LLM-Based Verilog Adder Generation and Verification Project

A Two-Part Project *for Hardware Design with Large Language Models*

Document Version: October 17, 2025

Table of Contents

1. Project Overview

This project is designed to explore the capabilities and limitations of Large Language Models (LLMs) in hardware design, specifically in generating and verifying Verilog code for digital adder circuits. The project consists of two interconnected parts that demonstrate the complete workflow from design generation to verification.

Project Objectives

- Understand how LLMs can interpret/generate H/W description language (HDL) code
- Practice reverse engineering Verilog designs into natural language specifications
- Evaluate the accuracy of LLM-generated Verilog code against golden reference designs
- Learn to leverage LLMs for automated testbench generation
- Gain hands-on experience with hardware simulation tools (Iverilog)
- Develop critical analysis skills for comparing synthesizable Verilog architectures

Learning Outcomes

- Ability to analyze and describe digital circuit architectures in natural language
- Understanding of different adder architectures and their trade-offs
- Proficiency in using LLM tools (ChipChat/AutoChip) for hardware design
- Skills in manual code review and verification of HDL designs
- Experience with testbench development and internal signal verification
- Competence in using Iverilog for RTL simulation and debugging

2. Prerequisites and Tools

Required Knowledge

- Basic understanding of digital logic design
- Familiarity with Verilog HDL syntax and semantics
- Knowledge of adder circuits (full adder, half adder, carry propagation)
- Understanding of combinational logic design
- Basic command-line skills for running simulation tools

Required Tools

- Access to LLM tools: ChipChat or AutoChip (for Verilog generation)
- LLM-aided testbench generation tool (enhanced version with internal signal checking)
- Iverilog (Icarus Verilog) - open-source Verilog simulation tool
- Optional: Waveform viewer (GTKWave) for signal analysis

Repository Access

The golden Verilog designs are available in the Verilog-Adders repository. Clone the repository using:

```
git clone https://github.com/FCHXWH823/Verilog-Adders.git
```

3. Available Adder Types

Verilog-Adders repository contains implementations of six different adder architectures. Each adder type has unique characteristics in terms of speed, area, and complexity.

Carry Ripple Adder (RCA)

The simplest adder architecture where carry bits ripple through each stage. Simple to implement but slower for large bit-widths. Files: RCA4.v, RCA8.v

Carry Lookahead Adder (CLA)

Improves speed by pre-computing carry bits using generate and propagate logic. Faster than RCA but requires more logic gates. Files: CLA4.v, CLA8.v

Carry Select Adder (CSA)

Uses parallel computation with two RCAs (assuming carry=0 and carry=1), then selects result. Good balance between speed and area. Files: CSA4.v, CSA8.v

Carry Skip Adder (CSkipA)

Allows carry to skip over blocks when all propagate signals are active. Moderate improvement over RCA with reasonable complexity. Files: CSkipA4.v, CSkipA8.v

Kogge-Stone Adder (KSA)

Advanced parallel prefix adder with logarithmic depth. Fastest architecture but highest area cost. Files: KSA4.v, KSA8.v

Hybrid Adder (HA)

Combines different adder architectures for optimal performance. Complex but efficient for specific use cases. Files: HA8.v

Selection Guidance

For this project, you must select TWO DIFFERENT adder types. Consider choosing:

- One simple architecture (RCA) and one advanced architecture (KSA or CLA) to understand the complexity spectrum
- Two similar architectures (CLA and CSA) to compare different optimization strategies
- Architectures with 8-bit

4. Part 1: LLM-Based Verilog Generation from Golden Designs

In this part, you will use LLMs to understand existing Verilog designs by first converting them to natural language descriptions, then regenerating the Verilog code to verify the LLM's understanding and generation capabilities.

4.1. Step 1: Select Two Adders

Tasks:

- Review the available adder types in Section 3
- Select TWO different adder architectures from the repository
- Ensure you understand the architectural differences between your chosen adders
- Document your selection with justification

Example Selection:

Adder 1: Carry Ripple Adder (RCA8.v) - chosen for its simplicity and educational value
Adder 2: Carry Lookahead Adder (CLA8.v) - chosen to understand optimized carry computation

4.2. Step 2: Generate Natural Language Descriptions

Tasks:

- For each selected adder, read and analyze the golden Verilog code
- Feed the complete Verilog code to your LLM tool
- Prompt the LLM to generate a comprehensive natural language description
- The description should include: architecture overview, module hierarchy, signal flow, key logic operations
- Review and refine the generated description for accuracy and completeness
- Save both descriptions for the next step

Suggested LLM Prompt:

"Analyze the following Verilog code and provide a detailed natural language description of the design. Include: (1) Overall architecture and purpose, (2) Module hierarchy and interfaces, (3) Signal flow and datapath, (4) Key logic operations, (5) Any special design features. Be specific about how the circuit implements its functionality."

