

ChipChat_Report

Summary

The first example i choose is the Example 1: binary_to_bcd

The second example i choose is the Example 5: Dice_Roller

The extension example i choose is the Example 8: Traffic Light

Firstly: Binary_to_bcd:

1、 The Generation Parameter:

```
completion = client.chat.completions.create(  
    model = "gpt-4o-mini",  
    messages=[{"role": "user", "content": verilog_generation_prompt}],  
    max_tokens=1024, # limits the maximum number of tokens (words or pieces of words) that the model will generate in its response.  
    stream=False  
)
```

As you can see in this picture , the model is “gpt-4o-mini” The max_tokens is 1024

2、 The Final Prompt i use:

verilog_generation_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?

, , ,

3、 The Output Module:

```
output_verilog_code = '''
module binary_to_bcd_converter (
    input wire [4:0] binary_input, // 5-bit binary input
    output reg [7:0] bcd_output     // 8-bit BCD output (4 bits for tens, 4 bits for ones)
);

// Internal registers for BCD representation
reg [3:0] bcd_tens; // Tens place
reg [3:0] bcd_ones; // Ones place
integer i; // Loop variable

always @* begin
    // Initialize BCD values to 0
    bcd_tens = 4'b0000;
    bcd_ones = 4'b0000;

    // Convert binary to BCD using double-dabble algorithm
    for (i = 0; i < 5; i = i + 1) begin
        // Shift left the BCD digits to make room for the next binary bit
        if (bcd_tens >= 5)
            bcd_tens = bcd_tens + 4'b0011; // Add 3 to tens place if necessary

        if (bcd_ones >= 5)
            bcd_ones = bcd_ones + 4'b0011; // Add 3 to ones place if necessary

        // Shift BCD values left to add a new binary bit
        {bcd_tens, bcd_ones} = {bcd_tens, bcd_ones} << 1;

        // Add the next bit of the binary input
        bcd_ones[0] = binary_input[4 - i]; // Add the next bit of the binary input
    end

    // Combine BCD places into the final output
    bcd_output = {bcd_tens, bcd_ones};
end

endmodule
'''
```

The Design explanation:

So basically the llm model gave out the final results which can perfectly fit the testbench.

The module interfaces includes the input signal which is 5bits wire binary_inputs and 8bits reg bcd_output.

And it defines two bc_tens and bcd_ones interval variables and also the interger i for the loop exclusively.

Standard binary counters carry over at 16 (2^4), but decimal (BCD) must carry over at 10. To fix this "gap," the algorithm follows two rules: Check: Before shifting, if any BCD digit is ≥ 5 , add 3. Shift: Shift everything left by 1 bit to process the next binary digit.

Secondly: Dice_Roller

1、 The Generation Parameter:

```
completion = client.chat.completions.create(  
    model = "gpt-4o-mini",  
    messages=[{"role": "user", "content": verilog_generation_prompt}],  
    max_tokens=1024, # limits the maximum number of tokens (words or pieces of words) that the model will generate in its response.  
    stream=False  
)
```

As you can see in this picture , the model is “gpt-4o-mini” The max_tokens is 1024

2、 The Final Prompt i use:

```
verilog_generation_prompt = '''  
I am trying to create a Verilog model for a simulated dice roller. It must meet the following specifications:  
  
- Inputs:  
    - clk  
    - Active-low rst_n  
    - die_select (2-bits)  
    - roll  
  
- Outputs:  
    - rolled_number (up to 8-bits)  
And the internal signal should be reg  
and try to keep the always sequential logic and combination logic  
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?  
'''
```

3、The Output Module:

```
output_verilog_code = '''
module dice_roller (
    input wire clk,
    input wire rst_n,
    input wire [1:0] die_select, // 00 for 4-sided, 01 for 6-sided, 10 for 8-sided, 11 for 20-sided
    input wire roll,
    output reg [7:0] rolled_number // Up to 8-bits
);

    // Pseudorandom number generator state
    reg [4:0] lfsr; // A 5-bit LFSR can provide enough pseudorandom results for our needs
    reg [3:0] max_value; // Maximum value based on die selection
    reg roll_triggered; // To handle roll output only once per trigger

    // Combinational logic for setting max_value based on die selection
    always @(*) begin
        case (die_select)
            2'b00: max_value = 4'd4; // 4-sided die
            2'b01: max_value = 4'd6; // 6-sided die
            2'b10: max_value = 4'd8; // 8-sided die
            2'b11: max_value = 4'd20; // 20-sided die
            default: max_value = 4'd0;
        endcase
    end

    // Sequential logic for LFSR and random number generation
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            lfsr <= 5'b10001; // Initial value for LFSR
            rolled_number <= 0;
            roll_triggered <= 0;
        end else begin
            // LFSR feedback
            lfsr <= {lfsr[3:0], lfsr[4] ^ lfsr[2]}; // Example feedback polynomial

            if (roll && !roll_triggered) begin
                // Use LFSR to generate a random number within the range of the die
                rolled_number <= (lfsr % max_value) + 1; // Ensure a range starting from 1
                roll_triggered <= 1; // Mark that roll was triggered
            end else if (!roll) begin
                // Reset the roll_triggered when roll signal goes low
                roll_triggered <= 0;
            end
        end
    end
endmodule
'''
```

