DSD F 18

Lab 7 State Machine Chart with ROM

Name_____Andrew Quick_____ abc123 ___qxt050___ Date_____10/9/2024_____

1. What are the Inputs to the ROM and how many rows must the ROM have?

Since 5 input bits (3 user inputs and 2 state inputs) can represent 2^5=32 combinations, the ROM must have 32 rows.

2. How many outputs and how many columns must the ROM have?

5 (2 for the next state and 3 for the outputs), therefore 5 columns.

3. How many effective address lines?

5 address lines, 3 user inputs and 2 state inputs.

4. Write the equations you could use if you implemented it with 2 FF and coded the 3 States 00, 01 and 10 for A and B FF.

Flip-Flop Input DA: DA=A′·X2
Flip-Flop Input DB: DB=(A′·B′·X1′)+(A′·B′·X1·X3′)+(A·B′)
Output Z3: Z3=(A·X1)+(A·B′·X2)
Output Z2: Z2=(A′·B·X3′)+(A·X1)
Output Z1: Z1=(A·X2·X3)+(A′·B′·X1)

5. Did you provide for Q and Q' outputs from your FF's? Why would this be helpful?

This would be helpful for debugging and simpler logic equations.

5. Which method required more resources on the FPGA (check the usage report)?

ROM uses more memory cells. Logic equations use more LUTs and FFs.

