Fpga - HW3

University: Shiraz University
Course: FPGA
Student: Parsa Haghighatgoo – 40030644
Assignment Title: Fpaga - HW3

1 Question One: Divisibility by 3 LSB-first Mealy FSM

Design a finite state machine (FSM) that receives the bits of a binary number **sequentially starting from the LSB** (least significant bit). The FSM must output:

$$\texttt{REM} = \begin{cases} 0 & \text{if the number is divisible by 3} \\ 1 & \text{otherwise} \end{cases}$$

The implementation must be **Mealy form**, and the Verilog code must not use the modulo operator (`%`) in the synthesizable design.

2 Key Idea and Mathematical Background

Let the input bits arrive as $b_0, b_1, b_2, \ldots$ where b_0 is the LSB. After receiving k bits, the partial value is:

$$N_k = \sum_{i=0}^{k-1} b_i 2^i$$

We need to track the remainder:

$$r_k = N_k \bmod 3$$

When the next bit b_k arrives, the new value becomes:

$$N_{k+1} = N_k + b_k 2^k \quad \Rightarrow \quad r_{k+1} = (r_k + b_k(2^k \bmod 3)) \bmod 3$$

Since:

$$2^0 \bmod 3 = 1,\ 2^1 \bmod 3 = 2,\ 2^2 \bmod 3 = 1,\ 2^3 \bmod 3 = 2,\ \ldots$$

the weight $(2^k \bmod 3)$ **alternates** between 1 and 2 for even/odd bit positions.

2.1 State Variables

Therefore the FSM must remember:

- The current remainder $r \in \{0, 1, 2\}$
- The parity of the current bit position $p \in \{0, 1\}$:

- $p = 0$: next weight is 1 (even index)
- $p = 1$: next weight is 2 (odd index)

Total states $= 3 \times 2 = 6$. Each state can be labeled as (p, r).

3 FSM Transition Logic (LSB-first)

Let the input bit be $x \in \{0, 1\}$. Then:

$$r^+ = \begin{cases} (r + x) \bmod 3 & p = 0 \ \text{(weight 1)} \\ (r + 2x) \bmod 3 & p = 1 \ \text{(weight 2)} \end{cases} \qquad p^+ = \overline{p}$$

So the parity toggles each time a bit is consumed.

3.1 Mealy Output

This design is **Mealy**, so the output depends on the **current state and the current input**. We compute the next remainder r^+ combinationally and output:

$$\texttt{REM} = \begin{cases} 0 & r^+ = 0 \\ 1 & r^+ \neq 0 \end{cases}$$

Meaning: `REM` reflects whether the number is divisible by 3 **after consuming the current input bit**.

4 Verilog Implementation

The module encodes the state as:

$$\texttt{state} = \{\texttt{parity}, \texttt{rem[1:0]}\}$$

where `rem` is 0,1,2 and `parity` selects whether the next weight is 1 or 2.

4.1 Design Module (`div3_lsb_mealy.v`)

```
// ------------------------------------------------------------
// div3_lsb_mealy: Mealy FSM that checks divisibility by 3
// Bits are received LSB first (one per clock).
// REM = 0 if divisible by 3 after the current bit is consumed,
// REM = 1 otherwise.
// ------------------------------------------------------------
module div3_lsb_mealy (
    input  wire clk,
    input  wire rst,      // synchronous active-high reset
    input  wire bit_in,   // incoming bit, LSB first
```

```
    output wire REM       // 0 if divisible by 3, 1 otherwise
);

    // State = {parity, rem[1:0]}
    // parity=0 => next bit weight is 1 (even index)
    // parity=1 => next bit weight is 2 (odd index)
    reg [2:0] state, next_state;

    wire parity = state[2];
    wire [1:0] rem = state[1:0];

    // Compute next remainder based on parity (weight 1 or 2)
    reg [1:0] next_rem;

    always @(*) begin
        // default
        next_rem = rem;

        if (!parity) begin
            // weight = 1: next_rem = (rem + bit_in) mod 3
            case ({rem, bit_in})
                3'b00_0: next_rem = 2'd0;
                3'b00_1: next_rem = 2'd1;
                3'b01_0: next_rem = 2'd1;
                3'b01_1: next_rem = 2'd2;
                3'b10_0: next_rem = 2'd2;
                3'b10_1: next_rem = 2'd0;
                default: next_rem = 2'd0; // not used
            endcase
        end else begin
            // weight = 2: next_rem = (rem + 2*bit_in) mod 3
            case ({rem, bit_in})
                3'b00_0: next_rem = 2'd0;
                3'b00_1: next_rem = 2'd2;
                3'b01_0: next_rem = 2'd1;
                3'b01_1: next_rem = 2'd0; // 1+2=3 -> 0
                3'b10_0: next_rem = 2'd2;
                3'b10_1: next_rem = 2'd1; // 2+2=4 -> 1
                default: next_rem = 2'd0; // not used
            endcase
        end

        // Toggle parity each consumed bit
        next_state = {~parity, next_rem};
    end

    // Mealy output depends on current state + input via next_rem
```

```
    assign REM = (next_rem == 2'd0) ? 1'b0 : 1'b1;

    // State register
    always @(posedge clk) begin
        if (rst)
            state <= 3'b0_00; // parity=0, rem=0
        else
            state <= next_state;
    end

endmodule
```

4.2 Testbench (`tb_div3_lsb_mealy.v`)

The testbench feeds numbers LSB-first and checks correctness. Note that using `%` is allowed inside the **testbench** (not synthesizable logic).

```
module tb_div3_lsb_mealy;

    reg clk = 0;
    reg rst = 1;
    reg bit_in = 0;
    wire REM;

    div3_lsb_mealy dut(
        .clk(clk),
        .rst(rst),
        .bit_in(bit_in),
        .REM(REM)
    );

    always #5 clk = ~clk;

    task feed_number;
        input integer value;
        input integer nbits;
        integer i;
        integer expected;
        begin
            // feed LSB first
            for (i = 0; i < nbits; i = i + 1) begin
                bit_in = (value >> i) & 1;
                @(posedge clk);
            end
            expected = (value % 3 != 0); // REM=1 if not divisible
            if (REM !== expected[0]) begin
                $display("FAIL value=%0d expected REM=%0d got REM=%0d", value,
```

```
    expected, REM);
                $stop;
            end else begin
                $display("PASS value=%0d REM=%0d", value, REM);
            end
        end
    endtask

    initial begin
        // reset
        @(posedge clk);
        rst = 1;
        @(posedge clk);
        rst = 0;

        // try a bunch
        feed_number(0, 1);
        feed_number(3, 2);
        feed_number(6, 3);
        feed_number(7, 3);
        feed_number(12, 4);
        feed_number(13, 4);
        feed_number(21, 5);
        feed_number(22, 5);

        // brute small range
        begin : brute
            integer v;
            for (v = 0; v < 64; v = v + 1) begin
                rst = 1; @(posedge clk); rst = 0;
                feed_number(v, 6);
            end
        end

        $display("ALL TESTS PASSED");
        $finish;
    end

endmodule
```

5 FSM

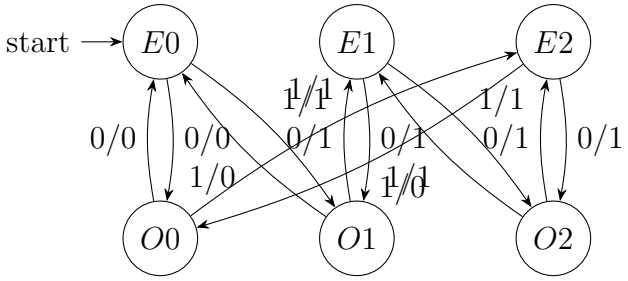

Figure 1: Mealy FSM for checking divisibility by 3 with LSB-first input. States encode (parity, remainder). Edge labels are `bit_in/REM`.

6 Simulation Results and Output Analysis

6.1 Behavioral Simulation Waveform

Figure 2 shows the functional (behavioral) simulation. Important observations:

- `bit_in` changes once per cycle to present the next LSB-first bit.
- Because this is a **Mealy** FSM, `REM` is computed from the **current input** and the **current state**, so it can change **within the same cycle** after `bit_in` changes.
- The final value of `REM` after the last bit of a number indicates whether the complete number is divisible by 3.

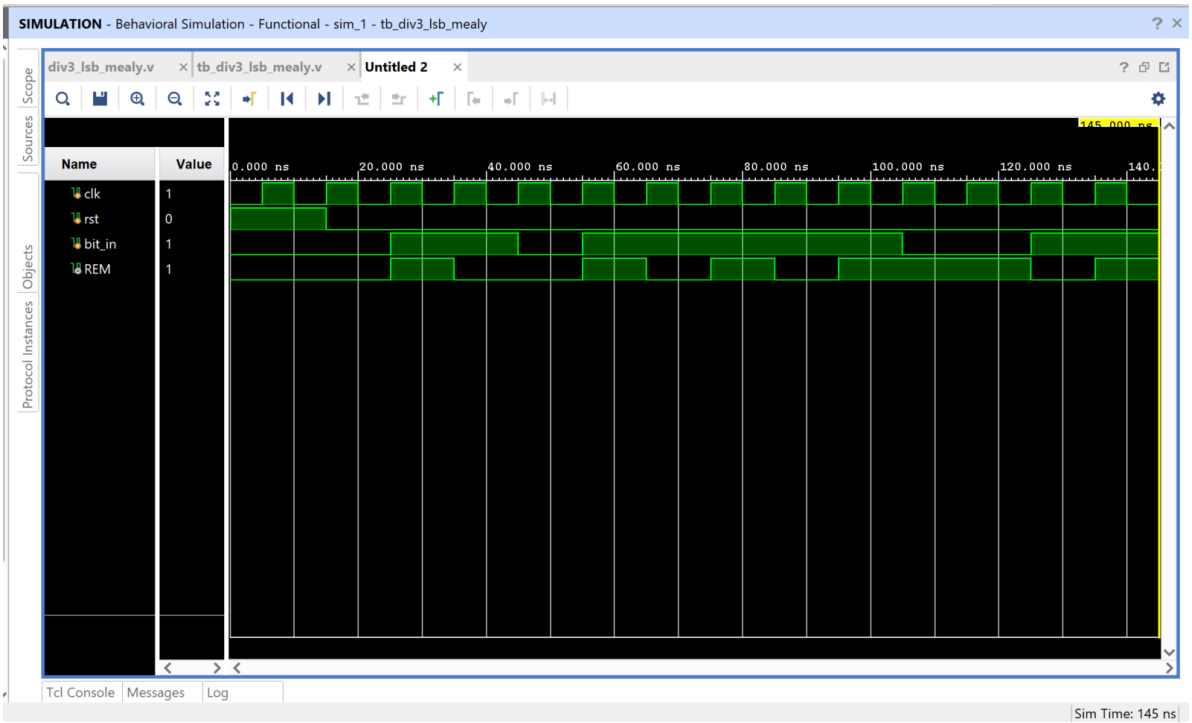

Figure 2: Behavioral simulation waveform for `div3_lsb_mealy`. The `REM` output reflects divisibility by 3 after consuming the current bit (Mealy behavior).

