthropic.com/news/claude-4
[12] Antmicro. 2022. *TileLink.* https://github.com/antmicro/cocotb-tilelink/blob/main/tests/designs/ULSingleMasterDriver/TLUL_ram.sv
[13] Antmicro. 2022. *TileLink testsuite.* https://github.com/antmicro/cocotb-tilelink
[14] ARM 2019. *AMBA AXI Protocol Specifications.* ARM. ARM IHI 0022G.
[15] ARM 2021. *AMBA AHB Protocol Specification.* ARM. ARM IHI 0033.
[16] ARM 2023. *AMBA APB Protocol Specifications.* ARM. ARM IHI 0024E.
[17] Yunsheng Bai, Ghaith Bany Hamad, Chia-Tung Ho, Syed Suhaib, and Haoxing Ren. 2025. FVDebug: An LLM-Driven Debugging Assistant for Automated Root Cause Analysis of Formal Verification Failures. *arXiv* (2025). https://arxiv.org/abs/2510.15906
[18] Chipmunk Logic. 2025. *APB.* https://github.com/iammituraj/apb/blob/main/apb_slave.sv
[19] Cocotb. 2025. *Cocotb framework.* https://www.cocotb.org/
[20] courageheart. 2018. *APB.* https://github.com/courageheart/AMBA_APB_SRAM/blob/master/rtl/apb_v3_sram.v
[21] Dan Gisselquist. 2017. *Wishbone Verification IP.* https://zipcpu.com/zipcpu/2017/11/07/wb-formal.html
[22] Daxzio. 2025. *APB testsuite.* https://github.com/daxzio/cocotbext-apb
[23] Abdelrahman Elnaggar and Benjamin Tan. 2025. Adding Context to LLM-Guided Verilog Repair. In *ISQED '25.*
[24] Alex Forencich. 2020. *Wishbone.* https://github.com/alexforencich/verilog-wishbone/blob/master/rtl/wb_ram.v
[25] GodelMachine. 2025. *AHB Interface SystemVerilog Module.* https://github.com/GodelMachine/AHB2/blob/master/rtl/ahb_intf.sv
[26] Google DeepMind. 2025. *Gemini.* https://ai.google.dev/gemini-api/docs/models
[27] Yiwei Hu, Zhen Li, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. 2025. SoK: Automated Vulnerability Repair: Methods, Tools, and Assessments. In *USENIX Security '25.*
[28] Yuchen Hu, Junhao Ye, Ke Xu, Jialin Sun, Shiyue Zhang, Xinyao Jiao, Dingrong Pan, Jie Zhou, Ning Wang, Weiwei Shan, et al. 2024. UVLLM: An Automated Universal RTL Verification Framework Using LLMs. *arXiv* (2024). https://arxiv.org/abs/2411.16238
[29] Kevin Laeufer, Brandon Fajardo, Abhik Ahuja, Vighnesh Iyer, Borivoje Nikolić, and Koushik Sen. 2024. RTL-Repair: Fast Symbolic Repair of Hardware Design Code. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.*
[30] LangChain. 2025. *LangChain.* https://www.langchain.com
[31] Ying Li, Faysal Hossain Shezan, Bomin Wei, Gang Wang, and Yuan Tian. 2025. SoK: Towards Effective Automated Vulnerability Repair. In *USENIX Security '25.*
[32] A. Liu et al. 2024. DeepSeek-V3 Technical Report. *arXiv* (2024). https://arxiv.org/abs/2412.19437
[33] Zizhen Liu, Deheng Yang, Xiaoguang Mao, Jiayu He, Guangda Zhang, Yan Lei, and Jiang Wu. 2025. SRepair: Symbolic Regression-Based Repair for Hardware Design Code. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025).
