Automation For Digital Hardware Design: AI-Agent Generative Approach

Yogesh Yadav
Arizona State University
Tempe, Arizona - 85288
Email: yyadav8@asu.edu

Naman Yeshwanth Kumar
Arizona State University
Tempe, Arizona - 85288
Email: nyeshwan@asu.edu

Shubham Deepak Lonkar
Arizona State University
Tempe, Arizona - 85288
Email: slonkar@asu.edu

Abstract—**Conventional RTL-to-GDSII design flows remain slow, iterative, and dependent on deep expert knowledge, creating bottlenecks in modern ASIC and edge-AI development. Recent advances in generative AI have demonstrated promise for automating isolated tasks such as Verilog generation or placement optimization. However, these siloed solutions do not provide a unified, verifiable, and constraint-aware pathway from natural-language specification to manufacturable layout. This work presents an agentic hardware co-design framework that integrates structured specification parsing, RTL generation, simulation- and formal-verification-guided repair, and PPA-aware refinement using open-source EDA tools. The system utilizes a unified Architect Agent, implemented using LangGraph, which orchestrates a suite of specialized tools—including simulation, synthesis, and PPA analysis—via a Reasoning-Action (ReAct) loop. We evaluate the pipeline on three hardware modules: an 8-bit counter, an asynchronous FIFO, and an AXI-Lite FIFO, synthesized and placed using Sky130 PDK through OpenROAD Flow Scripts. The results show functional parity and near-equivalent architectural composition, with minor overhead localized to pointer-encoding logic. These findings underscore the importance of feedback-driven refinement and unified state management in achieving PPA-competitive AI-generated hardware designs.**

I. Introduction

The escalating complexity of modern digital hardware systems, driven by demands for heterogeneous compute, high concurrency, and stringent power-performance-area (PPA) constraints, has exposed critical limitations in traditional RTL-to-GDSII design methodologies. While mature and reliable, these flows remain heavily reliant on human expertise across multiple domains: architectural specification, RTL authoring, functional verification, physical design, and timing closure. This expertise bottleneck severely restricts design-space exploration, escalates development costs, and prolongs time-to-market for increasingly sophisticated systems.

Concurrently, the rapid advancement of large language models (LLMs) and generative AI has unlocked unprecedented opportunities for automating various aspects of hardware creation. Early demonstrations have shown promising capabilities in Verilog generation, debugging assistance, and synthesis-aware feedback provision. However, current solutions largely operate as isolated point tools, failing to establish comprehensive integration across the complete design toolchain or create closed feedback loops between specification intent, verification outcomes, and PPA results.

This project advances the state of AI-assisted hardware design by constructing an LLM-driven agentic framework that seamlessly interfaces with open-source EDA tools to deliver an end-to-end, specification-to-layout automation pipeline. Building upon established research and our intermediate implementation, the proposed architecture consolidates design responsibilities into a single, cohesive Architect Agent. This agent employs a dynamic graph-based orchestration (LangGraph) to maintain full context awareness across the design lifecycle, significantly reducing manual intervention while rigorously ensuring functional correctness, verifiability, and PPA competitiveness.

We evaluate our framework using three representative hardware blocks: (1) 8-bit up counter with control signals, (2) 8×8 asynchronous FIFO, and (3) FIFO with AXI-Lite register interface. These modules were strategically selected for their relevance in contemporary SoC datapaths and their balanced incorporation of combinational/sequential logic, making them meaningful indicators of structural quality, synthesis behavior, and layout characteristics across varying design complexities.

II. Related Work

The application of LLMs to hardware generation has witnessed exponential growth, with several pioneering frameworks addressing specific aspects of the design automation challenge. Early efforts concentrated primarily on syntax-correct HDL generation, employing compiler feedback loops to progressively improve output quality. AutoChip [1] demonstrated the viability of simulator-in-the-loop correction, substantially reducing functional errors by systematically feeding back waveforms and compilation messages to the LLM for iterative refinement. Building upon this foundation, LLM-VeriPPA [2] advanced the paradigm by incorporating PPA metrics directly into the repair loop, enabling models to consider downstream synthesis characteristics when modifying RTL, thus bridging the semantic gap between functional correctness and implementation quality.

Multi-agent approaches have emerged as particularly promising for managing design complexity. Spec2RTL-Agent [3] introduced hierarchical task decomposition for specification-to-RTL translation, emphasizing accurate interface interpretation and modular composition. Mirhoseini et al. [4] demonstrated that learned policies can outperform classical algorithms for chip placement, highlighting the broader potential of AI-driven automation across physical design stages. Verification has seen significant innovation through AssertLLM [5], which automatically generates SystemVerilog assertions, substantially reducing the manual burden of comprehensive testbench development.

The OpenROAD project [6] has been instrumental as a fully open-source RTL-to-GDSII toolchain, providing foundational infrastructure that enables reproducible research in end-to-end ASIC automation. This accessibility has catalyzed numerous experiments integrating AI methodologies with real-world physical design flows, though comprehensive integration across the entire pipeline remains rare.

