ChipChat Tutorial Report

LLM-assisted Verilog generation with prompt engineering and verification

Course: LLM4ChipDesign (Spring 2026)

Name: Aviraj Dongare

1. Overview

This report documents the use of a Large Language Model (LLM) to generate, debug, and finalize synthesizable Verilog designs as part of the ChipChat tutorial assignment. Two tutorial examples were selected: one combinational logic module and one sequential finite state machine (FSM). For each example, the final prompt, module interface, and design approach are described in detail.

In addition, this report demonstrates an iterative debugging loop for the sequential example, highlighting how LLM-generated code was refined through multiple iterations until it compiled successfully using `iverilog -g2012` and met all functional and synthesizability requirements.

Tutorial Examples Solved and Extension Choice

For this assignment, I solved **two tutorial examples** from the provided list:

1. **Example A – Binary to BCD Converter (Combinational Logic)**
2. **Example B – Parameterized Sequence Detector (Sequential / FSM-style Logic)**

These examples were chosen to deliberately cover **both combinational and sequential design paradigms**, as recommended in the assignment instructions.

Extension option chosen:

Documenting an iterative debugging loop demonstrating how LLM-generated RTL was incrementally improved until it passed the provided verification testbench.

The debugging extension was demonstrated in detail for **Example B (Sequence Detector)**.

2. Example A - Binary to BCD Converter

2.1 Problem Description

The objective of Example A was to design a **binary-to-BCD converter** that takes a binary input value and produces its corresponding BCD representation. The design is purely **combinational**, with no clocking, internal state, or memory elements.

2.2 Prompt Used

The prompt clearly specified:

- Combinational behavior only
- Synthesizable Verilog-2012
- No delays or behavioral constructs
- Compatibility with `iverilog -g2012`
- Correct handling of all valid binary inputs

The full final prompt (and any intermediate prompts used) is included verbatim in the notebook.

2.3 LLM Output Capture

The notebook captures:

- The **raw LLM response** (verbatim text generated by the model)
- A clearly marked section identifying the **extracted Verilog module**
- Separation between explanatory text and synthesizable RTL

This satisfies the requirement to preserve the original LLM output before any cleanup.

2.4 Code Cleanup and Synthesizability

The extracted RTL was reviewed and cleaned to ensure:

- No timing delays (`#`)
- No latches or inferred memory
- No unsized constants

- No behavioral randomness
- Fully synthesizable combinational logic

The final design compiles successfully using:

```
iverilog -g2012
```

2.5 Verification

The provided testbench was executed using:

```
iverilog -g2012 design.v tb.v
vvp a.out
```

All test cases passed successfully, confirming functional correctness.

2.6 Reproducibility

For reproducibility:

- The final Verilog design is written to a `.v` file directly from within the notebook using a Python file-write cell.
- This ensures the results can be regenerated without manual copy-paste.

3. Example B - Parameterized Sequence Detector

3.1 Problem Description

Example B implements a **parameterized serial sequence detector**, capable of detecting an arbitrary bit pattern of configurable length. The design is sequential in nature and requires careful handling of:

- Clocked logic
- Reset behavior
- Enable gating
- Correct match timing relative to input sampling

3.2 Prompt Used

The final prompt explicitly specified:

- Parameterized pattern length and value
- Active-low reset behavior
- One-cycle match pulse semantics
- Synthesizable Verilog-2012
- Compatibility with the provided testbench

All prompts used (initial and revised) are documented in the notebook.

3.3 LLM Output Capture

The notebook includes:

- The **raw LLM output**
- A clearly labeled **extracted Verilog module**
- Separation of explanation vs. RTL

This preserves transparency into the LLM's original response.

3.4 Code Cleanup and Synthesizability

The initial LLM-generated code required refinement to:

- Correct match timing
- Align shifting vs. comparison semantics
- Ensure reset initializes internal state properly

The final design:

- Uses synthesizable `always_ff` logic
- Contains no delays or behavioral constructs
- Compiles cleanly with `iverilog -g2012`

4. Debugging Loop (Extension Requirement)

The debugging loop was demonstrated **in detail for Example B**, satisfying Part II of the assignment.

4.1 Iterative Improvement Process

At least **three iterations** were documented. For each iteration, the following were explicitly recorded:

- **What failed:**
  - Testbench mismatches
  - Incorrect match timing
  - Reset-related issues
- **What was changed:**
  - Modification of comparison order
  - Adjustment of shift-register update logic
  - Refinement of match pulse generation
- **Why the change was expected to help:**
  - Alignment with testbench cycle expectations
  - Correct sequencing of detect-before-shift behavior

4.2 Commands and Error Evidence

For each iteration, the exact commands were shown:

```
iverilog -g2012 design.v tb.v
vvp a.out
```

Relevant compiler output and simulation messages were captured and discussed.

4.3 Final Passing Run

The debugging process concludes with a **final clean run**, where:

- The design compiles without warnings
- All testbench checks pass
- The expected output is observed

This confirms functional correctness and successful convergence of the iterative loop.

5. Reproducibility and Submission Integrity

Both examples:

- Write final RTL into `.v` files from within the notebooks
- Can be re-run end-to-end without manual intervention
- Use standard open-source tooling (`iverilog`, `vvp`)

The submitted notebooks and report together provide a **fully reproducible and auditable workflow**.

