

# Finite State Machine & Memories

1.

```
module Tesla_Control #(
    parameter MIN_DISTANCE = 7'd40      // 40 meters
)(
    input wire      clk,
    input wire      rst,                // active high async reset
    input wire [7:0] speed_limit,        // allowable speed limit
    input wire [7:0] car_speed,          // current car speed
    input wire [6:0] leading_distance,    // distance to front vehicle

    output reg       unlock_doors,        // unlock doors when HIGH
    output reg       accelerate_car      // accelerate car when HIGH
);

    // FSM state encoding
    localparam STOP      = 2'b00;
    localparam ACCELERATE = 2'b01;
    localparam DECELERATE = 2'b10;

    // State registers
    reg [1:0] crnt_state, nxt_state;

    // Sequential block: state update
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            crnt_state <= STOP;
        end else begin
            crnt_state <= nxt_state;
        end
    end

    // Next state logic
    always @(*) begin
        nxt_state = crnt_state; // default hold
        case (crnt_state)
            STOP: begin
                if (leading_distance >= MIN_DISTANCE) begin
                    nxt_state = ACCELERATE;
                end else begin
                    nxt_state = STOP;
                end
            end
            ACCELERATE: begin
                if (leading_distance < MIN_DISTANCE || car_speed >
speed_limit) begin
                    nxt_state = DECELERATE;
                end
            end
        end
    end
end
```

```

        end else begin
            nxt_state = ACCELERATE;
        end
    end

    DECELERATE: begin
        if (car_speed == 0) begin
            nxt_state = STOP;
        end else if (leading_distance >= MIN_DISTANCE && car_speed <
speed_limit) begin
            nxt_state = ACCELERATE;
        end else begin
            nxt_state = DECELERATE;
        end
    end

    default: nxt_state = STOP;
endcase
end

// Output logic (Moore FSM: depends only on crnt_state)
always @(*) begin
    case (crnt_state)
        STOP: begin
            unlock_doors    = 1'b1;
            accelerate_car = 1'b0;
        end

        ACCELERATE: begin
            unlock_doors    = 1'b0;
            accelerate_car = 1'b1;
        end

        DECELERATE: begin
            unlock_doors    = 1'b0;
            accelerate_car = 1'b0;
        end

        default: begin
            unlock_doors    = 1'b1;
            accelerate_car = 1'b0;
        end
    endcase
end

endmodule

```

```

module Tesla_Control_tb();
    // Testbench signals
    reg clk;
    reg rst;
    reg [7:0] speed_limit;
    reg [7:0] car_speed;
    reg [6:0] leading_distance;

    wire unlock_doors;
    wire accelerate_car;

    // Instantiate DUT
    Tesla_Control DUT (
        .clk(clk),
        .rst(rst),
        .speed_limit(speed_limit),
        .car_speed(car_speed),
        .leading_distance(leading_distance),
        .unlock_doors(unlock_doors),
        .accelerate_car(accelerate_car)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 100MHz clock
    end

    // Stimulus
    initial begin
        // Initialize signals
        rst = 1;
        speed_limit = 8'd100; // 100 km/h speed limit
        car_speed = 8'd0;
        leading_distance = 7'd20; // 20m (too close)

        // Apply reset
        #12 rst = 0;

        // Car is too close (should be STOP)
        #20;

        // Increase distance (safe to accelerate)
        leading_distance = 7'd50;
        car_speed = 8'd20;
        #40;

        // Car accelerates until close to speed limit
    end

```

```

    car_speed = 8'd95;
    #40;

    // Exceed speed limit (should DECELERATE)
    car_speed = 8'd110;
    #40;

    // Reduce speed gradually
    car_speed = 8'd80;
    leading_distance = 7'd30; // too close again
    #40;

    // Car slows down to stop
    car_speed = 8'd0;
    #40;

    // Resume safe distance
    leading_distance = 7'd70;
    car_speed = 8'd50;
    #40;

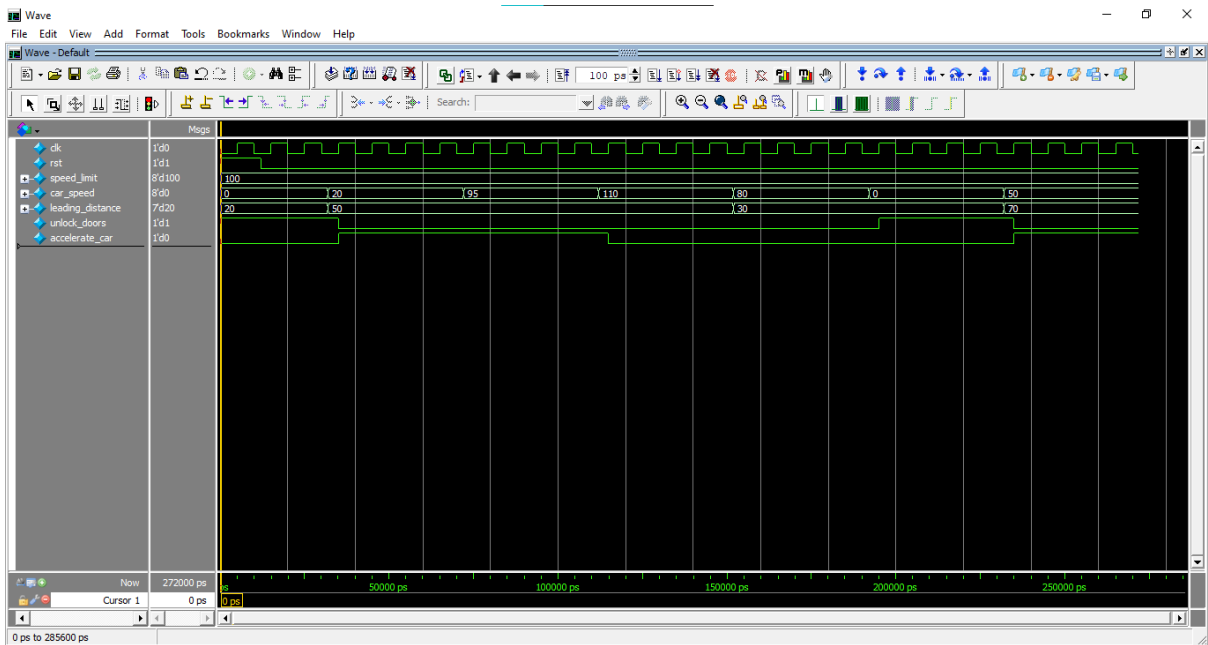
    // End simulation
    $stop;
end

// Monitor output
initial begin
    $display("Time\tState\tSpeed\tLimit\tDistance\tUnlock\tAccel");
    $monitor("%0t\t%d\t%d\t%d\t%d\t%b\t%b",
              $time, DUT.crnt_state, car_speed, speed_limit,
leading_distance,
              unlock_doors, accelerate_car);
end

endmodule

```

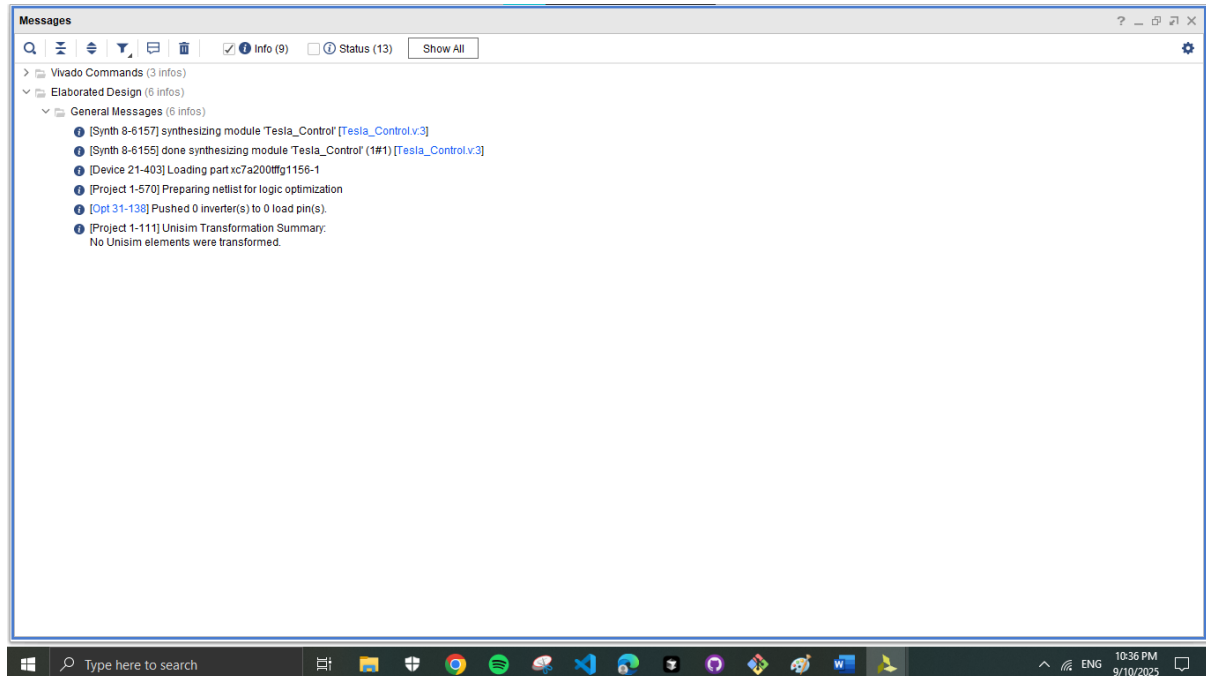
# QuestaSim snippet



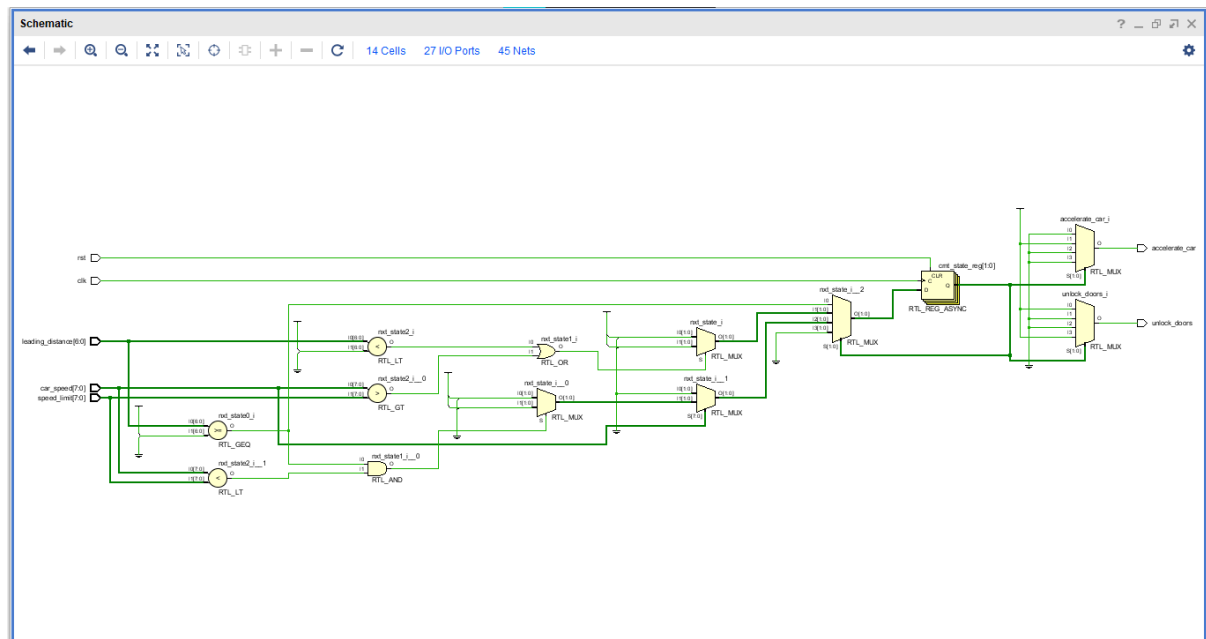
#	Time	State	Speed	Limit	Distance	Unlock	Accel
#	0	0	0	100	20	1	0
#	32000	0	20	100	50	1	0
#	35000	1	20	100	50	0	1
#	72000	1	95	100	50	0	1
#	112000	1	110	100	50	0	1
#	115000	2	110	100	50	0	0
#	152000	2	80	100	30	0	0
#	192000	2	0	100	30	0	0
#	195000	0	0	100	30	1	0
#	232000	0	50	100	70	1	0
#	235000	1	50	100	70	0	1