6.2 Implementation / Utilization (Optional but Included)

Figure 3 shows the Vivado utilization summary for the synthesized module. The design is very small (as expected for a 6-state FSM). In the shown run, the utilization includes approximately:

- Slice LUTs: 2
- Slice Registers: 3
- Bonded IOB: 4 (clk, rst, bit_in inputs + REM output)
- BUFG: 1 (clock buffer)

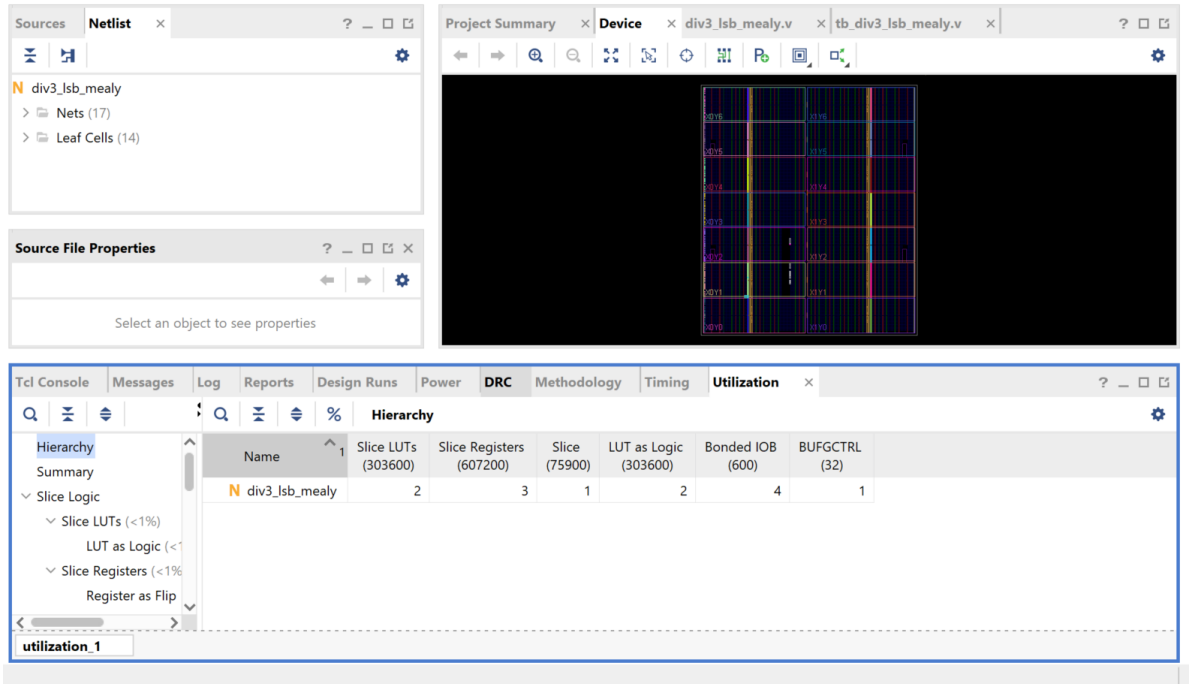

Figure 3: Vivado device view and utilization report for the synthesized FSM.

6.3 Schematic View (Optional but Included)

Figure 4 shows the schematic generated by the tool, confirming the implementation consists of a small amount of combinational logic (LUTs) plus flip-flops for the state register.

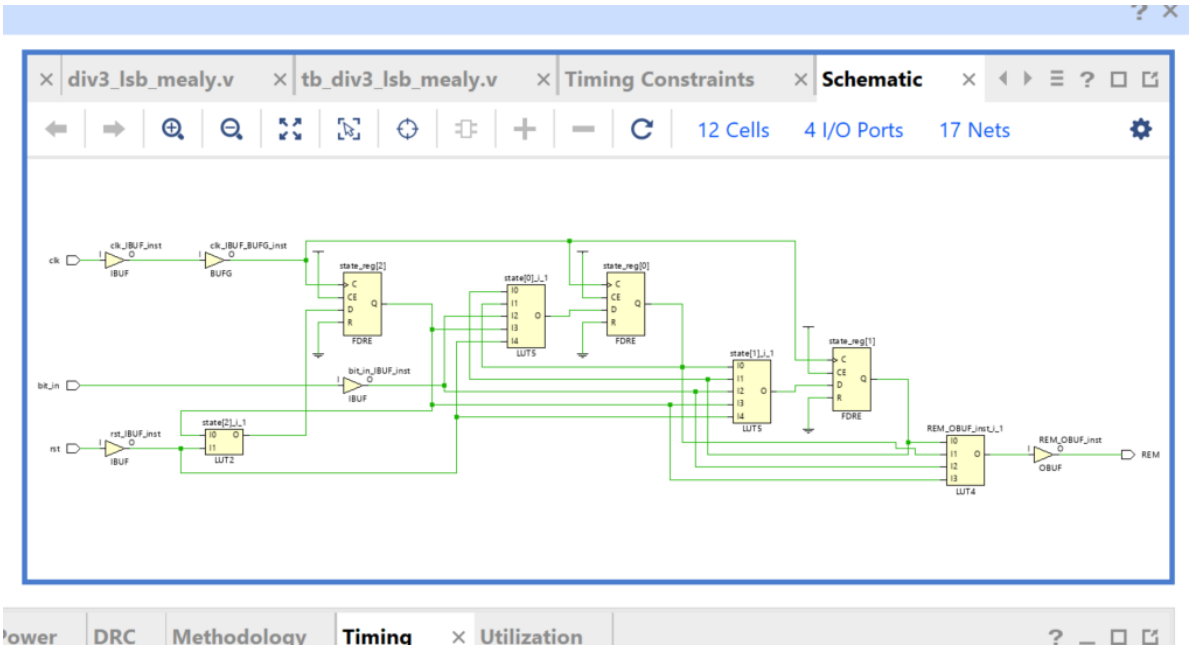

Figure 4: Schematic of the synthesized `div3_lsb_mealy` module (state FFs + LUT logic).

7 Conclusion

A Mealy FSM was designed to check divisibility by 3 for a binary number streamed **LSB-first**. Because the weights $2^k \bmod 3$ alternate between 1 and 2, the FSM tracks both the remainder modulo 3 and the parity of the bit position, resulting in 6 states. The Verilog implementation avoids the modulo operator in synthesizable logic and produces correct results in simulation.

8 Question 2(a): Clock Enable Generator without DCM

8.1 Problem Statement

The goal of this part is to design a hardware module that generates a **clock enable (CE)** signal using:

- a single system clock,
- an arbitrary 8-bit input value,
- **no DCM/PLL**, and
- **no gated clock**.

The generated CE signal must allow a target module, operating on the same system clock, to run at a speed from **1 to 256 times slower** than the maximum speed (i.e., the case where the module is active on every clock cycle).

8.2 Design Approach

Because clock gating is not allowed, the system clock must be distributed unchanged to all modules. The required slowdown is achieved by asserting a **clock enable (CE)** signal periodically. The target module executes its logic only when CE is asserted, while still using the main system clock.

An 8-bit input value `div` is used to control the slowdown factor. The relationship is defined as:

$$\text{Slowdown Factor} = \texttt{div} + 1$$

This mapping ensures:

- `div = 0` $\Rightarrow$ CE asserted every clock cycle ($1\times$ slowdown)
- `div = 255` $\Rightarrow$ CE asserted every 256 cycles ($256\times$ slowdown)

This choice avoids division by zero and provides the full required range.

8.3 Clock Enable Generation Logic

The CE signal is generated using a simple synchronous counter:

- The counter increments on every rising edge of the system clock.
- When the counter reaches the value `div`, a one-cycle CE pulse is generated.
- The counter is then reset to zero and the process repeats.

The CE signal is therefore a **single-clock-wide pulse** occurring every (`div`+1) clock cycles.

8.4 Clock Enable Generator Verilog Module

```
// ------------------------------------------------------------
// Clock Enable Generator (1x to 256x slowdown)
// ------------------------------------------------------------
module ce_gen_1_to_256 (
    input  wire       clk,
    input  wire       rst,      // synchronous active-high reset
    input  wire [7:0] div,      // 0..255 -> enable every (div+1) cycles
    output reg        ce        // one-cycle-wide enable pulse
);
    reg [7:0] cnt;

    always @(posedge clk) begin
        if (rst) begin
            cnt <= 8'd0;
            ce  <= 1'b0;
        end else begin
            if (cnt == div) begin
                cnt <= 8'd0;
                ce  <= 1'b1;
            end else begin
                cnt <= cnt + 8'd1;
                ce  <= 1'b0;
            end
        end
    end
endmodule
```

8.5 Target Module Using Clock Enable

To demonstrate correct operation, a simple counter is used as a target module. The counter increments only when CE is asserted, while still using the same system clock.

```
// ------------------------------------------------------------
// Example target module using CE (slow counter)
// ------------------------------------------------------------
module slow_counter (
    input  wire       clk,
    input  wire       rst,
```

```
    input  wire       ce,
    output reg [7:0]  q
);
    always @(posedge clk) begin
        if (rst)
            q <= 8'd0;
        else if (ce)
            q <= q + 8'd1;
    end
endmodule
```

8.6 Simulation and Output Analysis

Figure 5 shows the behavioral simulation of the clock enable generator together with the target counter.