6. Which method did you find was easier to do?

ROM

7. Paste in your code.

`timescale 1ns/10ps

module counting_state_machine

#

```
(
 parameter NUM_SEGMENTS = 8,
 parameter CLK_PER     = 10,   // Clock period in ns
 parameter REFR_RATE    = 1000, // Refresh rate in Hz
 parameter ASYNC_BUTTON = "SAFE" // "CLOCK", "NOCLOCK", "SAFE", "DEBOUNCE"
 )
(
 input wire                 clk,
 input wire                 BTNC,    // Change state
 input wire                 CPU_RESETN,
 input wire [2:0]            SW,      // Switch inputs: SW[15:11] for address, SW[7:0] for data
 output logic [NUM_SEGMENTS-1:0] anode,
 output logic [7:0]           cathode
 );

logic [NUM_SEGMENTS-1:0][3:0]      encoded;
logic [NUM_SEGMENTS-1:0]          digit_point;
logic                         reset;

seven_segment
  #
  (
   .NUM_SEGMENTS (NUM_SEGMENTS),
   .CLK_PER     (CLK_PER),   // Clock period in ns
   .REFR_RATE    (REFR_RATE)  // Refresh rate in Hz
   )
u_7seg
  (
```

```
  .clk        (clk),
  .reset      (reset),
  .encoded    (encoded),
  .digit_point  (digit_point),
  .anode      (anode),
  .cathode    (cathode)
  );


// Capture the rising edge of button press
logic                   last_button;
logic                   button;
(* mark_debug = "true" *) logic    button_down;

initial begin
  last_button = 1'b0;      // Initialize single-bit signals
  button     = 1'b0;
  button_down = 1'b0;
end

generate
  if (ASYNC_BUTTON == "SAFE") begin : g_CLOCK
    (* ASYNC_REG = "TRUE", mark_debug = "true" *) logic [2:0] button_sync;

    always @(posedge clk) begin
      // BTNC handling (Move to the next state)
      button_down <= 1'b0;  // Reset button_down
      button_sync <= {button_sync[1:0], BTNC};  // Shift in BTNC state
```

```
    if (button_sync[2:1] == 2'b01)
      button_down <= 1'b1;  // Button pressed
    else
      button_down <= 1'b0;  // No button press detected
  end

 end else begin : g_NOCLOCK
  always @(posedge clk) begin
    last_button <= button;
    button     <= BTNC;
    if (BTNC & ~last_button)
      button_down <= 1'b1;  // Button pressed (detect rising edge)
    else
      button_down <= 1'b0;
  end
 end
endgenerate

// Define states
parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10;

// State variables
reg [1:0] state, next_state;
logic [2:0] Z;  // Output from state machine to encoded

// Use SW[2:0] as input for state transitions
wire [2:0] sw_input = SW[2:0];  // Use the lowest 3 bits of SW for state transitions
```

```
// Button handling logic
(* ASYNC_REG = "TRUE" *) logic [2:0] button_sync;

// State machine next state logic
always @(posedge clk or negedge CPU_RESETN) begin
  if (!CPU_RESETN) begin
    state <= S0;  // Start in state S0 on reset
  end else if (button_down) begin
    state <= next_state;
  end
end

initial begin
  encoded     = '0;
  digit_point = '1;
end

// State transition logic
always @(*) begin
  case (state)
    S0: begin
      case (sw_input)
        3'b000: begin
          next_state = S1;
          Z = 3'b010;
        end
        3'b001: begin
          next_state = S1;
```

```
 Z = 3'b010;
end
3'b010: begin
 next_state = S2;
 Z = 3'b101;
end
3'b011: begin
 next_state = S1;
 Z = 3'b001;
end
3'b100: begin
 next_state = S1;
 Z = 3'b010;
end
3'b101: begin
 next_state = S1;
 Z = 3'b010;
end
3'b110: begin
 next_state = S2;
 Z = 3'b101;  // Set Z based on transition for S0
end
3'b111: begin
 next_state = S2;
 Z = 3'b001;
end
default: begin
 next_state = S0;
```

```
    end
  endcase
end
S1: begin
  Z = 3'b001;
  case (sw_input)
    3'b000: next_state = S1;
    3'b001: next_state = S1;
    3'b010: next_state = S2;
    3'b011: next_state = S2;
    3'b100: next_state = S1;
    3'b101: next_state = S1;
    3'b110: next_state = S2;
    3'b111: next_state = S2;
    default: next_state = S1;
  endcase
end
S2: begin
  Z = 3'b001;
  case (sw_input)
    3'b000: next_state = S1;
    3'b001: next_state = S0;
    3'b010: next_state = S1;
    3'b011: next_state = S0;
    3'b100: next_state = S1;
    3'b101: next_state = S0;
    3'b110: next_state = S1;
    3'b111: next_state = S0;
```

```
        default: next_state = S2;
      endcase
    end
    default: next_state = S0;
  endcase
end

// Seven-segment display encoding
always @(posedge clk) begin
  if (button_down) begin
    encoded[4] <= Z;
    encoded[0] <= next_state;
  end
  if (!CPU_RESETN) begin
    encoded <= 4'b0000;  // Reset encoded display
  end
end

endmodule

// seven_segment.sv
//
// Encapsulate multiple seven segment displays using the cathode driver plus an
// anode driver.
`timescale 1ns/10ps
module seven_segment
 #
 (
```

```
 parameter NUM_SEGMENTS = 8,
 parameter CLK_PER     = 10,  // Clock period in ns
 parameter REFR_RATE    = 1000 // Refresh rate in Hz
 )
(
 input wire                    clk,
 input wire                    reset, // active high reset
 input wire [NUM_SEGMENTS-1:0][3:0] encoded,
 input wire [NUM_SEGMENTS-1:0]     digit_point,
 output logic [NUM_SEGMENTS-1:0]    anode,
 output logic [7:0]              cathode
 );

localparam INTERVAL = int'(100000000 / (CLK_PER * REFR_RATE));

logic [$clog2(INTERVAL)-1:0]       refresh_count;
logic [$clog2(NUM_SEGMENTS)-1:0]    anode_count;
logic [NUM_SEGMENTS-1:0][7:0]      segments;

cathode_top ct[NUM_SEGMENTS]
  (
   .clk       (clk),
   .encoded    (encoded),
   .digit_point(digit_point),
   .cathode    (segments)
   );

initial begin
```

```
  refresh_count = '0;
  anode_count   = '0;
 end

 always @(posedge clk) begin
  if (refresh_count == INTERVAL) begin
   refresh_count        <= '0;
   anode_count          <= anode_count + 1'b1;
  end else refresh_count <= refresh_count + 1'b1;
  anode                 <= '1;
  anode[anode_count]     <= '0;
  cathode               <= segments[anode_count];
  if (reset) begin
   refresh_count        <= '0;
   anode_count          <= '0;
  end
 end

endmodule

// cathode_top.sv
// -------------------------------------
// Drive the cathodes of 7 segment display
// -------------------------------------
//
// input the encoded value from 0-F and generate the cathode signals
`timescale 1ns/10ps
module cathode_top
```

```
(
 input wire        clk,
 input wire [3:0]   encoded,
 input wire        digit_point,
 output logic [7:0] cathode
 );

always_ff @(posedge clk) begin
  cathode[7] <= digit_point;
  case (encoded)
    4'h0: cathode[6:0] <= 7'b1000000;
    4'h1: cathode[6:0] <= 7'b1111001;
    4'h2: cathode[6:0] <= 7'b0100100;
    4'h3: cathode[6:0] <= 7'b0110000;
    4'h4: cathode[6:0] <= 7'b0011001;
    4'h5: cathode[6:0] <= 7'b0010010;
    4'h6: cathode[6:0] <= 7'b0000010;
    4'h7: cathode[6:0] <= 7'b1111000;
    4'h8: cathode[6:0] <= 7'b0000000;
    4'h9: cathode[6:0] <= 7'b0010000;
    4'hA: cathode[6:0] <= 7'b0001000;
    4'hB: cathode[6:0] <= 7'b0000011;
    4'hC: cathode[6:0] <= 7'b1000110;
    4'hD: cathode[6:0] <= 7'b0100001;
    4'hE: cathode[6:0] <= 7'b0000110;
    4'hF: cathode[6:0] <= 7'b0001110;
  endcase
end
```

```
endmodule
```