ChipChat Tutorial Assignment Report

LLM-Assisted Verilog Generation, Debugging, Verification, and Extension

Tejas Attarde
LLM4ChipDesign

1 Introduction

Large Language Models (LLMs) have recently demonstrated strong capabilities in generating source code from natural language specifications. While this ability has been widely explored in software engineering, its application to hardware design introduces a distinct set of challenges. Hardware description languages such as Verilog require strict adherence to synthesizability constraints, deterministic behavior, and compatibility with fixed verification environments.

Unlike software programs, hardware designs cannot rely on runtime debugging or dynamic patching. Errors in RTL often manifest only during simulation or, worse, after synthesis. Therefore, the use of LLMs in hardware design must be approached with caution, emphasizing verification, manual review, and iterative refinement.

This assignment explores the ChipChat tutorial framework, which provides a disciplined workflow for integrating LLMs into RTL design. The framework treats the LLM as a code-generation assistant rather than an oracle, placing responsibility on the designer to validate and correct the generated output.

The primary objectives of this report are:

- To demonstrate correct use of the ChipChat workflow for RTL generation
- To validate LLM-generated designs using unmodified testbenches
- To document an iterative debugging loop grounded in hardware principles
- To extend a tutorial design in a meaningful and reproducible way

2 ChipChat Workflow and Methodology

The ChipChat tutorial enforces a structured methodology for LLM-assisted hardware design. This methodology mirrors real-world RTL development practices and consists of the following stages:

2.1 Specification Writing

The first step is to translate design intent into a precise natural language specification. Unlike informal descriptions, these specifications must explicitly define:

- Module name and interface
- Signal widths and directions
- Timing behavior (combinational vs sequential)
- Reset semantics
- Synthesizability constraints

Ambiguity at this stage often leads to incorrect or non-synthesizable LLM output.

2.2 LLM Invocation

The OpenAI API is used to query the model with the written prompt. In this assignment, the same invocation pattern demonstrated in the ChipChat tutorial notebook was followed exactly. The raw output is captured and displayed for inspection.

2.3 RTL Extraction and Cleanup

LLM-generated code frequently includes extraneous constructs such as comments, test logic, or non-synthesizable elements. Following the tutorial, only the core module definition is extracted and manually cleaned to ensure compatibility with `iverilog -g2012`.

2.4 Verification and Iteration

Verification is performed using provided testbenches. Testbenches are treated as immutable; all discrepancies must be resolved by modifying the RTL. This enforces a professional verification mindset and prevents "overfitting" the verification environment.

3 Models and Generation Parameters

All RTL generation in this assignment was performed using the OpenAI API.

3.1 Model Selection

- **Model**: `gpt-4o-mini`
- **Provider**: OpenAI

This model was selected for its balance between code quality and deterministic output.

3.2 Generation Parameters

- **Temperature**: 0
- **Max Tokens**: Default

A temperature of zero was chosen to minimize randomness and ensure reproducible results, which is essential in a hardware verification context.

4 Part I — Example A: Binary-to-BCD Converter

4.1 Problem Description

The Binary-to-BCD converter translates a binary number into a packed Binary Coded Decimal representation. The tutorial testbench assumes a purely combinational implementation and checks correctness across all valid input values.

4.2 Final Prompt

```
Generate a synthesizable Verilog module named binary_to_bcd_converter.
The module converts a 5-bit binary input into a packed BCD output.
Use purely combinational logic.
Do not use clocks, delays, or initial blocks.
The interface must exactly match the provided testbench.
```

4.3 Module Interface

```
module binary_to_bcd_converter (
    input  wire [4:0] binary_input,
    output reg  [7:0] bcd_output
);
```

4.4 Design Rationale

The design computes the ones and tens digits using modulo and division operations with constant divisors. These operations are synthesizable for small constant values and result in concise RTL. No sequential logic is used, ensuring immediate output stability.

4.5 Verification

```
iverilog -g2012 binary_to_bcd_converter.v binary_to_bcd_tb.v
vvp a.out
```

All test cases passed, confirming functional correctness.

5 Part I — Example B: Sequence Detector

5.1 Problem Description

The sequence detector monitors a stream of 3-bit input values and asserts an output when a predefined sequence is observed. The provided testbench samples outputs immediately after clock edges, placing strict timing requirements on the design.

5.2 Final Prompt

```
Generate a synthesizable Verilog module named sequence_detector.
Detect a specific sequence of 3-bit input values.
Use an active-low reset.
Assert the output for exactly one cycle.
The interface must exactly match the provided testbench.
```

5.3 Module Interface

```
module sequence_detector (
    input  wire       clk,
    input  wire       reset_n,
    input  wire [2:0]  data,
    output wire       sequence_found
);
```

5.4 Design Approach

The detector is implemented as an FSM where each state corresponds to a partial match of the target sequence. Detection is generated combinationally from the previous state and current input to ensure visibility in the same cycle as required by the testbench.

5.5 Verification

```
iverilog -g2012 sequence_detector.v sequence_detector_tb.v
vvp a.out
```

All test cases passed.

