ChipChat Assignment Report

LLM-Assisted Verilog Generation, Verification, and Debugging

Tejas Attarde

Abstract

Large Language Models (LLMs) have demonstrated strong performance in natural language reasoning and software code generation; however, their reliability in hardware description language (HDL) design remains an open challenge. This report investigates the use of LLMs as assistive tools for synthesizable Verilog generation using the ChipChat framework. Through structured prompt engineering, RTL cleanup, simulation-based verification, and iterative debugging, this work evaluates both the strengths and limitations of LLM- assisted hardware design. The results show that while LLMs can significantly accelerate early-stage RTL development, careful human oversight, disciplined verification, and hardware-aware prompting are essential to ensure correctness and synthesizability.

1. Introduction and Motivation

Hardware design traditionally requires deep domain expertise, careful reasoning about clocked behavior, and rigorous verification. Even small mistakes in synchronization or reset logic can result in functional failures that are difficult to diagnose. With the rapid advancement of large language models, there is growing interest in whether these models can meaningfully assist in hardware description language (HDL) development.

Unlike software programs, RTL designs describe cycle-accurate behavior and must adhere to strict synthesizability constraints. A design that appears logically correct may still fail due to subtle timing issues. This makes HDL generation a particularly challenging task for LLMs, which are primarily trained on natural language and software code.

The motivation for this assignment is to explore how LLMs behave when used as assistive tools for Verilog generation, and to understand how prompt engineering, verification, and iterative debugging can mitigate their limitations. Rather than treating the LLM as an oracle, this work treats it as a fallible collaborator whose outputs must be carefully validated.

2. Background: LLMs in Hardware Design

Large language models learn statistical patterns in syntax and structure from large corpora of text and code. While this allows them to generate syntactically correct Verilog, HDL design presents challenges that differ significantly from software

development. RTL code must satisfy timing, reset, and concurrency constraints that are not explicitly encoded in natural language descriptions.

Common failure modes observed in LLM-generated HDL include:

- Incorrect or missing reset behavior
- Mixing combinational and sequential logic
- Accidental latch inference
- Outputs asserted combinationally instead of synchronously

These issues motivated the structured ChipChat workflow, which emphasizes prompt precision, explicit RTL cleanup, and simulation-based verification. In this assignment, simulation results are treated as the ultimate authority on correctness.

3. Toolchain and Experimental Setup

All experiments were conducted in Google Colab to ensure reproducibility. The following tools were used:

- **LLM API:** OpenAI
- **Model:** gpt-4o-mini
- **Simulation Tool:** Icarus Verilog (`iverilog`)

The `iverilog -g2012` flag was used consistently to enforce Verilog-2012 compliance. All generated RTL and testbenches were written to disk programmatically from within the notebooks to ensure full reproducibility.

4. Prompt Engineering Methodology

Prompt engineering was a critical factor in achieving usable and synthesizable Verilog designs. Initial experiments with short or vague prompts often resulted in code that was syntactically valid but violated hardware design principles, such as the use of behavioral constructs, implicit latches, or missing reset logic.

To address this, all prompts were written with explicit hardware-centric constraints. Each prompt clearly specified the target Verilog standard (Verilog-2012), prohibited delays and behavioral timing constructs, and required the use of `always_ff` and `always_comb` blocks to enforce proper separation between sequential and combinational logic.

For sequential designs, reset behavior was explicitly defined as synchronous and active-high. Outputs that represented events (such as sequence detection) were required to be asserted for exactly one clock cycle. These constraints significantly reduced ambiguity in the LLM's responses and improved the quality of the generated RTL.

A temperature of zero was used for all model calls to ensure deterministic outputs. This decision was important for reproducibility, as it ensured that rerunning the notebooks would produce identical code generation results, making debugging and verification repeatable.

5.1 Example A: `binary_to_bcd`

Final Prompt

The LLM was instructed to generate a purely combinational Verilog module that converts an 8-bit binary input into three BCD digits. The prompt explicitly disallowed clocks, delays, and non-synthesizable constructs.

Module Interface

```
module binary_to_bcd(
    input logic [7:0] bin,
    output logic [3:0] hundreds,
    output logic [3:0] tens,
    output logic [3:0] ones
);
```

Design Approach

The final implementation uses the double-dabble (shift-and-add-3) algorithm implemented within an `always_comb` block. Although Icarus Verilog emits a known warning related to constant bit selects in `always_comb`, the design is fully

synthesizable and functionally correct. Verification was performed using the official ChipChat testbench.

Detailed Implementation Discussion

The binary-to-BCD conversion was implemented using the well-known double-dabble (shift-and-add-3) algorithm. This algorithm iteratively shifts the binary input and conditionally adds three to intermediate BCD digits when their value exceeds four. The approach is well suited for combinational implementation and avoids division or modulo operations, which are inefficient in hardware.

The implementation uses a fixed-width shift register to hold the binary input and intermediate BCD digits. A `for` loop performs a bounded number of iterations, which is synthesizable because the loop bounds are static and known at compile time. Although Icarus Verilog emits a warning related to constant bit selects in `always_comb` blocks, this is a known simulator limitation and does not affect functional correctness or synthesizability.