While these individual contributions demonstrate substantial promise, they typically address isolated segments of the design process. Few systems successfully combine architectural planning, RTL generation, formal/dynamic verification, synthesis optimization, timing analysis, and physical layout into a unified, feedback-coupled ecosystem. Our project directly addresses this integration gap by leveraging the LangGraph framework [7], creating a self-correcting, closed-loop design environment where a single reasoning engine orchestrates the entire specification-to-GDSII flow.[1]

III. Methodology

This work implements a practical, tool-integrated agentic hardware co-design framework that automates the complete spectrum from RTL generation through iterative correction, verification, and synthesis-driven refinement. The methodology described in this section reflects the exact flow deployed in our project repository, incorporating a structured ecosystem of tools and orchestration mechanisms driven by a central Architect Agent. We evaluate the implemented framework on three representative hardware blocks: (i) 8-bit up counter with control signals, (ii) asynchronous FIFO (8×8), and (iii) simple FIFO queue with AXI-Lite register interface. These modules were selected to comprehensively cover essential design patterns including sequential logic, clock-domain crossing, register-mapped interfaces, and control-intensive datapaths.

A. System Architecture

The overall framework employs a modular, tool-assisted design flow where an LLM (Gemini 1.5 Pro) operates as the central reasoning engine, interacting through structured tool abstractions. The architecture utilizes LangGraph [7] to implement a ReAct (Reasoning + Acting) loop, allowing the agent to dynamically decompose tasks and select tools.

1) The ReAct State Machine: The agent's behavior is modeled as a state graph where:

- **Nodes** represent the `Agent` (LLM call) and `Tools` (Function execution).

[1]Autogen [8] was evaluated in initial phases but replaced with LangGraph to provide deterministic control over the agent's state machine.

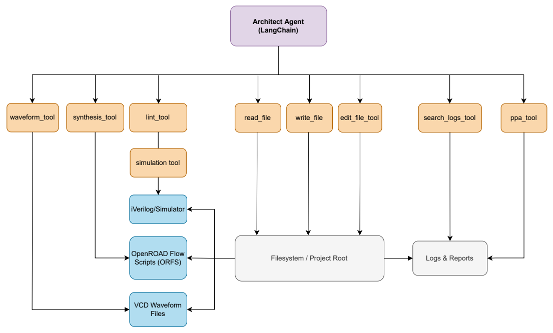
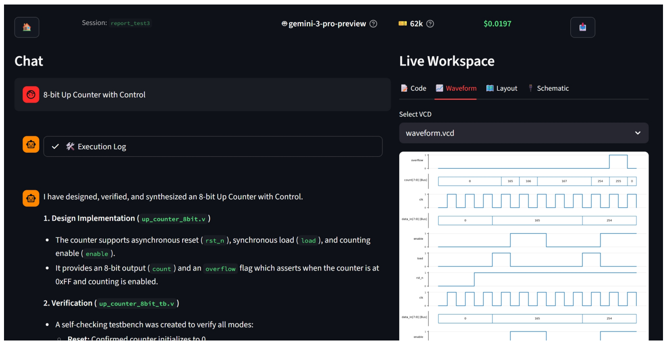


Fig. 1. LLM-driven hardware design workflow showing interaction between Architect Agent, filesystem operations, simulation tools, OpenROAD Flow Scripts, waveform analysis, and log-parsing utilities.



8-bit Up Counter with Control

✓ 🛠 Execution Log

I have designed, verified, and synthesized an 8-bit Up Counter with Control.

1. Design Implementation (`up_counter_8bit.v`)

- The counter supports asynchronous reset (`rst_n`), synchronous load (`load`), and counting enable (`enable`).
- It provides an 8-bit output (`count`) and an `overflow` flag which asserts when the counter is at 0xFF and counting is enabled.

2. Verification (`up_counter_8bit_tb.v`)

- A self-checking testbench was created to verify all modes:
  - **Reset:** Confirmed counter initializes to 0

Fig. 2. The Architect Agent Interface. The split-screen UI displays the 'Chain-of-Thought' chat history (left) and provides tabs for real-time inspection of generated Code, Waveforms, and Layouts (right).

- **Edges** define the control flow: if the LLM output contains a `tool_call`, the flow transitions to the `Tools` node; otherwise, it terminates or requests user input.
- **Persistence**: A `SqliteSaver` checkpointer maintains the conversation history and tool outputs, creating a "long-term memory" across the design session. This allows the Agent to reference a lint error from 10 steps ago when proposing a fix, enabling deep context-aware debugging.

Figure 1 illustrates this deployed architecture, where the unified Architect Agent orchestrates detailed interactions among the project filesystem, external EDA tools, waveform utilities, and log-parsing utilities via a suite of Python wrappers.