Key observations:

- The system clock `clk` runs continuously and is never gated.
- The CE signal is a single-cycle pulse.
- When `div` is increased, the spacing between CE pulses increases accordingly.
- The target counter increments only on cycles where CE is asserted, confirming that the effective operating speed is reduced.

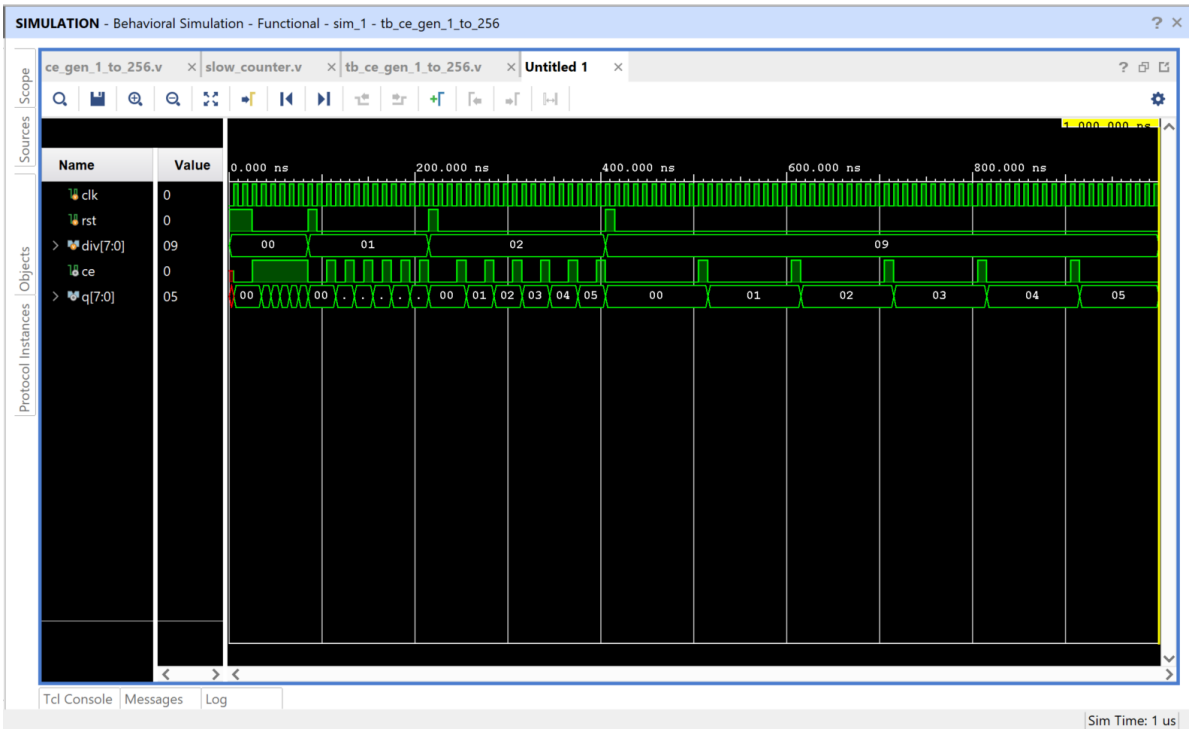

Figure 5: Behavioral simulation showing the clock enable signal and the slowed target counter for different values of `div`.

8.7 Discussion

This design satisfies all problem constraints:

- No clock gating is used.
- No DCM or PLL is required.
- The system clock is shared by all modules.
- The slowdown factor is programmable from 1× to 256× using an 8-bit input.

The solution is fully synthesizable and scales efficiently, using only a small counter and simple comparison logic.

8.8 Conclusion

A configurable clock enable generator was successfully designed to control the effective operating speed of a target module without modifying the system clock. Simulation results confirm correct timing behavior and proper slowdown over the full required range.

9 Question 2(b): Adjustable-Speed Counter Using Clock Enable

9.1 Problem Statement

Using the clock enable generator designed in Question 2(a), the objective of this part is to design a counter whose effective counting speed can be adjusted. The counter must:

- operate using the same system clock,
- avoid clock gating,
- use the generated clock enable (CE) signal to control speed.

9.2 Design Approach

The adjustable-speed counter is implemented by combining two modules:

1. the clock enable generator from Question 2(a), and
2. a synchronous counter that updates only when CE is asserted.

The system clock is distributed unchanged to both modules. The clock enable signal determines when the counter is allowed to increment, effectively controlling the counting speed without modifying the clock itself.

The slowdown factor is defined as:

$$\text{Counting Period} = (\texttt{div} + 1) \times T_{\text{clk}}$$

where `div` is the 8-bit control input and T_{clk} is the system clock period.

9.3 Adjustable-Speed Counter Implementation

The top-level module instantiates the CE generator and uses the CE signal to enable the counter update.

```
// ------------------------------------------------------------
// Adjustable-Speed Counter using Clock Enable
// ------------------------------------------------------------
module adjustable_counter #(
    parameter WIDTH = 8
)(
    input  wire             clk,
    input  wire             rst,      // synchronous active-high reset
    input  wire [7:0]       div,      // speed control
    output reg  [WIDTH-1:0] count,
    output wire             ce_out     // exposed for observation
);

```

```
    wire ce;

    // Clock enable generator from Part (a)
    ce_gen_1_to_256 u_ce (
        .clk(clk),
        .rst(rst),
        .div(div),
        .ce(ce)
    );

    // Counter updates only when CE is asserted
    always @(posedge clk) begin
        if (rst)
            count <= {WIDTH{1'b0}};
        else if (ce)
            count <= count + 1'b1;
    end

    assign ce_out = ce;

endmodule
```

9.4 Simulation Results and Analysis

Figure 6 shows the behavioral simulation of the adjustable-speed counter.

From the waveform, the following observations can be made:

- The system clock runs continuously and is never gated.
- The clock enable signal is asserted periodically based on the value of `div`.
- The counter increments only on clock cycles where CE is high.
- Increasing `div` increases the spacing between counter updates, demonstrating adjustable speed.

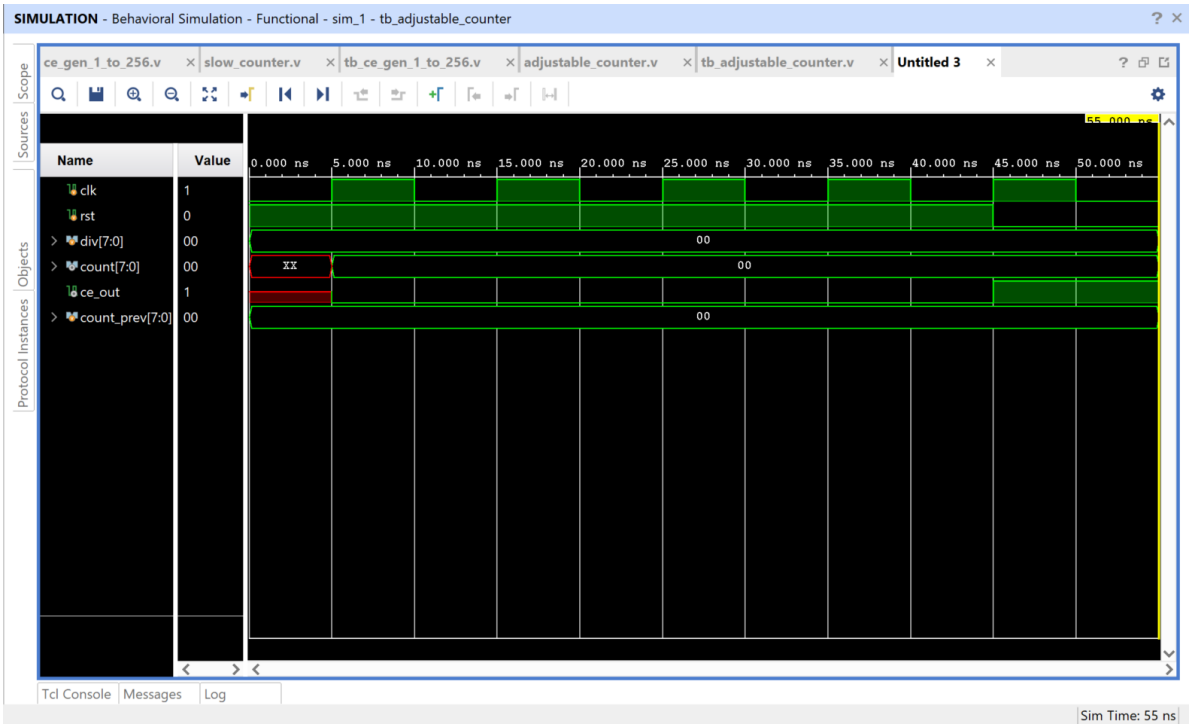

Figure 6: Behavioral simulation of the adjustable-speed counter showing the relationship between CE pulses and counter increments.

9.5 Schematic and Resource Utilization

Figure 7 shows the synthesized schematic of the design. The structure clearly consists of a clock enable generator and a counter connected by enable logic, with a single shared clock.

Figure 8 presents the resource utilization summary. The design uses a small number of LUTs and registers, demonstrating an efficient implementation.