# Elaboration

## “Messages” tab

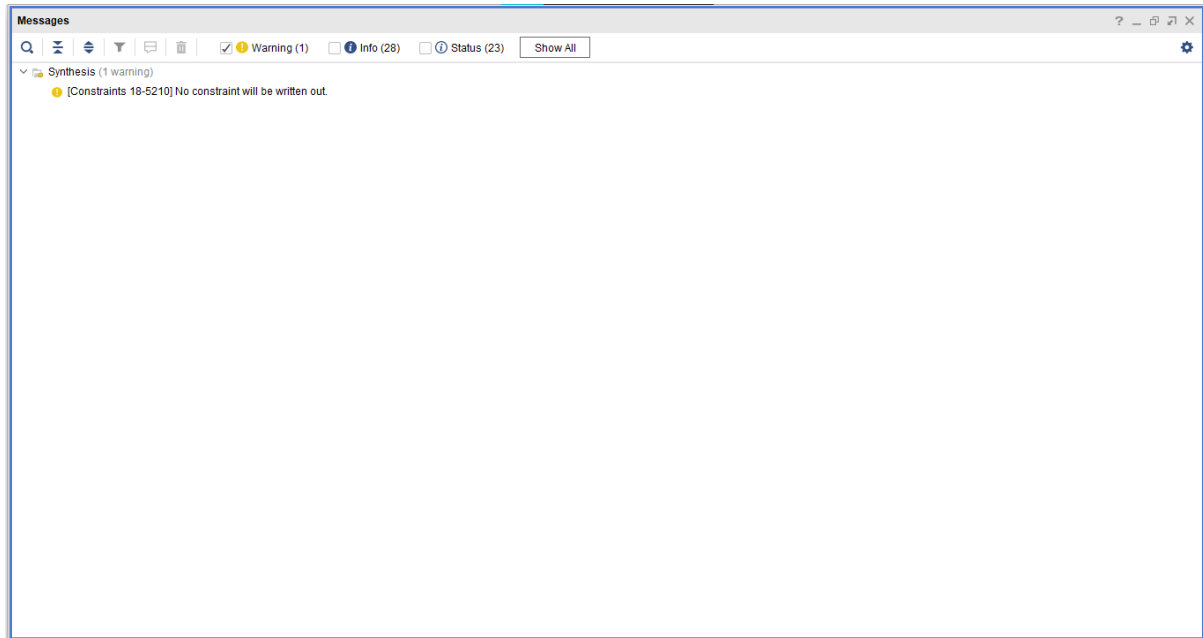


## Schematic snippet

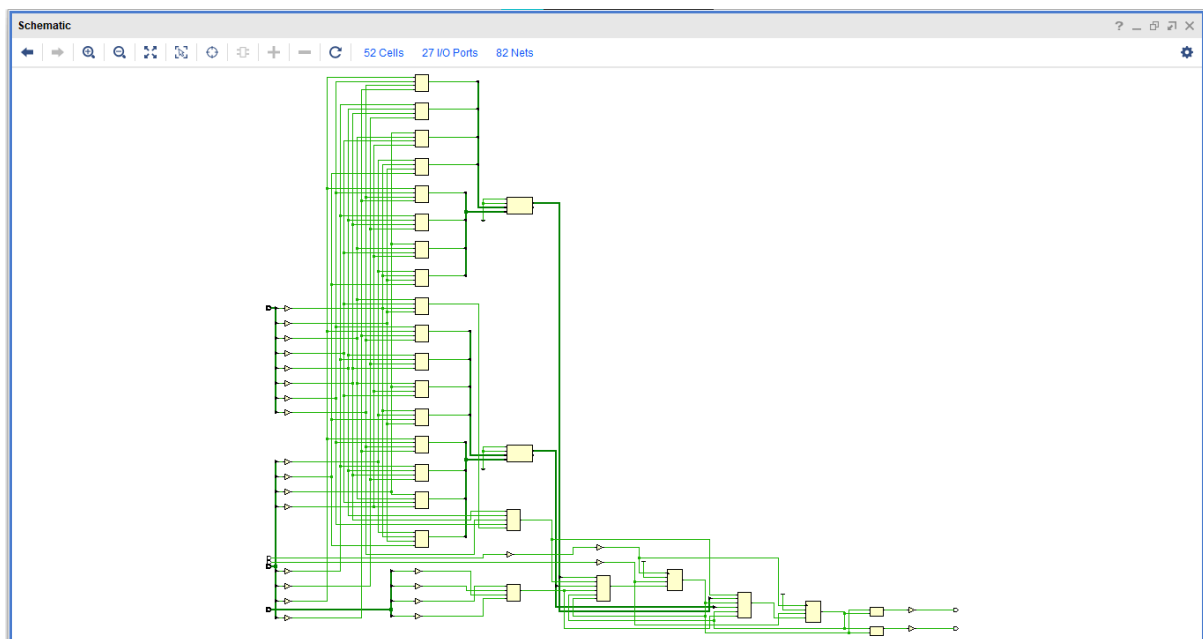


# Synthesis

## “Messages” tab

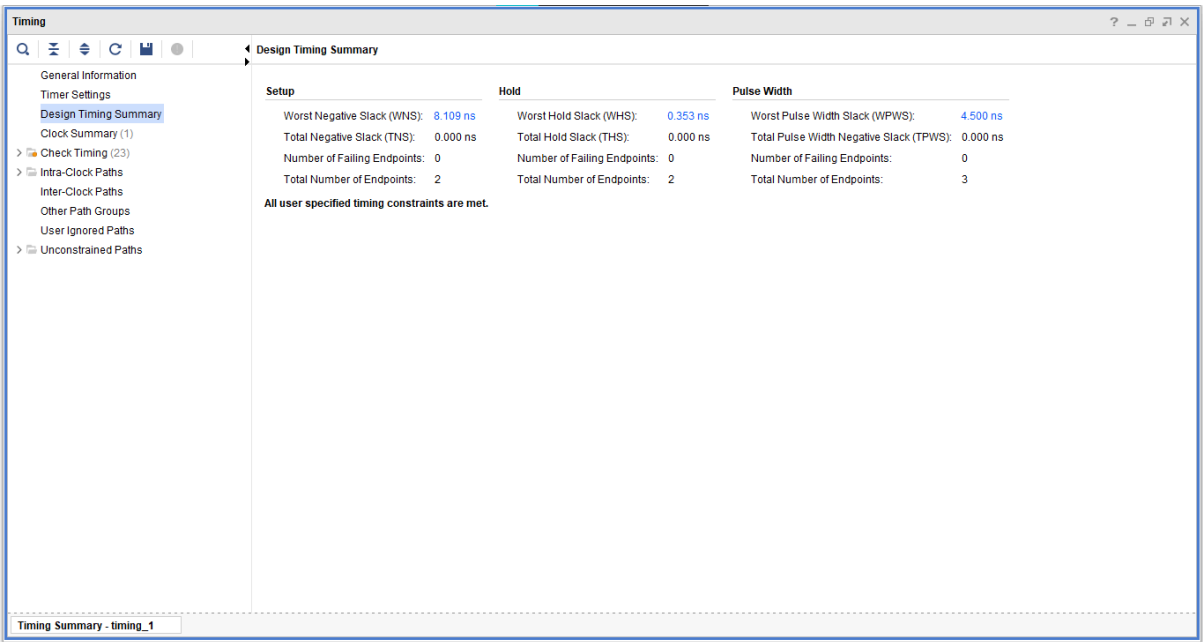


## Schematic snippet

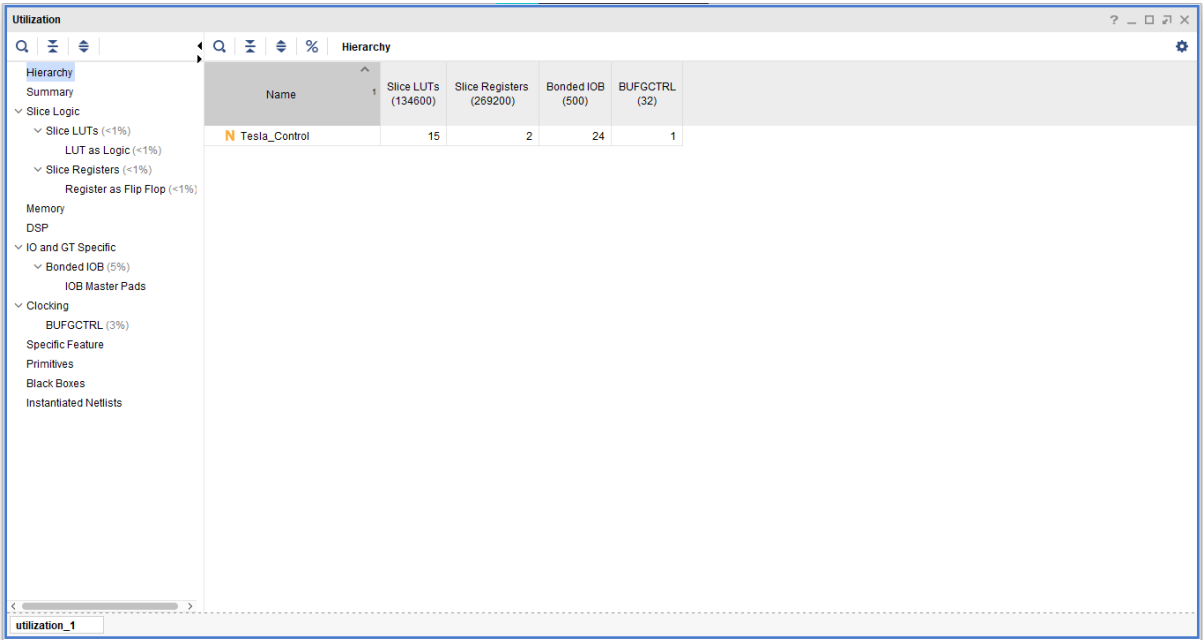




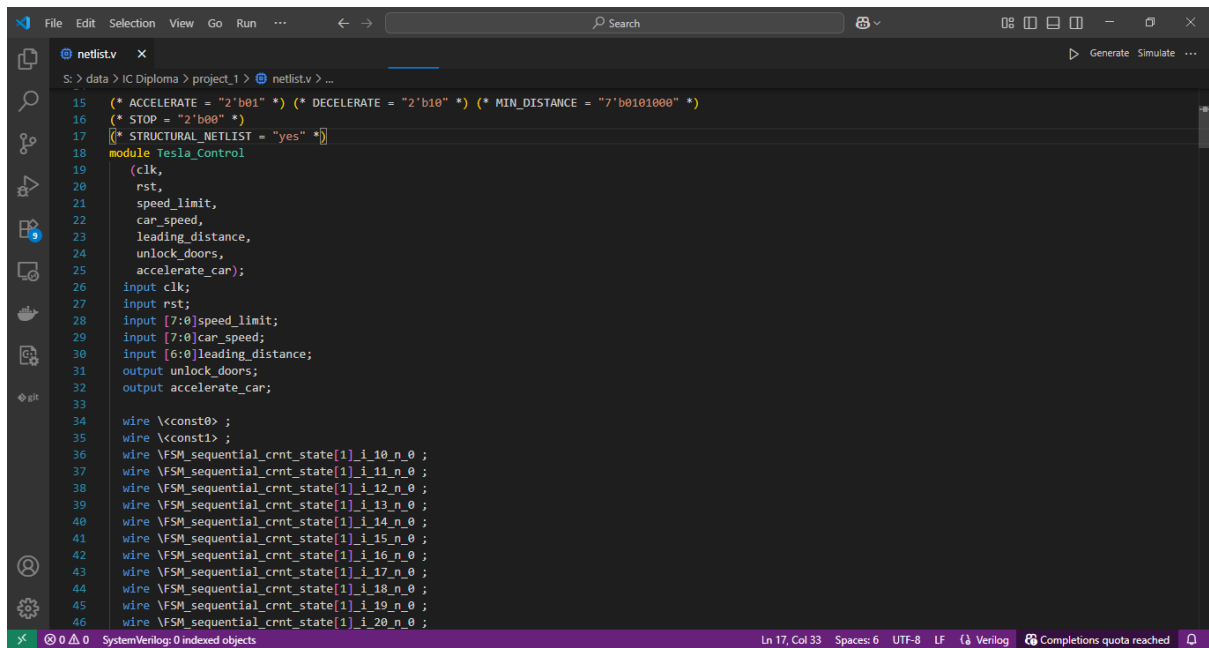
# Timing report



# Utilization report



# Netlist file



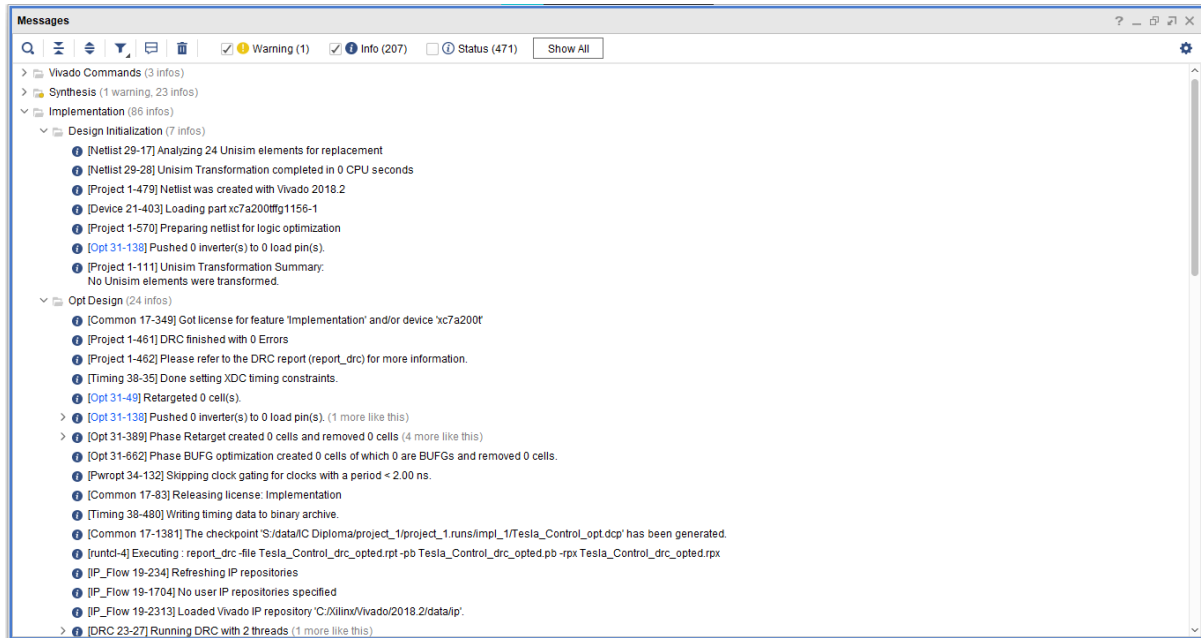
The screenshot shows a code editor window titled "netlist.v" with a dark theme. The editor contains Verilog code for a module named "Tesla\_Control". The code includes several conditional compilation directives at the top, followed by module ports and a large block of wire declarations for sequential counter states. The status bar at the bottom indicates "SystemVerilog: 0 indexed objects" and "Ln 17, Col 33".

```
15 (* ACCELERATE = "2'b01" *) (* DECELERATE = "2'b10" *) (* MIN_DISTANCE = "2'b0101000" *)
16 (* STOP = "2'b00" *)
17 (* STRUCTURAL_NETLIST = "yes" *)
18 module Tesla_Control
19     (clk,
20      rst,
21      speed_limit,
22      car_speed,
23      leading_distance,
24      unlock_doors,
25      accelerate_car);
26 input clk;
27 input rst;
28 input [7:0]speed_limit;
29 input [7:0]car_speed;
30 input [6:0]leading_distance;
31 output unlock_doors;
32 output accelerate_car;
33
34 wire \<const0> ;
35 wire \<const1> ;
36 wire \FSM_sequential_crnt_state[1]_i_10_n_0 ;
37 wire \FSM_sequential_crnt_state[1]_i_11_n_0 ;
38 wire \FSM_sequential_crnt_state[1]_i_12_n_0 ;
39 wire \FSM_sequential_crnt_state[1]_i_13_n_0 ;
40 wire \FSM_sequential_crnt_state[1]_i_14_n_0 ;
41 wire \FSM_sequential_crnt_state[1]_i_15_n_0 ;
42 wire \FSM_sequential_crnt_state[1]_i_16_n_0 ;
43 wire \FSM_sequential_crnt_state[1]_i_17_n_0 ;
44 wire \FSM_sequential_crnt_state[1]_i_18_n_0 ;
45 wire \FSM_sequential_crnt_state[1]_i_19_n_0 ;
46 wire \FSM_sequential_crnt_state[1]_i_20_n_0 ;
```

Ln 17, Col 33 Spaces: 6 UTF-8 LF Verilog Completions quota reached

# Implementation

## “Messages” tab



## Utilization report

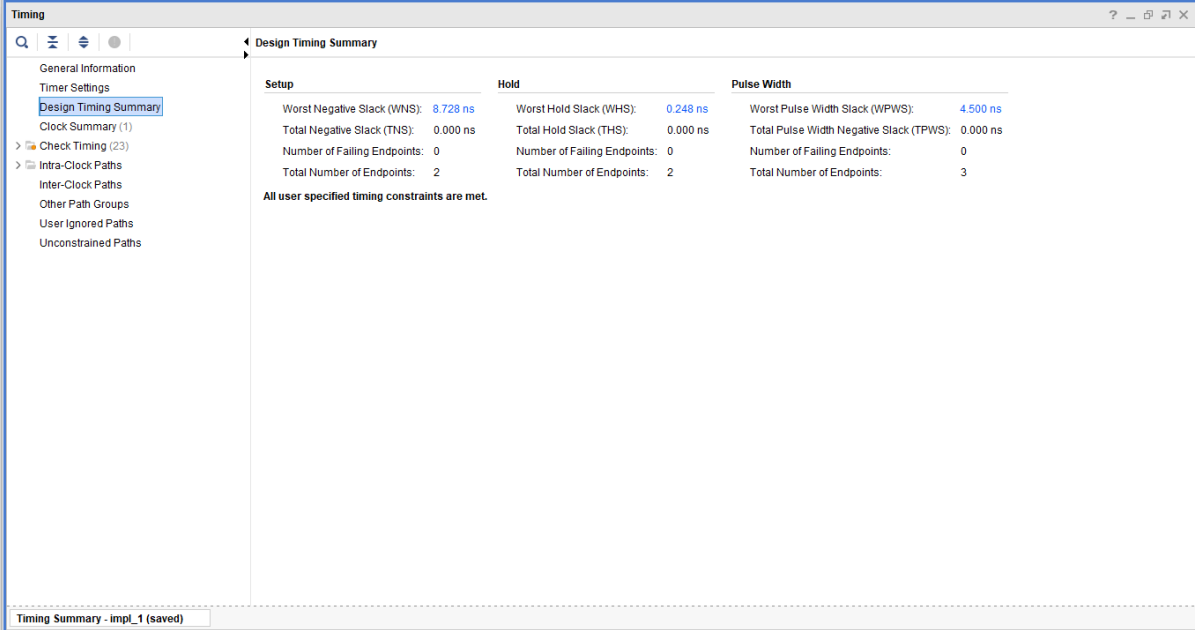
The screenshot shows the Vivado Utilization window for 'utilization\_1'. The table below represents the data shown in the main pane.

Name	Slice LUTs (133800)	Slice Registers (267600)	Slice (33450)	LUT as Logic (133800)	LUT Flip Flop Pairs (133800)	Bonded IOB (500)	BUFGCTRL (32)
Tesla_Control	15	2	6	15	2	24	1

The left sidebar shows a hierarchy of utilization categories:

- Hierarchy
  - Summary
  - Slice Logic
    - Slice LUTs (<1%)
      - LUT as Logic (<1%)
    - Slice Registers (<1%)
      - Register as Flip Flop (<1%)
    - Slice Logic Distribution
      - Slice (<1%)
        - SliceLUT
      - LUT Flip Flop Pairs (<1%)
        - LUT-FF pairs with one un...
        - LUT-FF pairs with one un...
      - LUT as Logic (<1%)
        - using O5 and O6
        - using O6 output only
    - Memory
      - DSP
    - IO and GT Specific
      - Bonded IOB (5%)
        - IOB Slave Pads
        - IOB Master Pads
    - Clocking
      - BUFGCTRL (3%)
    - Specific Feature
      - Primitives
      - Block Boxes

# Timing report

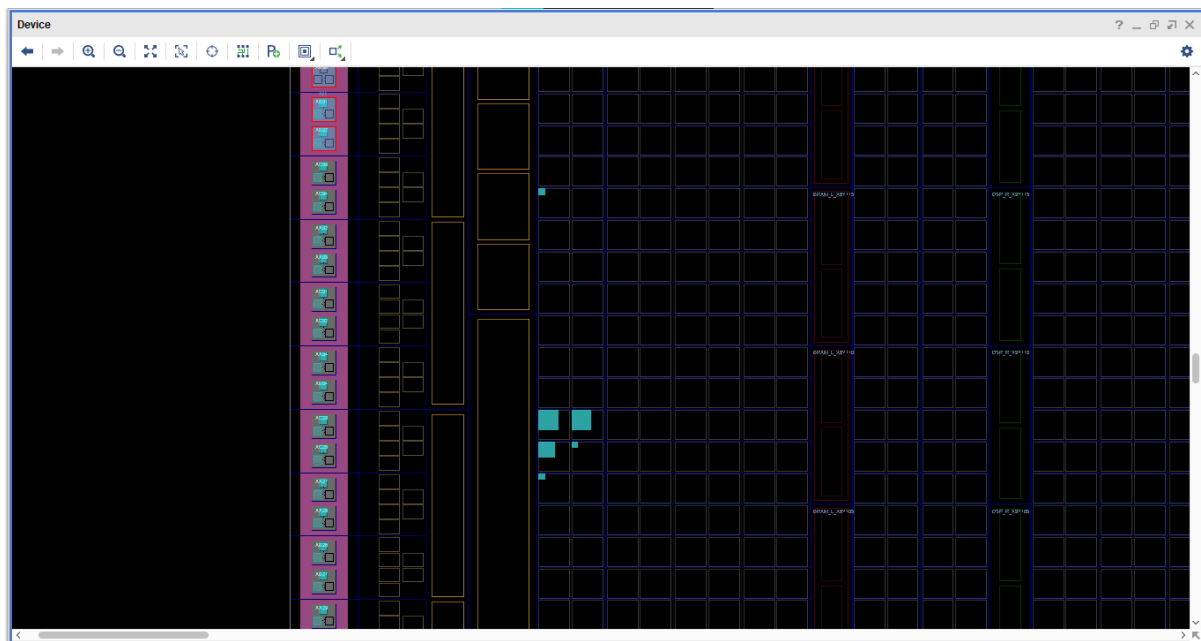


The image shows a screenshot of the 'Timing' window in a design tool, specifically the 'Design Timing Summary' tab. The window has a sidebar on the left with a tree view containing 'General Information', 'Timer Settings', 'Design Timing Summary' (selected), 'Clock Summary (1)', 'Check Timing (23)', 'Intra-Clock Paths', 'Inter-Clock Paths', 'Other Path Groups', 'User Ignored Paths', and 'Unconstrained Paths'. The main area displays a table with three columns: 'Setup', 'Hold', and 'Pulse Width'. Each column contains three rows of data: 'Worst Negative Slack (WNS)', 'Total Negative Slack (TNS)', and 'Number of Failing Endpoints'. The 'Setup' column shows WNS as 8.728 ns, TNS as 0.000 ns, and 0 failing endpoints. The 'Hold' column shows WHS as 0.248 ns, THS as 0.000 ns, and 0 failing endpoints. The 'Pulse Width' column shows WPPWS as 4.500 ns, TPWS as 0.000 ns, and 0 failing endpoints. A status message at the bottom of the table states 'All user specified timing constraints are met.' The window title bar reads 'Timing' and the status bar at the bottom says 'Timing Summary - impl\_1 (saved)'.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 8.728 ns	Worst Hold Slack (WHS): 0.248 ns	Worst Pulse Width Slack (WPWS): 4.500 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 2	Total Number of Endpoints: 2	Total Number of Endpoints: 3

All user specified timing constraints are met.

# Device snippet



2.

```
module Gray_Counter(
    input wire clk,
    input wire rst,      // active high async reset
    output reg [1:0] y    // 2-bit Gray output
);
    // FSM state encoding
    localparam A = 2'b00;
    localparam B = 2'b01;
    localparam C = 2'b11;
    localparam D = 2'b10;

    // State registers
    reg [1:0] crnt_state, nxt_state;

    // Sequential: State update
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            crnt_state <= A;    // Reset to state A (00)
        end else begin
            crnt_state <= nxt_state;
        end
    end

    // Combinational: Next-state logic
    always @(*) begin
        case (crnt_state)
            A: nxt_state = B;    // 00 -> 01
            B: nxt_state = C;    // 01 -> 11
            C: nxt_state = D;    // 11 -> 10
            D: nxt_state = A;    // 10 -> 00
            default: nxt_state = A;
        endcase
    end

    // Moore Output: depends only on crnt_state
    always @(*) begin
        case (crnt_state)
            A: y = 2'b00;
            B: y = 2'b01;
            C: y = 2'b11;
            D: y = 2'b10;
            default: y = 2'b00;
        endcase
    end
endmodule
```

```

module Gray_Counter_Golden_Model(
    input wire clk,
    input wire rst,           // active high async reset
    output reg [1:0] gray_out // 2-bit Gray output
);

    reg [1:0] binary_count;

    // Sequential binary counter
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            binary_count <= 2'b00; // reset binary counter
        end else begin
            binary_count <= binary_count + 1'b1; // increment
        end
    end

    // Binary-to-Gray conversion
    always @(*) begin
        gray_out[1] = binary_count[1]; // MSB same
        gray_out[0] = binary_count[1] ^ binary_count[0]; // XOR for LSB
    end

endmodule

module Gray_Counter_tb();

    // Testbench signals
    reg clk;
    reg rst;
    wire [1:0] gray_fsm; // DUT output
    wire [1:0] gray_golden; // Golden model output

    // Instantiate FSM-based Gray counter (DUT)
    Gray_Counter dut (
        .clk(clk),
        .rst(rst),
        .y(gray_fsm)
    );

    // Instantiate Binary-to-Gray counter (Golden model)
    Gray_Counter_Golden_Model golden (
        .clk(clk),
        .rst(rst),
        .gray_out(gray_golden)
    );

    // Clock generation: 100 MHz

```

```

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

// Stimulus
initial begin
    // Apply reset
    rst = 1;
    #12;
    rst = 0;

    // Run for some cycles
    #200;

    $stop;
end

// Monitor outputs
initial begin
    $display("Time\tFSM_Out\tGolden_Out\tMatch");
    $monitor("%0t\t%b\t%b\t%b",
             $time, gray_fsm, gray_golden,
             (gray_fsm == gray_golden));
end

endmodule

```

The screenshot shows the Waveform Viewer in the Wave IDE. The interface includes a menu bar (File, Edit, View, Add, Format, Tools, Bookmarks, Window, Help) and a toolbar with various icons for waveform manipulation. The main display area shows a digital waveform for four signals: clk, rst, gray\_fsm, and gray\_golden. The signals are plotted as green and blue lines on a black background. The time axis is marked with major ticks every 40,000 ps, ranging from 0 ps to 222600 ps. The signal names are listed in a table on the left.