6 Part II — Debugging and Iterative Improvement

6.1 Debugging Philosophy

Debugging was performed under the constraint that the testbench could not be modified. This reflects industry practice, where verification environments are often shared and immutable.

6.2 Iteration Table

Iteration	Issue Observed	Fix Applied	Outcome
1	Reset misalignment	Asynchronous reset	Fail
2	Output scheduling issue	Combinational output	Fail
3	State/input misalignment	Detect on previous state	Pass

6.3 Lessons Learned

This process highlighted the importance of understanding Verilog scheduling semantics, particularly the distinction between blocking and non-blocking assignments and the timing of testbench checks.

6.4 Detailed Iteration Analysis

This section provides a deeper explanation of each debugging iteration summarized in Table 1. Rather than treating debugging as a single-step correction, the ChipChat workflow emphasizes iterative refinement based on observed verification failures.

6.4.1 Iteration 1: Reset Semantics Mismatch

In the first iteration, the sequence detector failed to assert the `sequence_found` signal at the expected cycle. Initial inspection suggested that the FSM logic was structurally correct; however, closer analysis revealed a mismatch between reset semantics in the design and the behavior assumed by the testbench.

The initial design used a synchronous active-low reset. In contrast, the testbench deasserted reset on a rising clock edge. As a result, the FSM remained in reset for one additional cycle, causing the first valid input symbol to be ignored. This shifted the entire input sequence and prevented correct detection.

To address this issue, the reset logic was changed to an asynchronous active-low reset. This allowed the FSM to exit reset immediately when `reset_n` was deasserted, ensuring that the first input symbol was sampled correctly.

Despite this correction, the testbench continued to fail, indicating that reset semantics were not the only source of error.

6.4.2 Iteration 2: Output Visibility and Scheduling

In the second iteration, the reset issue was resolved, but the output signal was still not asserted at the expected cycle. This failure required a deeper understanding of Verilog simulation semantics.

The testbench checks the output signal immediately after the rising edge of the clock. In the design, `sequence_found` was originally assigned using a non-blocking assignment inside a clocked `always` block. Due to Verilog's event scheduling model, non-blocking assignments are updated in the NBA (Non-Blocking Assignment) region, which occurs after the testbench performs its check.

As a result, even though the FSM logically detected the sequence, the output signal was not visible at the time of checking. This led to the false impression that the detection logic was incorrect.

The fix in this iteration was to generate the output signal combinationally rather than sequentially. By deriving `sequence_found` directly from the FSM state and current input, the signal became visible in the same simulation cycle as required by the testbench.

This change improved timing alignment but revealed another subtle issue.

6.4.3 Iteration 3: State and Input Alignment

After resolving scheduling issues, the design still failed under certain conditions due to a misalignment between FSM state transitions and input evaluation. Specifically, the detection condition was based on the updated FSM state rather than the previous state combined with the current input.

Because FSM state updates occur on the clock edge, checking the updated state in the same cycle as the input caused the detection condition to be missed. This off-by-one-cycle error is a common pitfall in FSM design.

The final correction involved asserting the detection signal based on the FSM state prior to transition and the current input value. This ensured that the detection condition aligned exactly with the testbench's timing expectations.

After applying this fix, the design passed all test cases without requiring any modifications to the testbench.

6.5 Cycle-Level Debugging Insights

This debugging process highlights the importance of reasoning at the cycle level when designing sequential logic. Even small mismatches in reset timing or output visibility can cause failures that are not obvious from static code inspection.

The iterative approach encouraged by the ChipChat tutorial mirrors real industry practice, where designs are refined incrementally based on verification feedback rather than rewritten entirely.

6.6 Why Testbench Immutability Matters

A key constraint in this assignment was the requirement that the testbench remain unchanged. This constraint prevents designers from masking RTL errors by modifying verification logic and ensures that the final design is robust and portable.

By adhering to this constraint, the final sequence detector design is guaranteed to function correctly in any environment that follows the same interface and timing assumptions.

7 Part III — Extension: Parameterized Binary-to-BCD Converter

7.1 Extension Motivation

Parameterization improves reuse and scalability. Extending the Binary-to-BCD converter avoids FSM complexity while demonstrating meaningful design evolution.

7.2 Extended Interface

```
module binary_to_bcd_param #(
    parameter INPUT_WIDTH = 5
)(
    input  wire [INPUT_WIDTH-1:0] binary,
    output reg  [7:0] bcd
);
```

7.3 Verification

```
iverilog -g2012 binary_to_bcd_param.v binary_to_bcd_param_tb.v
vvp a.out
```

All extension test cases passed.

8 Reproducibility and Limitations

All notebooks were executed top-to-bottom without manual intervention except for API key insertion. While LLMs accelerate development, they require careful human oversight to ensure correctness and synthesizability.

9 Conclusion

This assignment demonstrates that LLMs can be valuable assistants in RTL development when paired with rigorous verification and disciplined debugging. The ChipChat workflow provides a structured approach for safely integrating LLMs into hardware design processes.

10 Detailed Prompt Engineering Analysis

Prompt engineering plays a critical role in successfully using large language models for hardware design. Unlike software generation, RTL generation requires precise structural and behavioral constraints. Even small ambiguities in the prompt can lead to incorrect interfaces, non-synthesizable constructs, or mismatches with verification environments.