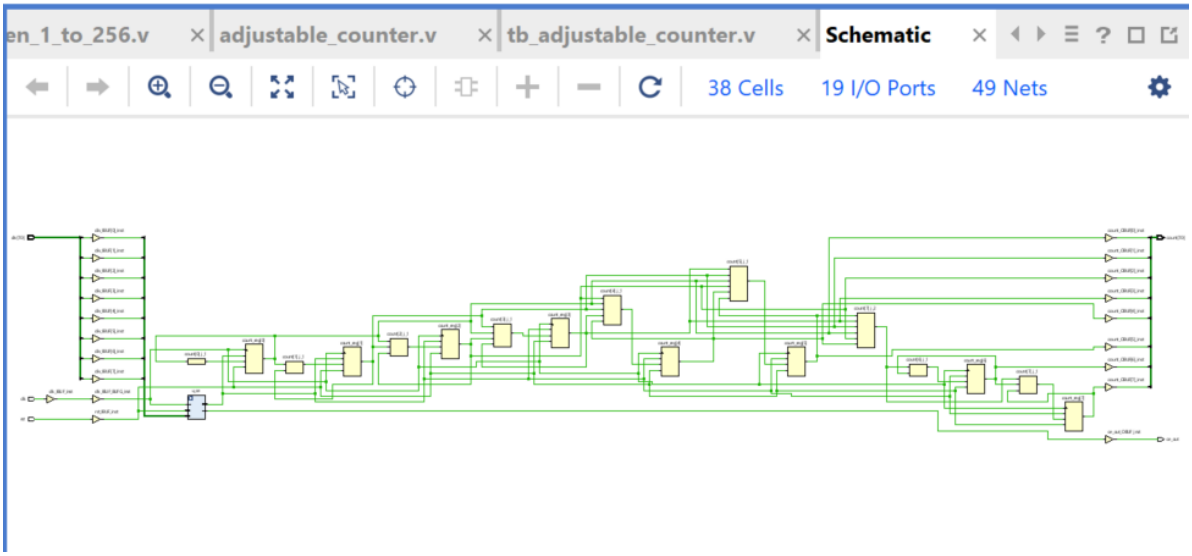

Figure 7: Synthesized schematic of the adjustable-speed counter.

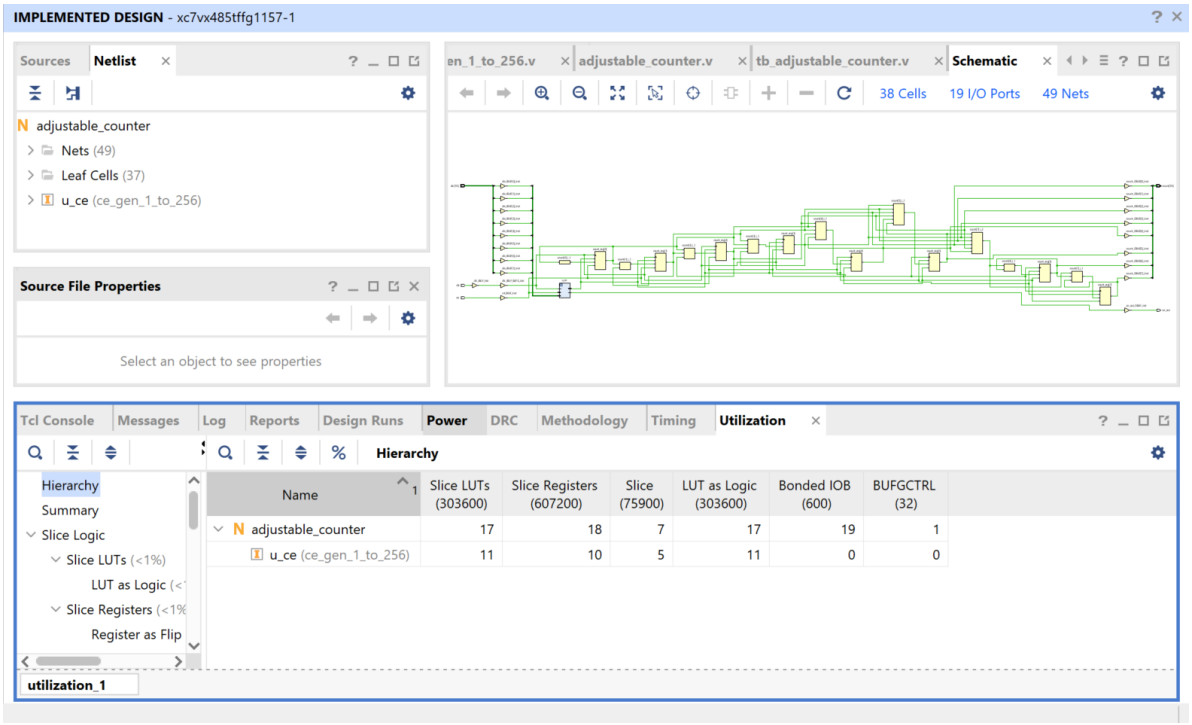

Figure 8: Resource utilization report for the adjustable-speed counter.

9.6 Conclusion

An adjustable-speed counter was successfully designed using the clock enable generator from Question 2(a). The counter operates on a fixed system clock and achieves speed control solely through a clock enable signal, satisfying all problem constraints. Simulation and synthesis results confirm correct functionality and efficient resource usage.

10 Question 3: FIFO Using Two Stacks

10.1 Problem Statement

The goal of this question is to design and implement a **parameterized FIFO (First-In First-Out) buffer** using **only two stack modules**, namely `StackA` and `StackB`. Implementing a FIFO as a circular buffer or using FIFO/RAM IP cores is not allowed.

The FIFO must satisfy:

- **FIFO order:** data written first must be read first.
- **Transfer on read:** on a read request, if the output stack is empty, transfer data from the input stack to the output stack while preserving FIFO order.
- **Synchronous operation:** all operations are synchronous to the system clock.
- **No gated clocks.**

10.2 Design Approach

The design follows the standard two-stack FIFO technique:

- **StackA (input stack):** all writes (`wr_en`) push into StackA.
- **StackB (output stack):** all reads (`rd_en`) pop from StackB.

10.2.1 Operating Principle

1. **Write (enqueue):** push the incoming data into StackA.
2. **Read (dequeue):**
   - If StackB is not empty, pop from StackB (fast path).
   - If StackB is empty and StackA has data, transfer elements from StackA to StackB: repeatedly pop StackA and push into StackB until StackA becomes empty.

Transferring from StackA to StackB reverses the order once, which ensures that the **oldest element becomes the top of StackB**. Therefore, popping StackB returns data in **FIFO order**.

10.2.2 Control Logic (FSM)

Because stack operations are synchronous, a small FSM is used to coordinate:

- accepting read and write requests,
- multi-cycle transfer from StackA to StackB when required,
- producing output data and a one-cycle `rd_valid` pulse.

All control signals are registered on the rising edge of the system clock, and no gated clocks are used.

10.3 Module Descriptions

10.3.1 Stack Modules

The FIFO instantiates exactly two stacks: `StackA` and `StackB`. Each stack is parameterized by data width and depth, and supports synchronous push/pop along with `empty` and `full` flags.

```
‘timescale 1ns/1ps
module StackA #(
    parameter WIDTH = 8,
    parameter DEPTH = 16
)(
    input  wire             clk,
    input  wire             rst,
    input  wire             push,
    input  wire             pop,
    input  wire [WIDTH-1:0] din,
    output reg  [WIDTH-1:0] dout,
    output wire             empty,
    output wire             full
);
    localparam AW = $clog2(DEPTH);

    reg [WIDTH-1:0] mem [0:DEPTH-1];
    reg [AW:0]      sp; // 0..DEPTH

    assign empty = (sp == 0);
    assign full  = (sp == DEPTH);

    always @(posedge clk) begin
        if (rst) begin
            sp   <= 0;
            dout <= 0;
        end else begin
            case ({(push && !full), (pop && !empty)})
                2’b10: begin // push only
                    mem[sp] <= din;
                    sp      <= sp + 1;
                end
                2’b01: begin // pop only
                    dout <= mem[sp-1];
                    sp   <= sp - 1;
                end
                2’b11: begin // push and pop together => replace top, sp
```

```
    unchanged
                    dout      <= mem[sp-1];
                    mem[sp-1] <= din;
                    sp        <= sp;
                end
                default: begin
                    sp <= sp;
                end
            endcase
        end
    end
endmodule
```

```
`timescale 1ns/1ps
module StackB #(
    parameter WIDTH = 8,
    parameter DEPTH = 16
)(
    input  wire             clk,
    input  wire             rst,
    input  wire             push,
    input  wire             pop,
    input  wire [WIDTH-1:0] din,
    output reg  [WIDTH-1:0] dout,
    output wire             empty,
    output wire             full
);
    localparam AW = $clog2(DEPTH);

    reg [WIDTH-1:0] mem [0:DEPTH-1];
    reg [AW:0]      sp;

    assign empty = (sp == 0);
    assign full  = (sp == DEPTH);

    always @(posedge clk) begin
        if (rst) begin
            sp   <= 0;
            dout <= 0;
        end else begin
            case ({(push && !full), (pop && !empty)})
                2'b10: begin
                    mem[sp] <= din;
                    sp      <= sp + 1;
                end
                2'b01: begin
                    dout <= mem[sp-1];
                    sp   <= sp - 1;
```

```
                end
                2'b11: begin
                    dout      <= mem[sp-1];
                    mem[sp-1] <= din;
                    sp        <= sp;
                end
                default: begin
                    sp <= sp;
                end
            endcase
        end
    end
endmodule
```