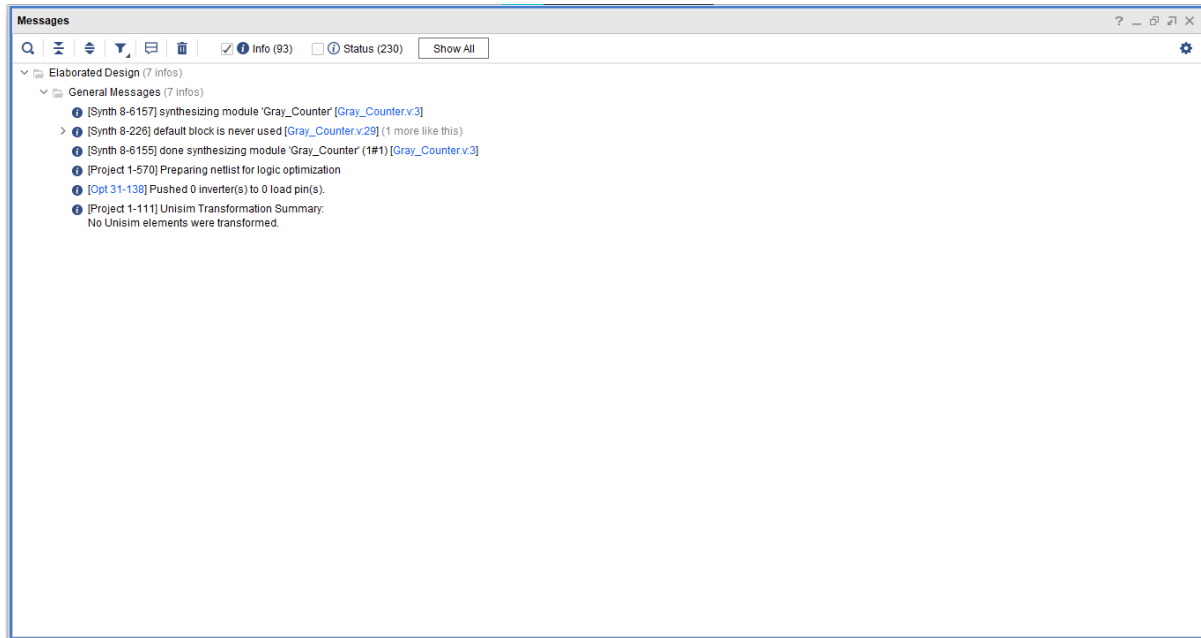
Signal	Value
clk	1
rst	0
gray_fsm	0
gray_golden	0

# Time	FSM_Out		Golden_Out	Match
# 0	00	00	1	
# 15000	01	01	1	
# 25000	11	11	1	
# 35000	10	10	1	
# 45000	00	00	1	
# 55000	01	01	1	
# 65000	11	11	1	
# 75000	10	10	1	
# 85000	00	00	1	
# 95000	01	01	1	
# 105000	11	11	1	
# 115000	10	10	1	
# 125000	00	00	1	
# 135000	01	01	1	
# 145000	11	11	1	
# 155000	10	10	1	
# 165000	00	00	1	
# 175000	01	01	1	
# 185000	11	11	1	
# 195000	10	10	1	
# 205000	00	00	1	

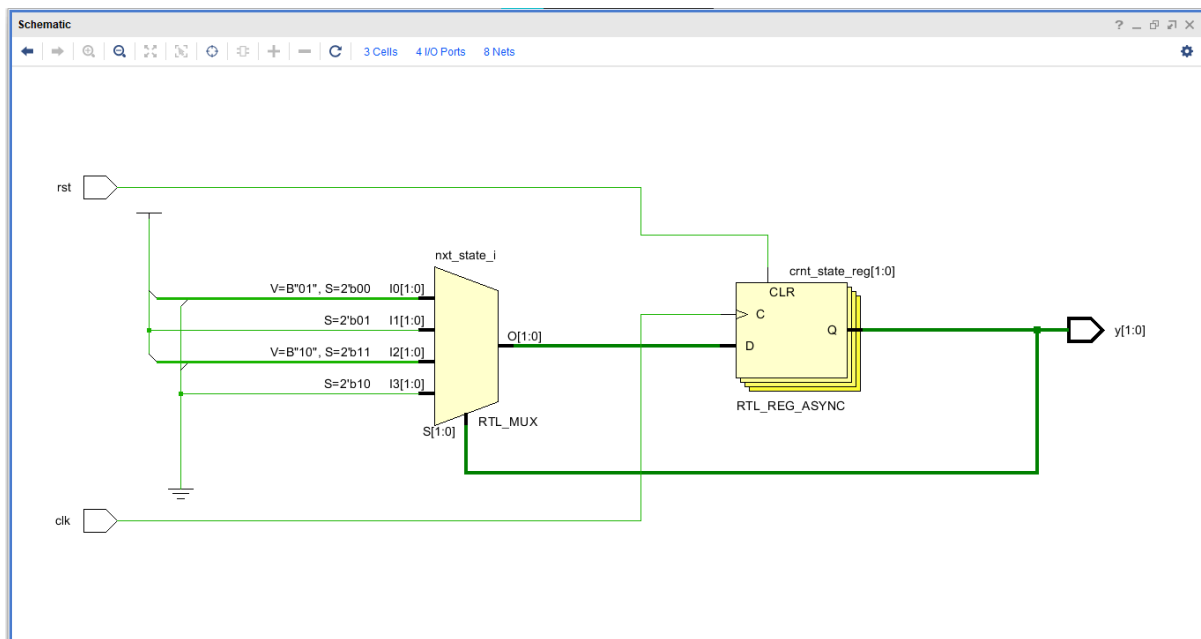


# Elaboration

## “Messages” tab

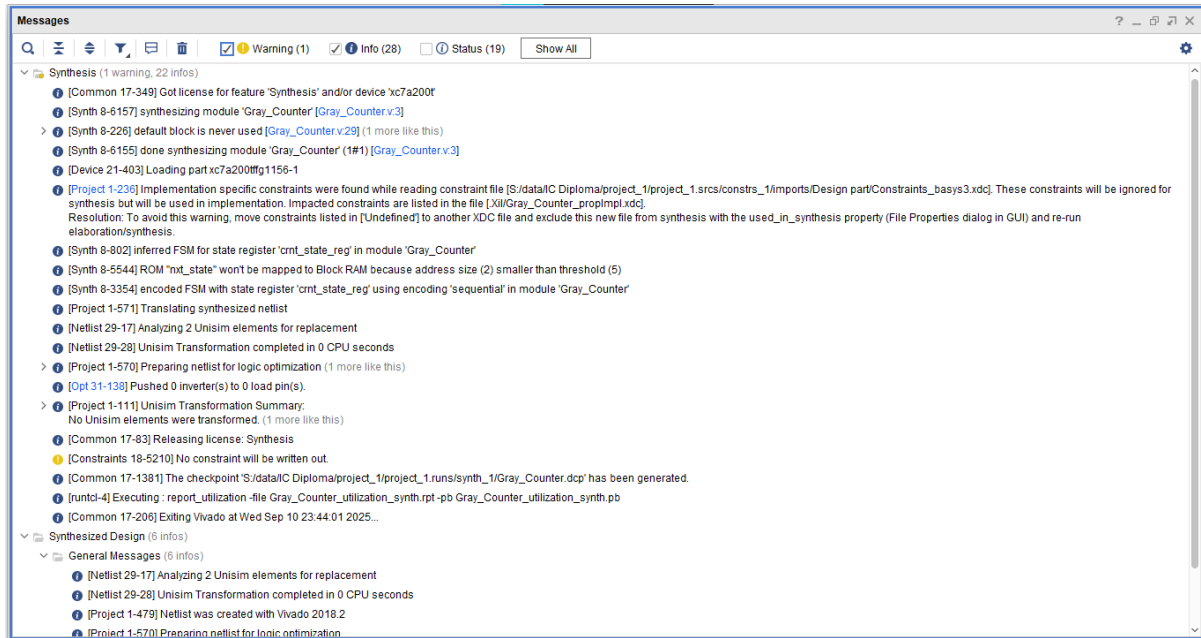


## Schematic snippet

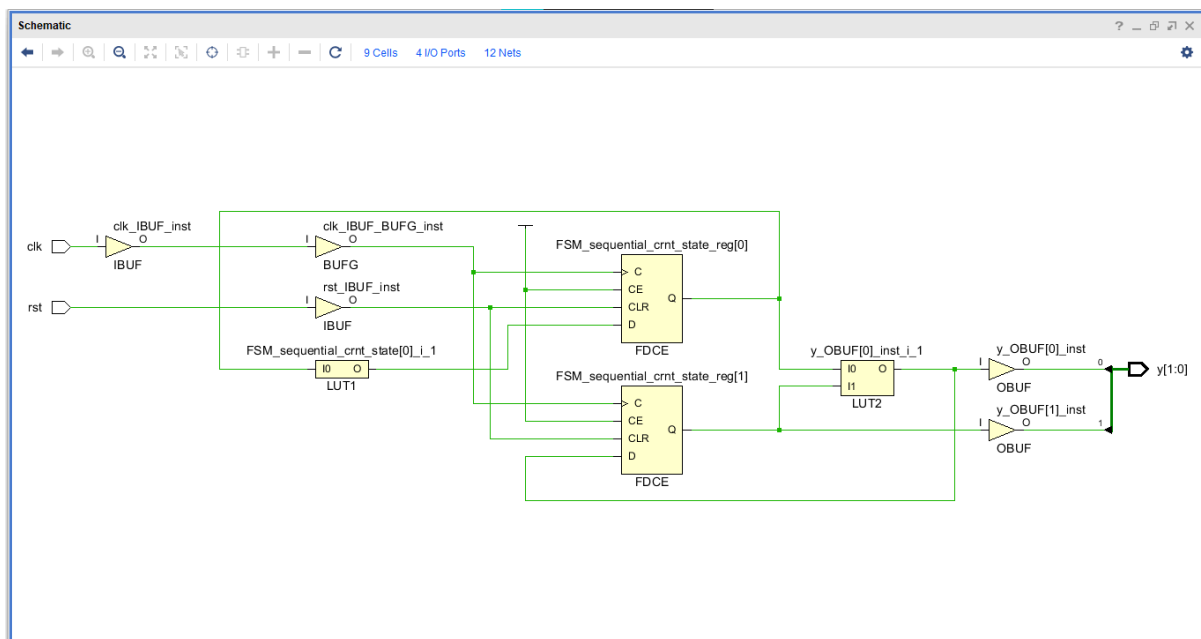


# Synthesis

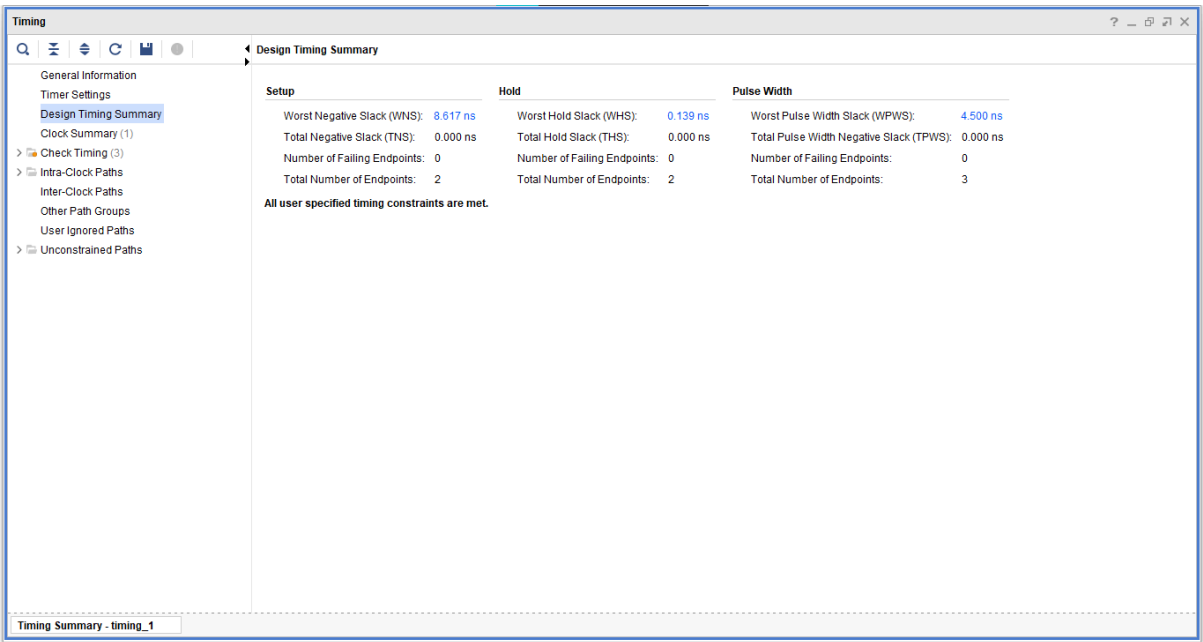
## “Messages” tab



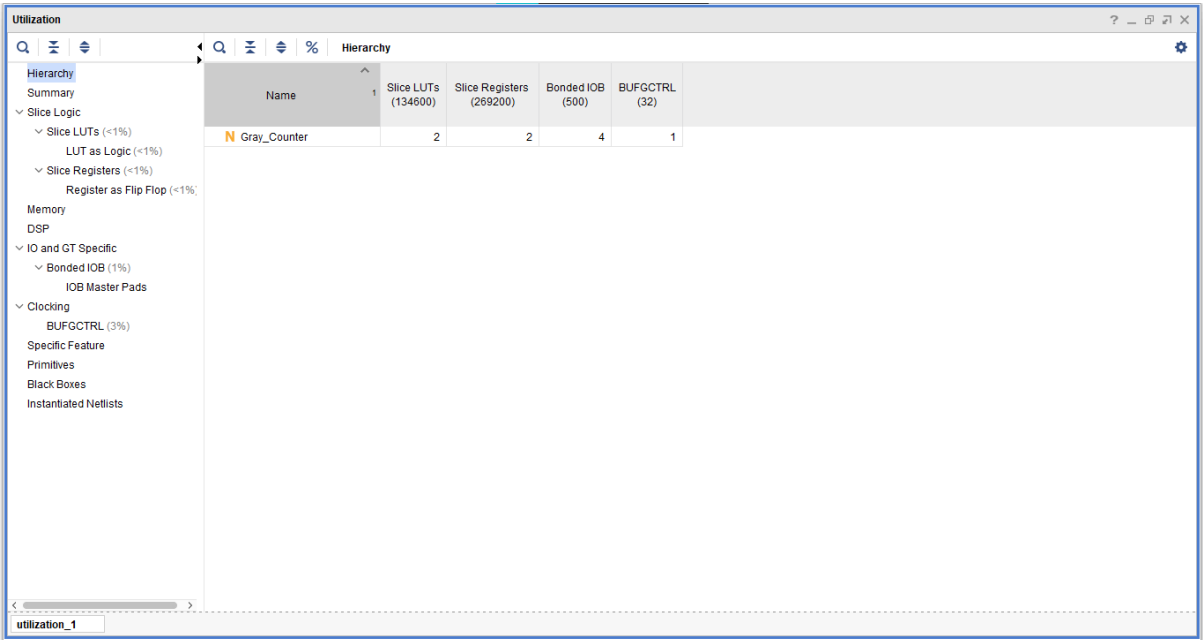
## Schematic snippet



# Timing report

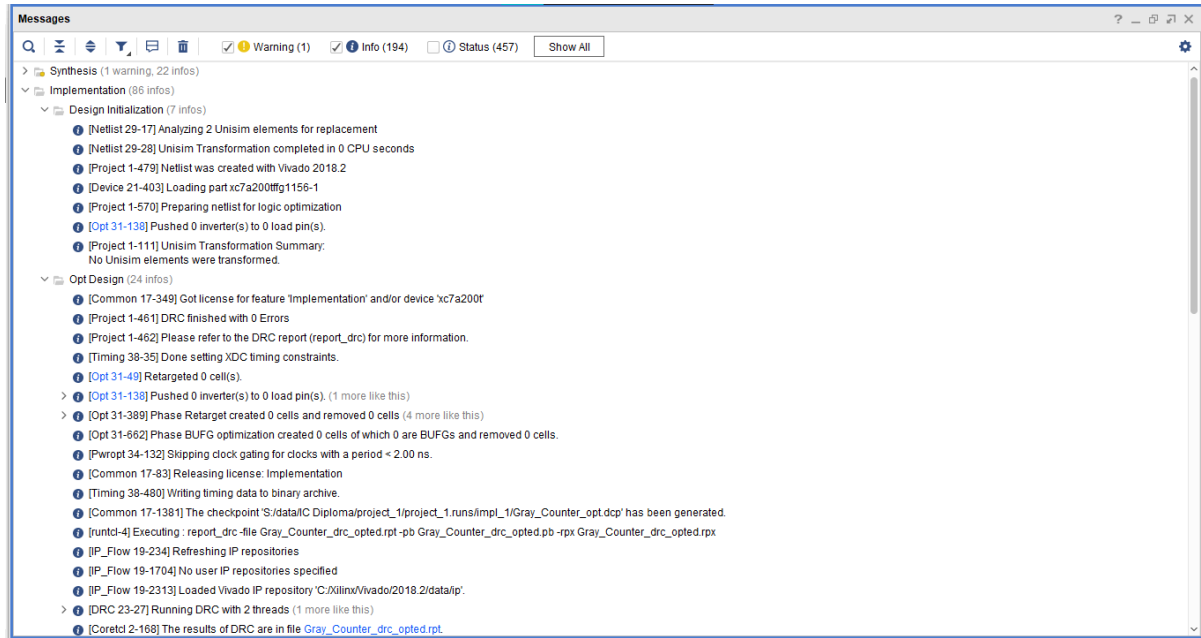


# Utilization report



# Implementation

## “Messages” tab



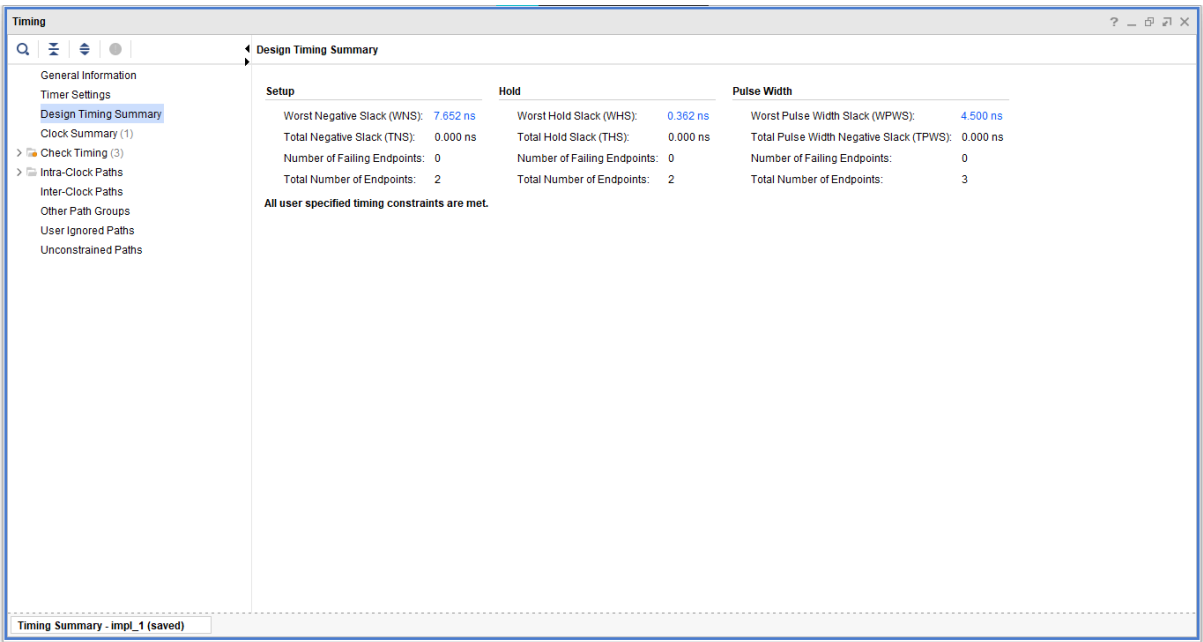
## Utilization report

The screenshot shows the Utilization window in Vivado, displaying the hierarchy of resources. The window is titled "Utilization" and has a toolbar with icons for search, filter, and other functions. The resources are organized into a tree view on the left, with the following categories:

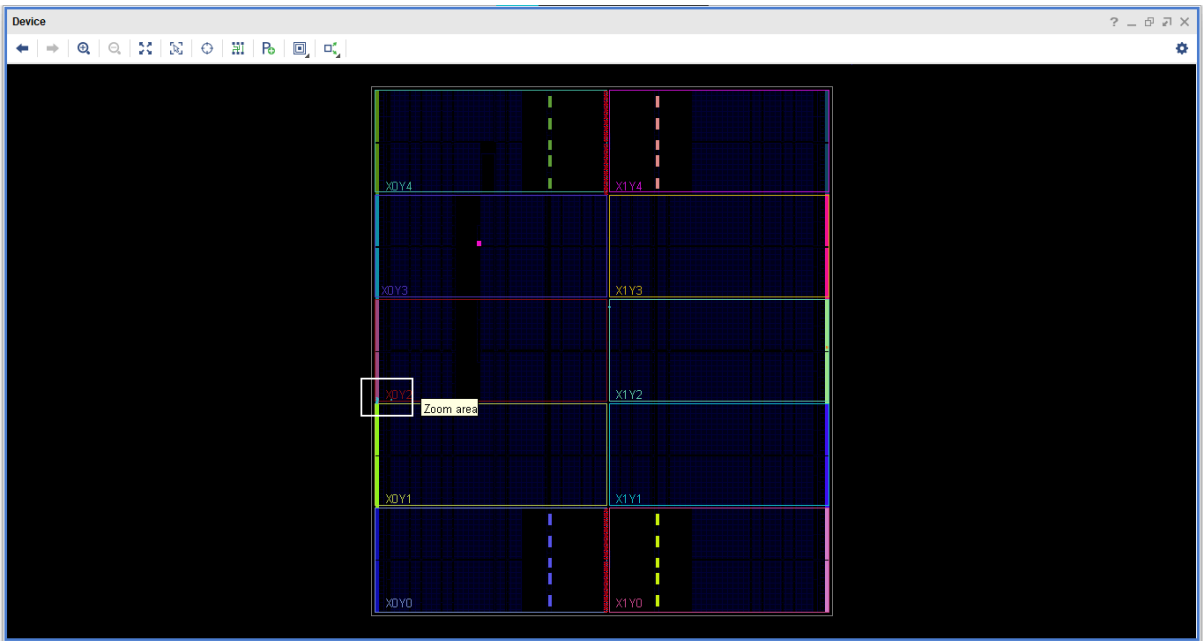
- Hierarchy
  - Summary
  - Slice Logic
    - Slice LUTs (<1%)
      - LUT as Logic (<1%)
    - Slice Registers (<1%)
      - Register as Flip Flop (<1%)
    - Slice Logic Distribution
      - Slice (<1%)
        - SLICEL
      - LUT Flip Flop Pairs (<1%)
        - LUT-FF pairs with one unus
      - LUT as Logic (<1%)
        - using O6 output only
    - Memory
    - DSP
    - IO and GT Specific
      - Bonded IOB (1%)
        - IOB Slave Pads
        - IOB Master Pads
    - Clocking
      - BUFGCTRL (3%)
    - Specific Feature
    - Primitives
    - Black Boxes
    - Instantiated Netlists

Name	Slice LUTs (133800)	Slice Registers (267600)	Slice (33450)	LUT as Logic (133800)	LUT Flip Flop Pairs (133800)	Bonded IOB (500)	BUFGCTRL (32)
Gray_Counter	2	2	1	2	1	4	1

# Timing report



# Device snippet



3.

```
module Seq_111_Detector(  
    input wire clk,  
    input wire rst,    // async active-high reset  
    input wire Din,    // serial data input (1 bit per clock)  
    output reg ERR     // error flag: high when "111" detected  
);  
  
    // State encoding  
    localparam START      = 3'b000;  
    localparam D0_IS_1    = 3'b001; // first 1 seen  
    localparam D1_IS_1    = 3'b010; // second consecutive 1 seen  
    localparam D0_NOT_1   = 3'b011; // first bit was 0  
    localparam D1_NOT_1   = 3'b100; // second bit was 0  
  
    reg [2:0] crnt_state, nxt_state;  
  
    // State register  
    always @(posedge clk or posedge rst) begin  
        if (rst) begin  
            crnt_state <= START;  
        end else begin  
            crnt_state <= nxt_state;  
        end  
    end  
  
    // Next-state logic + output (Mealy)  
    always @(*) begin  
        // defaults  
        nxt_state = crnt_state;  
        ERR = 1'b0;  
  
        case (crnt_state)  
            START: begin  
                if (Din) begin  
                    nxt_state = D0_IS_1;  
                end else begin  
                    nxt_state = D0_NOT_1;  
                end  
            end  
  
            D0_IS_1: begin  
                if (Din) begin  
                    nxt_state = D1_IS_1;  
                end else begin  
                    nxt_state = D1_NOT_1;  
                end  
            end  
        end  
    end
```

```

        end

        D0_NOT_1: begin
            // regardless of Din, move on
            nxt_state = D1_NOT_1;
        end

        D1_IS_1: begin
            // regardless of Din, back to start
            nxt_state = START;
            if (Din) begin
                // detected 111
                ERR = 1'b1;
            end
        end

        D1_NOT_1: begin
            // regardless of Din, back to start
            nxt_state = START;
        end

        default: begin
            nxt_state = START;
        end
    endcase
end

endmodule

```

```

module Seq_111_Detector_tb();

    reg clk;
    reg rst;
    reg Din;
    wire ERR;

    // Instantiate DUT
    Seq_111_Detector DUT (
        .Din(Din),
        .clk(clk),
        .rst(rst),
        .ERR(ERR)
    );

    // Clock generation (100 MHz ? 10 ns period)
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
endmodule

```

```

end

// Stimulus
initial begin
    // Apply reset
    rst = 1;
    Din = 0;
    #15;
    rst = 0;

    // Generate random serial data for 50 cycles
    repeat (50) begin
        @(posedge clk);
        Din = $random % 2;    // Random 0 or 1
    end

    // Finish simulation
    #50;
    $stop;
end

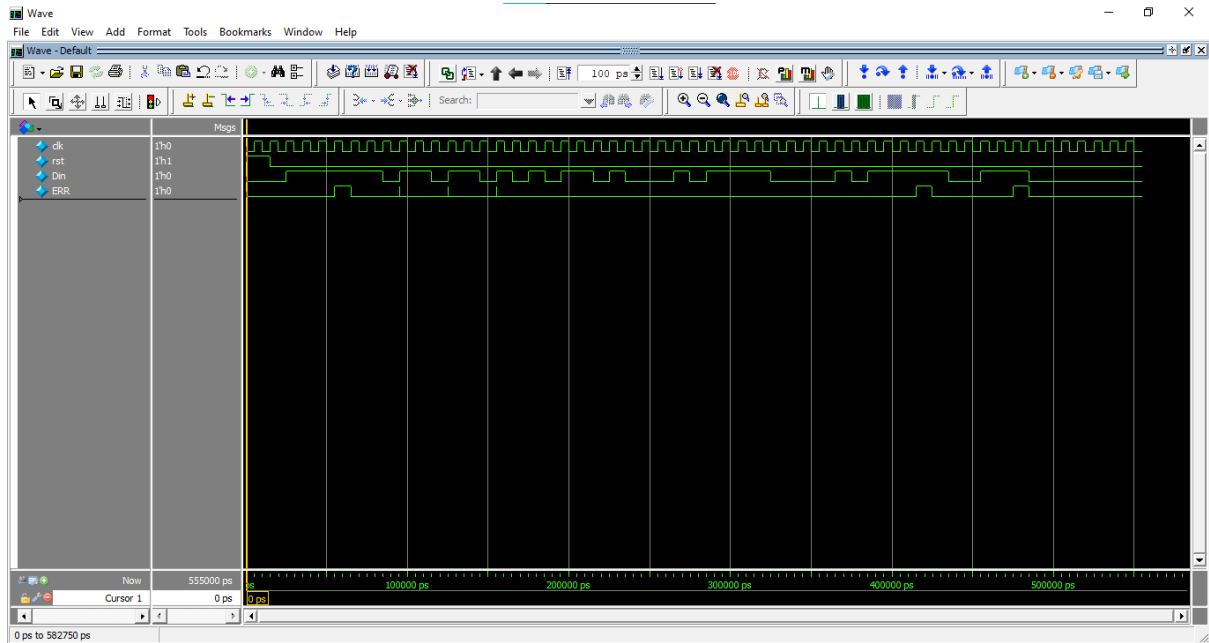
// Display activity in console
initial begin
    $display("Time\tDin\tERR\tState");
    $monitor("%0t\t%b\t%b\t%b", $time, Din, ERR, DUT.crnt_state);
end

endmodule

```



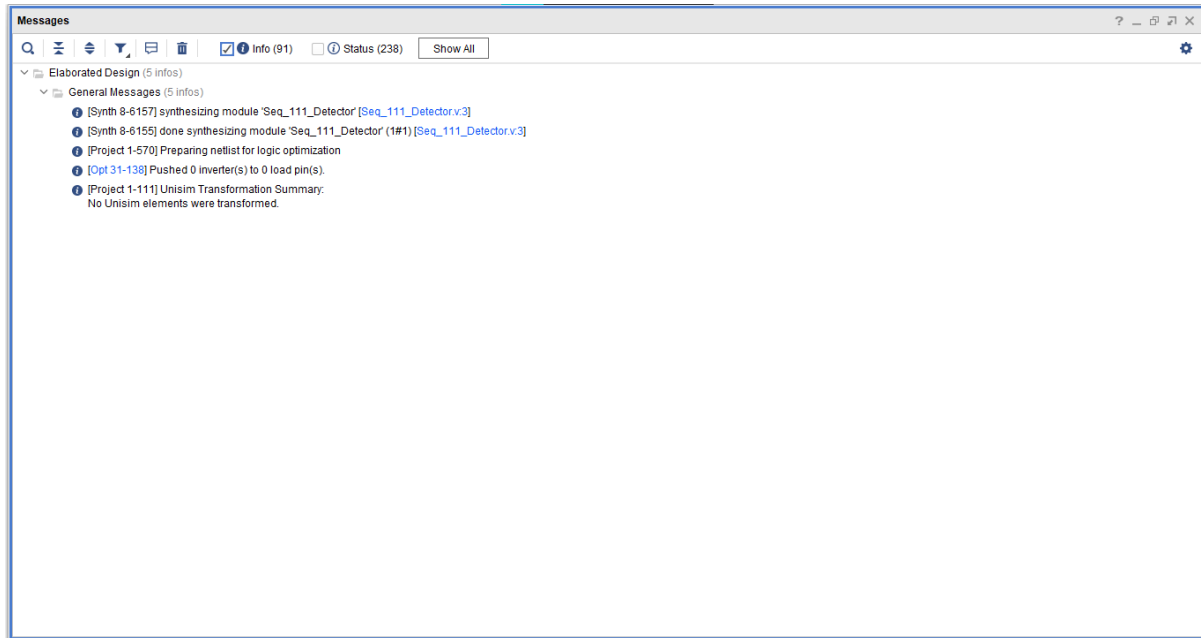
# QuestaSim snippet



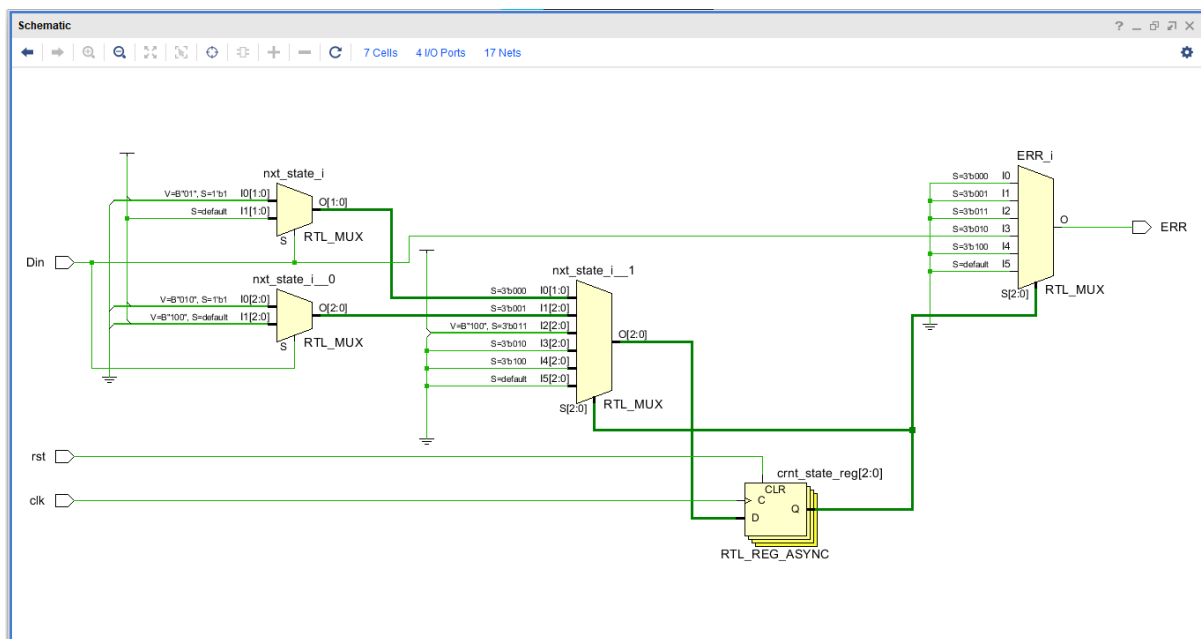
#	Time	Din	ERR	State
#	0	0	0	000
#	15000	0	0	011
#	25000	1	0	100
#	35000	1	0	000
#	45000	1	0	001
#	55000	1	1	010
#	65000	1	0	000
#	75000	1	0	001
#	85000	0	0	010
#	95000	1	0	000
#	105000	1	0	001
#	115000	0	0	010
#	125000	1	0	000
#	135000	1	0	001
#	145000	0	0	010
#	155000	1	0	000
#	165000	0	0	001
#	175000	1	0	100
#	185000	0	0	000
#	195000	1	0	011
#	205000	1	0	100
#	215000	0	0	000
#	225000	1	0	011
#	235000	0	0	100
#	245000	0	0	000
#	255000	0	0	011
#	265000	1	0	100