For the Binary-to-BCD converter example, the prompt explicitly specified that the design must be purely combinational and must not include clocks, delays, or initial blocks. This was necessary because the provided testbench assumes immediate output stabilization after input changes. Without this constraint, the LLM frequently attempts to introduce sequential logic, which would cause testbench failures.

In the Sequence Detector example, the prompt required careful specification of reset polarity, output pulse width, and interface matching. Early prompt iterations that did not explicitly specify output timing resulted in multi-cycle assertions of the detection signal. This reinforced the need for precise language when describing temporal behavior in hardware designs.

Overall, this assignment demonstrated that prompt engineering for RTL generation is closer to writing a formal specification than an informal description. The LLM performs best when the prompt resembles a natural language version of a hardware specification document.

11 LLM Failure Modes in Hardware Design

During this assignment, several recurring failure modes of LLM-generated hardware designs were observed. Understanding these failure modes is essential for safely integrating LLMs into hardware development workflows.

One common failure mode is incorrect reset handling. LLMs frequently default to synchronous resets unless explicitly instructed otherwise. However, verification environments may assume asynchronous resets or specific reset deassertion timing. This mismatch can lead to subtle off-by-one-cycle errors that are difficult to diagnose without waveform inspection.

Another failure mode is improper use of non-blocking assignments for output signals that are checked immediately after clock edges. While non-blocking assignments are generally recommended for sequential logic, their interaction with testbench scheduling semantics can cause outputs to appear delayed. This was directly observed in the Sequence Detector example.

LLMs also tend to omit default transitions in finite state machines. This can lead to undefined behavior when unexpected inputs occur. Manual review and completion of FSM transition logic were necessary to ensure robust operation.

These observations highlight that LLMs should be treated as productivity tools rather than authoritative designers. Human expertise remains essential for ensuring correctness.

12 Verification Methodology

Verification in this assignment was performed using the provided testbenches without any modification. This constraint mirrors industry verification flows, where design teams must adapt their RTL to fixed verification environments.

Each design was compiled using `iverilog -g2012`, ensuring SystemVerilog compatibility while maintaining broad tool support. Simulation was performed using `vvp`, and outputs were inspected for pass/fail messages emitted by the testbenches.

In addition to pass/fail outcomes, intermediate failures were analyzed by reasoning about clock cycles, reset timing, and signal visibility. Although waveform viewing tools such as GTKWave were not required, understanding simulation timing was critical for debugging.

This assignment reinforced the importance of simulation-based verification even for seemingly simple designs. Subtle issues related to scheduling and reset semantics can cause failures that are not immediately obvious from the RTL code alone.

13 Comparison of Raw LLM Output and Cleaned RTL

The ChipChat tutorial emphasizes the need to manually extract and clean LLM-generated RTL. This step proved essential throughout the assignment.

Raw LLM outputs often included excessive comments, alternative design approaches, or partially implemented logic. In some cases, the LLM generated multiple module definitions or included behavioral constructs unsuitable for synthesis.

Manual cleanup involved isolating the correct module, removing extraneous code, and verifying synthesizability. This process required understanding both the intent of the prompt and the assumptions made by the testbench.

The cleaned RTL was significantly shorter and clearer than the raw LLM output. This highlights an important insight: the value of LLMs lies in accelerating initial code generation, not in replacing careful human review.

14 Extension Design Rationale

The extension implemented in Part III focused on parameterization of the Binary-to-BCD converter. This choice was motivated by a desire to demonstrate design scalability without introducing additional timing complexity.

Parameterized designs are common in real-world RTL development, allowing a single module to be reused across multiple configurations. By parameterizing the input width, the same converter logic can be applied to different input ranges without rewriting the design.

This extension avoided state machines and sequential logic, reducing the risk of verification issues while still demonstrating meaningful design evolution. The accompanying self-checking testbench verified correctness across the full input range.

15 Reproducibility Checklist

To ensure reproducibility, the following conditions were satisfied:

- All notebooks execute top-to-bottom without manual edits
- The OpenAI API key is inserted in a single, clearly marked cell
- All generated Verilog files are written programmatically
- Verification commands are explicitly listed
- No testbench files are modified in Parts I or II

These steps ensure that another user can reproduce the results on a different system with minimal effort.

16 Threats to Validity

While the results presented in this report are correct within the scope of the assignment, several limitations should be noted. The designs are relatively small and do not stress the scalability limits of LLM-based generation. Larger designs may expose additional challenges not encountered here.

Additionally, only a single LLM configuration was evaluated. Different models or temperature settings may produce different results. Future work could explore comparative studies across models.

17 Final Reflections

This assignment provided valuable insight into the strengths and limitations of LLM-assisted hardware design. While LLMs can significantly reduce the time required to generate baseline RTL, they do not eliminate the need for verification expertise.

The ChipChat workflow provides a strong foundation for safely integrating LLMs into hardware design processes. With careful prompt engineering, disciplined verification, and iterative debugging, LLMs can become effective assistants in RTL development.