10.3.2 FIFO Wrapper (Two Stacks)

The FIFO wrapper instantiates StackA and StackB and implements the FSM and transfer logic described above.

```
`timescale 1ns/1ps
module fifo_two_stacks #(
    parameter WIDTH = 8,
    parameter DEPTH = 16
)(
    input  wire             clk,
    input  wire             rst,
    input  wire             wr_en,
    input  wire             rd_en,
    input  wire [WIDTH-1:0] din,
    output reg  [WIDTH-1:0] dout,
    output reg              rd_valid,
    output wire             empty,
    output wire             full
);

    // stack signals (REGISTERED, single driver)
    reg             pushA, popA, pushB, popB;
    reg [WIDTH-1:0] dinA, dinB;
    wire [WIDTH-1:0] doutA, doutB;
    wire            emptyA, fullA, emptyB, fullB;

    StackA #(.WIDTH(WIDTH), .DEPTH(DEPTH)) uA (
        .clk(clk), .rst(rst),
        .push(pushA), .pop(popA),
        .din(dinA), .dout(doutA),
        .empty(emptyA), .full(fullA)
    );

```

```
    StackB #(.WIDTH(WIDTH), .DEPTH(DEPTH)) uB (
        .clk(clk), .rst(rst),
        .push(pushB), .pop(popB),
        .din(dinB), .dout(doutB),
        .empty(emptyB), .full(fullB)
    );

    // FIFO occupancy count enforces total depth = DEPTH
    localparam CW = $clog2(DEPTH+1);
    reg [CW-1:0] count;

    assign empty = (count == 0);
    assign full  = (count == DEPTH);

    // FSM
    localparam [1:0]
        S_IDLE       = 2'd0,
        S_XFER_POPA  = 2'd1,
        S_XFER_PUSHB = 2'd2,
        S_READ_CAP   = 2'd3;

    reg [1:0] state;
    reg [WIDTH-1:0] xfer_data;

    always @(posedge clk) begin
        if (rst) begin
            state    <= S_IDLE;
            count    <= 0;
            dout     <= 0;
            rd_valid <= 0;

            pushA <= 0; popA <= 0; dinA <= 0;
            pushB <= 0; popB <= 0; dinB <= 0;

            xfer_data <= 0;
        end else begin
            // defaults each cycle
            pushA <= 0; popA <= 0;
            pushB <= 0; popB <= 0;
            dinA  <= din;
            dinB  <= xfer_data;

            rd_valid <= 1'b0;

            // do enqueue/dequeue bookkeeping only when truly accepted
            // (we can accept wr and rd in same cycle in IDLE if no transfer
    needed)
```

```
76
77              case (state)
78                  S_IDLE: begin
79                      // write
80                      if (wr_en && !full) begin
81                          pushA <= 1'b1;
82                          dinA  <= din;
83                          count <= count + 1;
84                      end
85
86                      // read
87                      if (rd_en && !empty) begin
88                          if (!emptyB) begin
89                              popB  <= 1'b1;
90                              count <= count - 1;
91                              state <= S_READ_CAP;
92                          end else begin
93                              // need transfer first (A must have data because FIFO
    not empty)
94                              state <= S_XFER_POPA;
95                          end
96                      end
97                  end
98
99                  // pop from A (produces doutA next cycle)
100                 S_XFER_POPA: begin
101                     if (!emptyA && !fullB) begin
102                         popA  <= 1'b1;
103                         state <= S_XFER_PUSHB;
104                     end else begin
105                         state <= S_IDLE;
106                     end
107                 end
108
109                 // capture doutA and push into B; loop until A empty
110                 S_XFER_PUSHB: begin
111                     xfer_data <= doutA;
112                     pushB     <= 1'b1;
113                     dinB      <= doutA;
114
115                     if (!emptyA) state <= S_XFER_POPA;
116                     else         state <= S_IDLE;
117                 end
118
119                 // capture popped data from B
120                 S_READ_CAP: begin
121                     dout     <= doutB;
```

```
                    rd_valid <= 1'b1;
                    state    <= S_IDLE;
                end

                default: state <= S_IDLE;
            endcase
        end
    end

endmodule
```

10.3.3 Testbench

A self-checking testbench was created to validate FIFO behavior. It writes a known sequence (e.g., 1..8) and then reads back the same number of elements, verifying that the output order matches the input order.

```
`timescale 1ns/1ps

module tb_fifo_two_stacks;

    reg clk;
    reg rst;

    reg wr_en;
    reg rd_en;
    reg [7:0] din;

    wire [7:0] dout;
    wire rd_valid;
    wire empty;
    wire full;

    fifo_two_stacks #(.WIDTH(8), .DEPTH(8)) dut (
        .clk(clk),
        .rst(rst),
        .wr_en(wr_en),
        .rd_en(rd_en),
        .din(din),
        .dout(dout),
        .rd_valid(rd_valid),
        .empty(empty),
        .full(full)
    );

    initial clk = 0;
    always #5 clk = ~clk;
```

```
31
32      task do_write(input [7:0] v);
33      begin
34          @(posedge clk);
35          wr_en <= 1'b1;
36          din   <= v;
37          @(posedge clk);
38          wr_en <= 1'b0;
39          din   <= 8'h00;
40      end
41      endtask
42
43      task do_read;
44      begin
45          @(posedge clk);
46          rd_en <= 1'b1;
47          @(posedge clk);
48          rd_en <= 1'b0;
49      end
50      endtask
51
52      initial begin
53          wr_en = 0;
54          rd_en = 0;
55          din   = 0;
56
57          rst = 1;
58          repeat(3) @(posedge clk);
59          rst = 0;
60
61          // Write values: expect FIFO read order 11,22,33,44,55,66
62          do_write(8'd11);
63          do_write(8'd22);
64          do_write(8'd33);
65          do_write(8'd44);
66
67          // Read 2 values (11,22)
68          do_read();
69          do_read();
70
71          // Write 2 more (55,66)
72          do_write(8'd55);
73          do_write(8'd66);
74
75          // Read remaining
76          repeat(10) do_read();
77
```

```
78          repeat(10) @(posedge clk);
79          $stop;
80      end
81
82      always @(posedge clk) begin
83          if (rd_valid) begin
84              $display("READ: %0d", dout);
85          end
86      end
87
88  endmodule
```

10.4 Simulation Results

Figure 9 shows the behavioral simulation waveform. The sequence written to the FIFO is observed at the output in the same order, confirming correct FIFO behavior. The `rd_valid` signal asserts for one clock cycle whenever a valid output word is produced.

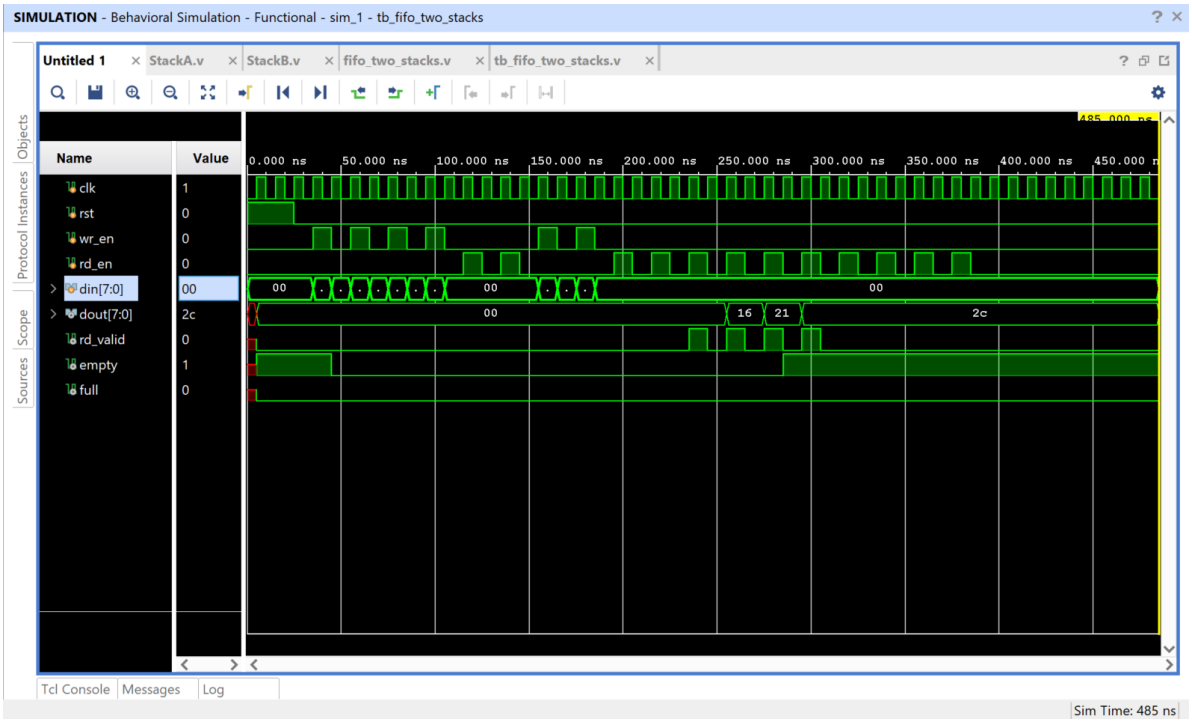

Figure 9: Behavioral simulation waveform of the FIFO implemented using two stacks.

10.5 Synthesis and Implementation Results

10.5.1 Resource Utilization

Figure 10 shows the Vivado utilization report. The design uses only logic resources (LUTs and registers) and does not use any FIFO/RAM IP cores.

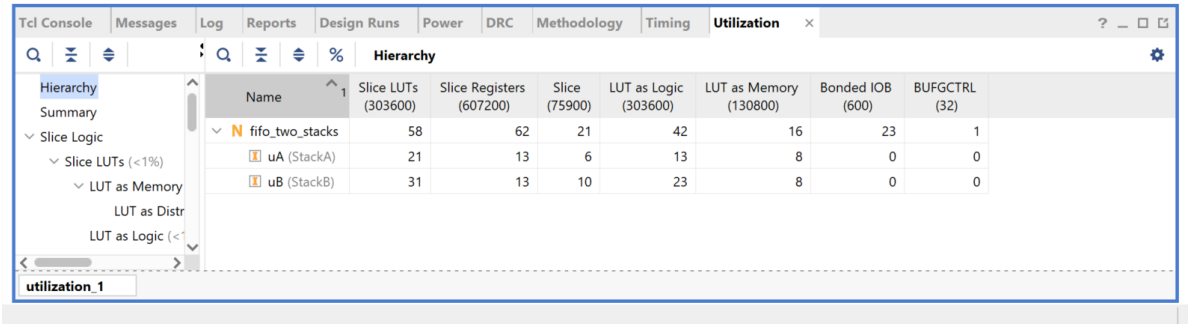

Figure 10: Resource utilization for the two-stack FIFO (Vivado report).