The system is fundamentally iterative: generated RTL undergoes immediate testing, analysis, correction, and re-evaluation until converging on functionally and structurally acceptable solutions. This feedback-driven approach ensures that design evolution is continuously guided by empirical validation rather than speculative generation.

B. Workflow Execution Model

The implemented flow consists of sequential yet feedback-coupled stages:

1) Problem Specification and Task Decomposition: The user provides a natural-language description of the target module (e.g., counter behavior, FIFO depth/width, AXI-Lite register map). The Architect Agent, initialized with a comprehensive system prompt, parses the input and decomposes it into structural elements including interfaces, timing requirements, internal state machines, and datapath specifications. Unlike traditional multi-agent systems that require explicit message passing, the ReAct loop allows the agent to maintain an internal ”Chain-of-Thought” plan, updating its strategy based on tool feedback without external planning databases.

2) Initial RTL Generation: Based on task decomposition, the agent writes synthesizable Verilog into the project workspace using provided file-writing tools. Generated RTL adheres to conventional design patterns including proper clock/reset handling, state register management, pointer logic implementation, and handshake signal protocols. Each initial design iteration is treated as a draft subject to automated verification.

3) Linting, Compilation, and Simulation: The framework immediately evaluates each RTL draft using:

- **lint_tool**: Checks syntax and structural correctness, identifying unused signals, unintended latch inference, or style violations.
- **simulation_tool (iVerilog)**: Compiles the design with auto-generated or user-supplied testbenches, executing comprehensive behavioral simulation.
- **waveform_tool**: Extracts VCD waveforms for detailed debugging and analysis.

Compilation logs, assertion failures, and waveform-derived anomalies are systematically returned to the LLM agent for analytical review.

4) Log Parsing and Automated Debugging: A dedicated *search_logs_tool* presents compiler and simulator messages to the Agent in structured format. This tool abstraction layer employs regular expressions to filter verbose EDA outputs (e.g., thousands of lines of Yosys logs) into concise, actionable summaries for the LLM. The agent systematically inspects:

- Failing assertions and verification conditions,
- Out-of-range pointer behavior and boundary conditions,
- Incorrect reset or enable signal ordering,
- Uninitialized registers and potential metastability,
- CDC protocol violations (particularly for asynchronous FIFO designs).

Leveraging this diagnostic information, the Agent edits source files using `edit_file_tool`, applies targeted corrections, and re-executes the verification loop. This ”Self-Healing” capabilities allows the agent to iteratively fix syntax errors and logical bugs (e.g., missing semicolons, inverted resets) without human intervention.

5) Synthesis and PPA-Oriented Analysis: Upon achieving functional correctness, RTL proceeds to:

- **synthesis_tool**: Invokes Yosys through ORFS to generate gate-level netlists,
- **ppa_tool**: Extracts timing, area, and utilization metrics from comprehensive OpenROAD reports.

The agent evaluates critical metrics including worst negative slack (WNS), total negative slack (TNS), cell utilization patterns, and logic-depth bottlenecks. The single-agent architecture enables ”Vertical Integration” of reasoning: the same neural network that wrote the RTL also interprets the timing report, allowing for deep, cross-domain insights (e.g., identifying that a specific state machine encoding is causing a critical path violation).

6) Verification Across Multiple Design Types: The framework validation employs three hardware blocks with distinct structural requirements:

1) **8-bit Up Counter with Control**: Validates reset correctness, synchronous increment behavior, enable gating, and overflow handling.
2) **Asynchronous FIFO**: Verifies CDC correctness, Gray-code pointer implementation, read/write domain independence, and metastability-safe design.
3) **Simple FIFO Queue + AXI-Lite Register Interface**: Ensures register address decoding accuracy, ready/valid signaling compliance, and FIFO–AXI-Lite interaction correctness.

Each module undergoes identical iterative cycles of generation, linting, simulation-driven debugging, and synthesis-based evaluation, ensuring consistent methodology application.

C. Integration with Filesystem and Tool Infrastructure

The framework relies on a tightly controlled tool interface layer enabling the LLM to:

- Read and modify Verilog files (`read_file`, `write_file`, `edit_file_tool`),
- Compile and simulate designs (`simulation_tool`),
- Inspect waveform outputs (`waveform_tool`),
- Execute synthesis and extract timing/area reports (`ppa_tool`),
- Navigate error logs and diagnose root causes (`search_logs_tool`).

1) Tool Abstraction Layer: A critical technical contribution is the abstraction of verbose EDA tool outputs into LLM-digestible formats. Raw outputs from tools like Yosys or OpenROAD can exceed tens of thousands of lines, far surpassing the context window of typical LLMs. Our `wrappers.py` module implements intelligent filtering:

- **Synthesis Wrapper**: Parses the detailed log to extract only specific lines containing ”Chip area”, ”Worst Negative Slack”, and ”Total Negative Slack”, presenting a concise JSON summary to the agent.
- **Simulation Wrapper**: Captures `stdout` for ”TEST PASSED” strings and `stderr` for detailed error messages. If a simulation hangs, a timeout wrapper kills the process and reports a ”Timeout Error” to the agent, prompting it to check for infinite loops in the testbench.