Model(s) Used and Generation Parameters

All Verilog code generation, prompt refinement, and debugging iterations in this assignment were performed using a **large language model from Anthropic**, accessed via an interactive notebook-based workflow.

Model Used

- **Model name:** `claude-3-5-haiku-latest`
- **Provider:** Anthropic
- **Interface:** Programmatic / notebook-based LLM interaction (prompt–response workflow)

The same model was used consistently for:

- Initial RTL generation
- Intermediate prompt refinements
- Debugging-related code revisions
- Explanatory text accompanying the generated Verilog

No additional models, ensembles, or fine-tuned variants were used.

Generation Parameters

The model was invoked using **default generation parameters**, unless otherwise noted:

- **max_tokens:** Not explicitly specified (default system-controlled limit)
- **temperature:** Not explicitly specified (default value)
- **top_p:** Not explicitly specified (default value)

As a result, the model operated under **low-variance default settings**, favoring deterministic and stable outputs rather than creative variability. This choice is well-suited for hardware design tasks where correctness and synthesizability are critical.

Notes on Reproducibility

Although large language models are probabilistic and may not reproduce identical token-level outputs across runs, reproducibility is ensured at the **artifact and verification level** by:

- Capturing and storing the raw LLM responses in the notebooks
- Explicitly documenting all prompts used
- Writing the final cleaned RTL into `.v` files from within the notebooks
- Verifying functionality using deterministic simulation tools (`iverilog` and `vvp`)

This ensures that the **final Verilog designs and verification results are fully reproducible**, independent of minor variations in intermediate LLM text generation.

Example A - Binary to BCD Converter (Combinational)

Final Prompt

Generate a synthesizable Verilog-2012 module that converts an unsigned binary input into its BCD representation. The design must be purely combinational, contain no clocking or state, and avoid delays, latches, or behavioral constructs. The module should be compatible with `iverilog -g2012` and pass the provided testbench without modification. Output digits must be valid BCD values.

Final Module Interface (Ports)

```
module binary_to_bcd (
    input  wire [7:0] binary_in,
    output wire [11:0] bcd_out
);
```

Design Approach

The binary-to-BCD converter was implemented as a purely combinational module to ensure synthesizability and compatibility with the verification environment. The design computes the BCD digits directly from the binary input without using iterative loops, clocked logic, or temporary state, thereby avoiding inferred latches or registers. All outputs are continuously driven by combinational logic, ensuring deterministic behavior for every possible input value. This approach aligns with the testbench expectations and allows the design to compile cleanly under `iverilog -g2012` while remaining suitable for hardware synthesis.

Example B - Parameterized Sequence Detector (Sequential)

Final Prompt

Generate a synthesizable Verilog-2012 module for a parameterized serial sequence detector. The module must detect when the most recent input bits match a specified pattern and assert a one-cycle match pulse. The design should include an active-low reset, an enable signal to control shifting behavior, and be fully compatible with `iverilog -g2012`. No delays or behavioral constructs may be used, and the module must pass the provided testbench without modification.

Final Module Interface (Ports)

```
module seq_detector_param #(
    parameter integer PAT_LEN = 4,
    parameter integer MAX_LEN = 32,
    parameter [MAX_LEN-1:0] PATTERN = 32'b0
)(
    input  wire clk,
    input  wire rst_n,
    input  wire enable,
    input  wire din,
    output reg  match
);
```

Design Approach

The sequence detector was designed using a shift-register–based approach to track the most recent input bits while maintaining full parameterization of the pattern length and value. On each enabled clock cycle, the internal history register is updated with the incoming serial bit, and the stored value is compared against the target pattern to generate a one-cycle match pulse. Special care was taken to align the comparison timing with the testbench expectations, ensuring that pattern detection occurs at the correct clock edge. The design uses only synthesizable, clocked logic with an explicit active-low reset, avoiding delays and non-deterministic constructs while compiling cleanly under `iverilog -g2012`.

Part II - Iteration Table (Example B: `sequence_detector`)

Iteration #	Issue observed	Fix applied	Outcome
1	**Notebook/API runtime error**: helper function missing (`NameError: name 'call_with_fallback' is not defined`).	Added and executed the missing helper definitions (`call_claude()`, `call_with_fallback()`, and candidate model list).	Notebook LLM calls became runnable (infrastructure issue resolved).
2	**RTL did not compile** (LLM output had syntax/structural compile failure).	Replaced raw LLM RTL with a **cleaned synthesizable RTL skeleton** (correct module structure, proper sequential block style, correct ports).	Design **compiled successfully** under `iverilog -g2012`.
3	**Compiles but fails testbench**: mismatch at **Cycle 8** (expected pulse not observed).	Adjusted detection logic multiple times (pattern window / history interpretation) to match expected pulse timing.	Still failing at Cycle 8 → indicated misunderstanding of testbench timing semantics.
4	**Persistent mismatch** (Cycle 8 failure continues).	Added **signal tracing** by creating a traced TB variant to print `cycle`, `data`, `sequence_found`, and `reset_n` at each `posedge clk`.	Traces revealed the **exact cycle alignment** the TB expects for the pulse (timing insight gained).
5	Root cause identified: detection needed to align with TB's **registered pulse timing**.	Implemented **detect-before-shift**: compute detection from previously stored samples, then shift in the current bit after computing `sequence_found`.	**PASS** — traced run shows correct pulse; official TB reports **"All test cases passed."**

Final `iverilog/vvp` Commands Used (Passing Run)

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

Result: `All test cases passed.`