4.3. Step 3: Generate Verilog Code from Descriptions

Tasks

- Use ChipChat or AutoChip LLM Verilog generation tool
- Feed the natural language description from Step 2 to the tool
- Instruct the tool to generate Verilog code with the SAME architecture as described
- Ensure the generated code maintains module names, port names, and hierarchical structure
- Save the LLM-generated Verilog code for each adder
- Generate code for both selected adders

Suggested Generation Prompt:
"Based on the following description, generate Verilog code that implements this exact architecture. Maintain *the same module hierarchy, signal names, and design approach described. Use structural Verilog with gate-level primitives where specified."*

4.4. Step 4: Manual Verification

Tasks

- Compare the LLM-generated Verilog code with the golden reference design
- Check for architectural consistency (same module hierarchy, signal flow)
- Verify port declarations (names, widths, directions)
- Compare logic implementation (gate-level vs behavioral, internal structure)
- Identify any differences in implementation details
- Document findings in a comparison report

Verification Checklist:

- Module names match
- Port names and widths match
- Internal signal declarations match
- Module hierarchy is preserved
- Logic implementation is equivalent (functionally)
- Design style matches (structural vs behavioral)
- Any sub-modules are correctly instantiated

Expected Outcomes:
The LLM-generated code may not be identical to the golden design but should be functionally equivalent and maintain the same architectural approach. Document any deviations and assess whether they represent valid alternative implementations or errors.

5. Part 2: LLM-Based Testbench Generation and Simulation

In this part, you will use an enhanced LLM-aided testbench generation tool to create comprehensive testbenches that verify both the functionality and internal signals of your LLM-generated designs.

5.1. Step 1: Prepare Inputs

Actions:

- Gather the natural language description from Part 1, Step 2
- Gather the LLM-generated Verilog code from Part 1, Step 3
- Review the design to identify critical internal signals that should be verified
- List internal signals: carry bits, intermediate sums, propagate/generate signals, etc.
- Prepare these inputs for the testbench generation tool

Internal Signal Identification:

For adders, important internal signals typically include:

- Carry bits between stages (e.g., c[1], c[2], c[3])
- Propagate and generate signals (for CLA, KSA)
- Intermediate sum values (for CSA)
- Block propagate signals (for Skip adder)

5.2. Step 2: Generate Testbench with Internal Signal Checking

Actions:

- Use the enhanced LLM-aided testbench generation tool
- Feed both the natural language description AND the LLM-generated Verilog code
- MODIFY the generation prompt to explicitly request internal signal verification
- Specify which internal signals should be monitored and checked
- Request assertion-based verification or expected value checking
- Generate the testbench code
- Review the generated testbench to ensure it includes internal signal checks

Enhanced Prompt for Internal Signal Checking:

"Generate a comprehensive Verilog testbench for the following design. The testbench should:
1. Test all input combinations or a representative set of test vectors
2. Verify the output signals (sum and carry_out)
3. IMPORTANTLY: Monitor and verify internal signals including [list specific signals]
4. Include assertions or checks for expected internal signal values
5. Report any mismatches between expected and actual values
6. Use $display statements to show internal signal values during simulation
7. Include a summary of test results at the end"

Testbench Requirements:

- Must instantiate the DUT (Design Under Test) correctly
- Should include comprehensive test vectors
- Must verify primary outputs (sum, carry_out)
- Must include checks for critical internal signals
- Should use $monitor or $display to show signal values
- Should report PASS/FAIL status for each test
- Should include timing controls (delays) appropriate for simulation

5.3. Step 3: Simulation with Iverilog

Actions:

- Save the LLM-generated testbench as a .v file (e.g., adder_tb.v)
- Ensure the LLM-generated design file is in the same directory or provide correct path
- Compile the design and testbench using Iverilog
- Run the simulation and capture output
- Analyze the simulation results
- If errors occur, debug and potentially regenerate parts of the testbench

Iverilog Commands:

Compilation:

```
iverilog -o adder_sim adder_design.v adder_tb.v
```

Execution:

```
vvp adder_sim
```

Waveform viewing (optional):

```
gtkwave adder_waveform.vcd
```

Debugging Tips:

- If compilation fails, check for syntax errors in the generated code
- Verify module names match between design and testbench instantiation
- Check signal names and widths are consistent
- Use $dumpfile and $dumpvars in testbench to generate VCD waveforms
- Review internal signal values in simulation output to verify correctness
- Compare internal signal behavior with expected values based on architecture

5.4. Step 4: Analysis and Verification

Actions:

- Review simulation output for PASS/FAIL status
- Examine internal signal values during different test cases
- Verify that carry propagation behaves correctly

- Check that intermediate calculations match expected values
- Document any discrepancies between expected and actual behavior
- If issues found, analyze root cause (LLM generation error, testbench error, or design bug)
- Iterate as needed to resolve issues

Analysis Questions:

- Do all test cases pass for the primary outputs (sum, carry_out)?
- Do the internal signals match the expected behavior of the architecture?
- Are there any unexpected signal values or transitions?
- Does the LLM-generated design behave identically to the golden design?
- What differences exist between the LLM design and golden design?
- Are these differences acceptable or do they indicate errors?