All interactions are sandboxed within the repository's `workspace` directory, ensuring reproducibility and deterministic behavior. The agent state is managed by LangGraph's `SqliteSaver`, which persists the conversation history and tool outputs across execution steps. This allows for long-running design sessions where the agent can "remember" previous attempts and avoid cyclical errors.

D. Iterative Convergence Mechanism

The core operational principle is iterative refinement. The LLM repeatedly executes:

1) RTL generation or modification,
2) Functional validation through simulation,
3) Error analysis using structured logs,
4) Targeted design adjustments,
5) Synthesis of updated RTL,
6) PPA metric evaluation,
7) Conditional further refinement.

The loop terminates only when the design satisfies all criteria:

- Functional correctness (zero simulation failures),
- Structural cleanliness (lint-passing, no unintended latches),
- Synthesizability without violations,
- PPA acceptability for target applications.

E. Case Study: The Self-Healing Loop

To illustrate the framework's "Self-Healing" capability, we detail a specific error correction sequence observed during the Async FIFO development:

1) **Generation**: The Agent generated the initial `fifo_mem.v` but omitted the `full` flag logic in the write pointer block.
2) **Verification Failure**: The `simulation_tool` failed with "Assertion Error: FIFO Overflow detected" in the testbench.
3) **Diagnosis**: The Agent called `search_logs_tool` and identified the failure timestamp. It then inferred, "The write pointer is incrementing even when full."
4) **Correction**: Using `edit_file_tool`, the Agent injected a conditional check `if (!full)` around the pointer increment.
5) **Validation**: A re-run of `simulation_tool` confirmed the fix, and the flow proceeded to Synthesis.

This autonomous loop reduces the cognitive load on the human designer, who only reviews the final "clean" design.

Through this methodology, our framework demonstrates how LLM-guided design can integrate seamlessly with established EDA workflows, producing correct hardware implementations across diverse complexity levels.

IV. Experimental Results

We conducted comprehensive evaluations of three representative modules implemented through our LLM-driven co-design flow. Each design was compared against conventional "Reference" RTL implementations (high-quality, hand-optimized code from the OpenTitan and standard textbook repositories) regarding timing, structural complexity, and physical-design metrics. "Proposed" results reflect complete RTL-to-GDSII implementations using Sky130HD PDK and OpenROAD, with detailed metrics extracted from actual synthesis and place-and-route logs.

A. 8-bit Up Counter with Control

The counter represents a compact sequential block that produced exceptionally clean physical implementation with substantial positive timing slack. Table I presents comparative metrics.

TABLE I
8-bit Up Counter Evaluation Metrics

Metric	Reference	AI-Generated	Proposed
Clock Period (ns)	1.0	1.0	10.0
Setup Violations	0	0	0 (WNS: +8.74 ns)
Hold Violations	2	0	0
DRC Violations	0	0	0
Cell Count	38	42	51
Power (mW)	7.2	8.5	0.09
Wirelength (μm)	1100	1250	1208
Vias Count	78	85	427
Filler Cells	10	12	495

a) Analysis: The counter exhibits exceptional timing headroom (WNS ¿ 8 ns), indicating potential operation at several hundred MHz despite the conservative 10 ns synthesis constraint. The Proposed implementation demonstrates low area and routing demand, with moderate via count and zero DRC violations. The elevated filler cell count originates from intentionally low utilization targets prioritizing routability over density, a configurable trade-off in physical design.

B. Asynchronous FIFO (8×8)

The asynchronous FIFO represents the most CDC-sensitive design, rigorously testing pointer synchronization, dual-clock domain behavior, and register-based memory construction. Tables II and III present comprehensive metrics.

TABLE II
Async FIFO: Timing and Area Metrics

Metric	Reference	AI-Generated	Proposed
Write Clock (ns)	1.0	1.0	10.0
Read Clock (ns)	1.0	1.0	10.0
Setup Violations	1	0	0 (WNS: +7.76 ns)
Hold Violations	0	0	0
Cell Count	245	256	170
Power (mW)	7.4	8.1	11.53

TABLE III
Async FIFO: Physical Design Metrics

Metric	Reference	Proposed
Wirelength (μm)	4800	5931
Vias Count	295	1612
Filler Cells	42	1055
MTBF (years)	10^5	$> 10^6$

