Lab Report-AutoChip Colab

Course/Lab: Lab 2-AutoChip Colab

Student: D. Akhash Krishna

NetID: AD7252

Date: February 13th, 2026

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

Tools and Settings:

LLM model(s) & version(s): `arcee-ai/trinity-large-preview:free`

Interface: Google Colab (Python SDK via `openai` client with

`base_url="https://openrouter.ai/api/v1"`)

Simulation: Icarus Verilog for compile/sim

\

PART 1

a) Example 1-Binary to BCD Converter

Initial Prompt:

/*

Create a Verilog module named top_module that is compatible with iverilog.

The module should have a 5-bit input, binary_input, to handle values from 0 to 31.
It must also have an 8-bit output, bcd_output, where the most significant nibble (bits 7 down to 4) represents the tens digit and the least significant nibble (bits 3 down to 0) represents the units digit of the decimal equivalent.

The implementation should use division and modulo operations to perform the conversion.
The intermediate results should be assigned to wires, and the final bcd_output should be a concatenation of the lower 4 bits of the tens and units wires.

The module declaration should be as follows:

*/

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

// Module body goes here

endmodule
```

Important AutoChip parameters:

"model_family": "ChatGPT",

"model_id": "arcee-ai/trinity-large-preview:free",

"num_candidates": 5,

"iterations": 5,

"outdir": "outputs_binary_to_bcd",

"ensemble": false

Trajectory explanation:

The module uses division and modulo to convert the binary input to BCD.

Intermediate results (tens and units) are stored in wires.

The final BCD output is the concatenation of tens and units.

The module is compatible with Icarus Verilog (iverilog) for simulation.

b) Example 5: Dice Roller

Initial Prompt:

/* You are generating synthesizable Verilog-2001.

Output ONLY the complete Verilog-2001 design module named top_module as raw code, ending with endmodule.
Do not include any markdown, prose, comments, attributes, `timescale, or a testbench.
Produce exactly one module.

Functional spec (must match exactly):

- Module:
  module top_module (
      input wire   clk,
      input wire   rst_n,   // active-low reset
      input wire [1:0] die_select,   // 00=4-sided, 01=6-sided, 10=8-sided, 11=20-sided
      input wire   roll,   // roll trigger
      output reg [7:0] rolled_number   // 1..20
  );

- Internals and behavior:
  * Declare: reg [7:0] lfsr;
  * Rising-edge detect on roll:
      reg roll_prev;
      wire roll_rising_edge = roll & ~roll_prev;
  * LFSR feedback taps (exactly these bits): wire feedback = lfsr[7] ^ lfsr[6] ^ lfsr[5] ^ lfsr[4];
  * Single sequential block: always @(posedge clk or negedge rst_n)
      - If (!rst_n):
          lfsr   <= 8'h01;   // non-zero seed

```
        rolled_number <= 8'h00;
        roll_prev     <= 1'b0;
    - Else:
        roll_prev <= roll;
        // free-run LFSR every cycle so value changes between rolls
        lfsr <= {lfsr[6:0], feedback};
        // on rising edge of roll, update output once
        if (roll_rising_edge) begin
            case (die_select)
                2'b00: rolled_number <= (lfsr % 8'd4)  + 8'd1;   // 1..4
                2'b01: rolled_number <= (lfsr % 8'd6)  + 8'd1;   // 1..6
                2'b10: rolled_number <= (lfsr % 8'd8)  + 8'd1;   // 1..8
                2'b11: rolled_number <= (lfsr % 8'd20) + 8'd1;   // 1..20
                default: rolled_number <= 8'h00;
            endcase
        end
  * Use nonblocking assignments (<=) in the sequential block.
  * Do not use $random or system tasks.

Close the module with endmodule. Output only that single module.

*/
```

Important AutoChip parameters:

"model_family": "ChatGPT",
"model_id": "arcee-ai/trinity-large-preview:free",
"num_candidates": 5,
"iterations": 5,
"outdir": "outputs_dice_roller",
"ensemble": false

Trajectory explanation:

Module Inputs/Outputs:

Inputs: clk, rst_n (reset), die_select (die type), and roll (trigger).

Output: rolled_number (8-bit value representing the rolled die number).

Internal Registers:

lfsr: 8-bit register for the pseudo-random number.

roll_prev: Stores the previous state of roll for rising-edge detection.

Feedback: Calculated from selected bits in lfsr.

Sequential Block:

Resets lfsr and rolled_number on rst_n.

Updates lfsr and rolled_number on the rising edge of clk, based on the roll signal and die_select:

For each die_select value, rolled_number is calculated using lfsr % (sides of die) + 1.

Nonblocking Assignments: Ensures proper sequential logic behavior with the <= operator.

This module simulates a pseudo-random die roll, with the number generated based on the selected die size (4, 6, 8, or 20 sides).

b) Manual RTL

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

    // Wires to hold the intermediate results of the conversion
    wire [3:0] tens;
    wire [3:0] units;

    // Calculate tens and units using division and modulo operations
    assign tens = binary_input / 10;
    assign units = binary_input % 10;

    // Concatenate tens and units to form the final BCD output
    assign bcd_output = {tens, units};

endmodule
```

Explanation:

The module receives a 5-bit binary input (binary_input).
It calculates the **tens** and **units** of the decimal equivalent of the binary input:

- **Tens** are calculated by dividing the binary value by 10.
- **Units** are calculated by taking the modulus of the binary value with 10.
  The final **BCD output** is the concatenation of the tens and units.