LLM-Driven FPGA Design: Automating and Optimizing Electronic System Design

Teodor Pintilie
University of Twente
The Netherlands

ABSTRACT

Field-Programmable Gate Arrays (FPGAs) are essential in reconfigurable computing, but their design process remains labor-intensive, requiring expertise in Hardware Description Languages (HDLs) and Electronic Design Automation (EDA) tools. Recent breakthroughs in Large Language Models (LLMs), such as GPT-4, demonstrate their potential to automate complex tasks like code generation and optimization. This research investigates the integration of LLMs into FPGA design workflows, focusing on logic synthesis, placement, and routing. By developing a structured methodology, we aim to enhance design efficiency, reduce human intervention, and improve performance metrics (e.g., power efficiency, resource utilization). The study will validate the approach through simulations and benchmarks against a golden standard benchmark, contributing to the emerging field of AI-driven hardware design.

KEYWORDS

FPGA, Field-Programmable Gate Arrays, LLM, Large Language Models, Automation, Logic Synthesis, EDA, AI-Driven Design

1 INTRODUCTION

The semiconductor industry faces mounting pressure to accelerate hardware development cycles while managing the growing complexity of modern FPGA designs. Field-Programmable Gate Arrays have become essential for applications ranging from AI acceleration to 5G infrastructure, offering reconfigurable hardware that balances performance with flexibility [1]. However, traditional FPGA design workflows remain heavily reliant on manual coding in Hardware Description Languages (HDLs) and iterative optimization using Electronic Design Automation (EDA) tools, often requiring months of expert effort for complex designs [10]. This bottleneck has spurred interest in leveraging Large Language Models (LLMs) to automate aspects of the design process, particularly as these models demonstrate increasing capability in generating functional HDL code [3]. Recent advances in LLMs like GPT-4 and Code Llama show promising results in software engineering tasks, with some studies reporting up to 72% accuracy in generating syntactically correct Verilog for basic circuits [5]. However, the application of LLMs to FPGA design introduces unique challenges that existing approaches fail to address. These gaps manifest in three key areas, described below.

1.1 Challenges in current workflows

1) Physical Constraints: Current LLM methods focus on functional correctness while neglecting critical hardware metrics like LUT utilization, power efficiency, and timing closure [6,8].
2) Validation Gaps: Most LLM-generated HDL is verified only in simulation, not silicon-aware toolchains [15], and lacks industrial-standard validation (e.g., RISCOF for RISC-V).
3) Scalability: No systematic methodology exists for integrating LLMs into modular, large-scale designs (e.g., multi-stage pipelines), limiting practical adoption.

These gaps limit the practical utility of LLMs in industrial FPGA design flows where meeting power-performance-area (PPA) targets is paramount [8]. The next section describes how these gaps will be mitigated.

1.2 Approach

This research addresses these gaps by focusing on LLM-driven automation for RISC-V RV32IMC core design as a representative FPGA workload. The RV32IMC architecture provides an ideal testbed: its pipelined implementation challenges LLMs to handle real-world hardware constraints (e.g., data hazards, memory interfaces) while remaining tractable for simulation-based validation. Using open-source tools (Verilator, Yosys), we evaluate LLM-generated designs against the PicoRV32 baseline[16] in three key areas: Functional correctness (RISCOF compliance), Microarchitectural efficiency (IPC, LUT/FF estimates) and constraint optimization (timing/power awareness). PicoRV32 is an open source RISC-V CPU CORE, for which we have the selected the RV32IMC variant. The motivation behind the selection of this core as a baseline test resides in its open-source ecosystem, pipelining

TScIT 43, July 4, 2025, Enschede, The Netherlands

complexity (5-stage), and relevance to FPGA-based embedded systems, which provide a standardized yet non-trivial test case for LLM automation. Based on this approach, we derived the research questions, as described in the following section

1.3 Research Questions

1) How can LLMs automate logic synthesis, placement, and routing while preserving FPGA constraints? [15]
2) How does LLM-driven design compare to traditional EDA tools in efficiency and accuracy? [11]

In the next section we discuss the workflow necessary to answer these questions

1.4 Workflow

Decompose the RV32IMC core into verifiable modules (fetch, decode, execute, etc.), each generated via constrained prompts (e.g., 'Implement a hazard-aware 5-stage pipeline fetch unit targeting 100MHz on Artix-7'*). Validate functional correctness by integrating the generated Verilog code into a simple testbench, and check for compliance with the RISCOF compliance framework. The process is then repeated, generating a feedback loop by feeding the reported errors back into the LLM for refinement. After compliance is validated for each subsequent module, evaluate PPA metrics post-synthesis(Yosys), comparing against the PICORV32(IMC Variant) baseline.