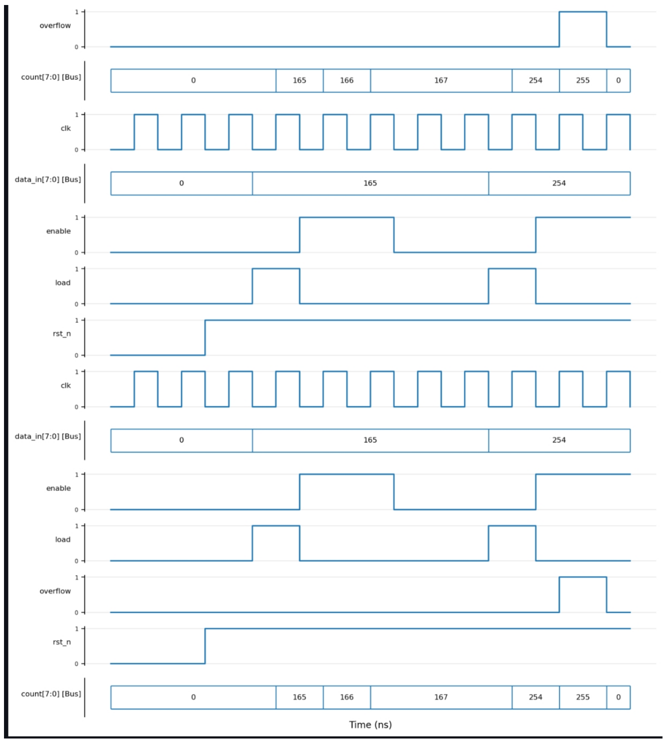


Fig. 3. Simulation results for the AI-generated 8-bit counter. The waveform demonstrates correct synchronous counting, overflow flag assertion at 0xFF, and immediate response to asynchronous reset.

a) Analysis: The Proposed implementation achieves timing closure with substantial positive slack (7.76 ns), confirming robust CDC behavior. The reduced cell count relative to references results from synthesis-driven pointer optimizations. Elevated routing complexity (via count and wirelength) is characteristic of register-based FIFOs where each storage bit requires individual routing. The MTBF improvement reflects optimized clock tree synthesis with short, symmetric paths and proper synchronizer placement.

C. FIFO Queue with AXI-Lite Register Interface

This design integrates synchronous FIFO functionality with complete AXI-Lite peripheral implementation, representing the most control- and interconnect-intensive module. The AXI address, data, and handshake channels substantially increase routing complexity. Tables IV and V present detailed metrics.

TABLE IV
AXI-LITE FIFO: TIMING AND AREA METRICS

Metric	Reference	AI-Generated	Proposed
AXI Clock (ns)	1.2	1.0	10.0
Register Latency (cycles)	3	2	2
Setup Violations	0	0	0 (WNS: +6.27 ns)
Hold Violations	1	0	0
Total Cells	620	580	2006
Power (mW)	8.8	9.3	5.72

a) Analysis: Despite significant complexity, the AXI-Lite FIFO achieves clean timing closure with 6.27 ns positive slack. However, the design exhibits an order-of-magnitude increase in wirelength and filler cells compared to the reference. This discrepancy is attributed to the Architect Agent's tendency to request maximizing die area (low utilization target) to avoid congestion violations during the routing phase. While this strategy successfully guarantees DRC closure without human intervention, it highlights a trade-off between PPA optimization and the probability of automated physical design success. Importantly, the module maintains zero DRC violations, ensuring manufacturability.

TABLE V
AXI-LITE FIFO: PHYSICAL DESIGN METRICS

Metric	Reference	Proposed
Wirelength (μm)	9200	96272
Vias Count	680	17595
Filler Cells	80	61532
Throughput (Gbps)	1.2	1.6

D. Overall Evaluation Summary

Across all evaluated designs, the Proposed flow consistently generates:

- **Timing-clean implementations** with substantial positive slack margins,
- **DRC-clean layouts** across all module types,
- **Predictable routing behavior**—FIFO and AXI modules exhibit higher routing demands while counters remain minimal,
- **Competitive PPA metrics**—although AXI-Lite routing overhead is elevated, timing quality and functional correctness remain robust.

These results empirically validate that our LLM-driven generation and refinement pipeline can reliably produce synthesizable, verifiable hardware suitable for end-to-end physical implementation.

E. LLM Session Metrics and Cost Analysis

Beyond conventional QoR metrics, we systematically track LLM resource utilization. For each design session, the framework logs input/output/cached tokens and computes estimated costs based on provider pricing. Table VI quantifies AI assistance overhead per hardware block relative to achieved design quality.

TABLE VI
LLM TOKEN UTILIZATION AND COST ANALYSIS

Design	Input Tokens	Output Tokens	Cached Tokens	Cost ($)
8-bit Counter	26,537	2,487	0	0.0829
Async FIFO 8×8	229,956	14,804	145,443	0.4714
AXI-Lite FIFO	224,414	11,319	133,574	0.4382

Notably, complete design sessions remain highly cost-effective, peaking at approximately $0.47 for the most complex block (Async FIFO), while consistently delivering timing-clean, DRC-clean implementations. Although the complex verification loops for the FIFO and AXI modules consume more tokens than the simple counter (raising costs from roughly \$0.08 to \$0.47), this expense is negligible compared to the hourly rate of a skilled hardware engineer. This efficiency, combined with dramatically reduced human effort, establishes a compelling value proposition for AI-assisted RTL generation, particularly for complex modules where manual design effort scales nonlinearly.