6. Expected Deliverables

Upon completion of this project, you should submit the following deliverables:

Adder Selection Document

A document explaining which two adders you selected and why. Include architectural differences.

Natural Language Descriptions

Complete natural language descriptions for both selected adders, generated in Part 1, Step 2.

LLM-Generated Verilog Designs

The Verilog code files generated by ChipChat/AutoChip for both adders (Part 1, Step 3).

Verification Report (Part 1)

A detailed comparison between LLM-generated code and golden reference designs. Include similarities, differences, and assessment of equivalence.

LLM-Generated Testbenches

Complete testbench files with internal signal checking for both adders (Part 2, Step 2).

Simulation Results

Iverilog simulation output showing test execution and results. Include screenshots or captured console output.

Analysis Report (Part 2)

Analysis of simulation results, including verification of internal signals, any issues encountered, and how they were resolved.

Final Project Report

A comprehensive report discussing:

- Your experience with LLM-based hardware design
- Accuracy and limitations of LLM-generated code
- Quality of LLM-generated testbenches
- Lessons learned and best practices
- Recommendations for using LLMs in hardware design

7. Evaluation Criteria

Your project will be evaluated based on the following criteria:

Completeness (25%)

- All required steps completed for both adders
- All deliverables submitted
- Documentation is thorough and well-organized

Technical Accuracy (30%)

- Natural language descriptions accurately reflect the Verilog designs
- LLM-generated Verilog is functionally correct
- Testbenches properly verify both outputs and internal signals
- Simulation results are correctly interpreted

Analysis Quality (25%)

- Verification reports show deep understanding of designs
- Differences between LLM and golden designs are well analyzed
- Simulation results are thoroughly examined
- Critical thinking applied to LLM capabilities and limitations

Problem-Solving (10%)

- Ability to debug and resolve issues during the project
- Effective iteration on prompts to improve LLM outputs
- Appropriate modifications to generated code when necessary

Documentation (10%)

- Clear and professional writing
- Proper formatting of code and results
- Logical organization of materials
- Inclusion of relevant examples and evidence

8. Appendix: Sample Workflows

Sample Workflow for RCA (4-bit Ripple Carry Adder)

Part 1 Sample

1. Selected Adder: RCA4.v from Carry Ripple Adder directory

2. Generated Description (excerpt):

"This design implements a 4-bit Ripple Carry Adder using four full adder modules. The architecture consists of a Full Adder (FA) module and the top-level RCA4 module. Each full adder computes one bit of the sum and generates a carry output that feeds into the next stage. The carry ripples from LSB to MSB, with the first stage receiving a carry-in of 0..."

3. LLM Generation: Feed description to ChipChat

4. Verification: Compare generated code structure, verify FA module definition, check carry chain

Part 2 Sample

1. Prepare natural language description and LLM-generated RCA4 design

2. Identify internal signals: c[1], c[2], c[3] (carry bits between stages)

3. Prompt for testbench generation with internal checks

4. Simulate with iverilog

5. Sample output:

```
Test 1: a=0001, b=0001, expected_sum=0010, expected_c1=0, expected_c2=0, expected_c3=0
Result: PASS - sum=0010, cout=0, internal carries correct
Test 2: a=1111, b=0001, expected_sum=0000, expected_c1=1, expected_c2=1, expected_c3=1
Result: PASS - sum=0000, cout=1, internal carries correct
```

Tips and Best Practices

- Start with simpler adders (RCA) before attempting complex ones (KSA)
- Be specific in your prompts to the LLM - include architectural details
- When comparing code, focus on functional equivalence rather than exact match
- Test edge cases: all zeros, all ones, carry propagation across all bits
- Use waveform viewers to visualize internal signal behavior
- Iterate on your prompts if initial LLM output is unsatisfactory
- Keep detailed notes throughout the process for your final report
- Don't hesitate to manually fix minor syntax errors in generated code
- Verify that internal signals make logical sense based on the architecture
- Compare simulation results between LLM design and golden design if possible

Common Challenges and Solutions

LLM generates behavioral code instead of structural

Solution: Explicitly request structural/gate-level implementation in your prompt. Provide examples of desired style.

Testbench doesn't include internal signal checking

Solution: Be very explicit in your prompt. List specific signal names and what should be checked.

Simulation fails with compilation errors

Solution: Check for syntax errors, verify module names match, ensure all signals are declared.

Internal signal values seem incorrect

Solution: Trace the logic manually for simple test cases. Use waveform viewer to see signal transitions.

LLM-generated code has different module hierarchy

Solution: This may be acceptable if functionally equivalent. Document the differences and assess impact.

Conclusion

This project provides hands-on experience with the emerging field of LLM-assisted hardware design. By working through both the generation and verification workflows, you will gain insights into the capabilities and current limitations of AI tools in the hardware domain. The skills developed in this project - from architectural analysis to verification methodologies - are valuable for modern digital design practices.

Remember that LLMs are powerful tools but require careful validation. Always verify generated designs through simulation and, ideally, formal verification methods. Use these tools to augment your expertise, not replace your critical thinking and domain knowledge.

Good luck with your project!