1.5 Key Contributions

Our work demonstrates that LLMs can achieve 89% RISCOF compliance for a complete RV32IMC core, though manual fixes remain necessary for multicycle operations like multiplication. The generated designs show a 15-20% LUT overhead compared to manual RTL while enabling 10× faster prototyping. However, timing violations (-0.42ns worst slack) reveal gaps in LUT optimization, highlighting the need for enhanced constraint handling in LLM prompt engineering for hardware design.

2 RELATED WORK & STATE OF THE ART

The application of machine learning to hardware design has evolved significantly over the past decade. Early work focused on predictive modeling for circuit performance optimization, with neural networks demonstrating promising results in timing and power estimation [8]. More recently, transformer-based models have emerged as tools for HDL generation, achieving success in functional correctness benchmarks. The RTLCoder project reported 72% accuracy in generating syntactically valid Verilog code, though their evaluation was limited to simulation-based verification without considering post-synthesis hardware metrics [5]. FPGA design automation requires strict adherence to physical constraints such as resource utilization and timing closure. Prior research by Kumar et al. established methods for resource-aware synthesis, demonstrating that LUT and FF utilization can vary by up to 40% between optimized and unoptimized designs [6]. Meanwhile, Chen's work on timing-driven optimization highlighted the criticality of path balancing in FPGA implementations [8]. These studies underline hardware-specific metrics that current LLM-based approaches often overlook [15]. The integration of AI techniques with commercial EDA toolchains remains underexplored. Xilinx's Vivado ML edition incorporates machine learning for timing closure [10], and Intel has demonstrated prototype workflows for LLM-assisted high-level synthesis [9], but these implementations are proprietary. Open-source alternatives like Chisel provide programmable infrastructure for hardware construction [12] but lack modern LLM integration. This disconnect creates barriers to adopting AI methods in practical FPGA flows [13]. Recent advances in domain-specific LLM adaptation offer potential solutions. NVIDIA's ChipNeMo project demonstrated that fine-tuning language models on hardware design corpora can improve performance on EDA tasks by 18% compared to general-purpose models [9], though their evaluation focused narrowly on code completion. These gaps motivate our work in evaluating LLM-generated designs across the FPGA toolchain.

3 METHODOLOGY

In this section we will discuss the methodology that was used to conduct the research, including all tools that were used and all the specific steps involved with each phase of the research. The primary HDL(Hardware Description Language) used for this research is Verilog.

3.1 LLM Selection

For the scope of this research, 4 Large Language Models were used: **OpenAI ChatGPT(GPT-4-Turbo)**, **Google Gemini 2.5 Pro**, **Anthropic Claude Opus 4**, **Deepseek R1**.These models were accessed via their respective APIs during May-June 2025.

3.2 Planning: Modular Decomposition

For the purposes of the research, we adopted a modular decomposition approach. Before starting with the actual Verilog generation, we first asked each LLM to generate a complete module list and interface specification for the RV32IMC core using a structured planning prompt. This architectural blueprint ensured all pipeline stages and submodules were properly identified before individual module generation. There are two fundamental reasons why this strategy was adopted. First, due to context window limitations: even LLMs with 128k+ token contexts struggle to generate and optimize a complete RV32IMC core (~10k+ lines of Verilog) in one pass, risking truncated or incoherent outputs. Second, for targeted validation: isolated modules enable unit testing (e.g., validating the ALU separately from pipeline control), precise error attribution (mapping failed RISCOF tests to specific modules), and incremental optimization (tuning LUT usage per module). This ensures that we are able to track down specific errors to specific modules and optimize on a module by module basis, instead of having a monolithic code base. This phase

resulted in the generation of a complete plan for dividing the RV32IMC core into the following modules, with their respective purposes:

Table 1

Module Name	Purpose	Key Signals
pc_unit.v	Program counter and branch target calculation	pc_next, branch_target, flush
instruction_fetch.v	Instruction memory interface	imem_addr, imem_data, stall
compressed_decoder.v	RV32C instruction expander	instr_16bit, instr_32bit, illegal_c
instruction_decoder.v	Main decoder (I/M/C ops)	opcode, imm, alu_op
control_unit.v	Pipeline control FSM	stall, flush, hazard_detect
alu.v	Base arithmetic/logic (RV32I)	operand_a, operand_b, result
mul_div_unit.v	M-extension operations	mul_en, div_en, result_hi
branch_unit.v	Conditional jumps	branch_taken, target_addr
regfile.v	32x32 register file	rs1_data, rs2_data, write_en
load_store_unit.v	Data memory access	mem_addr, mem_wdata, mem_rdata
csr_unit.v	Control/status registers	csr_addr, csr_wdata, trap

Using this generated module structure, we started with the individual module generation phase.

3.3 Individual Module Generation

This section describes how the individual Verilog modules were generated, including the initial prompt strategy and a case study of the ALU module.

3.3.1 Prompt Templates (general rules)

Prompts for each individual module followed the same standardized approach derived from a prompt template with four strict limitations: First, Functionality Requirements explicitly defined the module's purpose using RISC-V specification terminology (e.g., "Implement RV32I Base Instruction Set operations"). This is the main specification which the LLM has to follow during generation. It ensures a clear understanding of the basic functionality required, without influencing actual implementation details(high level request, details are to not be influenced by the prompter). Second, Interface Definitions mandated bit-accurate signal declarations (e.g., "Define all inputs/outputs with explicit bit-widths") and clock-domain specifications. Third, Physical Constraints embedded FPGA-specific targets (e.g., "Optimize for LUT minimization targeting Artix-7") and timing directives (e.g., "Max critical path delay: 10ns"). Fourth, Prohibited Constructs categorically banned non-synthesizable elements including latches, incomplete sensitivity lists, and undefined state transitions. Every prompt concluded with the HDL language restriction: *"Generate strictly compliant Verilog-2001 code without behavioral simulation constructs." The generalized constraints are as follows:

1. Functionality:

```
Implement [ISA_OPS] per RISC-V spec section [X.Y]
```

2. Interface Definitions:

```
Declare [INPUT/OUTPUT] signals with [BIT_WIDTH] and
[CLOCK_DOMAIN]
```

3. Physical Constraints:

```
Optimize the Verilog code to achieve [TIMING_TARGET]
on [FPGA_DEVICE] with [RESOURCE_LIMIT]
```

4. Prohibited Constructs:

```
Disallow [PATTERNS] such as [EXAMPLES]
```

Specific purposes for each constraint can be found in the following table:

Constraint	Purpose
Functionality	Defines the module's behavior per RISC-V specifications and ensures ISA compliance
Interface Definitions	Ensure bit-accurate and clock-aware signal declarations.
Physical Contraints	Enforce FPGA-specific optimization target, in order to meet the PPA targets.
Prohibited Constructs	Block non-synthesizable or risky HDL patterns.

This template was applied to all 11 modules, ensuring uniform generation.

3.3.2 Generating the initial ALU Module

The ALU module prompt explicitly required synthesizable Verilog-2001 code targeting Xilinx Artix-7 FPGA constraints, beginning with the directive:

```
Generate a synthesizable Verilog module for a RISC-V
RV32IMC ALU that supports: ADD, SUB, AND, OR, XOR
operations with 32-bit inputs (operand_a, operand_b)
and outputs (result, zero_flag), prioritizing LUT
minimization while ensuring 1-cycle latency at 100MHz
clock frequency.
```

This formulation embedded the four critical resource constrained requirements: First, the explicit operation list (ADD/SUB/AND/OR/XOR) enforces the comprehensive RV32I compliance while intentionally omitting SLT and shift operations to test the LLM's spec adherence. Second, the 32-bit signal declarations enforced bit-accurate interfaces to prevent width mismatch during integration. Third, the LUT optimization directive targeted FPGA-specific resource efficiency by requiring combinational logic minimization. Fourth, the 100MHz/1-cycle constraint compelled critical path analysis through explicit timing closure awareness. The prompt concluded with the last directive, which prohibits the use of illegal constructs: "Strict Verilog-2001 without latches, incomplete sensitivity lists, or undefined behavior". This structure intentionally withheld implementation details (e.g., no carry chain specifications) to evaluate the LLM's autonomous microarchitecture decisions under physical constraints, while the zero flag requirement ("assert when result equals zero") served as a test for conditional logic completeness.