10.5.2 Schematic View

Figure 11 shows the synthesized schematic confirming that the FIFO is composed of **exactly two stack instances** plus the control logic.

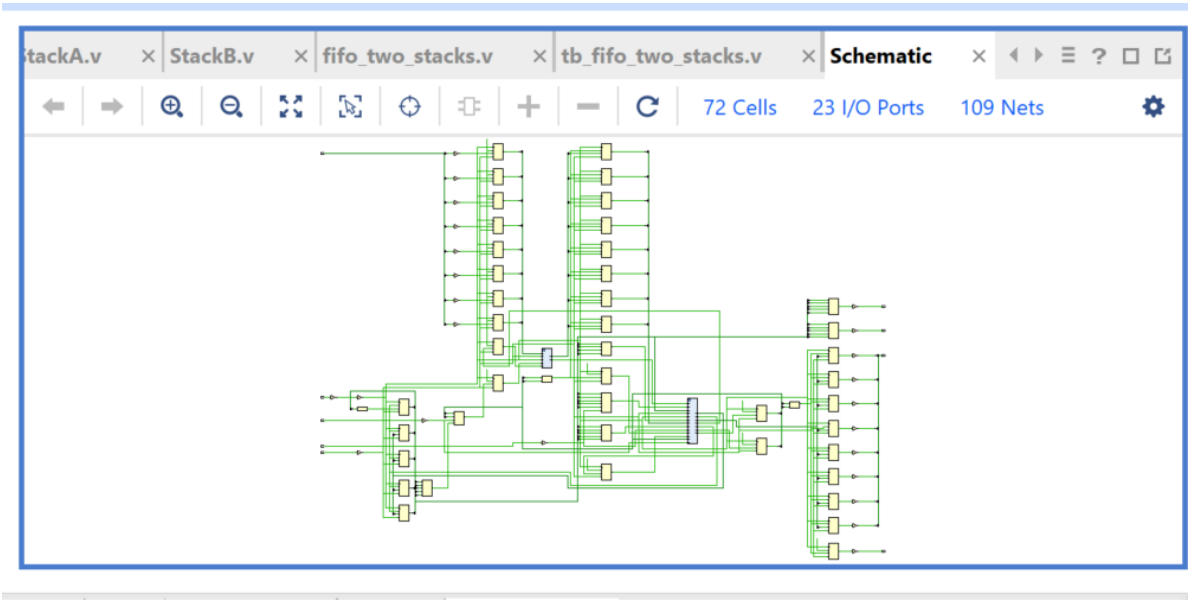

Figure 11: Post-synthesis schematic of the FIFO using two stacks.

10.5.3 Timing Summary (Optional)

If a timing report is provided, it can be included to show the design meets timing constraints (no setup/hold violations).

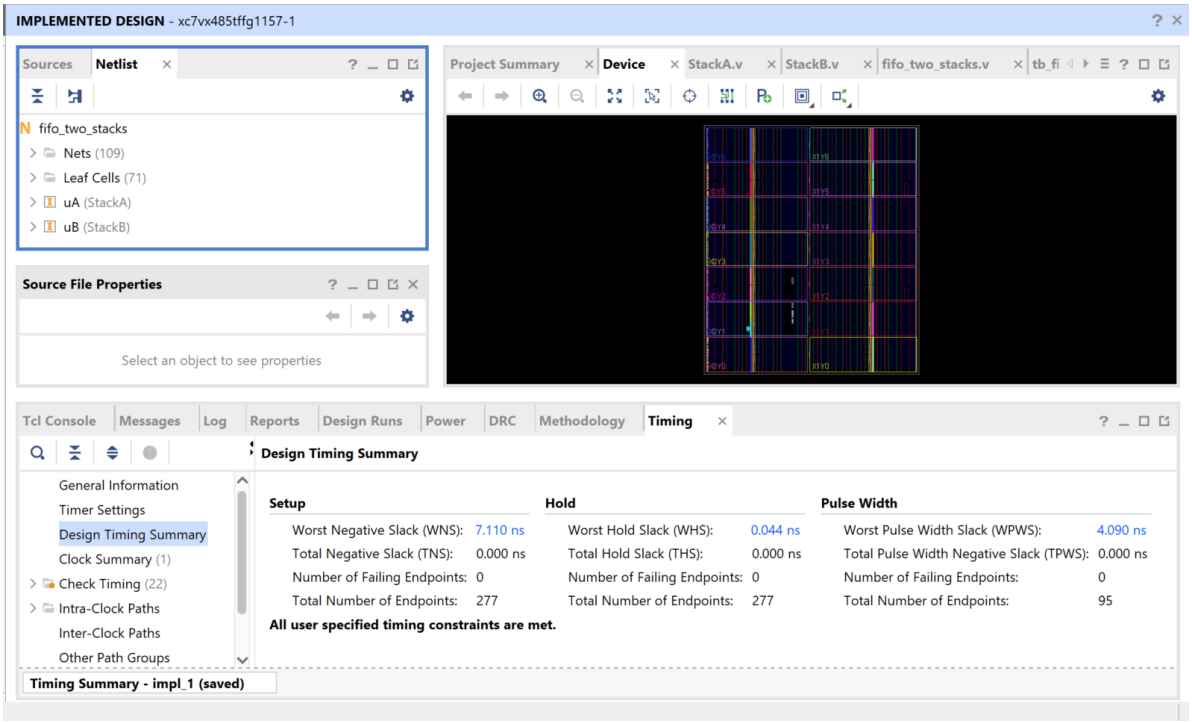

Figure 12: Post-implementation timing summary (Vivado report).

10.6 Discussion

This implementation satisfies all requirements: FIFO order is preserved, transfer occurs only when the output stack is empty, and all operations are synchronous. A small additional latency may occur on the first read after StackB becomes empty, because a transfer from StackA to StackB is required. This latency is expected and is the trade-off for implementing a FIFO using only two stacks.

10.7 Conclusion

A parameterized FIFO was successfully implemented using only two stack modules. Simulation verified correct FIFO behavior, and synthesis results confirm that the design uses only standard FPGA logic resources while avoiding gated clocks and FIFO/RAM IP.

11 Question four

The objective of this assignment is to design a controller for a smart application using a single mechanical push button. During hardware testing, the following issues were observed:

- The push button input is asynchronous with respect to the system clock.
- A single physical button press can cause multiple state transitions due to mechanical bouncing.

- The controller may enter unexpected or illegal states.

The controller must operate with the following four states:

- **Reset**: Initial state after power-up, all outputs deasserted.
- **Idle**: Appliance is off and waits for a valid button event.
- **Active**: Appliance is on for exactly 10 clock cycles.
- **Error**: Fault condition entered if a button event occurs during the Active state.

All state transitions must be fully synchronous, and the FSM must respond only to clean, clock-aligned button events.

12 Design Methodology

To solve the identified issues, the design was divided into two main blocks:

1. **Button Conditioning Block**
2. **Finite State Machine (FSM)**

12.1 Asynchronous Input Synchronization

Since the push button is asynchronous, a two flip-flop synchronizer is used to safely bring the signal into the clock domain and avoid metastability.

12.2 Debouncing and Event Generation

Mechanical button bouncing can produce multiple transitions for a single press. A debounce counter ensures the signal remains stable for a fixed number of clock cycles before acceptance. A rising-edge detector then generates a single-cycle pulse (`btn_event`) for each valid button press.

12.3 FSM Operation

A Moore-type FSM is implemented using fully synchronous logic. An internal counter ensures the Active state lasts exactly 10 clock cycles. A default state recovery mechanism ensures the controller cannot remain in unexpected states.

13 FSM State Diagram

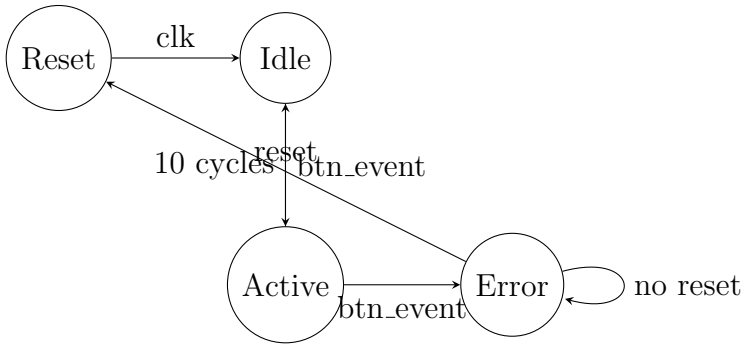

Figure 13: Finite State Machine of the Smart Controller

14 Verilog Implementation

14.1 Button Conditioner

This module synchronizes the asynchronous button, debounces it, and generates a single clean clock-aligned event.

```
module button_conditioner #(
    parameter integer DEBOUNCE_CYCLES = 50000  // adjust for your clock (e.g., 50 MHz => ~1ms)
)(
    input  wire clk,
    input  wire rst_n,      // active-low reset
    input  wire btn_async,  // raw mechanical button (asynchronous + bouncy)
    output wire btn_event,  // 1-clock pulse on clean rising edge
    output wire btn_level   // debounced level (optional)
);

    // ------------------------------------------------------------------------
    // 1) Two-flop synchronizer: mitigates metastability from async input
    // ------------------------------------------------------------------------
    reg btn_ff1, btn_ff2;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            btn_ff1 <= 1'b0;
            btn_ff2 <= 1'b0;
        end else begin
            btn_ff1 <= btn_async;
            btn_ff2 <= btn_ff1;
        end
    end