# Elaboration

## “Messages” tab

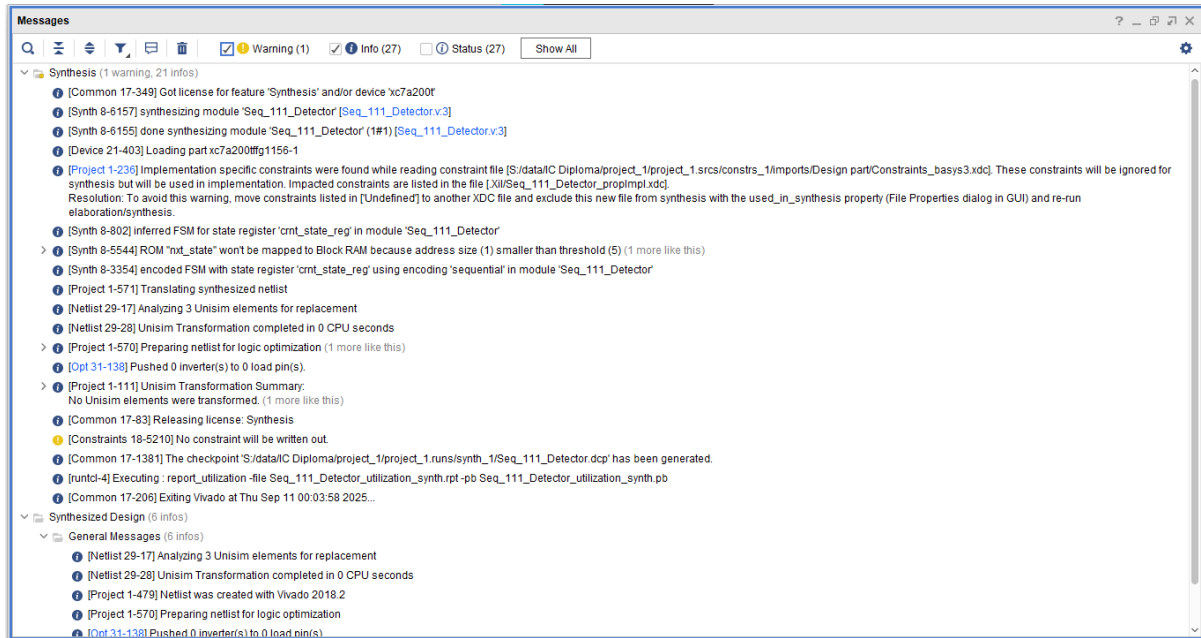


## Schematic snippet

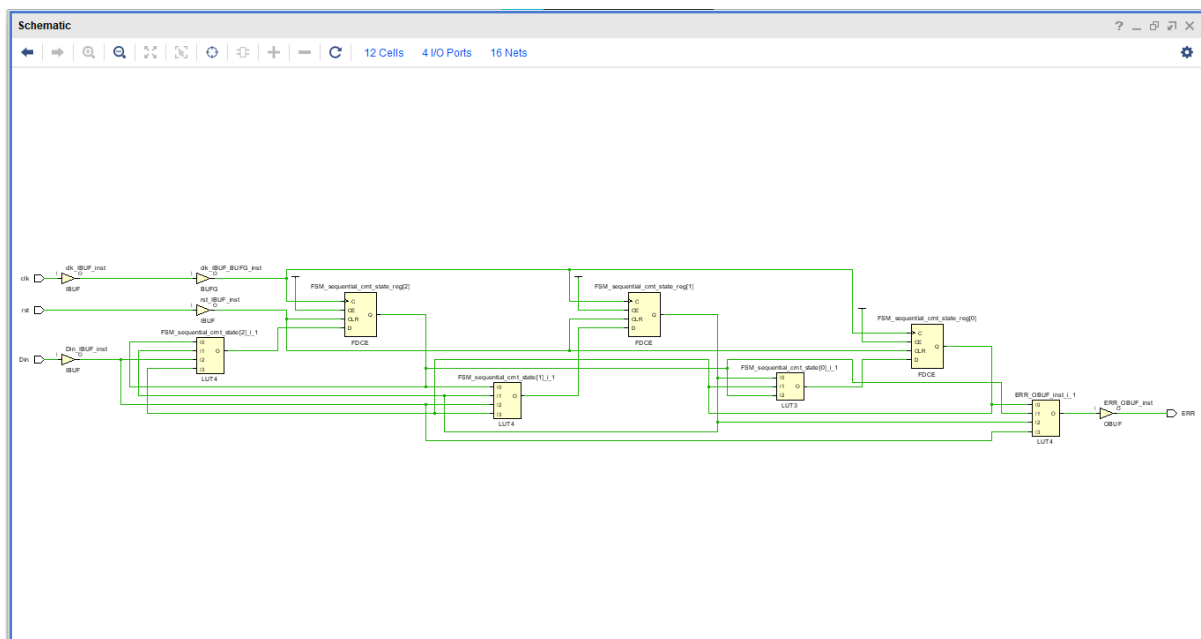


# Synthesis

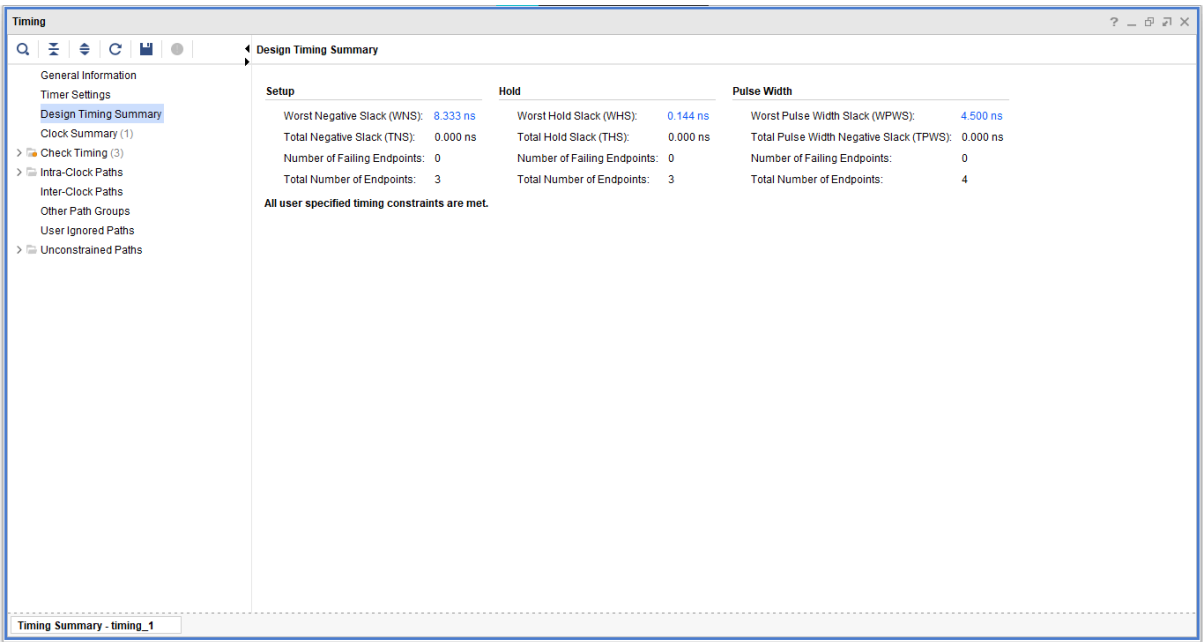
## “Messages” tab



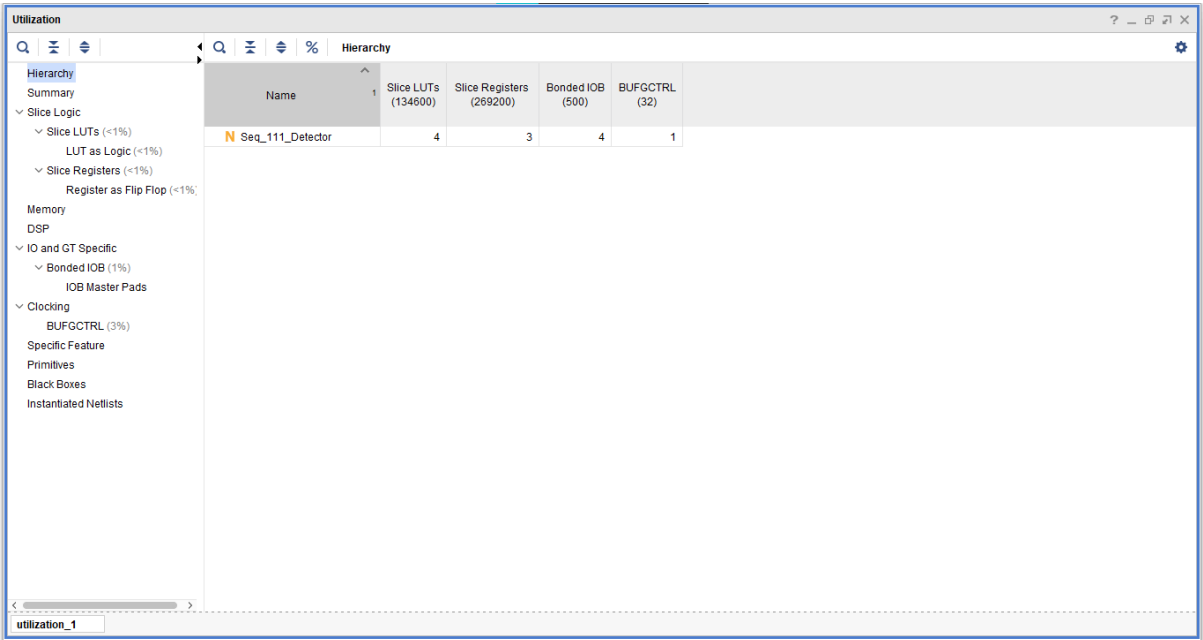
## Schematic snippet



# Timing report

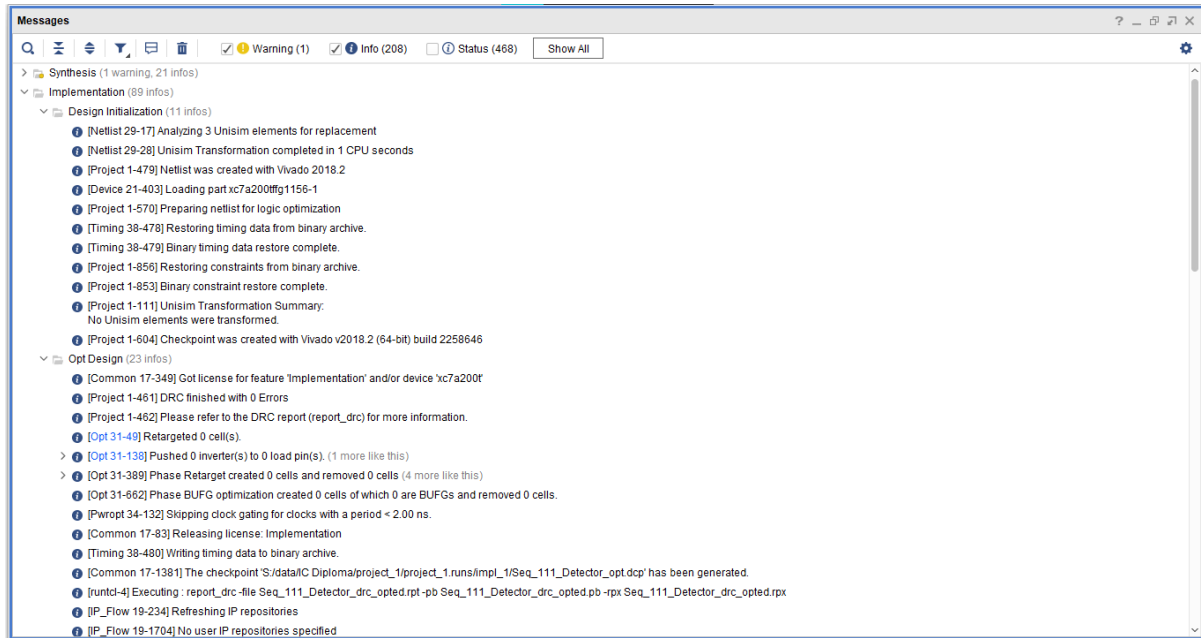


# Utilization report



# Implementation

## “Messages” tab



## Utilization report

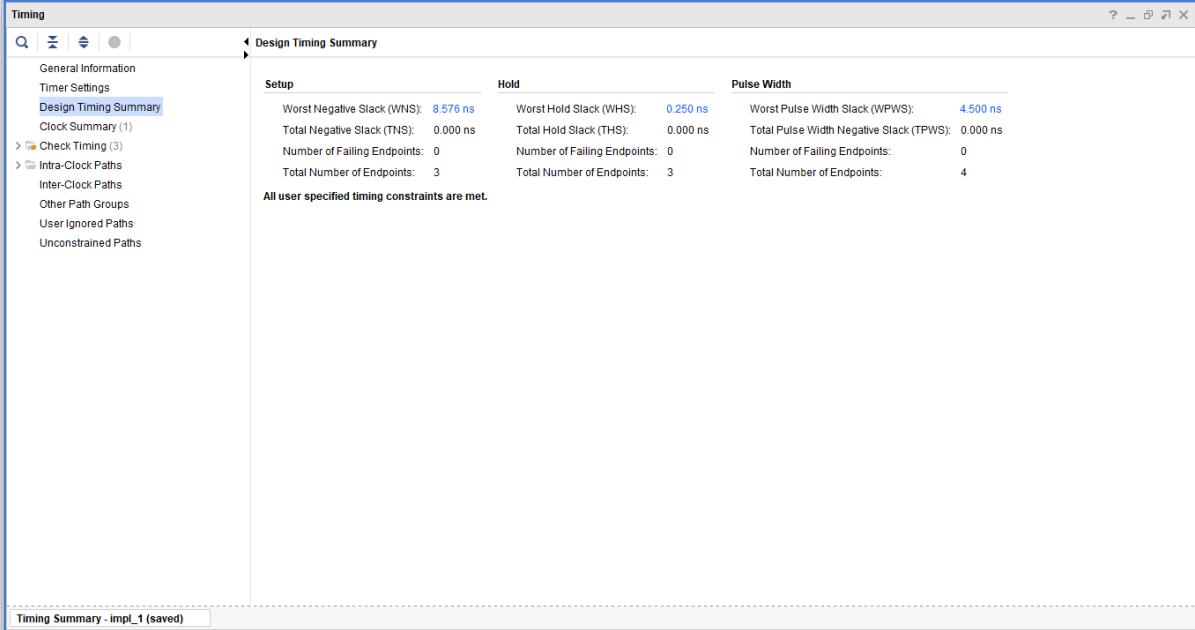
The screenshot shows the Utilization window in Vivado, displaying the Hierarchy tab. The window has a toolbar with icons for search, zoom, and other functions. The left sidebar shows a tree view of the utilization categories, including:

- Hierarchy
- Summary
- Slice Logic
  - Slice LUTs (<1%)
    - LUT as Logic (<1%)
  - Slice Registers (<1%)
    - Register as Flip Flop (<1%)
  - Slice Logic Distribution
    - Slice (<1%)
      - SLICEL
    - LUT Flip Flop Pairs (<1%)
      - LUT-FF pairs with one unused
      - LUT-FF pairs with one unused
    - LUT as Logic (<1%)
      - using O6 output only
  - Memory
  - DSP
  - IO and GT Specific
    - Bonded IOB (1%)
      - IOB Slave Pads
      - IOB Master Pads
  - Clocking
    - BUFGCTRL (3%)
  - Specific Feature
  - Primitives
  - Block Boxes
  - Instantiated Netlists

The main table displays the utilization data for the selected hierarchy. The table has the following columns:

Name	Slice LUTs (133800)	Slice Registers (267600)	Slice (33450)	LUT as Logic (133800)	LUT Flip Flop Pairs (133800)	Bonded IOB (500)	BUFGCTRL (32)
Seq_111_Detector	4	3	1	4	3	4	1

# Timing report



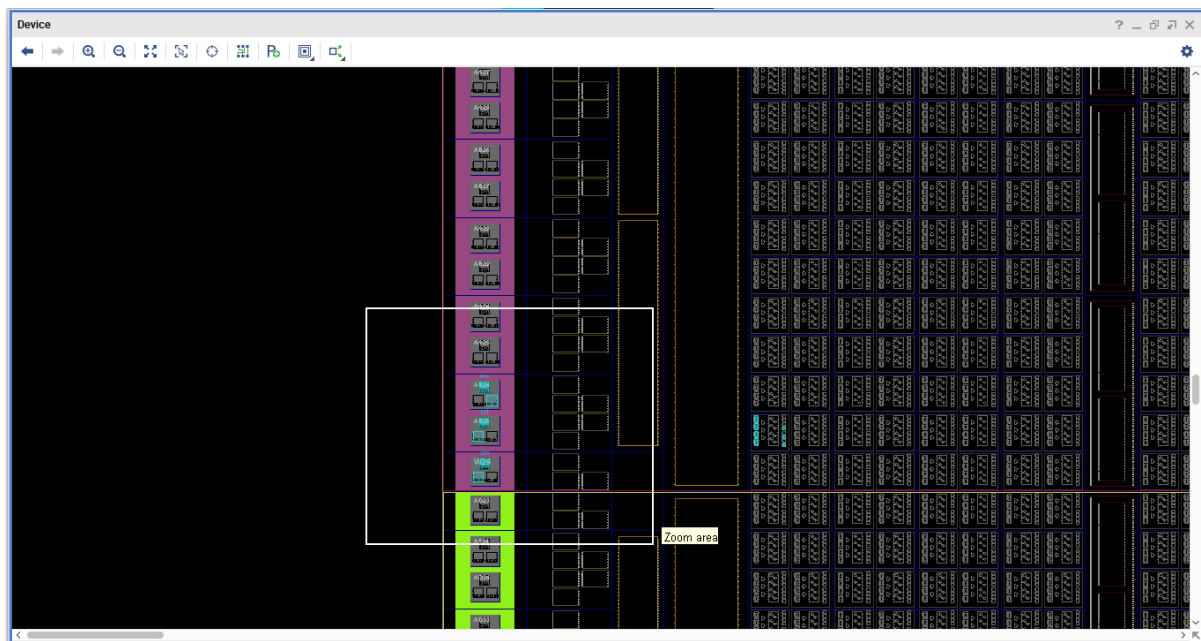
The image shows a 'Timing' window with a 'Design Timing Summary' tab. The left sidebar lists various timing-related categories. The main content area displays a summary of timing metrics for Setup, Hold, and Pulse Width constraints. All metrics indicate that the design meets the specified constraints.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.576 ns	Worst Hold Slack (WHS):	0.250 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	3	Total Number of Endpoints:	3	Total Number of Endpoints:	4

All user specified timing constraints are met.

Timing Summary - impl\_1 (saved)

# Device snippet



4.

```
module Single_Port_Sync_RAM #(
    parameter MEM_WIDTH      = 16,
    parameter MEM_DEPTH      = 1024,
    parameter ADDR_SIZE      = $clog2(MEM_DEPTH),
    parameter ADDR_PIPELINE  = "FALSE", // "TRUE" or "FALSE"
    parameter DOUT_PIPELINE  = "TRUE",  // "TRUE" or "FALSE"
    parameter PARITY_ENABLE  = 1
)(
    input wire clk,
    input wire rst, // synchronous active high
reset
    input wire wr_en, // write enable
    input wire rd_en, // read enable
    input wire blk_select, // chip enable
    input wire [ADDR_SIZE-1:0] addr, // input address
    input wire [MEM_WIDTH-1:0] din, // data input
    input wire addr_en, // enable for pipelined
address
    input wire dout_en, // enable for pipelined data
out
    output reg [MEM_WIDTH-1:0] dout, // data output
    output wire parity_out // even parity of dout
);

// Memory array
reg [MEM_WIDTH-1:0] mem [0:MEM_DEPTH-1];

// Address pipeline register
reg [ADDR_SIZE-1:0] addr_reg;
wire [ADDR_SIZE-1:0] eff_addr;

generate
    if (ADDR_PIPELINE == "TRUE") begin
        always @(posedge clk) begin
            if (rst) begin
                addr_reg <= {ADDR_SIZE{1'b0}};
            end else if (addr_en) begin
                addr_reg <= addr;
            end
        end
        assign eff_addr = addr_reg;
    end else begin
        assign eff_addr = addr;
    end
endgenerate
```

```

// RAM write operation
always @(posedge clk) begin
    if (blk_select && wr_en) begin
        mem[eff_addr] <= din;
    end
end

// Read data (direct or pipelined)
reg [MEM_WIDTH-1:0] dout_reg;

generate
    if (DOUT_PIPELINE == "TRUE") begin
        always @(posedge clk) begin
            if (rst) begin
                dout_reg <= {MEM_WIDTH{1'b0}};
            end else if (blk_select && rd_en && dout_en) begin
                dout_reg <= mem[eff_addr];
            end
        end
        always @(*) begin
            if (blk_select && rd_en) begin
                dout = dout_reg;
            end else begin
                dout = {MEM_WIDTH{1'b0}};
            end
        end
    end else begin
        always @(*) begin
            if (blk_select && rd_en) begin
                dout = mem[eff_addr];
            end else begin
                dout = {MEM_WIDTH{1'b0}};
            end
        end
    end
endgenerate

// Even parity output
generate
    if (PARITY_ENABLE == 1) begin
        assign parity_out = (blk_select && rd_en) ? ^dout : 1'b0;
    end else begin
        assign parity_out = 1'b0;
    end
endgenerate

endmodule

```



```

module Single_Port_Sync_RAM_tb();
    // Parameters
    localparam MEM_WIDTH = 16;
    localparam MEM_DEPTH = 1024;
    localparam ADDR_SIZE = $clog2(MEM_DEPTH);

    // DUT signals
    reg                clk;
    reg                rst;
    reg                wr_en;
    reg                rd_en;
    reg                blk_select;
    reg [ADDR_SIZE-1:0] addr;
    reg [MEM_WIDTH-1:0] din;
    reg                addr_en;
    reg                dout_en;
    wire [MEM_WIDTH-1:0] dout;
    wire                parity_out;

    // Instantiate DUT
    Single_Port_Sync_RAM #(
        .MEM_WIDTH(MEM_WIDTH),
        .MEM_DEPTH(MEM_DEPTH),
        .ADDR_SIZE(ADDR_SIZE),
        .ADDR_PIPELINE("TRUE"),
        .DOUT_PIPELINE("TRUE"),
        .PARITY_ENABLE(1)
    ) DUT (
        .clk(clk),
        .rst(rst),
        .wr_en(wr_en),
        .rd_en(rd_en),
        .blk_select(blk_select),
        .addr(addr),
        .din(din),
        .addr_en(addr_en),
        .dout_en(dout_en),
        .dout(dout),
        .parity_out(parity_out)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 100 MHz clock
    end

    // Stimulus

```

```

integer i;
initial begin
    // Initialize
    rst = 1;
    wr_en = 0;
    rd_en = 0;
    blk_select = 0;
    addr = 0;
    din = 0;
    addr_en = 0;
    dout_en = 0;

    // Release reset
    #20 rst = 0;

    // Perform 20 random write/read operations
    for (i = 0; i < 20; i = i + 1) begin
        @(posedge clk);

        blk_select = 1;
        addr_en     = 1;
        dout_en     = 1;

        // Randomize control signals
        wr_en = $random % 2;
        rd_en = $random % 2;

        // Randomize address and data
        addr = $random % MEM_DEPTH;
        din  = $random;

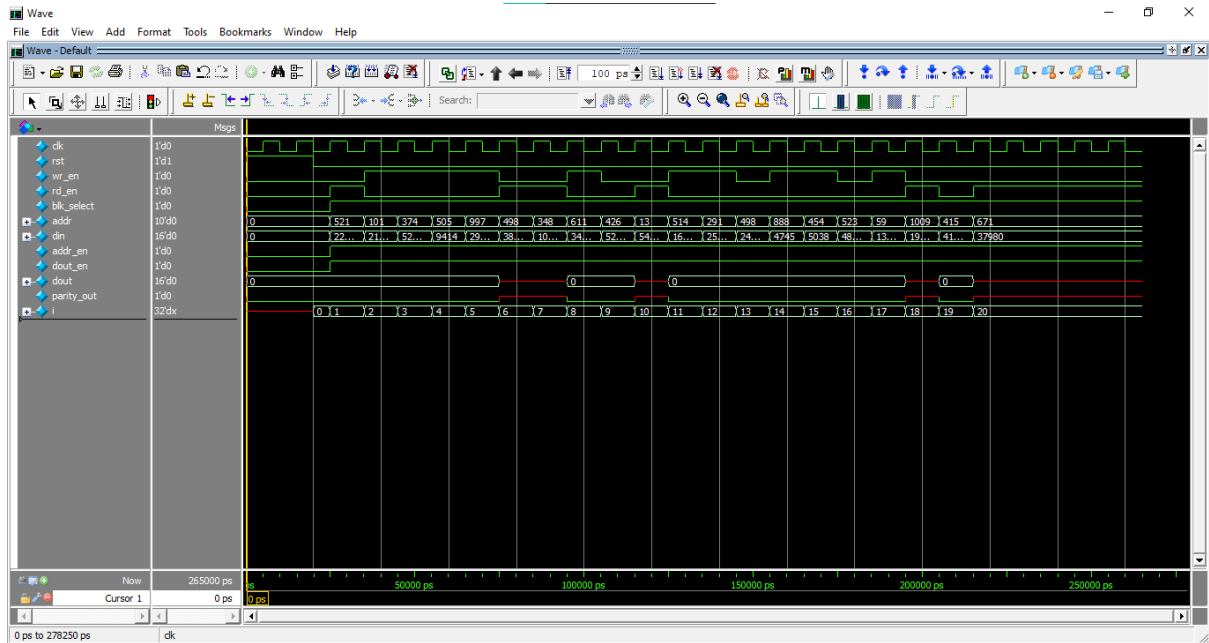
        // Ensure no simultaneous read & write (for clean demo)
        if (wr_en && rd_en)
            rd_en = 0;
    end

    // End simulation
    #50 $stop;
end

// Monitor outputs
initial begin
    $display("Time\tblk_sel\twr_en\trd_en\taddr\tdin\tdout\tparity");
    $monitor("%0t\t%b\t%b\t%b\t%0d\t%h\t%h\t%b",
             $time, blk_select, wr_en, rd_en, addr, din, dout,
parity_out);
    end
endmodule