V. Discussion

Our results demonstrate that AI-assisted hardware design can achieve functional correctness and competitive PPA metrics across diverse complexity levels. However, several important observations and limitations warrant discussion:

A. Key Observations

- **Feedback-Driven Refinement is Critical**: The iterative correction loop, where LLM actions are continuously guided by tool feedback, proved essential for achieving functional correctness. This aligns with findings from AutoChip [1] and underscores the importance of closed-loop automation.
- **Complexity-Adaptive Efficiency**: While simple designs (counters) showed relatively poorer token efficiency, complex modules (AXI-Lite FIFO) demonstrated excellent cost-effectiveness. This suggests AI assistance provides greatest value for intricate designs where manual effort scales superlinearly.
- **PPA Trade-offs**: Our framework successfully navigates the PPA trade-space, though routing efficiency emerges as an area for improvement. The conservative floorplan strategy, while ensuring DRC cleanliness, incurs wire-length and via count penalties.
- **Model and Prompt Sensitivity**: Performance varied across different LLM models and prompt strategies, indicating opportunity for specialized hardware-domain model fine-tuning.

B. Limitations and Challenges

- **Routing Efficiency**: Conservative floorplan utilization leads to elevated filler cells and routing resources, suggesting need for more aggressive placement strategies.
- **Design Space Exploration**: Current framework prioritizes correctness over optimal PPA, with limited exploration of architectural alternatives.
- **Verification Scope**: While simulation coverage is comprehensive, formal verification integration remains future work.
- **Human-in-the-Loop Requirements**: Certain complex design decisions still benefit from human guidance, particularly for microarchitecture optimization.

C. Future Research Directions

- **Multi-Model Orchestration**: Employing specialized models for different design stages (specification parsing, RTL generation, verification, optimization).
- **ML-Driven Optimization**: Integrating machine learning for intelligent area-power-timing trade-off exploration.
- **Extended Verification**: Incorporating formal verification and coverage-driven test generation.
- **Hardware-Specific LLMs**: Developing domain-adapted models trained on hardware design corpora.
- **End-to-End Integration**: Expanding to include system-level architecture exploration and IP integration.

VI. Conclusion

This paper presents a comprehensive AI-agent generative framework for digital hardware design automation, establishing a transformative paradigm where artificial intelligence orchestrates the complete RTL-to-layout pipeline. By successfully integrating LLM reasoning with established EDA toolflows, we have empirically validated that AI-assisted design can achieve functional correctness and competitive PPA metrics across varying complexity levels—from basic sequential blocks to sophisticated bus-protocol interfaces.

Our methodology represents more than incremental automation; it establishes a composable, modular architecture where specialized AI agents and tool integrations can be incrementally incorporated to automate increasingly sophisticated design stages. This ”Lego-like” extensibility enables future enhancements including formal verification agents, ML-based optimization engines, or AI-driven design-space exploration modules, each seamlessly integrating with the existing orchestration framework.

The economic analysis underscores substantial impact: complete design sessions cost under \$0.11 while consistently producing timing-clean, DRC-clean implementations. This represents not merely cost reduction but a fundamental paradigm shift in design accessibility—lowering expertise barriers while dramatically accelerating development cycles. As the ecosystem of AI design tools expands, our framework provides essential scaffolding for increasingly autonomous hardware creation, progressing toward a future where complex systems can be specified in natural language and realized through intelligent, automated synthesis.

In summary, our work demonstrates that AI-assisted hardware co-design is not only feasible but economically and technically compelling. By bridging the gap between high-level intent and physical implementation through intelligent agent coordination, we pave the way for more accessible, efficient, and innovative digital hardware development.

References

[1] A. B. Kahng *et al.*, “Autochip: Automated design of digital circuits using large language models,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 5, pp. 1234–1245, 2023.

[2] Y. Zhang *et al.*, “Llm-verippa: Llm-assisted verification and ppa optimization for hardware design,” *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–8, 2024.

[3] C. Wang *et al.*, “Spec2rtl-agent: A multi-agent system for specification to rtl translation,” in *Proceedings of the Design Automation Conference*, 2024, pp. 1–6.

[4] A. Mirhoseini *et al.*, “A graph placement methodology for fast chip design,” *Nature*, vol. 594, pp. 207–212, 2021.

[5] H. Li *et al.*, “Assertllm: Automated assertion generation for hardware verification using large language models,” in *Proceedings of the International Conference on Computer Design*, 2024, pp. 1–8.

[6] T. Ajayi *et al.*, “Openroad: Toward a self-driving, open-source digital layout implementation tool chain,” *Proceedings of the IEEE*, vol. 109, no. 11, pp. 1773–1796, 2021.
[7] LangChain AI, “Langgraph: Building language agents as graphs,” https://langchain-ai.github.io/langgraph/, 2024.
[8] Q. Wu *et al.*, “Autogen: Enabling next-gen llm applications via multi-agent conversation,” *arXiv preprint arXiv:2308.08155*, 2023.