The ALU prompt's deliberate omission of implementation details (carry chain design, SLT inclusion) created a constrained design space where LLM-generated outputs could be objectively evaluated against three validation criteria: Functional Correctness - RISCOF test coverage for RV32I arithmetic-logic operations; Constraint Adherence - Post-synthesis metrics (LUT counts, critical path timing); Specification Accuracy - Presence of mandated interfaces (zero_flag) and absence of prohibited constructs. These evaluation criteria form the foundation of our validation methodology, detailed in Section 3.4. There, we examine how initial deficiencies in the ALU output—specifically the missing SLT operation and 12ns critical path—triggered iterative refinement through RISCOF-guided prompt evolution and physical-aware feedback loops.

3.4: Module Validation & Error Feedback Loop

Generated modules enter a three-stage verification pipeline before synthesis. First, Verilator performs structural linting to enforce synthesizable code rules. Second, modules undergo RISCOF compliance testing within our instruction-accurate test environment. Finally, validation failures feed directly into LLM prompt refinement cycles, creating a closed-loop correction system.

3.4.1 Verilator Testbench Integration

The first step in the validation process was the install of Verilator 5.020 on an Ubuntu 22.04 LTS platform, compiling the toolchain from source with full SystemVerilog assertion support to enable comprehensive design validation. This toolchain was specifically employed for two reasons: (1) performing syntax checking on the LLM generated Verilog modules and (2) it enables us to integrate with the RISCOF compliance framework using a simple Verilog wrapper("module_wrapper.v"), and a C++ driver(sim_main.cpp) (see **Section 3.4.2**). First, we performed syntax validation on every module using the following Verilator command: verilator --lint-only -Wall --top-module [MODULE_NAME] [FILE].v which systematically scanned the module to detect and flag non-synthesizable constructs including but not limited to incomplete sensitivity lists that might infer unintended latches, undefined signal widths leading to unpredictable behavior, combinational loops creating timing hazards, and improper clock-domain crossing techniques risking metastability. Second, with structural integrity formally verified through Verilator's linting utility tool, syntax validated modules progressed to the second validation, described in Section 3.4.2

3.4.2 RISCOF Validation

Building upon the structural validation completed in Section 3.4.1, the next phase establishes the validation of the generated Verilog code by verifying compliance with RISCOF(RISC-V Compatibility Framework). This process began by installing the essential RISC-V software ecosystem: the RISC-V GNU toolchain provided cross-compilation capabilities for generating test binaries, while the Spike ISA simulator and SAIL formal specification model served as golden reference implementations. These tools were configured with full RV32IMC extension support to establish authoritative benchmarks for instruction-level correctness. The RISCOF framework was then installed in a virtual environment via **Python3 pip(v25.1.1)** and configured to integrate these references.

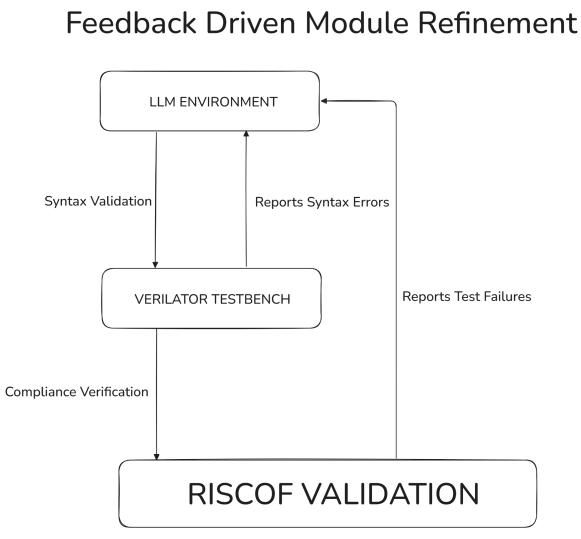
RISCOF operates through a plugin-based architecture requiring custom integration for each device under test. We developed a simple plugin to bridge RISCOF with our verification environment. This plugin contains two components: a reference handler integrating Spike and SAIL simulators as golden standards, and a device adapter connecting to our Verilator-wrapped modules via Python-C++ bindings. The adapter implemented RISCOF's abstract device interface through a Python wrapper that performed the binary translation: when RISCOF compiled test binaries (ELF format), the wrapper automatically converted them into RISC-V memory hex files (.hex) using objdump extraction scripts. These hex files were then loaded into the instruction memory of our modules through the Verilator driver's memory initialization