```

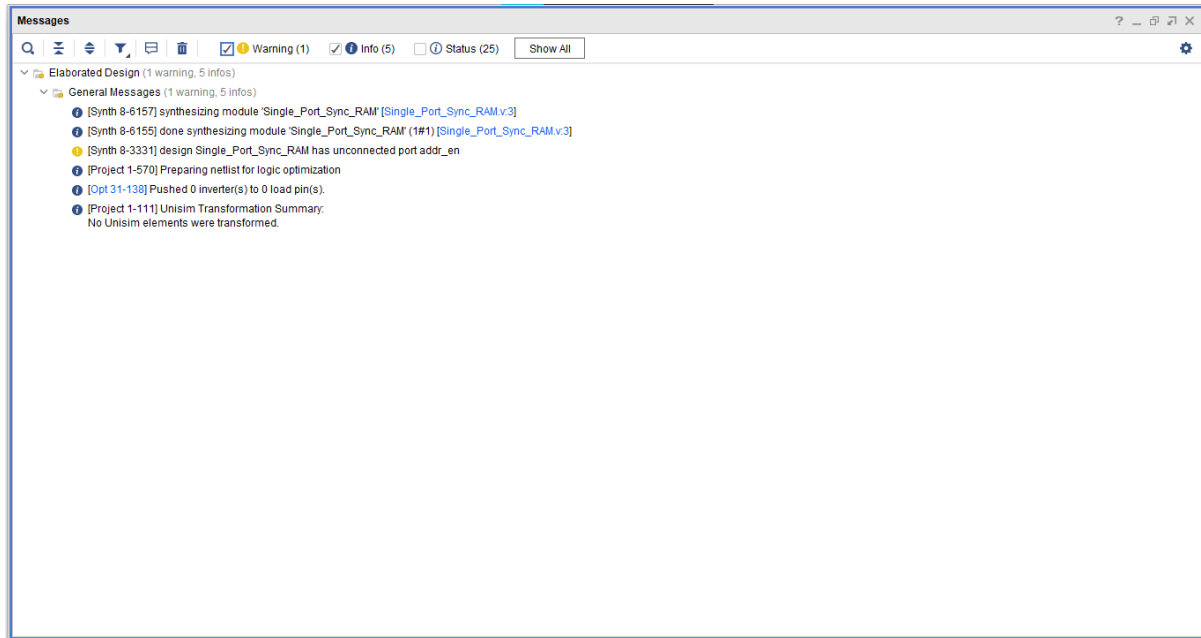
# QuestaSim snippet



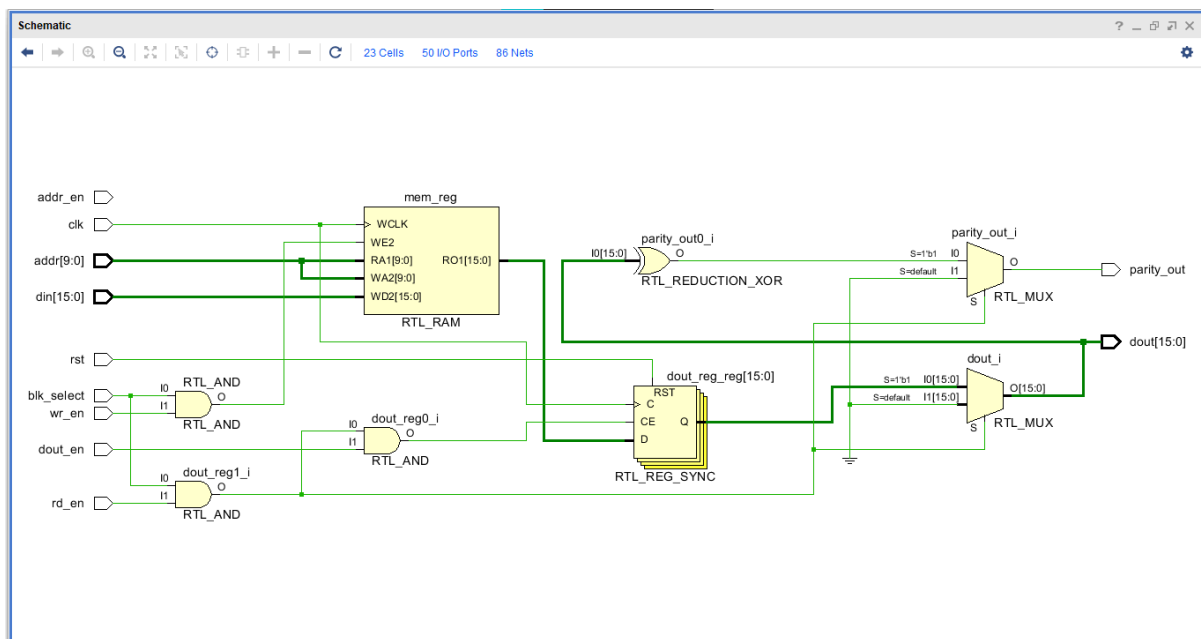
# Time	blk_sel	wr_en	rd_en	addr	din	dout	parity
# 0	0	0	0	0000	0000	0	
# 25000	1	0	1	521	5663	0000	0
# 35000	1	1	0	101	5212	0000	0
# 45000	1	1	0	374	cd3d	0000	0
# 55000	1	1	0	505	24c6	0000	0
# 65000	1	1	0	997	7277	0000	0
# 75000	1	0	1	498	96ce	xxxx	x
# 85000	1	0	1	348	28bd	xxxx	x
# 95000	1	1	0	611	870a	0000	0
# 105000	1	0	0	426	cc9d	0000	0
# 115000	1	0	1	13	d653	xxxx	x
# 125000	1	1	0	514	3eae	0000	0
# 135000	1	1	0	291	650a	0000	0

# Elaboration

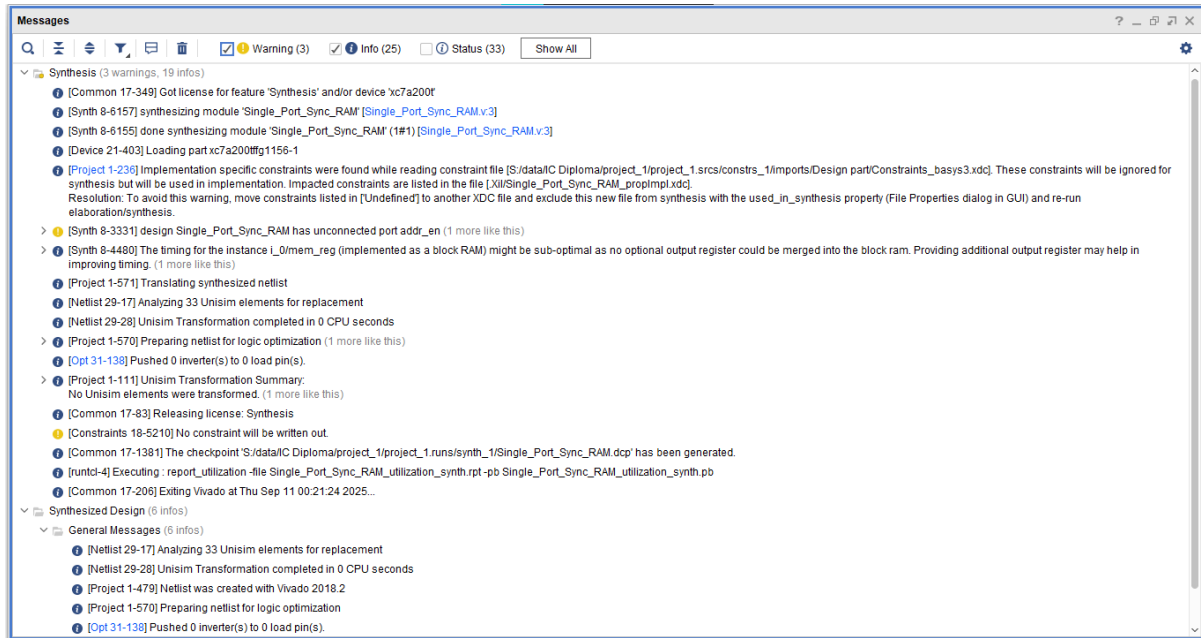
## “Messages” tab



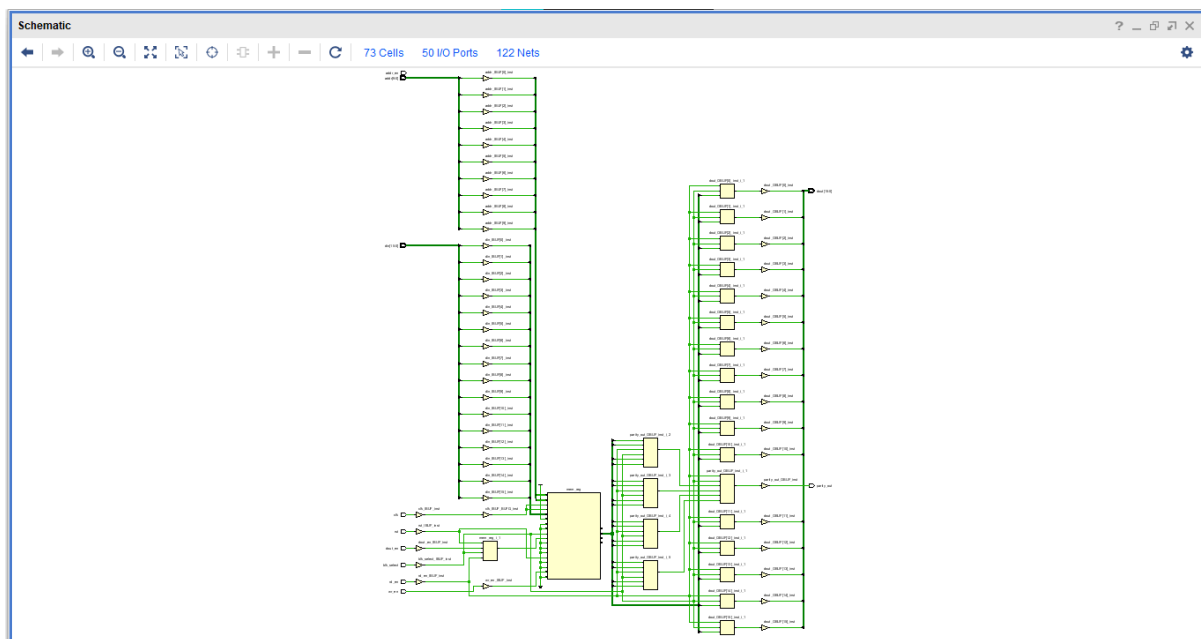
## Schematic snippet



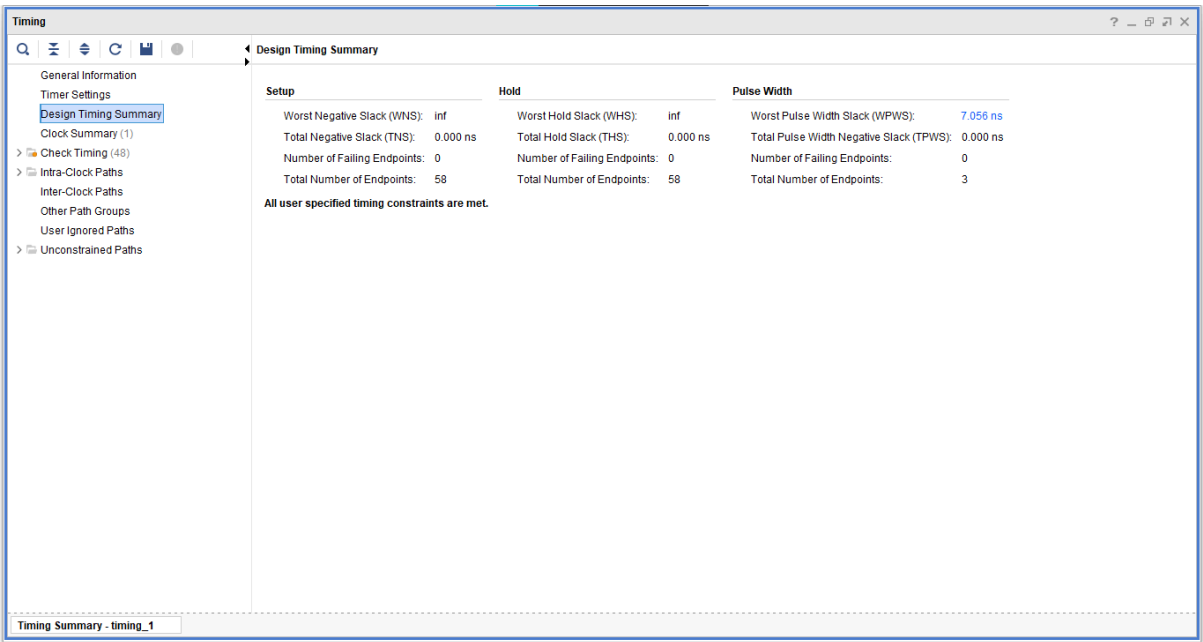
## “Messages” tab



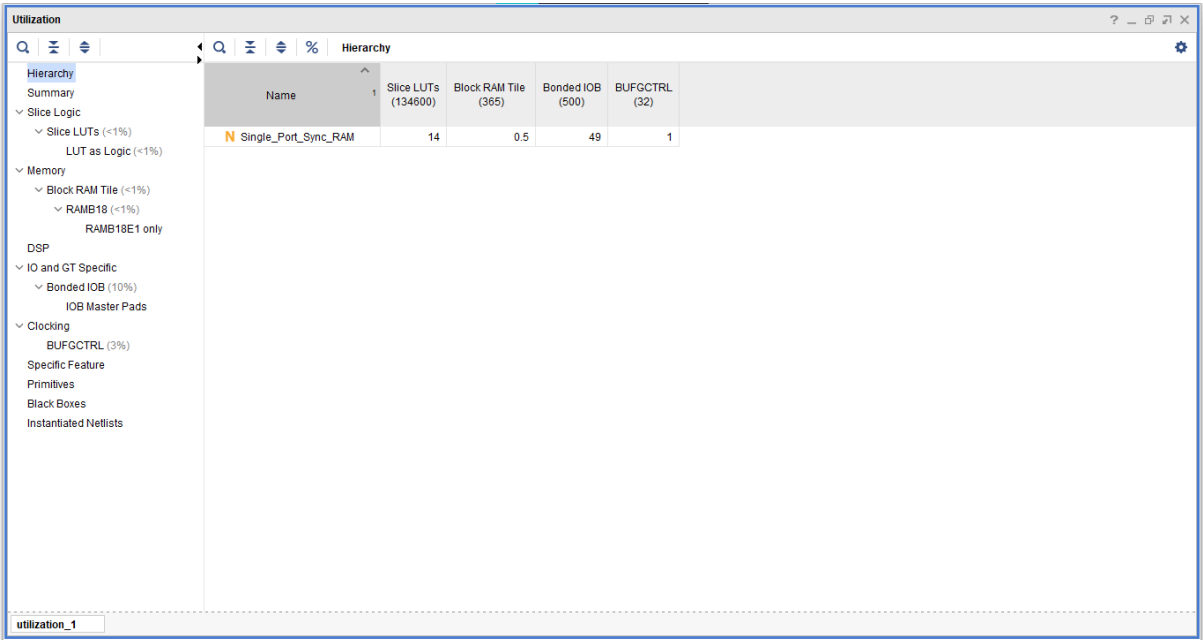
## Schematic snippet



# Timing report

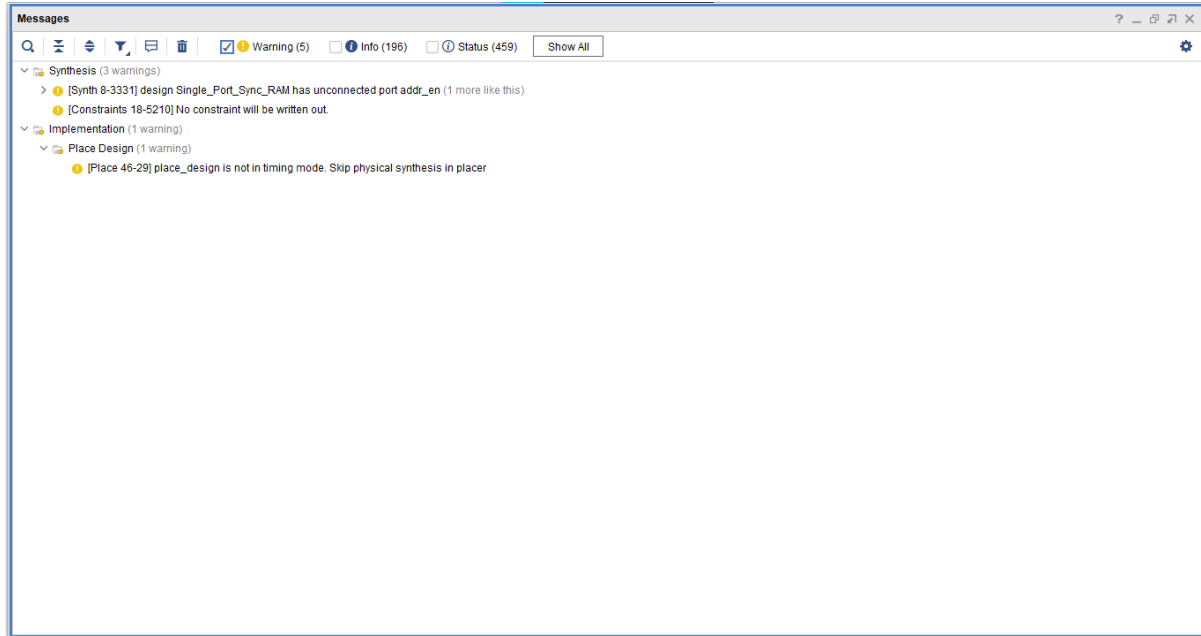


# Utilization report



# Implementation

## “Messages” tab



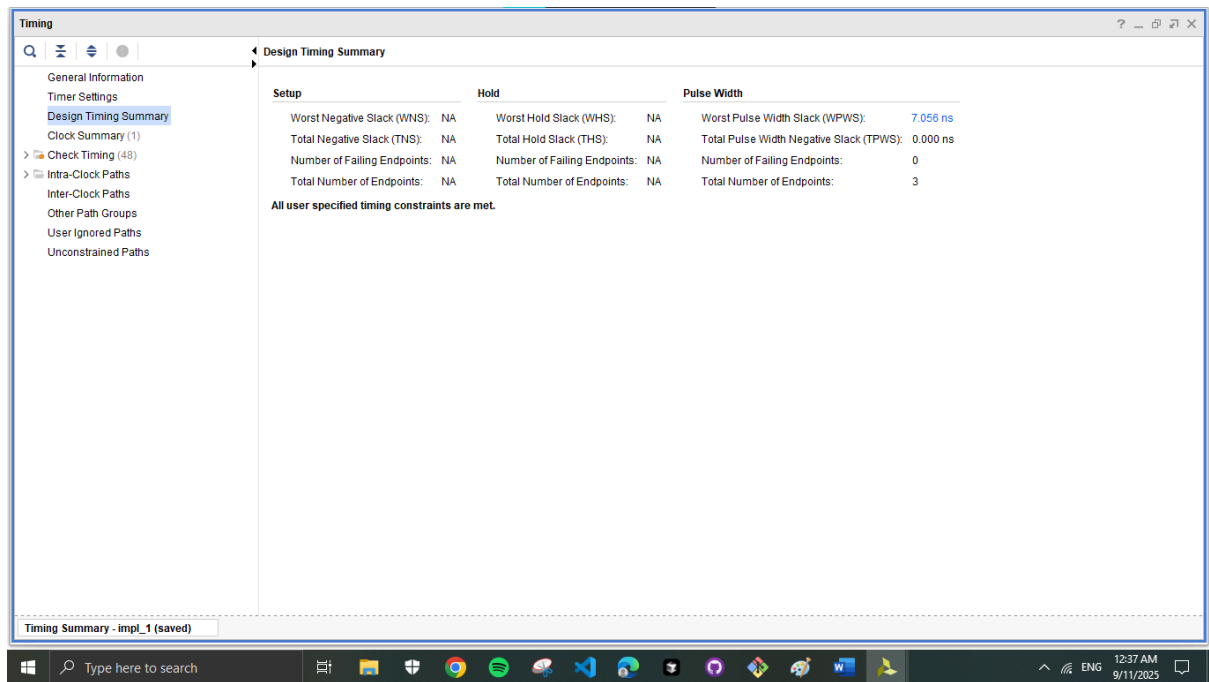
## Utilization report

The Utilization window displays a hierarchy of resource usage. The table shows the following data:

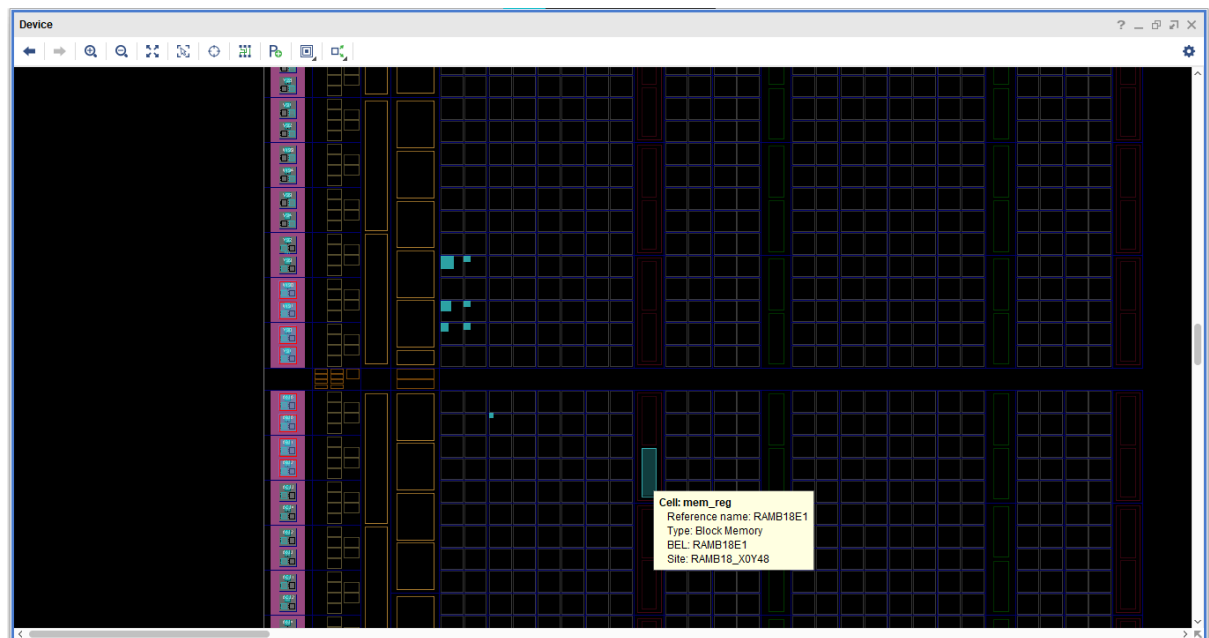
Name	Slice LUTs (133800)	Slice (33450)	LUT as Logic (133800)	Block RAM Tile (365)	Bonded IOB (500)	BUFGCTRL (32)
Single_Port_Sync_RAM	15	7	15	0.5	49	1

utilization\_1

# Timing report



# Device snippet





## 5.

```
module Async_FIFO #(
    parameter FIFO_WIDTH = 16,
    parameter FIFO_DEPTH = 512,
    parameter ADDR_SIZE  = $clog2(FIFO_DEPTH)
)(
    input wire          clk_a,    // write clock
    input wire          clk_b,    // read clock
    input wire          rst,      // sync reset

    // Write port
    input wire [FIFO_WIDTH-1:0] din_a,
    input wire              wen_a,

    // Read port
    input wire              ren_b,
    output reg [FIFO_WIDTH-1:0] dout_b,

    // Status
    output wire             full,
    output wire             empty
);

// Internal memory
reg [FIFO_WIDTH-1:0] mem [0:FIFO_DEPTH-1];

// Write and read pointers
reg [ADDR_SIZE:0] wr_ptr; // one bit wider for full detection
reg [ADDR_SIZE:0] rd_ptr;

// Write logic
always @(posedge clk_a) begin
    if (rst) begin
        wr_ptr <= 0;
    end else if (wen_a && !full) begin
        mem[wr_ptr[ADDR_SIZE-1:0]] <= din_a;
        wr_ptr <= wr_ptr + 1;
    end
end

// Read logic
always @(posedge clk_b) begin
    if (rst) begin
        rd_ptr <= 0;
        dout_b <= {FIFO_WIDTH{1'b0}};
    end else if (ren_b && !empty) begin
        dout_b <= mem[rd_ptr[ADDR_SIZE-1:0]];
    end
end
```

```

        rd_ptr <= rd_ptr + 1;
    end
end

// Status flags
assign empty = (wr_ptr == rd_ptr);
assign full   = ( (wr_ptr[ADDR_SIZE] != rd_ptr[ADDR_SIZE]) &&
                  (wr_ptr[ADDR_SIZE-1:0] == rd_ptr[ADDR_SIZE-1:0]) );

endmodule

module Async_FIFO_tb();

    // Parameters
    localparam FIFO_WIDTH = 16;
    localparam FIFO_DEPTH = 16;    // small for testing
    localparam ADDR_SIZE  = $clog2(FIFO_DEPTH);

    // DUT signals
    reg                clk_a;
    reg                clk_b;
    reg                rst;
    reg [FIFO_WIDTH-1:0] din_a;
    reg                wen_a;
    reg                ren_b;
    wire [FIFO_WIDTH-1:0] dout_b;
    wire                full;
    wire                empty;

    // Instantiate DUT
    Async_FIFO #(
        .FIFO_WIDTH(FIFO_WIDTH),
        .FIFO_DEPTH(FIFO_DEPTH),
        .ADDR_SIZE(ADDR_SIZE)
    ) DUT (
        .clk_a(clk_a),
        .clk_b(clk_b),
        .rst(rst),
        .din_a(din_a),
        .wen_a(wen_a),
        .ren_b(ren_b),
        .dout_b(dout_b),
        .full(full),
        .empty(empty)
    );

    // Clock generation
    initial begin

```

```

    clk_a = 0;
    forever #5 clk_a = ~clk_a;    // 100 MHz
end

initial begin
    clk_b = 0;
    forever #7 clk_b = ~clk_b;    // ~71 MHz
end

// Stimulus
integer i;
initial begin
    // Init
    rst    = 1;
    wen_a = 0;
    ren_b = 0;
    din_a = 0;

    // Release reset
    #20 rst = 0;

    // Run for 50 cycles of write clock
    for (i = 0; i < 50; i = i + 1) begin
        @(posedge clk_a);

        // Randomize write
        wen_a = $random % 2;
        din_a = $random;

        // Randomize read (using read clock domain)
        @(posedge clk_b);
        ren_b = $random % 2;

        // Avoid write when full
        if (full) wen_a = 0;

        // Avoid read when empty
        if (empty) ren_b = 0;
    end

    #50 $stop;
end

// Monitor activity
initial begin
    $display("Time\tclk_a\twen\tdin\tfull\tempty\tclk_b\tren\tdout");
    $monitor("%0t\t%b\t%b\t%h\t%b\t%b\t%b\t%b\t%h",

```

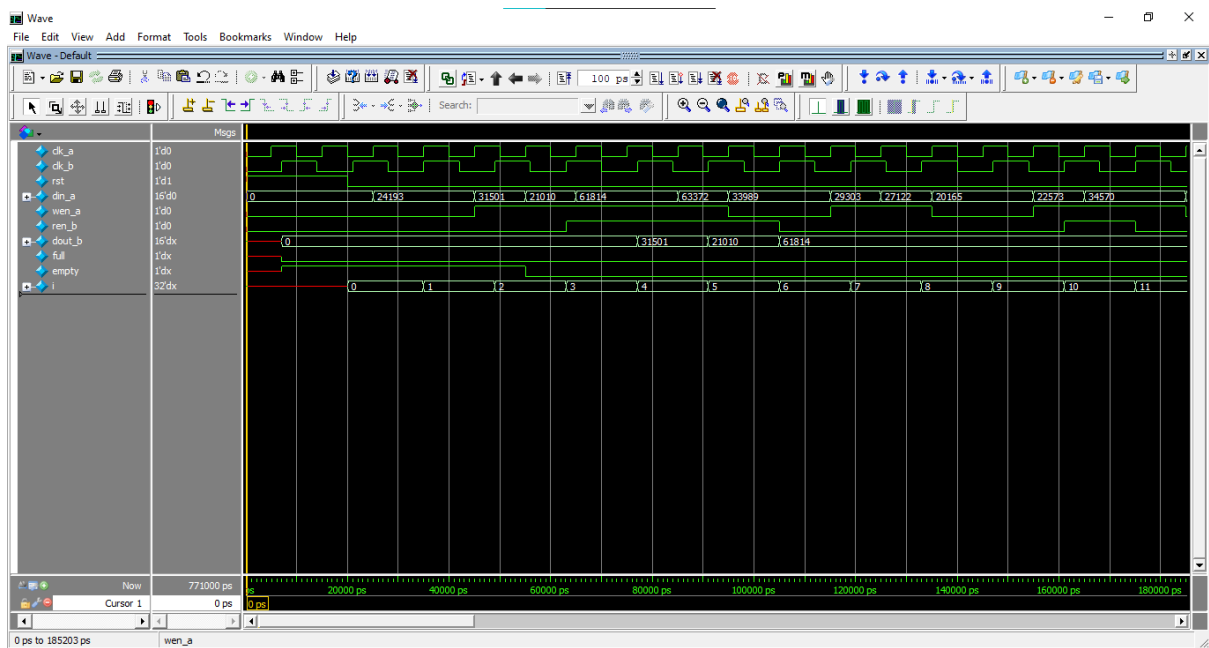
```

$time, clk_a, wen_a, din_a, full, empty, clk_b, ren_b,
dout_b);
end

endmodule

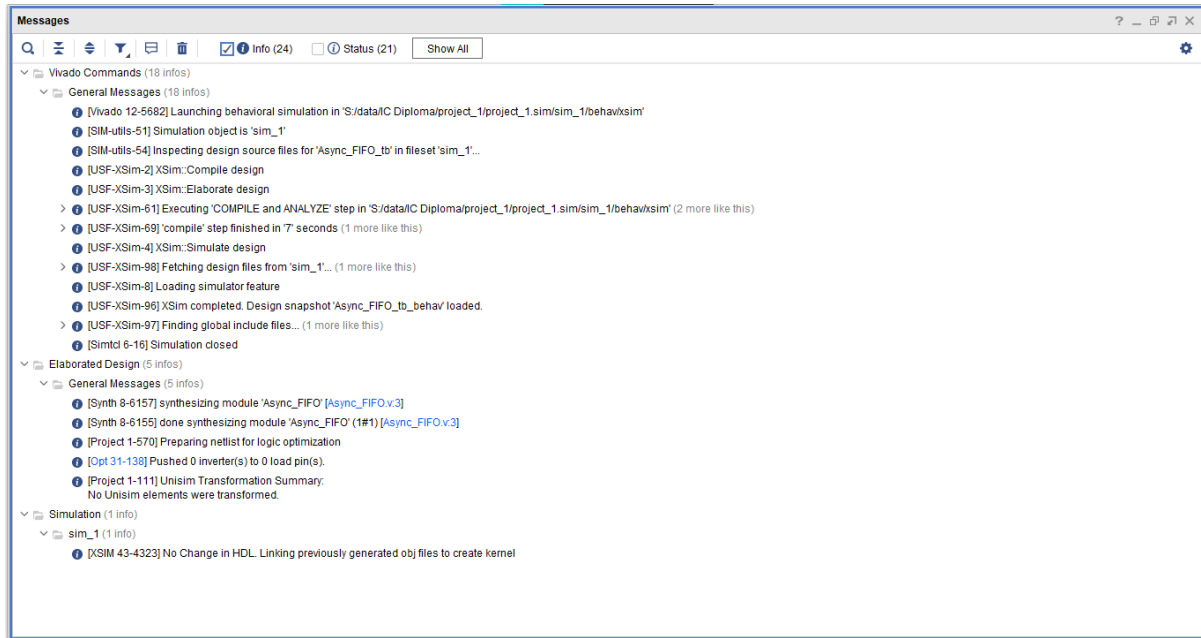
```