```

```
    // ---------------------------------------------------------------------
    // 2) Debounce logic:
    //    Only change debounced output after input remains stable for N cycles.
    // ---------------------------------------------------------------------
    localparam integer CNT_W = (DEBOUNCE_CYCLES <= 1) ? 1 : $clog2(
    DEBOUNCE_CYCLES);
    reg [CNT_W-1:0] stable_cnt;
    reg debounced;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            debounced  <= 1'b0;
            stable_cnt <= {CNT_W{1'b0}};
        end else begin
            if (btn_ff2 == debounced) begin
                // input matches current debounced state -> reset counter
                stable_cnt <= {CNT_W{1'b0}};
            end else begin
                // input differs -> count how long it stays different
                if (DEBOUNCE_CYCLES <= 1) begin
                    debounced <= btn_ff2; // immediate accept if N=1
                end else if (stable_cnt == DEBOUNCE_CYCLES-1) begin
                    debounced  <= btn_ff2;              // accept new stable
    value
                    stable_cnt <= {CNT_W{1'b0}};
                end else begin
                    stable_cnt <= stable_cnt + 1'b1;
                end
            end
        end
    end

    assign btn_level = debounced;

    // ---------------------------------------------------------------------
    // 3) Edge detection: one clean event per press (rising edge only)
    // ---------------------------------------------------------------------
    reg debounced_d;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) debounced_d <= 1'b0;
        else        debounced_d <= debounced;
    end

    assign btn_event = debounced & ~debounced_d; // 1-clock pulse on rising edge

endmodule
```

Listing 1: Button Conditioner Module

14.2 Smart Controller FSM

This module implements the FSM and output logic.

```
module smart_controller #(
    parameter integer ACTIVE_CYCLES   = 10,
    parameter integer DEBOUNCE_CYCLES = 50000
)(
    input  wire clk,
    input  wire rst_n,      // active-low reset
    input  wire btn_async,  // raw push button
    output reg  appliance_on,
    output reg  fault
);

    // Clean, clock-aligned, single-cycle button event
    wire btn_event;
    wire btn_level; // optional

    button_conditioner #(.DEBOUNCE_CYCLES(DEBOUNCE_CYCLES)) u_btn (
        .clk(clk),
        .rst_n(rst_n),
        .btn_async(btn_async),
        .btn_event(btn_event),
        .btn_level(btn_level)
    );

    // ------------------------------------------------------------
    // State encoding (plain Verilog)
    // ------------------------------------------------------------
    localparam [1:0]
        S_RESET  = 2'b00,
        S_IDLE   = 2'b01,
        S_ACTIVE = 2'b10,
        S_ERROR  = 2'b11;

    reg [1:0] state, next_state;

    // ------------------------------------------------------------
    // Active counter width (Verilog-friendly)
    // ------------------------------------------------------------
    function integer clog2;
        input integer value;
        integer i;
        begin
            clog2 = 0;
            for (i = value-1; i > 0; i = i >> 1)
                clog2 = clog2 + 1;
```

```
        end
    endfunction

    localparam integer ACW = (ACTIVE_CYCLES <= 1) ? 1 : clog2(ACTIVE_CYCLES);
    reg [ACW-1:0] active_cnt;

    // State register
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            state <= S_RESET;
        else
            state <= next_state;
    end

    // Active counter
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            active_cnt <= {ACW{1'b0}};
        end else begin
            if (state != S_ACTIVE) begin
                active_cnt <= {ACW{1'b0}};
            end else begin
                if (active_cnt == ACTIVE_CYCLES-1)
                    active_cnt <= {ACW{1'b0}};
                else
                    active_cnt <= active_cnt + 1'b1;
            end
        end
    end

    // Next-state logic
    always @(*) begin
        next_state = state;

        case (state)
            S_RESET: begin
                // go to Idle on the first clock after reset is released
                next_state = S_IDLE;
            end

            S_IDLE: begin
                if (btn_event)
                    next_state = S_ACTIVE;
            end

            S_ACTIVE: begin
                if (btn_event)
```

```
                    next_state = S_ERROR;
                else if (active_cnt == ACTIVE_CYCLES-1)
                    next_state = S_IDLE;
            end

            S_ERROR: begin
                next_state = S_ERROR;
            end

            default: begin
                next_state = S_RESET;
            end
        endcase
    end

    // Outputs (Moore)
    always @(*) begin
        appliance_on = 1'b0;
        fault        = 1'b0;

        case (state)
            S_ACTIVE: begin
                appliance_on = 1'b1;
                fault        = 1'b0;
            end
            S_ERROR: begin
                appliance_on = 1'b0;
                fault        = 1'b1;
            end
            default: begin
                appliance_on = 1'b0;
                fault        = 1'b0;
            end
        endcase
    end

endmodule
```

Listing 2: Smart Controller FSM

14.3 Test bench

This module implements the Test bench and output.

```
`timescale 1ns/1ps

module tb_smart_controller;

```

```
    // ------------------------------------------------------------
    // Clock: 100 MHz (period = 10 ns)
    // ------------------------------------------------------------
    reg clk;
    initial clk = 1'b0;
    always #5 clk = ~clk;

    // ------------------------------------------------------------
    // DUT I/O
    // ------------------------------------------------------------
    reg  rst_n;
    reg  btn_async;

    wire appliance_on;
    wire fault;

    // ------------------------------------------------------------
    // Instantiate DUT
    // NOTE: Use small DEBOUNCE_CYCLES for SIMULATION only
    // ------------------------------------------------------------
    smart_controller #(
        .ACTIVE_CYCLES(10),
        .DEBOUNCE_CYCLES(3)      // small so btn_event triggers quickly in sim
    ) dut (
        .clk(clk),
        .rst_n(rst_n),
        .btn_async(btn_async),
        .appliance_on(appliance_on),
        .fault(fault)
    );

    // ------------------------------------------------------------
    // Helpful tasks
    // ------------------------------------------------------------

    // Hold button stable HIGH long enough to pass debounce
    task press_clean;
        input integer hold_cycles; // number of clk cycles to hold high
        integer i;
        begin
            btn_async = 1'b1;
            for (i = 0; i < hold_cycles; i = i + 1) begin
                @(posedge clk);
            end
            btn_async = 1'b0;
            // wait a couple cycles after release
            @(posedge clk);
```

```
            @(posedge clk);
        end
    endtask

    // Simulate bounce: rapid toggles, then stable high long enough to debounce
    task press_bouncy_then_stable;
        input integer stable_cycles;
        begin
            // "bounce" (fast toggles not aligned to clk)
            #3  btn_async = 1'b1;
            #4  btn_async = 1'b0;
            #2  btn_async = 1'b1;
            #3  btn_async = 1'b0;
            #2  btn_async = 1'b1;

            // now hold stable long enough for debouncer
            repeat (stable_cycles) @(posedge clk);

            // release
            btn_async = 1'b0;
            @(posedge clk);
            @(posedge clk);
        end
    endtask

    // ------------------------------------------------------------
    // Test sequence
    // ------------------------------------------------------------
    initial begin
        // init
        rst_n     = 1'b0;
        btn_async = 1'b0;

        // apply reset for a few cycles
        repeat (3) @(posedge clk);
        rst_n = 1'b1;  // release reset

        // wait a couple cycles -> should go to IDLE
        repeat (2) @(posedge clk);

        // ============================================================
        // 1) IDLE press with bounce -> should generate ONE btn_event
        //    and enter ACTIVE
        // ============================================================
        press_bouncy_then_stable(4);  // stable 4 cycles > DEBOUNCE_CYCLES(3)

        // wait 2 cycles, should be in ACTIVE now (appliance_on=1)
```

```
99          repeat (2) @(posedge clk);
100
101         // ============================================================
102         // 2) While in ACTIVE, press again (clean) -> should go ERROR
103         // ============================================================
104         press_clean(4);  // hold high long enough for debounce -> btn_event
105
106         // wait a few cycles to observe ERROR holding
107         repeat (5) @(posedge clk);
108
109         // ============================================================
110         // 3) Reset should clear ERROR -> back to IDLE
111         // ============================================================
112         rst_n = 1'b0;
113         repeat (2) @(posedge clk);
114         rst_n = 1'b1;
115         repeat (3) @(posedge clk);
116
117         // done
118         $finish;
119     end
120
121 endmodule
```

Listing 3: Smart Controller FSM Test Bench

14.4 Timing Constraints File

The Timing Constraint File

```
1 create_clock -period 10.000 -name clk -waveform {0.000 5.000} [get_ports clk]
```