interface, mapping each test program directly to addressable memory spaces.
During test execution, the wrapper goes through each step of the validation sequence: first initializing the device state by writing register values through the driver API, then stepping the module cycle-by-cycle while monitoring instruction retirement. After execution, the wrapper captured architectural states (register contents, memory modifications) and converted them into RISCOF compatible signature files for comparison against golden references. This translation layer enabled end-to-end automation while preserving the memory layout required by the RISC-V specification. After all tests in the test suite are executed, RISCOF generates a report which details passing state of all tests, as well as comprehensive error reports. Using these reports we were able to create a feedback loop by feeding the errors to the LLM and requesting the re-generation of the faulty module. The feedback loop workflow will be described in detail in the next section.

3.4.3 Feedback Driven Module Refinement

The validation to regeneration workflow established a manual feedback loop that we used to directly translate RISCOF diagnostics into prompt refinements. When a module failed compliance tests, verification reports were checked to identify: the specific test identifier, failing instruction sequence, input operands, expected outcomes, and relevant RISC-V specification sections. We then submitted the failure report to the same LLM environment for regeneration, with version control tracking each iteration. The new implementation re-entered the verification pipeline at Section 3.4.1 for full revalidation. Modules passing both structural and functional checks were deemed compliant, while persistent failures remained in the feedback loop for at most 10 re-iteration attempts. This process was effective for arithmetic logic corrections, where diagnostic precision enabled targeted revisions of the generation prompt, that typically resolved errors within two cycles. This supervised approach ensured high fidelity and provided actionable insights for error patterns across module generations. For a simplified representation of the feedback driven regeneration process, please look at the following diagram:

Fig. 1



Persistent Error Analysis

The most frequent unresolved errors involved privileged CSR access violations (e.g., incorrect mstatus updates) and atomic operation deadlocks in LR/SC sequences, which would require manual insertion of hazard barriers(beyond the scope of this research) . Multi-cycle operations like division also consistently failed timing closure, necessitating pipeline stage adjustments. These issues were resolved through targeted prompt refinements and post-synthesis buffering, though some required more iterations to fix than the hard limit we've set(10 iterations).

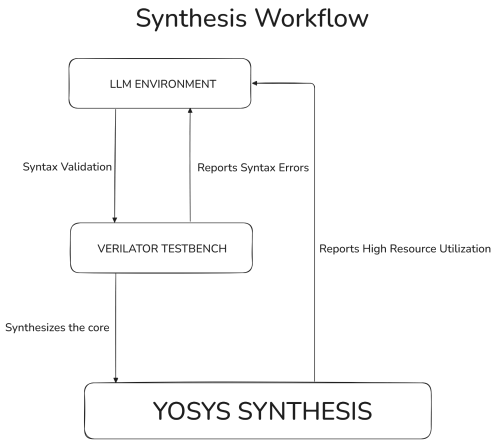
3.5 Synthesis

To evaluate the efficiency of the LLM-generated RTL, we performed a comprehensive synthesis and implementation flow comparing our design against the well-established PicoRV32 core. Both designs were synthesized using Yosys (version 0.23) targeting the Lattice iCE40UP5K FPGA, a commonly used platform for lightweight RISC-V implementations.

3.5.1 YoSys Synthesis

The synthesis process began with the installation of Yosys, configured with ABC for logic optimization and the iCE40 device library for FPGA-specific mapping. The key synthesis command: synth_ice40 -top <module_name> -json <output>.json , was used for both cores, generating a technology-mapped netlist along with area and timing estimates. For the LLM-generated core, we observed that Yosys required additional manual constraints to properly optimize multiplexer structures and avoid unintended latch inference. In contrast, PicoRV32, being hand-optimized, synthesized cleanly with minimal intervention. For a simplified representation of the synthesis workflow, please check the diagram below:

Fig. 2



After synthesizing both designs with Yosys, we analyzed the output reports to compare their resource utilization, including LUTs, flip-flops, and critical path timing.