## QuestaSim snippet

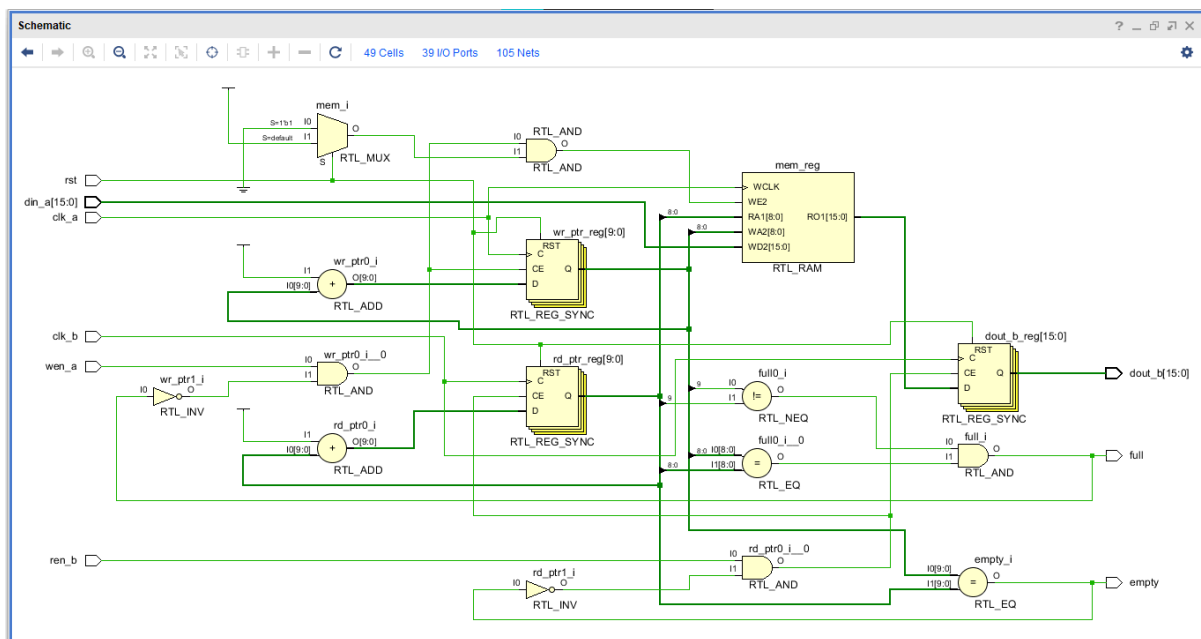


# Elaboration

## “Messages” tab

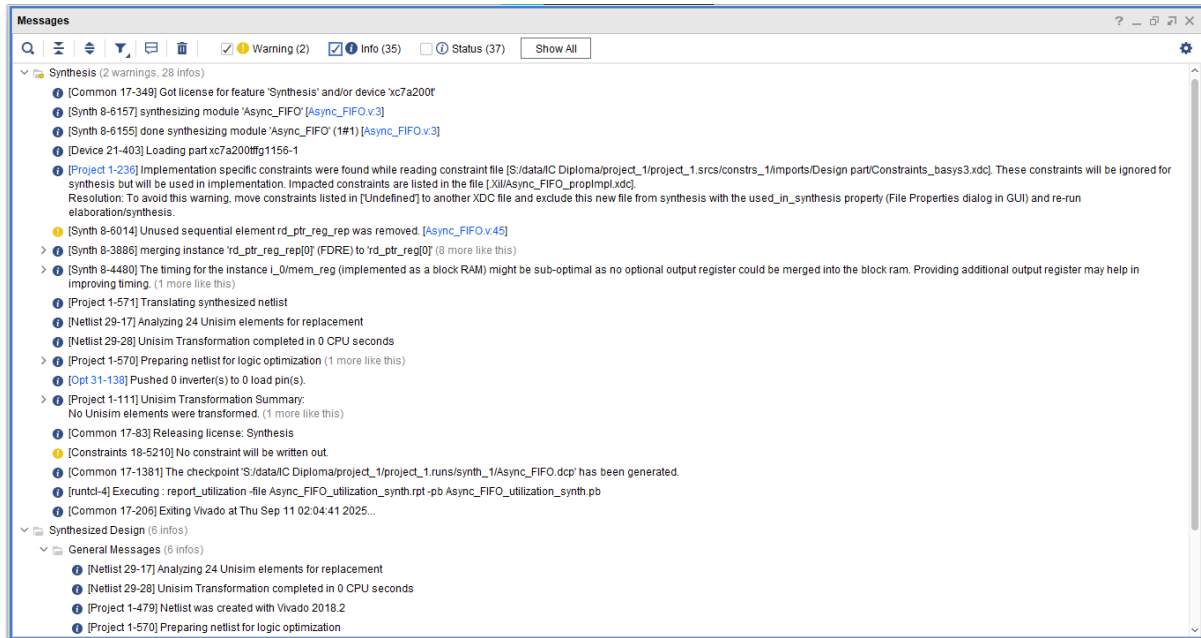


## Schematic snippet

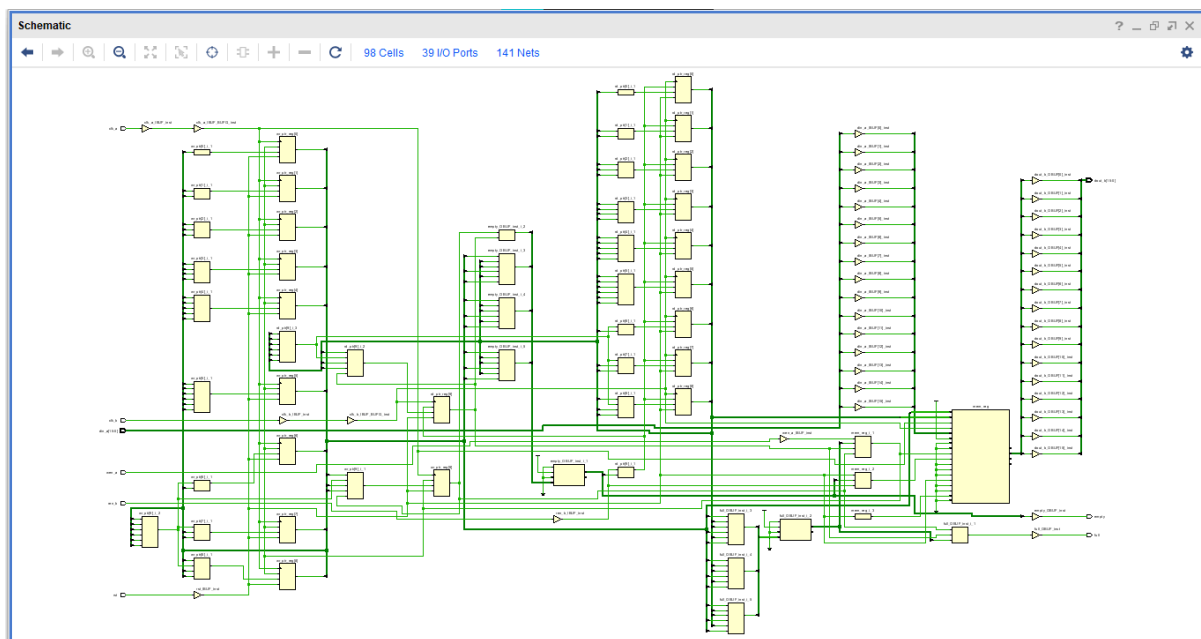


# Synthesis

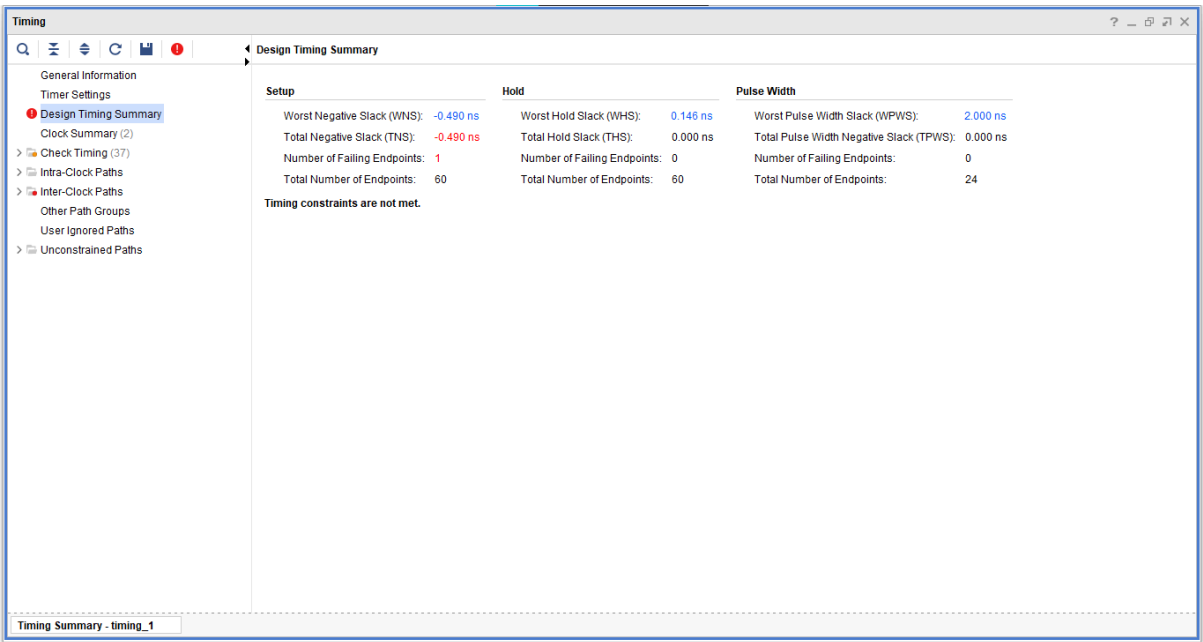
## “Messages” tab



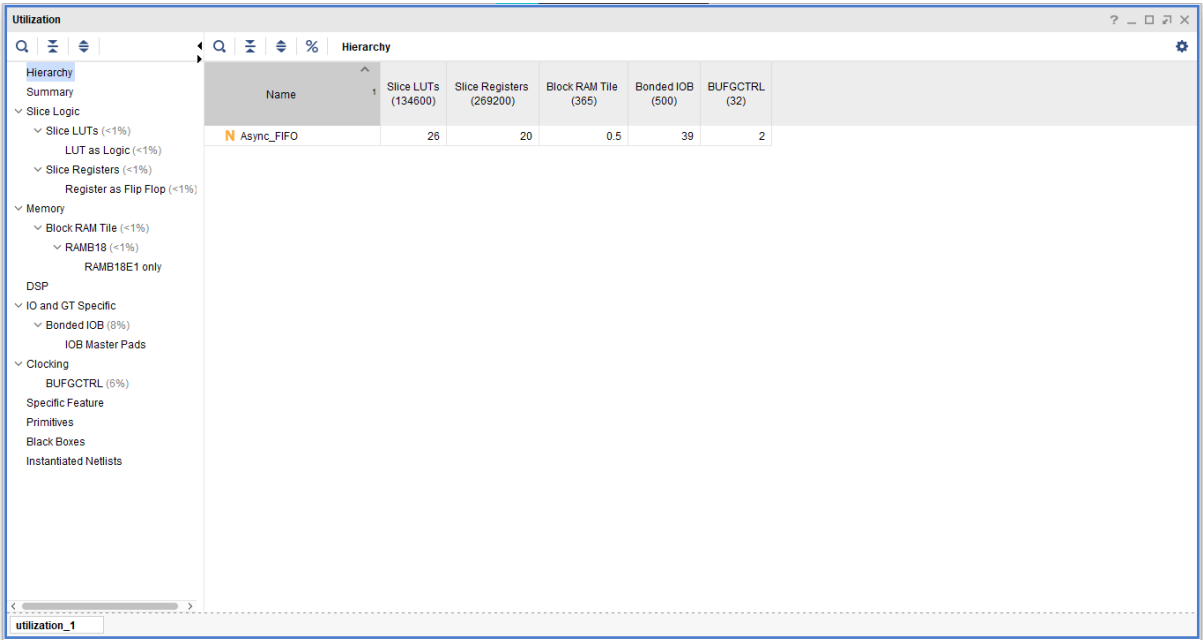
## Schematic snippet



# Timing report

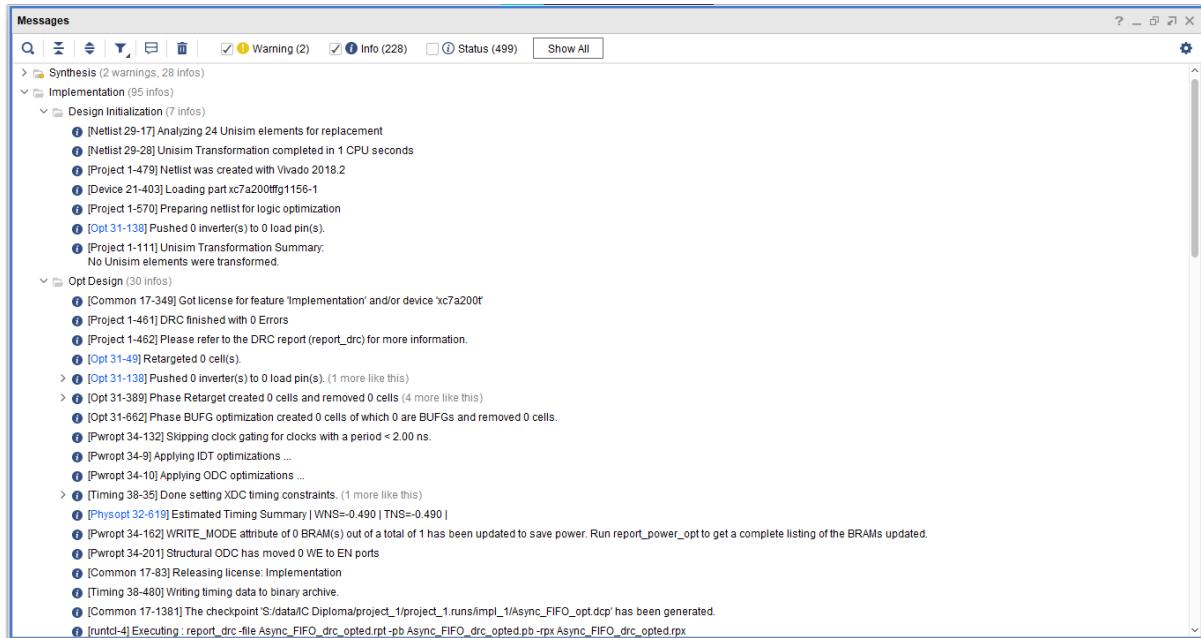


# Utilization report



# Implementation

## “Messages” tab



## Utilization report

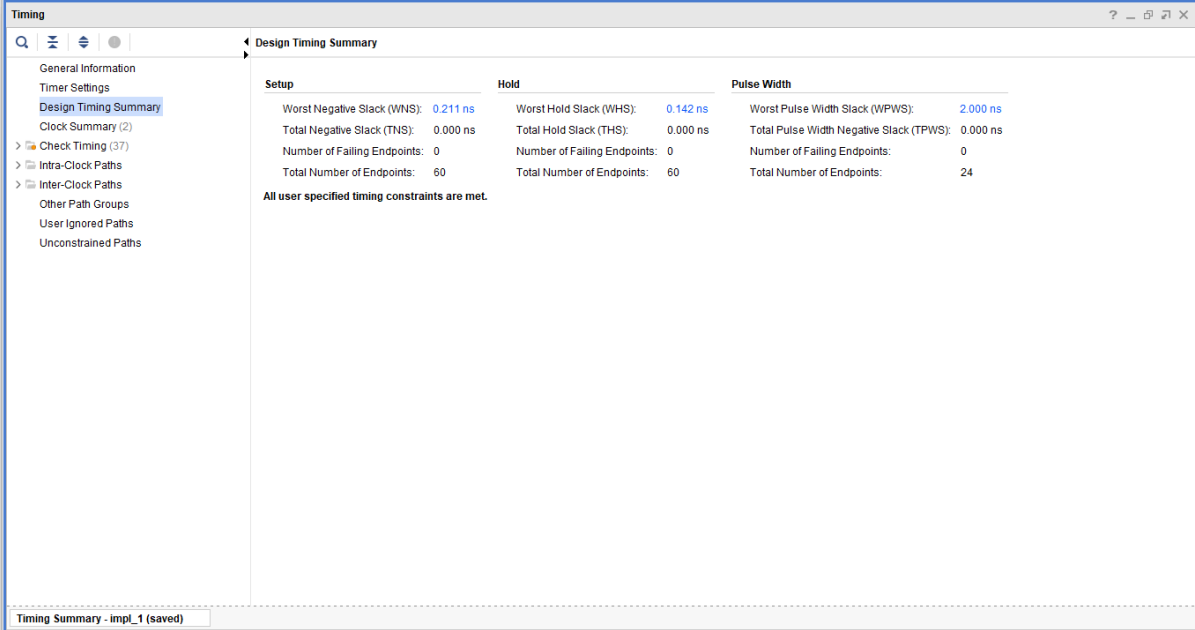
The screenshot shows the Utilization window in Vivado, displaying a hierarchy of utilization metrics. The metrics are organized into a tree structure on the left, and a table on the right shows the utilization for each metric. The table has columns for Name, Slice LUTs (133800), Slice Registers (267600), Slice (33450), LUT as Logic (133800), LUT Flip Flop Pairs (133800), Block RAM Tile (365), Bonded IOB (500), and BUFGCTRL (32).

Name	Slice LUTs (133800)	Slice Registers (267600)	Slice (33450)	LUT as Logic (133800)	LUT Flip Flop Pairs (133800)	Block RAM Tile (365)	Bonded IOB (500)	BUFGCTRL (32)
Async_FIFO	27	20	8	27	12	0.5	39	2

utilization\_1



# Timing report



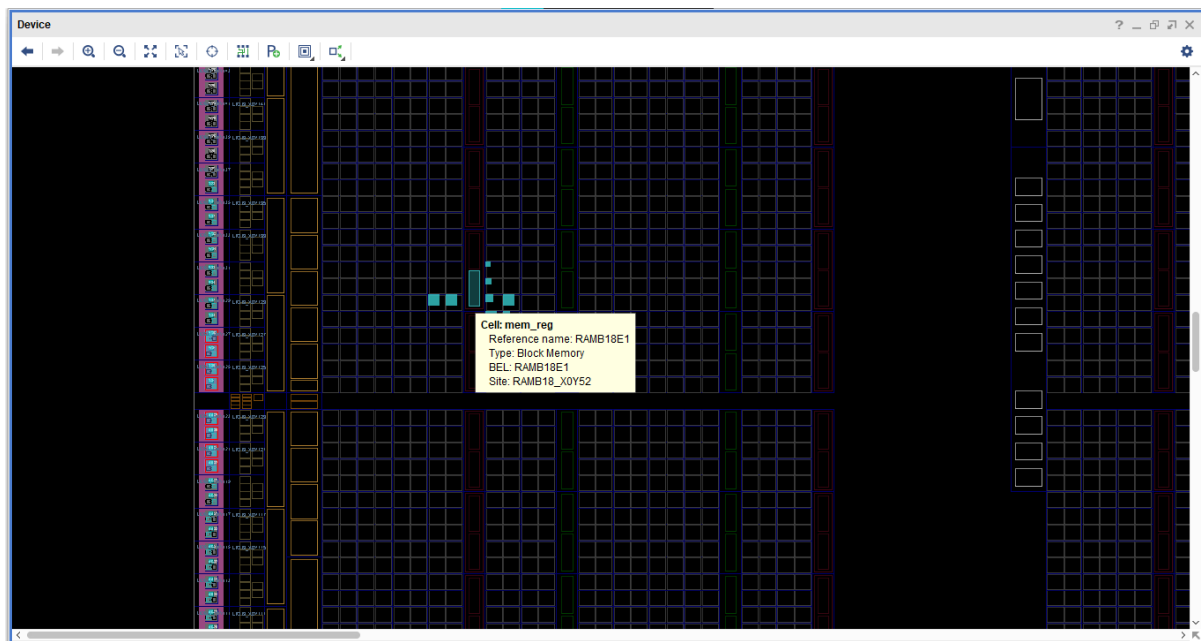
The image shows a 'Timing' window with a 'Design Timing Summary' tab. The left sidebar lists various timing-related options, with 'Design Timing Summary' selected. The main area displays a table of timing metrics for Setup, Hold, and Pulse Width, all of which are within specified limits. A status message at the bottom confirms that all user-specified timing constraints are met.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.211 ns	Worst Hold Slack (WHS):	0.142 ns	Worst Pulse Width Slack (WPWS):	2.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	60	Total Number of Endpoints:	60	Total Number of Endpoints:	24

All user specified timing constraints are met.

Timing Summary - impl\_1 (saved)

# Device snippet



## 6.

```
module ALSU #(
    parameter INPUT_PRIORITY = "A", // "A" or "B"
    parameter FULL_ADDER     = "ON" // "ON" or "OFF"
)(
    input wire      clk,
    input wire      rst,
    input wire      cin,
    input wire      red_op_A,
    input wire      red_op_B,
    input wire      bypass_A,
    input wire      bypass_B,
    input wire      direction,
    input wire      serial_in,
    input wire [2:0] opcode,
    input wire [2:0] A,
    input wire [2:0] B,

    output reg [15:0] leds,
    output reg [5:0] out
);

// Registered inputs
reg cin_reg, red_op_A_reg, red_op_B_reg, bypass_A_reg, bypass_B_reg;
reg direction_reg, serial_in_reg;
reg [2:0] opcode_reg, A_reg, B_reg;

// IP outputs
wire [3:0] out_add; // adder output (3-bit + carry)
wire [5:0] out_mult; // multiplier output

// Invalid operation detection
wire invalid_red_op, invalid_opcode, invalid;
assign invalid_red_op = (red_op_A_reg | red_op_B_reg) & (opcode_reg[1] |
opcode_reg[2]);
assign invalid_opcode = opcode_reg[1] & opcode_reg[2];
assign invalid       = invalid_red_op | invalid_opcode;

// Pipeline inputs
always @(posedge clk or posedge rst) begin
    if (rst) begin
        cin_reg      <= 0;
        red_op_B_reg  <= 0;
        red_op_A_reg  <= 0;
        bypass_B_reg  <= 0;
        bypass_A_reg  <= 0;
        direction_reg <= 0;
    end
end
```

```

        serial_in_reg <= 0;
        opcode_reg    <= 0;
        A_reg         <= 0;
        B_reg         <= 0;
    end else begin
        cin_reg        <= cin;
        red_op_B_reg   <= red_op_B;
        red_op_A_reg   <= red_op_A;
        bypass_B_reg   <= bypass_B;
        bypass_A_reg   <= bypass_A;
        direction_reg  <= direction;
        serial_in_reg  <= serial_in;
        opcode_reg     <= opcode;
        A_reg          <= A;
        B_reg          <= B;
    end
end

// LEDs blink if invalid
always @(posedge clk or posedge rst) begin
    if (rst) begin
        leds <= 0;
    end else begin
        if (invalid)
            leds <= ~leds;
        else
            leds <= 0;
    end
end

// Generate block for adder depending on FULL_ADDER
generate
    if (FULL_ADDER == "ON") begin : ADDER_FULL
        c_addsub_0 a1 (
            .A      (A_reg),      // input wire [2:0]
            .B      (B_reg),      // input wire [2:0]
            .C_IN   (cin_reg),    // input wire
            .S      (out_add)     // output wire [3:0]
        );
    end else begin : ADDER_NO_CIN
        c_addsub_0 a1 (
            .A      (A_reg),      // input wire [2:0]
            .B      (B_reg),      // input wire [2:0]
            .C_IN   (1'b0),      // force carry-in = 0
            .S      (out_add)     // output wire [3:0]
        );
    end
endgenerate

```

```

// Multiplier IP
mult_gen_1 m1 (
    .A (A_reg), // input wire [2:0]
    .B (B_reg), // input wire [2:0]
    .P (out_mult) // output wire [5:0]
);

// ALSU output logic
always @(posedge clk or posedge rst) begin
    if (rst) begin
        out <= 0;
    end else begin
        if (bypass_A_reg && bypass_B_reg) begin
            out <= (INPUT_PRIORITY == "A") ? A_reg : B_reg;
        end else if (bypass_A_reg) begin
            out <= A_reg;
        end else if (bypass_B_reg) begin
            out <= B_reg;
        end else if (invalid) begin
            out <= 0;
        end else begin
            case (opcode_reg)
                3'h0: begin // AND / reduction
                    if (red_op_A_reg && red_op_B_reg)
                        out <= (INPUT_PRIORITY == "A") ? &A_reg : &B_reg;
                    else if (red_op_A_reg)
                        out <= &A_reg;
                    else if (red_op_B_reg)
                        out <= &B_reg;
                    else
                        out <= A_reg & B_reg;
                end

                3'h1: begin // XOR / reduction
                    if (red_op_A_reg && red_op_B_reg)
                        out <= (INPUT_PRIORITY == "A") ? ^A_reg : ^B_reg;
                    else if (red_op_A_reg)
                        out <= ^A_reg;
                    else if (red_op_B_reg)
                        out <= ^B_reg;
                    else
                        out <= A_reg ^ B_reg;
                end

                3'h2: out <= out_add; // Use IP adder
                3'h3: out <= out_mult; // Use IP multiplier
            endcase
        end
    end
end

```

```

3'h4: begin // Shift with serial input
    if (direction_reg)
        out <= {out[4:0], serial_in_reg};
    else
        out <= {serial_in_reg, out[5:1]};
    end

3'h5: begin // Rotate
    if (direction_reg)
        out <= {out[4:0], out[5]};
    else
        out <= {out[0], out[5:1]};
    end

    default: out <= 0;
endcase
end
end
end
endmodule

```