Verification was performed using the official ChipChat testbench, which exhaustively checks representative input values. Successful simulation confirmed that the combinational logic produces correct BCD outputs for all tested cases.

5.2 Example B: sequence_detector

Final Prompt

The LLM was prompted to generate a synthesizable FSM that detects the overlapping sequence `1011`, uses enumerated states, supports synchronous reset, and asserts a one-cycle detection pulse.

Module Interface

```
module sequence_detector (
    input logic clk,
    input logic reset,
    input logic in_bit,
    output logic detected
);
```

Design Approach

The FSM was implemented using a state register, next-state combinational logic, and a registered output. Enumerated states improved readability and reduced the risk of encoding errors. The final design compiles with `iverilog -g2012` and passes the provided testbench.

FSM Structure and State Encoding

The sequence detector FSM was implemented using an enumerated state type, which improves readability and reduces the likelihood of state encoding errors. Each state represents partial progress toward detecting the target sequence, allowing overlapping sequences to be handled naturally.

The FSM follows a clear separation of concerns: a state register updated on the rising edge of the clock, a combinational next-state logic block, and a registered output block. This structure aligns with best practices in synchronous digital design and simplifies both reasoning and debugging.

Special attention was paid to output timing. The `detected` signal represents an event rather than a level, and therefore must be asserted for exactly one clock cycle. This requirement influenced both the FSM design and the subsequent debugging process documented in Part II.

6. Part II: Debugging Loop Documentation

This section documents a real debugging process encountered during the development of the `sequence_detector` FSM. Rather than presenting a contrived example, this section reflects an actual functional issue identified through simulation and the reasoning process used to resolve it.

6.1 Iteration 1: Initial Failure

The initial FSM implementation compiled successfully, indicating that the syntax and overall structure were valid. However, simulation using the provided testbench resulted in a failure message indicating that the expected sequence was not detected.

At this stage, the FSM appeared logically correct when inspected manually. The state transitions matched the intended behavior, and the final state was reached when the input sequence was applied. This made the failure non-obvious and highlighted the importance of simulation-based debugging.

6.2 Root Cause Analysis

Closer inspection revealed that the `detected` output was asserted combinationally within the next-state logic. As a result, the detection pulse was extremely short and not aligned with a clock edge. Because the testbench samples outputs synchronously, the pulse was missed entirely.

This type of bug is particularly subtle because it does not manifest as a compile error or syntax warning. Instead, it arises from a mismatch between the designer's intent and the temporal behavior of the circuit.

6.3 Iteration 2: Applying the Fix

To resolve the issue, the `detected` signal was moved into a clocked `always_ff` block. The signal was asserted when the FSM was in the appropriate state and the final input bit was observed. This change ensured that the output remained asserted for a full clock cycle.

6.4 Iteration 3: Final Verification

After applying the fix, the design was recompiled and simulated again. This time, the testbench passed successfully, confirming that the timing issue had been resolved. No further modifications were required.

7. Part III: Prompt-Engineering Extension

Design Rationale for Extension

The extension was designed to explore how additional constraints affect LLM-generated RTL. Adding an `enable` input required the FSM to conditionally update its state, introducing a level of control logic not present in the original example. Exposing the internal state through `state_out` improved observability and made debugging easier without affecting functional behavior.

These changes required careful prompt wording to ensure that the LLM did not introduce unintended latch behavior or violate synchronous design principles. The

final design reflects a more realistic hardware module that balances functionality, observability, and control.

Extension Description

The original FSM was extended by changing the detected sequence, adding an `enable` input to gate state transitions, and exposing the FSM state via a `state_out` output.

Prompt Impact

Explicitly specifying gating behavior and state observability in the prompt forced the LLM to generate more structured FSM logic. The resulting RTL was cleaner and easier to verify.

Verification

A modified self-checking testbench was written, and the extended design passed all simulations.

8. Verification Strategy and Reproducibility

All designs were verified using simulation with `iverilog` and `vvp`. Both RTL and testbench files were generated programmatically within the notebooks. Running each notebook top-to-bottom on a fresh Colab runtime reproduces identical results.

```
iverilog -g2012 -o simv design.v tb_design.v
vvp simv
```

All simulations concluded with:

```
All tests passed!
```

9. Limitations and Failure Modes

Despite careful prompt engineering, several limitations were observed:

- LLMs may generate temporally incorrect logic
- Outputs are often asserted combinationally
- FSM edge cases require human intervention

These observations reinforce the need for verification-driven workflows.

10. Lessons Learned and Future Work

This assignment demonstrated that LLMs can accelerate early-stage RTL development but cannot replace human reasoning. Future work could explore formal verification integration, automated prompt refinement, or HDL-specific model training.

11. Conclusion

This work evaluated LLM-assisted Verilog generation using the ChipChat framework. While LLMs proved useful for rapid RTL prototyping, correctness was only achieved through careful prompt engineering, manual cleanup, simulation- based verification, and iterative debugging. Treating the LLM as an assistive tool rather than an oracle was key to the successful completion of this assignment.