The design approach:

The module uses `die_select` to determine the maximum dice value (4, 6, 8, or 20). The LFSR continuously updates every clock cycle to provide pseudorandom values. When the roll signal is asserted, and no previous roll has been triggered, the current LFSR value is mapped to the valid dice range and assigned to `rolled_number`. The `roll_triggered` signal ensures that only one dice roll occurs per roll assertion, and the output remains unchanged until a new roll trigger occurs.

```

cd dice_roller/ && iverilog -o dice_roller.vvp dice_roller.v dice_roller_tb.v && vvp dice_roller.vvp
)

dice_roller.v:21: warning: Numeric constant truncated to 4 bits.
VCD info: dumpfile my_design.vcd opened for output.
Results for die_select 00:
  Rolled 1: 225 times
  Rolled 2: 261 times
  Rolled 3: 257 times
  Rolled 4: 257 times
Results for die_select 01:
  Rolled 1: 161 times
  Rolled 2: 193 times
  Rolled 3: 162 times
  Rolled 4: 162 times
  Rolled 5: 161 times
  Rolled 6: 161 times
Results for die_select 10:
  Rolled 1: 97 times
  Rolled 2: 129 times
  Rolled 3: 128 times
  Rolled 4: 128 times
  Rolled 5: 129 times
  Rolled 6: 129 times
  Rolled 7: 130 times
  Rolled 8: 130 times
Results for die_select 11:
  Rolled 1: 226 times
  Rolled 2: 258 times
  Rolled 3: 259 times
  Rolled 4: 257 times
Testbench completed successfully.

```

For the Part II:

The Table looks as below:

Iteration times	Issue_observed	Fix_applied	outcome	What_fail	Why i expext it can work
1	<p>dice_roller_tb.v:10: error: port ``die_select" is not a port of dut.</p> <p>1 error(s) during elaboration.</p>	I changed the original port die_sel to the die_select.	<p>dice_roller_tb.v:10: error: port ``die_select" is not a port of dut.</p> <p>1 error(s) during elaboration.</p>	The port name die_select is not correspondi ng	Because the dut in the testbench is not corresponding with the .v document
2	<p>1、The rst_n is somehow different name which is not corresponding to the .v documnet</p> <p>2、The max_value which is the internal signal which should be reg type instead of the wire type</p>	<p>I change the reset_n in the testbench document and make it align with the testbench.</p> <p>I change the max_value type from wire to reg</p>	<p>dice_roller_tb.v:10: error: port ``rst_n" is not a port of dut.</p> <p>dice_roller.v:17: error: max_value is not a valid l-value in tb_dice_roller.dut.</p> <p>dice_roller.v:12: : max_value is declared here as wire.</p> <p>dice_roller.v:18: error: max_value is not a valid l-value in tb_dice_roller.dut.</p> <p>dice_roller.v:12: : max_value is declared here as wire.</p> <p>dice_roller.v:19: error: max_value is not a valid l-value in tb_dice_roller.dut.</p> <p>dice_roller.v:12: :</p>	<p>The signal name and type which has a conflict with the dut module and some violations with the verilog rules</p>	<p>Because the name is not corresponding which cant make the compiler to locate the signal which is a big problem.</p> <p>Also, max_value is a value should be restore in the registers so basically you cant just using the assign in the always module . I mean you can use the wire signal , but you cant make it assign value in always module.</p>

