Lab Report-ChipChat Colab (GenAI Workflow & RTL Examples)

Course/Lab: Lab 1-ChipChat Colab

Student: **Naveen Kumar Senthil Kumar**

NetID: **ns6503**

Date: September 17, 2025

Tools: Icarus Verilog (iverilog), Google Colab, ChipChat (GenAI), OpenRouter API

1) Tools & Settings

LLM model(s) & version(s): `openrouter/sonoma-sky-alpha`

Interface: Google Colab (Python SDK via `openai` client with `base_url="https://openrouter.ai/api/v1"`)

Key parameters controlled:
- `max_tokens = 1,700,000`
- `stream = False`
- `messages = [{"role":"user","content": verilog_generation_prompt}]`
- Temperature / top_p / system prompt: defaults (not explicitly set)

Simulation: Icarus Verilog for compile/sim

Versioning & Files: prompts and iterations logged inline per example; VCDs saved per testbench when applicable.

2) Reproducible GenAI Workflow

1. Initial prompt: Describe the RTL spec (I/O, resets, timing/behavior, parameters).

2. Generate code (Iteration N): Ask LLM for Verilog/SystemVerilog module that matches the testbench's port names.

3. Simulate: `iverilog -g2012 -o sim.out design.v tb.v && vvp sim.out`

4. Collect feedback: Copy exact compile/run errors.

5. Refine prompt: Paste Tool Feedback (TF) and Simple Human Feedback (SHF).

6. Repeat until pass.

7. Record results.

Example 1-Binary to BCD Converter

Specification:

Inputs: 5-bit binary

Outputs: 8-bit BCD

Behavior: Convert 0-31 into packed BCD.

Initial Prompt:

'''

I am trying to create a Verilog model binary_to_bcd_converter for a binary to binary-coded-decimal converter. It must meet the following specifications:

- Inputs:
  - Binary input (5-bits)
- Outputs:
  - BCD (8-bits: 4-bits for the 10's place and 4-bits for the 1's place)

How would I write a design that meets these specifications?

'''

Iteration Log:

- **Iter 1 TF:**
  - binary_to_bcd_tb.v:8: error: port 'binary_input' is not a port of uut.
  - ... 'bcd_output' is not a port of uut.

    Root cause: Port-name mismatch between DUT and TB.
- **Iter 2 SHF:** "Match the signals with the testbench."
  Result: *All test cases passed.*

Outcome

- **Status:** PASS
- **Key fix:** Align DUT port names/types with TB.

Lesson(s) Learned

- Always mirror TB signal names exactly; mismatch is the most common first-iteration failure.

Example 2 - Sequence Detector

Specification:

Inputs: clk, reset_n, data[2:0];

Output: sequence_found.

Initial Prompt:

'''
I am trying to create a Verilog model for a sequence detector. It must meet the following specifications:
- Inputs:
- Clock
- Active-low reset
- Data (3 bits)
- Outputs:
- Sequence found
While enabled, it should detect the following sequence of binary input values:
- 0b001
- 0b101
- 0b110
- 0b000
- 0b110
- 0b110
- 0b011
- 0b101
How would I write a design that meets these specifications?

'''

Iteration Log:

- **Iter 1**

  TF:

a) ... 'reset_n' is not a port of dut.
b) ... 'data' is not a port of dut.
c) ... 'sequence_found' is not a port of dut.
Root cause: Port-name mismatch.

- **Iter 2**

  SHF: “Port mismatch found—match with testbench.”
  TF: Error: Cycle 8, Expected: 1, Got: 0 (functional error—FSM transition misaligned with TB’s check).
  Result: *All test cases passed.*

Outcome

- **Status:** PASS
- **Key fixes:**
  - Ports aligned with TB
  - Corrected FSM next-state logic to assert sequence_found in the correct cycle.

Lesson(s) Learned

- After ports compile, remaining failures are usually **edge alignment** (when to assert pulse) or **partial-match fallback** in FSM.

Example 3 - Shift Register

Specification:

Inputs: clk, reset_n, shift_en, din;

Output: dout[7:0].

Initial Prompt:

'''

I am trying to create a Verilog model for a shift register. It must meet the following specifications:
- Inputs:
- Clock
- Active-low reset
- Data (1 bit)
- Shift enable
- Outputs:
- Data (8 bits)
How would I write a design that meets these specifications?

'''

Iteration Log

- **First pass:** *All test cases passed.*

Outcome

- **Status:** PASS

Example 4 - Dice Roller

Specification

Inputs: clk, reset_n, die_select[1:0], roll

Output: rolled_number (width up to 8 bits)

Behavior: On rising roll, output a value uniform over:

- 00→ d4 (1–4)
- 01→ d6 (1–6)
- 10→ d8 (1–8)
- 11→ d20 (1–20)

Initial Prompt:

'''

I am trying to create a Verilog model for a simulated dice roller. It must meet the following specifications:
- Inputs:
- Clock
- Active-low reset
- Die select (2-bits)
- Roll
- Outputs:
- Rolled number (up to 8-bits)

The design should simulate rolling either a 4-sided, 6-sided, 8-sided, or 20-sided die, based on the input die select. It should roll when the roll input goes high and output the random number based on the number of sides of the selected die.

How would I write a design that meets these specifications?