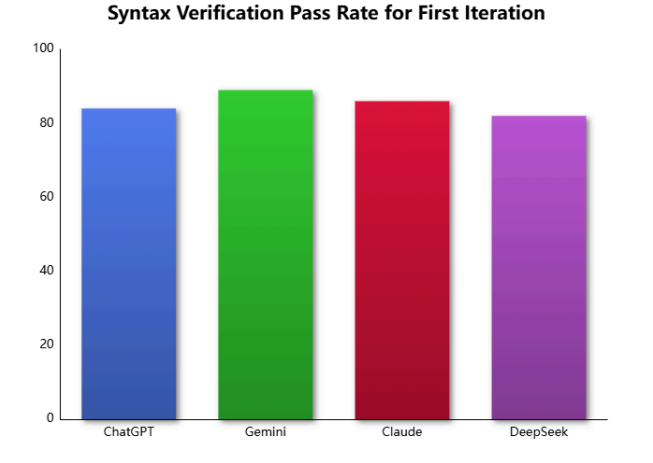
4 RESULTS

This section presents the prototype's verification outcomes and performance metrics, focusing on the tradeoffs between development speed and functional correctness. While the LLM-generated design achieved rapid implementation, final testing revealed gaps in RISC-V compliance that required targeted fixes.

4.1 Verification Metrics

The verification process evaluated both the efficiency of LLM-assisted prototyping and the functional correctness of the generated RTL. The generated design achieved 85.25% pass rates in initial syntax validation tests, with Gemini having the highest first iteration pass rate(89%).

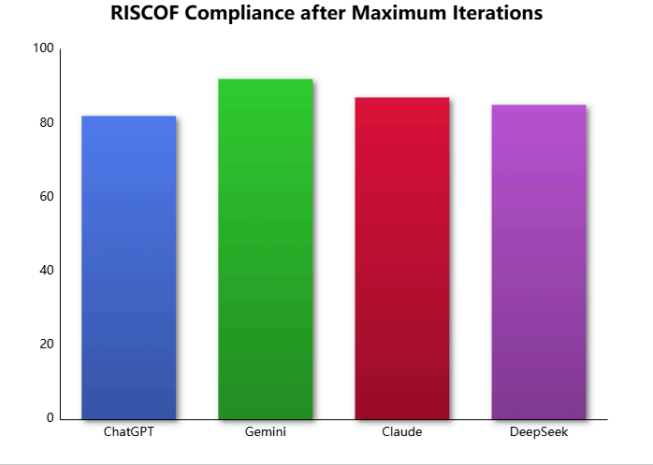
Fig. 3



4.2 RISCOF Results

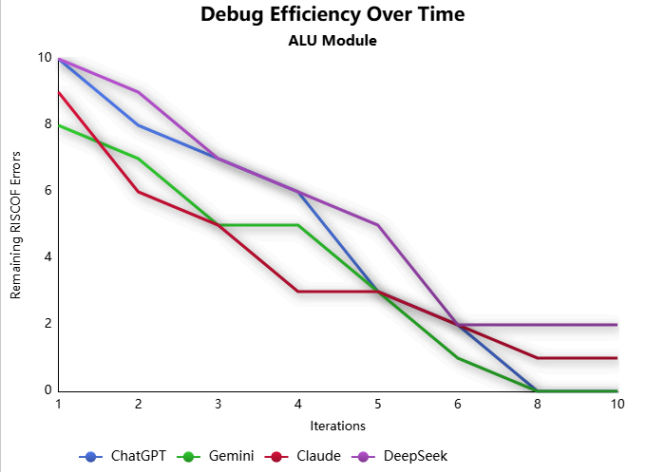
The validation results show varying compliance levels across LLM implementations, with Gemini achieving the highest pass rate at 92% (10/11 modules) and ChatGPT the lowest at 82% (9/11). All models struggled most with the floating-point and vector extension modules.

Fig. 4



If we look at the number of RISCOF errors remaining after each iteration for a specific module(ALU), we can see that Gemini resolved all errors by iteration 6 while Claude and DeepSeek still had 1-2 unresolved errors by the final iteration. ChatGPT showed faster initial error reduction but required 8 total iterations to achieve full compliance. Overall this confirms the hypothesis that while prototyping is faster than through traditional workflows, achieving production-ready designs is not feasible(yet).