			<p>max_value is declared here as wire.</p> <p>dice_roller.v:20: error: max_value is not a valid l-value in tb_dice_roller.dut.</p> <p>dice_roller.v:12: : max_value is declared here as wire.</p> <p>dice_roller.v:21: error: max_value is not a valid l-value in tb_dice_roller.dut.</p> <p>dice_roller.v:12: : max_value is declared here as wire.</p> <p>6 error(s) during elaboration.</p>																								
3	The signal called nreset still regenerated wrong	I do a iteration again	module dice_roller (input wire clk, input wire nreset, // Active-low reset input wire [1:0] die_select, // 00 - 4-sided, 01 - 6-sided, 10 - 8-sided, 11 - 20-sided input wire roll, // Signal to roll the die output reg [7:0] rolled_number // 8-bit output to represent rolled number);	The name of the signal is not corresponding	The logic as i said above																						
4	Right now its finally work	Change the .v file by generating new prompt and apply into the file	<p>dice_roller.v:21: warning: Numeric constant truncated to 4 bits.</p> <p>VCD info: dumpfile my_design.vcd opened for output.</p> <p>Results for die_select 00:</p> <table><tr><td>Rolled</td><td>1:</td></tr><tr><td>225 times</td><td></td></tr><tr><td>Rolled</td><td>2:</td></tr><tr><td>261 times</td><td></td></tr><tr><td>Rolled</td><td>3:</td></tr><tr><td>257 times</td><td></td></tr><tr><td>Rolled</td><td>4:</td></tr><tr><td>257 times</td><td></td></tr></table> <p>Results for die_select 01:</p> <table><tr><td>Rolled</td><td>1:</td></tr><tr><td>161 times</td><td></td></tr><tr><td>Rolled</td><td>2:</td></tr></table>	Rolled	1:	225 times		Rolled	2:	261 times		Rolled	3:	257 times		Rolled	4:	257 times		Rolled	1:	161 times		Rolled	2:	N/A	N/A
Rolled	1:																										
225 times																											
Rolled	2:																										
261 times																											
Rolled	3:																										
257 times																											
Rolled	4:																										
257 times																											
Rolled	1:																										
161 times																											
Rolled	2:																										

			193 times		
			Rolled	3:	
			162 times		
			Rolled	4:	
			162 times		
			Rolled	5:	
			161 times		
			Rolled	6:	
			161 times		
			Results for die_select 10:		
			Rolled	1:	
			97 times		
			Rolled	2:	
			129 times		
			Rolled	3:	
			128 times		
			Rolled	4:	
			128 times		
			Rolled	5:	
			129 times		
			Rolled	6:	
			129 times		
			Rolled	7:	
			130 times		
			Rolled	8:	
			130 times		
			Results for die_select 11:		
			Rolled	1:	
			226 times		
			Rolled	2:	
			258 times		
			Rolled	3:	
			259 times		
			Rolled	4:	
			257 times		
			Testbench	completed	
			successfully.		

Extention: Traffic_Light

The Generation file:

```
from openai import OpenAI

client = OpenAI(
    api_key = "sk-proj-4V8chbXUGbelQs5eCkrm2Hv393a5Ma6hs1F1wJ2eme0i3m#DnHQ6pYfqq6Glnvq4UxkPfw32Qf3BlbkfJLH85GnTzG1Rd_E_HDSmmWvCx25fpsvqz-#HqQoXlDFulbZt_6A7QkUp9tUNFZyqdTD_#FQU2wA"
)

completion = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "user", "content": verilog_generation_prompt}],
    max_tokens=1024, # limits the maximum number of tokens (words or pieces of words) that the model will generate in its response.
    stream=False
)

print(completion.choices[0].message.content)
```