'''

Iteration Log (condensed)

- **Iter 1 TF:**

a) ... 'die_select' is not a port of dut.
b) ... 'rolled_number' is not a port of dut.
Root cause: Port-name mismatch.

- **Iter 2 TF:** Same port mismatch corrected; simulation proceeds.
  Result: Distribution tables printed for each die; TB completed successfully.

Outcome

- **Status:** PASS
- **Observation:** Counts across many rolls are approximately uniform for each die selection.

Lesson(s) Learned

- With randomized modules, validate **distribution** rather than single-cycle expected values; TB should run many iterations and summarize.

Example 5 — Token Bucket Rate Limiter

Specification

- **Module:** token_bucket
- **Parameters:**
    - DEN=16 (token fraction denominator)
    - RATE_NUM=3 (tokens added/cycle)
    - BURST_MAX=8 (max burst in requests)
    - TOKEN_COST=DEN (tokens per request)
- **Inputs:** clk, rst_n, req_i
- **Outputs:** grant_o, ready_o
- **Behavior:**
    - Token counter saturates at BURST_MAX*DEN
    - Add RATE_NUM tokens per cycle (up to cap)
    - If req_i and tokens >= TOKEN_COST: assert grant_o one cycle and subtract cost
    - ready_o high when tokens >= TOKEN_COST
    - Reset fills bucket (full capacity), grant_o=0

Initial Prompt:

'''

Write a Verilog module named token_bucket that implements a token bucket rate limiter.
Parameters:
DEN (default 16): denominator for fractional rate control (tokens per request).

RATE_NUM (default 3): number of tokens added per cycle.

BURST_MAX (default 8): maximum burst size, expressed in requests.

TOKEN_COST (default = DEN): tokens consumed per granted request.

Inputs:

clk: clock.

rst_n: active-low synchronous reset.

req_i: request input.

Outputs:

grant_o: asserted when a request is accepted and tokens are consumed.

ready_o: asserted whenever enough tokens are available for a request.

Functional Behavior:

Maintain an internal token counter (width at least 32 bits).

On every clock cycle, add RATE_NUM tokens, but do not exceed the maximum capacity (BURST_MAX * DEN).

If req_i is high and tokens ≥ TOKEN_COST:

Assert grant_o for one cycle.

Subtract TOKEN_COST tokens.

Otherwise, keep grant_o low.

ready_o is high whenever tokens ≥ TOKEN_COST.

On reset (rst_n = 0), initialize the token counter to full capacity (BURST_MAX * DEN) and clear grant_o.

'''

Iteration Log

- **First pass:** *All test cases passed.*

Outcome

- **Status:** PASS

Lesson(s) Learned

- Token arithmetic works reliably when using sufficiently wide counters (≥32-bit) and saturating adds.

Common Failure Modes & Fix Patterns

1. **Port mismatches (names or widths):**

- **Symptom:** “port <name> is not a port of dut/uut.”
- **Fix:** Mirror TB’s exact names/types; show a small *port map checklist* in prompt.

2. **Pulse timing / FSM edge cases:**

- **Symptom:** “Expected 1, Got 0” at a specific cycle.
- **Fix:** Revisit next-state & output assertions; ensure single-cycle pulse and correct overlap handling (if relevant).

3. **Reset behavior:**

- **Symptom:** Random initial states or sporadic mismatches at start.
- **Fix:** Define synchronous or asynchronous reset clearly; test reset release cycle.

4. **Parameterization & saturation:**

- **Symptom:** Counter overflow or underflow.
- **Fix:** Use saturating add/sub and explicit capacity clamps.

Compile and Run command

```
iverilog -o <output_executable>.vvp <design_file.v> <testbench_file.v>

vvp <output_executable>.vvp
```

New Learnings

1. **Error Feedback as Iterative Design Aid**

- ➢ Compiler and simulator errors (e.g., *"port not a member of dut"*) were not just obstacles but useful debugging signals.
- ➢ Treating error messages as structured feedback allowed the GenAI model to refine outputs effectively with minimal human input.

2. **Prompt Engineering Improves Reliability**

- ➢ Providing the LLM with structured constraints (module name, ports, reset type, timing expectations) produced more stable RTL code compared to open-ended prompts.
- ➢ This shows how **clear, constraint-driven prompts** reduce the number of iterations required.

3. **FSM Design Requires Explicit Output Timing**

- ➢ The sequence detector revealed that LLMs often mis-handle **pulse generation timing**.
- ➢ Explicitly stating *"assert for one cycle only when pattern completes"* led to correct designs, reinforcing the importance of temporal clarity in digital design specs.

4. **Importance of Distribution-Based Verification**

- ➢ For the dice roller, single-case verification was insufficient; statistical testing across many iterations was necessary to confirm correctness.
- ➢ This highlighted the value of **probabilistic validation** for randomized designs.

5. **Token Bucket as a Case Study in Rate Control**

- ➢ The project showed how counter width, saturation logic, and reset conditions directly impact long-term rate stability.
- ➢ Verifying average throughput matched theory reinforced understanding of **rate shaping** in hardware systems.

6. **Human-in-the-Loop Efficiency**

- ➢ Minimal but **targeted human feedback (SHF)** accelerated convergence. Instead of rewriting full prompts, small clarifications (e.g., "match testbench port names") guided the LLM efficiently.
- ➢ This demonstrates the synergy of **human domain knowledge + GenAI code generation**.

**GenAI Tools was used to summarize learnings and structure the report*