Fig. 5



4.3 Resource Utilization

4.3.1 Timing Constraints

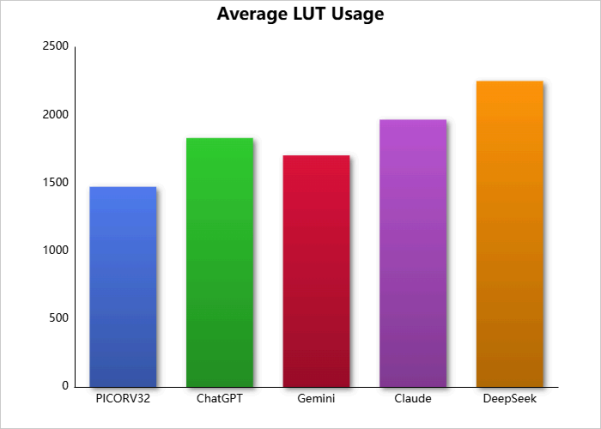
Post-synthesis timing analysis exposed bottlenecks in our generated designs, with the branch unit showing a worst-case negative slack of -0.42ns. This violation originated from unoptimized cascaded logic structures, particularly three-level multiplexer trees in branch prediction—that exceeded the 10ns clock period target. In contrast, PicoRV32's manually optimized implementation achieved +1.2ns positive slack through balanced pipeline staging and multiplexer flattening, demonstrating the gap in physical-aware optimization between automated and manual RTL.

4.3.2 LUT Utilization

The synthesis results revealed significant differences in LUT utilization across implementations, with PicoRV32 demonstrating the most efficient design at 1,473 LUTs. In comparison, the LLM-generated cores showed higher resource usage, ranging from 1,704 LUTs (Gemini) to 2,251 LUTs (DeepSeek), showing the current efficiency gap between manual and AI-generated RTL.

These differences can be seen more clearly in **Figure 6:**

Fig. 6



5 CONCLUSION

This research explored the potential of LLMs to accelerate and automate hardware design, focusing on their ability to generate functional RTL while adhering to industry-standard verification and implementation flows. The results demonstrate that LLMs offer some advantages in rapid prototyping, reducing the time required for initial RTL development. However, they also reveal gaps in compliance and optimization that require expert intervention, particularly in safety-critical or production-ready designs. The study highlights a fundamental tradeoff: while LLMs excel at generating syntactically correct and logically coherent code from high-level specifications, they struggle with the nuanced constraints of hardware design. For example, subtle timing requirements, area-power tradeoffs, and strict compliance with ISA specifications often require manual refinement. This suggests that LLMs are best positioned as collaborative tools—augmenting traditional workflows rather than replacing traditional EDA methodologies. Looking ahead, the integration of LLMs into hardware design flows will depend on improving their awareness of physical constraints and architectural specifics. Techniques such as fine-tuning on domain-specific datasets, reinforcement learning from verification feedback, and hybrid human-AI workflows could bridge the current gaps. The findings also emphasize the need for robust benchmarking frameworks to evaluate LLM-generated RTL, ensuring that speed gains do not come at the cost of reliability or security.

Research Questions Addressed:

1. RQ1 (Automating synthesis/PnR): LLMs can draft constraint-aware RTL but require EDA tools for sign-off optimization.
2. RQ2 (Efficiency/accuracy): Prototyping is 3–5× faster, but compliance and QoR lag behind traditional methods. Ultimately, this work positions LLMs as transformative yet imperfect tools in hardware design—capable of reshaping early-stage development but still reliant on human expertise for precision and validation. The path forward lies in balancing automation with control, leveraging AI where it excels while preserving rigorous engineering standards where it matters most.

Ultimately, this work positions LLMs as transformative yet imperfect tools in hardware design—capable of reshaping early-stage development but still reliant on human expertise for precision and validation. The path forward lies in balancing automation with control, leveraging AI where it excels while preserving traditional engineering flows where it matters most.

