AutoChip Assignment Report

Course: LLM Chip Design **Student Name:** Fangyi Yu

Part I(a): AutoChip Trajectories

In this section, I document the generation trajectory for two Verilog designs using the AutoChip workflow.

Example 1: Sequence Detector ('1011')

1. Task Description Design a Finite State Machine (FSM) to detect the sequence 1011 in a serial input stream. The system must handle overlapping sequences
(e.g., 1011011 triggers twice).

2. AutoChip Parameters

- **Model:** gpt-3.5-turbo
- **Max Iterations:** 15 (Increased to allow for self-correction)
- **Temperature:** Default (as per script)

3. Initial Prompt

"Design a Verilog module named 'sequence_detector' that detects the sequence '1011'. Use a Finite State Machine."

4. Trajectory & Analysis The generation process required significant prompt engineering to resolve timing mismatches.

- **Iteration 0 (Vague Prompt):** The model generated a generic FSM. However, compilation failed because the port names did not match the testbench expectations (e.g., using reset instead of rst_n). **Rank: -1 (Compilation Error).**
- **Iteration 1-4 (Interface Fix, Logic Mismatch):** I updated the prompt to explicitly define the interface. The code compiled successfully (iverilog passed), but simulation failed with mismatches (Expected 1, Got 0).
    - **Analysis:** The testbench checks the output **immediately** (in the same cycle the input changes). The LLM inherently biased towards generating a **Moore Machine** (output depends only on state), which introduces a 1-cycle latency. The testbench required a **Mealy Machine**.
- **Iteration 5+ (Feedback Loop & Prompt Refinement):** The automated feedback loop passed the simulation error (Error: Input=1, Expected=1, Got=0) back to the

model. However, the model struggled to fix the architecture purely based on error logs.
- **Final Fix (One-Shot Prompting):** I modified the prompt to explicitly strictly require a **Mealy Machine** and provided a **One-Shot Example** (a template for a '101' detector). This guided the LLM to implement the output as combinational logic (assign out = (state == S_101 && in == 1)).
- **Result:** The model successfully generated a correct Mealy FSM that passed all test cases.

Example 2: 5-bit Binary to BCD Converter

1. Task Description Convert a 5-bit binary input into two BCD digits (Tens, Ones) using the Double Dabble algorithm or combinational logic.

2. AutoChip Parameters

- **Model:** gpt-4o-mini (Based on logs)
- **Max Iterations:** 10

3. Initial Prompt

"Write a 5-bit Binary to BCD Converter. Input: [4:0] binary_input. Output: [7:0] bcd_output. Algorithm: Double Dabble."

4. Trajectory & Analysis

- **Iteration 0 (Syntax/Format Error):** As seen in the logs (binary.pdf), the simulation output parsing initially failed. The model likely produced Verilog code that compiled but didn't output the simulation results in the format the Python script expected (e.g., missing specific $display statements in the testbench or module).
- **Iteration 1 (Functional Errors):** The simulation ran but reported functional failures: Error: Test case 5 failed. Expected BCD: 8'b101, Got: 8'b1000.
    - **Analysis:** The "Double Dabble" algorithm (shift and add 3 if > 4) is tricky to implement in Verilog using always blocks without proper loop unrolling. The model initially miscalculated the "Add 3" condition for specific binary values (like 28 or 31).
- **Auto-Correction:** AutoChip fed the mismatch error back to gpt-4o-mini. The model corrected the logic by adjusting the if (temp > 4) temp = temp + 3; logic inside the conversion loop.
- **Result:** After iterations, the logic converged, and the design passed verification.

Part I(b): Manual RTL Design & Explanation

For this part, I manually designed the **Sequence Detector** to demonstrate the correct implementation of the timing requirements that the LLM initially struggled with.

1. Manual Design Choices

The key challenge in this assignment was the **timing of the output signal**.

- **Moore vs. Mealy:** The provided testbench checks out at time + 1 after the input changes, *before* the next clock edge. This necessitates a **Mealy Machine**, where the output is a function of the current state AND the current input.
- **State Encoding:** I used 4 states (S_IDLE, S_1, S_10, S_101). I did *not* use a state for S_1011 because the output needs to go high *during* the transition from S_101 back to S_1 (or S_10).
- **Overlap Handling:** Upon detecting 1011, if the input is 1, the state transitions to S_1 (treating the last 1 as the start of a new sequence).

```
`timescale 1ns / 1ps

module sequence_detector (

    input wire clk,

    input wire rst_n,

    input wire in,

    output reg out

);

    // State Definitions

    localparam [1:0] S_IDLE = 2'd0,

                     S_1    = 2'd1,

                     S_10   = 2'd2,

                     S_101  = 2'd3;

    reg [1:0] state, next_state;
```

```
// Sequential Logic: State Update

always @(posedge clk or negedge rst_n) begin

    if (!rst_n)

        state <= S_IDLE;

    else

        state <= next_state;

end


// Combinational Logic: Next State & Output (Mealy)

always @(*) begin

    // Defaults to prevent latches

    next_state = S_IDLE;

    out = 0;


    case (state)

        S_IDLE: next_state = (in) ? S_1 : S_IDLE;


        S_1:    next_state = (in) ? S_1 : S_10;


        S_10:   next_state = (in) ? S_101 : S_IDLE;


        S_101: begin

            if (in) begin

                // Sequence 1011 detected!
```

```
                    // Output goes high IMMEDIATELY (Mealy behavior)

                    out = 1;

                    next_state = S_1; // Handle Overlap: 1011...1 is the start of next

                end else begin

                    // Sequence 1010

                    out = 0;

                    next_state = S_10;

                end

            end

            default: next_state = S_IDLE;

        endcase

    end

endmodulend-Written RTL Code (sequence_detector_manual.v)
```