Listing 4: Timing Constraint File

15 Simulation Results

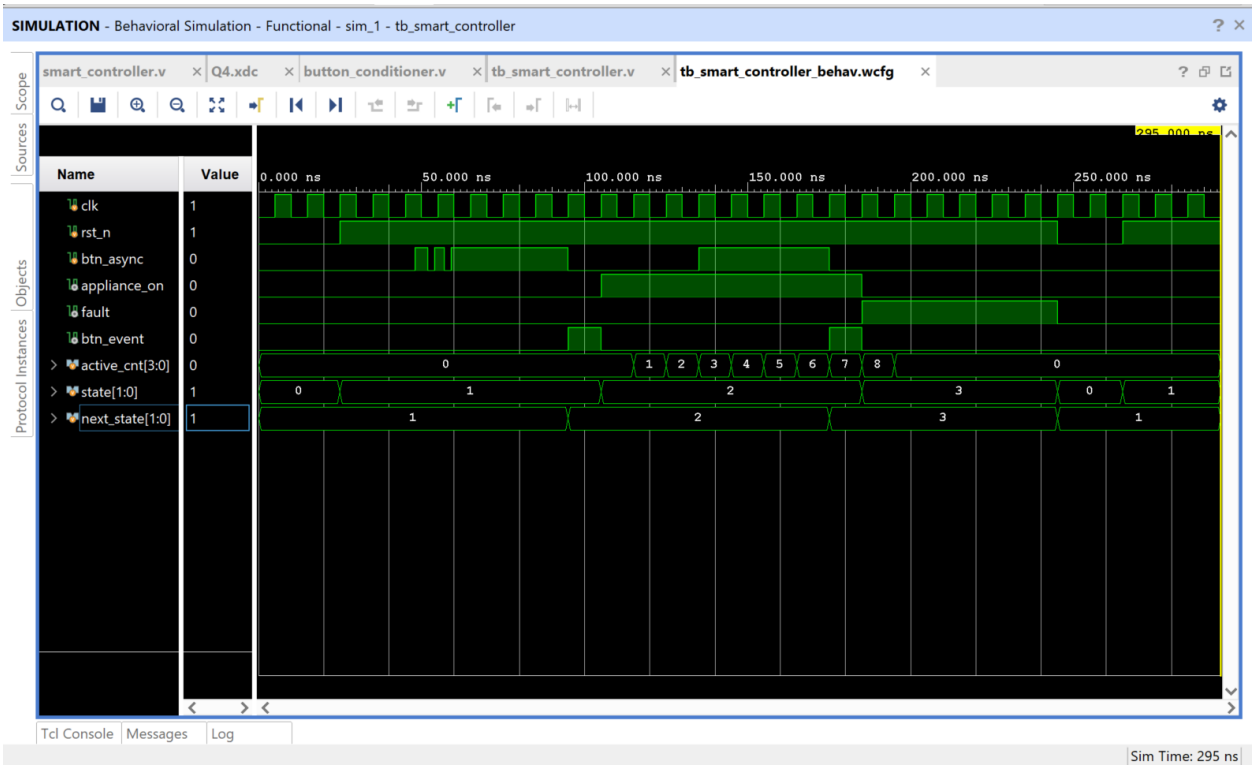

Figure 14: Behavioral Simulation Showing Idle to Active Transition

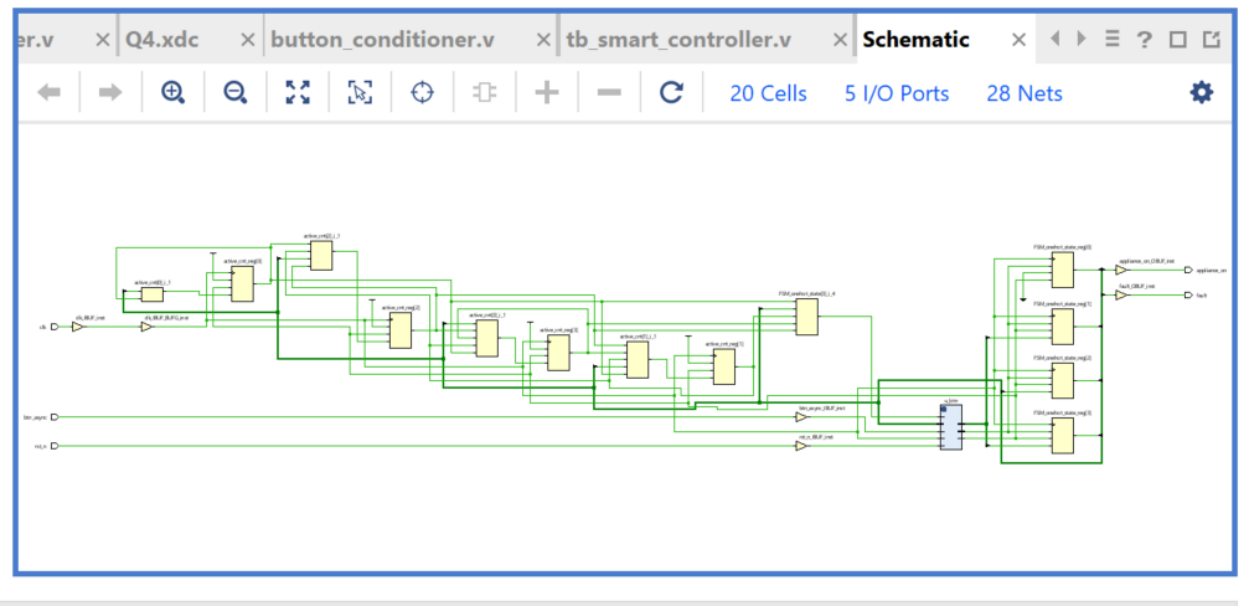

Figure 15: FSM Schematic After Synthesis

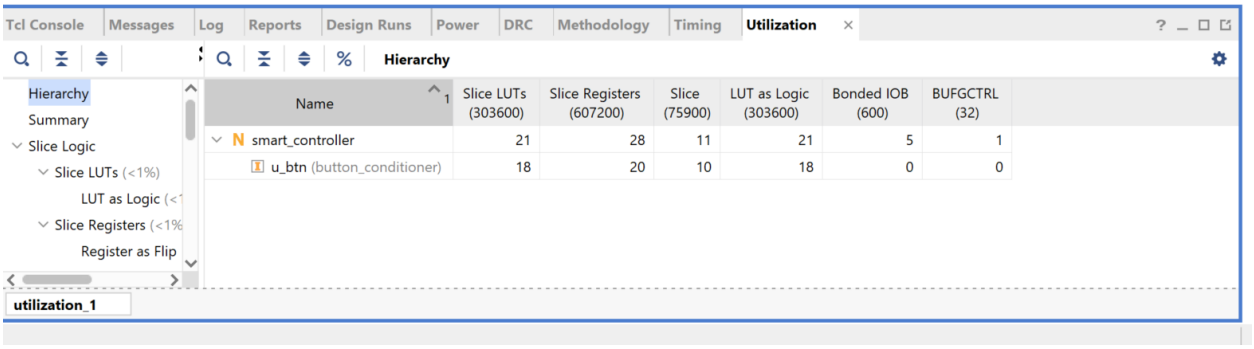

Figure 16: Device Utilization Summary

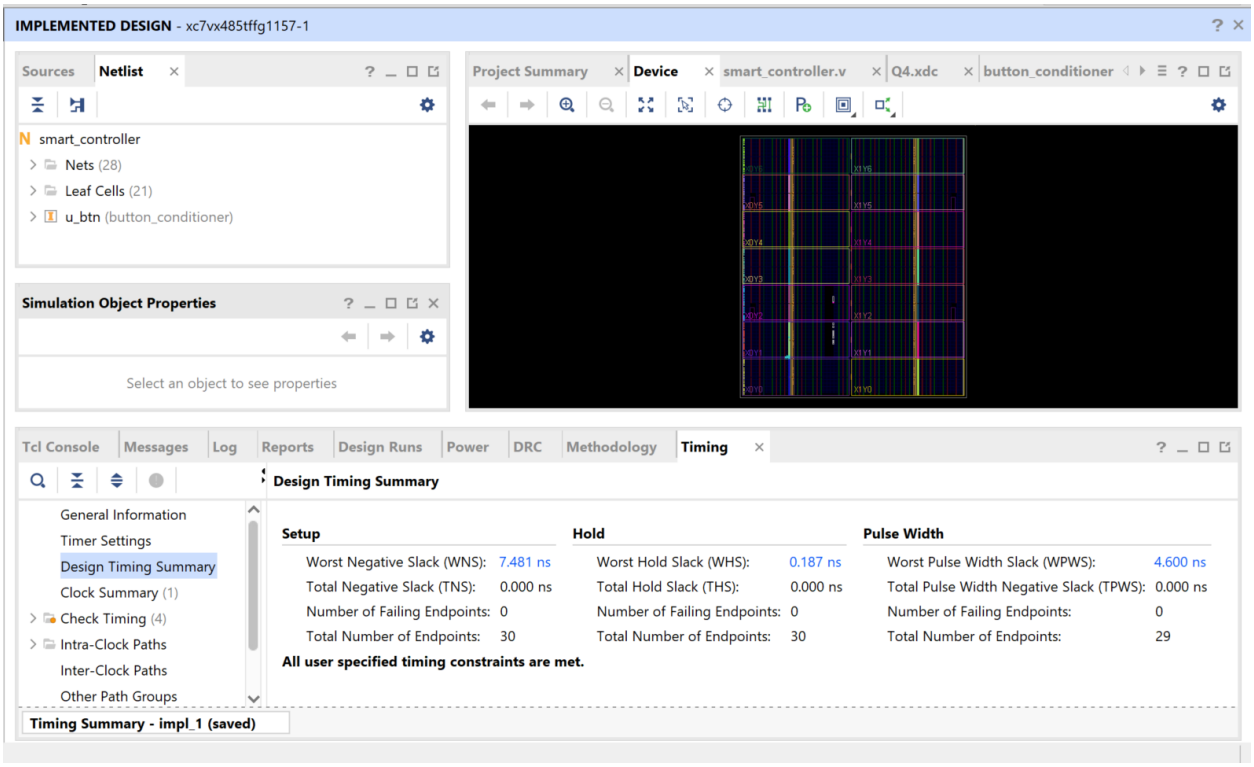

Figure 17: Post-Implementation Timing Summary

16 Result Analysis

Simulation results verify that button bouncing does not cause multiple state transitions and that the FSM responds only to clean button events. The Active state remains asserted for exactly 10 clock cycles, and any button event during this state immediately transitions the system to the Error state.

Post-implementation timing analysis shows positive slack and zero failing endpoints, confirming that all timing constraints are met. Device utilization is minimal, indicating an efficient and well-structured design.

17 Conclusion

A reliable smart controller was successfully designed using fully synchronous FSM principles. Asynchronous input issues were resolved using synchronization and debouncing techniques. Simulation and implementation results confirm that the design meets all functional and reliability requirements specified in the assignment.