The final prompt for the extension:

```
verilog_generation_prompt = '''

module traffic_light_fsm(
    input wire clk,           // Clock input
    input wire reset_n,       // Active-low reset
    input wire enable,        // Enable signal
    output reg red,           // Red light output
    output reg yellow,        // Yellow light output
    output reg green          // Green light output
);

// State encoding
parameter RED = 2'b00;
parameter GREEN = 2'b01;
parameter YELLOW = 2'b10;

reg [1:0] current_state, next_state;

// Timers for each state
reg [5:0] counter; // 6 bits can count up to 63 (max timer count needed)

// Combinational logic for state transitions
always @(*) begin
    // Default assignments
    next_state = current_state;

    case (current_state)
        RED: begin
            if (counter == 31) begin // Changed from 32 to 31
                next_state = GREEN;
            end
        end

        GREEN: begin
            if (counter == 19) begin // Changed from 20 to 19
                next_state = YELLOW;
            end
        end

        YELLOW: begin
            if (counter == 6) begin // Changed from 7 to 6
                next_state = RED;
            end
        end
    endcase
end
```



```

        YELLOW: begin
            if (counter == 6) begin // Changed from 7 to 6
                next_state = RED;
            end
        end
    endcase
end

// Sequential logic to handle the clock and reset
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        // Reset the state machine to RED and timer
        current_state <= RED;
        counter <= 0;
        {red, yellow, green} <= 3'b100; // Red ON, others OFF
    end
    else if (enable) begin
        // Update counter based on current state
        counter <= counter + 1;

        // Update outputs based on current state
        case (current_state)
            RED: begin
                {red, yellow, green} <= 3'b100; // Red ON
            end
            GREEN: begin
                {red, yellow, green} <= 3'b001; // Green ON
            end
            YELLOW: begin
                {red, yellow, green} <= 3'b010; // Yellow ON
            end
        endcase

        // State transition
        if (next_state != current_state) begin
            current_state <= next_state;
            counter <= 0; // Reset the counter for the next state
        end
    end
end

endmodule

now "Act as a Verilog engineer. I need to modify the traffic_light_fsm. Spec Change Requirements:

Add an input emergency_btn.

Add a new state EMERGENCY_RED.

If emergency_btn is high:

From GREEN, transition to YELLOW normally, then instead of going to GREEN next, go to EMERGENCY_RED.

From RED or YELLOW, transition to EMERGENCY_RED.

In EMERGENCY_RED, the output is red=1, and the FSM stays here as long as emergency_btn is high.

When emergency_btn goes low, return to the normal RED state and restart the timer. Please provide the Verilog-2012 code."
...

```

For the prompt , i added additional information about the logic for the traffic operation logic and add a specific design to the module like adding an emergency red light

Add an input emergency_btn.

Add a new state EMERGENCY_RED.

If emergency_btn is high:

From GREEN, transition to YELLOW normally, then instead of going to GREEN next, go to EMERGENCY_RED.

From RED or YELLOW, transition to EMERGENCY_RED.

In EMERGENCY_RED, the output is red=1, and the FSM stays here as long as emergency_btn is high.

When emergency_btn goes low, return to the normal RED state and restart the timer. Please provide the Verilog-2012 code."

```
!cd traffic_light/ && iverilog -o traffic_light.vvp traffic_light.v traffic_light_tb.v && vvp traffic_light.vvp

Test case 1 passed - FSM transitioned to RED after 0 clock cycles.
Test case 2 passed - FSM transitioned to GREEN after 32 clock cycles.
Test case 3 passed - FSM transitioned to YELLOW after 20 clock cycles.
Test case 4 passed - FSM transitioned to RED after 7 clock cycles.
All test cases passed!
```

This is for the basic traffic_light

```
!cd traffic_light/ && iverilog -o traffic_light.vvp traffic_light.v traffic_light_tb2.v && vvp traffic_light.vvp

Case 1 passed - FSM in correct state (R:1 Y:0 G:0).
Case 2 passed - FSM in correct state (R:0 Y:0 G:1).
Case 3 passed - FSM in correct state (R:0 Y:1 G:0).
Simulating Emergency Mode...
Case 4 passed - FSM in correct state (R:1 Y:0 G:0).
Checking if locked in RED during emergency...
Case 5 passed - FSM in correct state (R:1 Y:0 G:0).
Emergency released, waiting for next transition...
Case 6 passed - FSM in correct state (R:0 Y:0 G:1).
-----
SUCCESS: All extension test cases passed!
-----
```

The final pass results for the emergency function testbench.v

```
module traffic_light_fsm(
    input wire clk,           // Clock input
    input wire reset_n,       // Active-low reset
    input wire enable,        // Enable signal
    input wire emergency_btn,  // Emergency button
    output reg red,           // Red light output
    output reg yellow,        // Yellow light output
    output reg green          // Green light output
);
```

Design Implementation and Timing Strategy:

"In the revised verilog2.v, I implemented a look-ahead state transition logic to strictly adhere to the 32-cycle specification. The core logic is partitioned into a dedicated combinational block and a sequential block to ensure precise timing:

Combinational Logic (Condition Detection): I placed the state transition and light detection logic in a combinational block. When the counter reaches 31 (during the 32nd clock cycle), the next_state is immediately evaluated and set to GREEN. This allows the FSM to 'prepare' the transition before the next clock arrival.

Sequential Logic (State Update): The actual status update occurs on the rising edge of the 32nd clock (which marks the beginning of the 33rd cycle). On this edge, the current_state is updated from RED to GREEN via non-blocking assignments.

By updating the next_state during the 32nd cycle and committing it to the register at the 32nd clock edge, the design ensures that the RED light occupies exactly 32 full clock periods. The transition to GREEN is thus visible and stable at the start of the 33rd cycle, perfectly aligning with the simulation behavior observed in the testbench."