[34] LowRISC. 2025. *TileLink verification IP.* https://github.com/lowRISC/opentitan/blob/master/hw/ip/tlul/rtl/tlul_assert.sv
[35] Lucky-wfw. 2020. *AHB implementation.* https://github.com/lucky-wfw/ARM_AMBA_Design/blob/main/AHB/ahb_slave.v
[36] Zizhang Luo, Fan Cui, Kexing Zhou, Runlin Guo, Mile Xia, Hongyuan Hou, and Yun Liang. 2025. R3A: Reliable RTL Repair Framework with Multi-Agent Fault Localization and Stochastic Tree-of-Thoughts Patch Generation. *arXiv* (2025). https://arxiv.org/abs/2511.20090
[37] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. 2022. Debugging in the Brave New World of Reconfigurable Hardware. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.*
[38] maomran. 2018. *APB implementation.* https://github.com/maomran/APB-Slave/blob/master/APB_Slave.v
[39] Meta AI. 2024. *The Future of AI: Built with Llama.* https://ai.meta.com/blog/future-of-ai-built-with-llama/
[40] Kyungjun Min, Seonghyeon Park, Hyeonwoo Park, Jinoh Cho, and Seokhyeong Kang. 2025. Improving LLM-Based Verilog Code Generation with Data Augmentation and RL. In *DATE '25.*
[41] Mmxsrup. 2019. *AXI-Lite secondary implementation.* https://github.com/mmxsrup/axi4-interface/blob/master/axi4-lite/axi_lite_slave.sv
[42] MohamedHussein27. 2024. *APB implementation.* https://github.com/MohamedHussein27/AMPA_APB4_Protocol/blob/main/RTL/APB_Slave.v
[43] OpenAI. 2024. *Hello GPT-4o.* https://openai.com/index/hello-gpt-4o/
[44] OpenAI. 2025. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/
[45] PULP. 2025. AXI-Lite Pulp secondary implementation. https://github.com/olofk/axi_node/blob/master/axi_regs_top.sv
[46] Khushboo Qayyum, Muhammad Hassan, Sallar Ahmadi-Pour, Chandan Kumar Jha, and Rolf Drechsler. 2024. From Bugs to Fixes: HDL Bug Identification and Patching Using LLMs and RAG. In *2024 IEEE LLM Aided Design Workshop (LAD).*
[47] Khushboo Qayyum, Chandan Kumar Jha, Sallar Ahmadi-Pour, Muhammad Hassan, and Rolf Drechsler. 2025. LLM-Assisted Bug Identification and Correction for Verilog HDL. *ACM Transactions on Design Automation of Electronic Systems* (2025).
[48] Haomin Qi, Yuyang Du, Lihao Zhang, Soung Chang Liew, Kexin Chen, and Yining Du. 2025. VeriRAG: A Retrieval-Augmented Framework for Automated RTL Testability Repair. *arXiv* (2025). https://arxiv.org/abs/2507.15664
[49] Qwen Team. 2024. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115* (2024). https://arxiv.org/abs/2412.15115
[50] Shawshank96. 2020. *AHB implementation.* https://github.com/shawshank96/AMBA-APB/blob/main/apb_slave.sv
[51] Siemens EDA. 2025. *Questa Property Checking.* https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/property-checking/
[52] SiFive. 2025. *TileLink Specifications.* https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf
[53] STITCH. 2026. *Evaluation data.* https://Xprotocols-lab.github.io/STITCH/
[54] Suoglu. 2023. *AXI-Lite secondary implementation.* https://gitlab.com/suoglu/axi-lite-slave/-/blob/main/Sources/ip_repo/axi_lite_slave_1.0/hdl/axi_lite_slave_v1_0.v
[55] Yan Tan, Xiangchen Meng, Zijun Jiang, and Yangdi Lyu. 2025. AutoVeriFix: Automatically Correcting Errors and Enhancing Functional Correctness in LLM-Generated Verilog Code. *arXiv* (2025). https://arxiv.org/abs/2509.08416
[56] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In *DATE.*
[57] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2024. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Model. In *DAC '24.*
[58] Wallento. 2024. *Wishbone testsuite.* https://github.com/wallento/cocotbext-wishbone
[59] Jing Wang, Shang Liu, Yao Lu, and Zhiyao Xie. 2025. HLSDebugger: Identification and Correction of Logic Bugs in HLS Code with LLM Solutions. *arXiv* (2025). https://arxiv.org/abs/2507.21485
[60] Wangjidwb123. 2021. *AHB implementation.* https://github.com/wangjidwb123/AHB-SRAMC/blob/main/rtl/ahb_slave_if.v
[61] Wishbone. 2025. *Wishbone System-on-Chip.* https://wishbone-interconnect.readthedocs.io/en/latest/
[62] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *ICSE '23.*
[63] Jiazheng Zhang, Cheng Liu, and Huawei Li. 2025. Understanding and Mitigating Errors of LLM-Generated RTL Code. *arXiv* (2025). https://arxiv.org/abs/2508.05266
[64] Jie Zhou, Youshu Ji, Ning Wang, Yuchen Hu, Xinyao Jiao, Bingkun Yao, Xinwei Fang, Shuai Zhao, Nan Guan, and Zhe Jiang. 2025. Insights from Rights and Wrongs: A Large Language Model for Solving Assertion Failures in RTL Design. *arXiv* (2025). https://arxiv.org/abs/2503.04057
[65] ZipCPU repo. 2025. axi lite bug read. https://github.com/ZipCPU/wb2axip/blob/master/bench/formal/xlnxdemo.v. Accessed: Nov. 17, 2025. [Online].
[66] Melisande Zonta-Roudes, Andres Meza, Nora Hinderling, Lucas Deutschmann, Francesco Restuccia, Ryan Kastner, and Shweta Shinde. 2024. eXpect: On the Security Implications of Violations in AXI Implementations. In *ICCAD '24.*