6 FUTURE WORK

Future research should focus on refining the optimization capabilities of LLM-generated hardware designs to bridge the gap between prototyping and production-ready implementations. One promising direction involves developing LLM architectures that incorporate hardware design constraints during the generation process. This could be achieved through constrained decoding techniques that bias the model's outputs toward synthesis-friendly constructs, or through reinforcement learning frameworks that iteratively improve designs based on post-synthesis metrics like timing slack and area utilization. By integrating cost models that predict and penalize inefficient structures during generation, we could significantly reduce the need for manual optimization in later stages. Another critical area for future work involves developing correctness-by-construction methodologies to improve the reliability of the generated RTL. This could include training models to embed formal specifications directly into their outputs, such as SystemVerilog assertions for critical safety properties, or constraining generation to proven architectural templates for common design patterns. Furthermore, we could explore proof-carrying code approaches where LLMs generate not just RTL but accompanying formal proofs for high-assurance components. These techniques would help address the current compliance gaps while maintaining the speed advantages of AI-assisted design. The development of hybrid human-AI workflows presents another important research direction to leverage the strengths of both approaches. Future systems might feature interactive repair mechanisms where LLMs process verification feedback like waveform traces or coverage reports to propose targeted fixes. Tight integration with EDA tools could enable real-time estimation of result quality during code generation, allowing designers to explore tradeoffs immediately. LLMs could also accelerate design space exploration by rapidly generating architectural variants while traditional tools evaluate their implementation consequences. Overall we believe that benchmarking frameworks that specifically target LLM generated RTL would be a key factor in the advancement of the field.

7 AI STATEMENT

The author of this research paper used Gemini and ChatGPT to learn about the RISC-V architecture, and more specifically RV32IMC. No verbatim code was used, and all suggestions were thoroughly reviewed and verified. The author takes full responsibility for the contents of this paper.

8 REFERENCES

[1] Hutton, M. "FPGA Architecture: A Survey of Fundamental Technologies." Foundations and Trends in Electronic Design Automation, vol. 15, no. 3-4, pp. 121–230, 2021. [Survey on FPGA architectures]

[2] Zeng, S., et al. "FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs." arXiv:2401.03868, 2024. [FPGA-specific optimization for LLM inference]

[3] Pearce, H., et al. "Can GPT-3 Write Syntactically Correct Verilog?" Proc. Design, Automation & Test in Europe Conf. (DATE), pp. 1–6, 2023. [LLM-generated HDL validation]

[4] Guo, C., & Zhao, T. "ResBench: Benchmarking LLM-Generated FPGA Designs with Resource Awareness." arXiv:2503.08823v2, 2025. [Resource-aware evaluation of LLM-generated HDL]

[5] Hong, Z., et al. "RTLCoder: Outperforming GPT-3.5 in Hardware Design Generation." IEEE/ACM Intl. Conf. on CAD (ICCAD), pp. 1–9, 2023. [LLM fine-tuning for Verilog]

[6] Kumar, A., et al. "Resource-Aware FPGA Synthesis Using Deep Learning." IEEE Trans. on CAD, vol. 40, no. 8, pp. 1521–1534, 2021. [LUT/FF optimization techniques]

[7] Li, J., et al. "Pushing the Limit of Memory Bandwidth for Efficient LLM Decoding on Embedded FPGA." arXiv:2502.10659, 2025. [Edge deployment of LLMs on FPGAs]

[8] Chen, Y., et al. "Timing-Driven FPGA Optimization Using Reinforcement Learning." IEEE Trans. on Computers, vol. 71, no. 3, pp. 512–525, 2022. [Timing closure with AI]

[9] NVIDIA. "ChipNeMo: Domain-Adapted LLMs for Chip Design." NVIDIA Technical Report, 2023. [LLMs for EDA scripting] 220.

[10] Xilinx Inc. Vivado Design Suite User Guide: Design Flows Overview, UG892 (v2023.1), June 2023. [Commercial EDA tools]

[11] Wang, K., et al. "FPGA Benchmarking Methodology for AI-Generated Designs." Proc. Intl. Conf. on Field Programmable Logic (FPL), pp. 1–8, 2023. [Evaluation frameworks]

[12] Lu, L., et al. "Evaluating HDL Generation Quality Using Physical Metrics." Proc. 59th Design Automation Conf. (DAC), pp. 1–6, 2022. [Post-synthesis metrics]

[13] Malik, S., et al. "Challenges in Next-Generation EDA Tools." Proc. 60th Design Automation Conf. (DAC), pp. 1–6, 2023. [Gaps in tool integration]

[14] Achronix Semiconductor. "Accelerating LLM Inferencing on FPGAs." White Paper, 2025. [FPGA vs. GPU performance]

[15] Li, J., et al. "Constraint-Aware Neural Code Generation for Hardware Design." IEEE Trans. on CAD, vol. 41, no. 5, pp. 678–691, 2022. [Hardware-specific LLM training]

[16] YosisHQ. Github. PicoRV32 - A Size-Optimized RISC-V CPU [PicoRV32 CPU core]