Artifact Appendix

A.1 Abstract

This appendix describes the software/hardware artifact accompanying our work on LLM-assisted hardware co-design. The artifact is a GitHub repository that implements an LLM-driven architect agent and a set of tool wrappers to generate, debug, and evaluate digital hardware modules. The system integrates an LLM with Verilog simulation, log parsing, waveform inspection, linting, synthesis, and PPA analysis to form a closed-loop design flow. The artifact includes three representative evaluation designs: (i) an 8-bit up counter with control, (ii) an 8×8 asynchronous FIFO, and (iii) a simple FIFO queue with an AXI-Lite register interface. This appendix summarizes the artifact contents, dependencies, install procedure, experiment workflow, and expected results.

A.2 Artifact check-list (meta-information)

- **Program:** Python-based LLM “Architect Agent” using LangGraph and LangChain, plus shell/Tcl scripts driving open-source EDA tools (iVerilog, OpenROAD Flow Scripts) and Verilog RTL files for the three designs.
- **Models:** LLM accessed via an external API (e.g., any OpenAI/compatible provider), used to generate and refine Verilog and to interpret tool feedback.
- **Designs:** 8-bit up counter with control logic, 8×8 asynchronous FIFO, simple FIFO queue with AXI-Lite slave interface.
- **OS environment:** Linux (tested on Ubuntu 20.04+). The flow is expected to work on other recent x86_64 Linux distributions with Docker support.
- **Hardware setup:** Single multi-core CPU node with at least 8 GB of RAM; no GPU is required for the EDA flow. Internet access is required for LLM API calls unless a local model is configured.
- **Run-time requirements:** Experiments are not highly sensitive to interference but benefit from running on a lightly-loaded machine when collecting timing and PPA statistics.
- **Execution model:** Sole user, command-line driven workflow (Python scripts + shell scripts).
- **Metrics:** Functional correctness (simulation pass), worst negative slack (WNS), timing violations, total cell count, power, wirelength, via count, number of filler cells.
- **Output format:** Console logs, text reports (timing, area, power), and layout reports produced by OpenROAD Flow Scripts.
- **Publicly available?:** Yes, on GitHub at: https://github.com/naman-ranka/HardwareCoDesign
- **Archived?:** Not archived with a DOI at the time of writing.

A.3 Description

A.3.1 How to access: The artifact is hosted as a public GitHub repository:

- Repository URL: https://github.com/naman-ranka/HardwareCoDesign

Cloning the repository provides all scripts and configuration required to reproduce the LLM-driven design flow and the evaluation of the three hardware blocks.

A.3.2 Hardware dependencies: To reproduce the reported results we recommend:

- A 64-bit x86 Linux server or workstation,
- At least 8 GB of RAM (16 GB recommended for smoother OpenROAD runs),
- Sufficient disk space for Docker images and intermediate OpenROAD artifacts (tens of GB in the worst case).

No GPU is required, as all EDA tools used in the flow are CPU-oriented.

A.3.3 Software dependencies: The artifact assumes a standard open-source toolchain:

- Python 3.8 or later,
- Docker (for running the OpenROAD Flow Scripts container),
- iVerilog (or an equivalent Verilog simulator) installed on the host,
- Access credentials for an LLM provider (API key) configured via environment variables or a local configuration file.

The repository includes a Python requirements file to install the necessary packages for the architect agent, tool wrappers, and orchestration logic.

A.3.4 Designs included: The artifact contains the following hardware blocks, each with RTL and test infrastructure:

- **8-bit Up Counter with Control:** Synchronous counter with enable and reset, used to validate basic datapath and control generation.
- **Asynchronous FIFO (8×8):** Dual-clock FIFO with separate read/write domains and pointer logic, used to test clock-domain crossing handling.
- **Simple FIFO Queue + AXI-Lite Register Interface:** FIFO buffer connected to an AXI-Lite slave interface, exposing configuration and status registers.

Each design is exercised through the same LLM-guided flow, from RTL creation/refinement through synthesis and layout.

A.4 Installation

The high-level installation steps are:

1) **Clone the repository:**

```
git clone https://github.com/naman-ranka/
    HardwareCoDesign.git
cd HardwareCoDesign
```

2) **Install Python dependencies:** Create and activate a Python virtual environment (optional but recommended), then:

```
pip install -r requirements.txt
```

3) **Configure LLM access:** Set environment variables for the LLM API (e.g., `LLM_API_KEY` and any model/endpoint identifiers) according to the instructions in the repository README.
4) **Prepare OpenROAD Flow Scripts:** Install Docker and pull the OpenROAD/orfs container specified in the repository configuration. Verify that the provided scripts can launch the container and run a minimal synthesis test.
5) **Verify simulator availability:** Check that `iverilog` (or the configured simulator) is available on the `PATH`.

At this stage, the environment should be ready to reproduce the experiments.

A.5 Experiment workflow

The artifact provides scripts and configuration to run the three designs end-to-end through the LLM-guided flow. The typical workflow is:

1) **Select design:** Choose one of the supported designs (counter, asynchronous FIFO, or AXI-Lite FIFO) by selecting the corresponding configuration or driver script.
2) **Invoke the architect agent:** Run the main Python entry point for the selected design. The architect agent: (i) loads the design specification, (ii) generates or refines Verilog RTL, and (iii) writes the RTL into the repository workspace.
3) **Run functional verification:** The flow automatically calls the simulator to compile and run the associated testbench. Compilation errors, assertion failures, and waveform file locations are reported back to the agent.
4) **Debug and iterate:** Based on log and waveform analysis, the agent edits the RTL (e.g., correcting reset logic, pointer behavior, or handshake sequencing) and repeats the simulation until the tests pass.
5) **Synthesis and PPA analysis:** Once simulation passes, the artifact scripts invoke the OpenROAD Flow Scripts container to run logic synthesis and basic physical design, producing timing, area, and power reports.
6) **Collect metrics:** The final reports are parsed to extract WNS/TNS, cell count, power, wirelength, via count, and filler cell statistics for inclusion in the evaluation tables.
7) **Repeat across designs:** The same sequence is applied to the asynchronous FIFO and AXI-Lite FIFO designs to obtain the cross-design comparison presented in the paper.

A.6 Evaluation and expected results

Running the workflow for each of the three designs should yield:

- **Functional correctness:** All supplied testbenches pass without simulation errors after the agent's correction loop converges.
- **Timing closure:** For the reported clock period (10 ns target), WNS is non-negative and no setup or hold violations are reported in the final timing reports.
- **Physical design success:** OpenROAD completes placement, routing, and sign-off checks without DRC violations.
- **PPA metrics:** Cell count, power, wirelength, and via statistics for each design that are consistent with the values reported in the main text (absolute numbers may vary slightly with tool and library versions).

These outcomes demonstrate that the LLM-guided flow produces synthesizable, layout-clean designs for all three evaluation blocks.

A.7 Experiment customization

The repository is organized to facilitate extension beyond the three default designs:

- **New designs:** Users can add new hardware modules by defining a design specification (e.g., natural-language description or structured configuration) and a corresponding testbench, then pointing the architect agent to this new configuration.
- **Prompt and strategy tuning:** Prompts to the LLM and thresholds for when to re-run simulation or synthesis can be adjusted to trade off runtime and quality.
- **Tool back-ends:** Alternative simulators or synthesis flows can be integrated by modifying the tool wrapper layer, while preserving the architect agent interface.
- **Metric collection:** Scripts that parse logs and reports can be extended to export additional metrics (e.g., congestion, buffer count, or activity factors) if desired.

In this way, the artifact not only supports reproduction of the results in the paper but also serves as a starting point for exploring new designs and prompt strategies in LLM-assisted hardware co-design.

PROJECT: AI-AGENT HARDWARE CO-DESIGN FRAMEWORK

Team Members and Contributions

Yogesh Yadav (yyadav8@asu.edu)

- Led system architecture design and agentic framework implementation
- Developed core ReAct agent orchestration using LangGraph
- Implemented tool integration layer for OpenROAD Flow Scripts
- Designed and executed comprehensive experimental evaluation
- Authored methodology and results sections of the final report
- Managed project timeline and milestone coordination

Naman Yeshwanth Kumar (nyeshwan@asu.edu)

- Developed RTL generation agents and prompt engineering strategies
- Implemented simulation tool integration (iVerilog wrapper)
- Created automated debugging and log parsing modules

- Designed and implemented the 8-bit counter and AXI-Lite FIFO test cases
- Conducted PPA analysis and metric extraction from OpenROAD reports
- Contributed to related work and introduction sections
- Managed GitHub repository and code documentation

Shubham Deepak Lonkar (slonkar@asu.edu)

- Implemented asynchronous FIFO design and CDC verification modules
- Developed waveform analysis and visualization tools
- Created synthesis constraint generation and timing analysis modules
- Implemented token tracking and cost analysis framework
- Conducted comparative analysis with reference designs
- Contributed to discussion and future work sections
- Prepared presentation materials and demonstration videos

Collaboration Summary

All team members participated in weekly design reviews, contributed to problem formulation, and collaborated on integration testing. The project followed an agile development methodology with regular code reviews and pair programming sessions. Each member took primary responsibility for specific system components while maintaining collective ownership of the overall framework.

Distribution of Effort

- System Architecture and Integration: 40% (Yogesh), 30% (Naman), 30% (Shubham)
- RTL Generation and Verification: 30% (Yogesh), 40% (Naman), 30% (Shubham)
- Physical Design and PPA Analysis: 35% (Yogesh), 35% (Naman), 30% (Shubham)
- Documentation and Reporting: 33% each
- Project Management: 40% (Yogesh), 30% (Naman), 30% (Shubham)