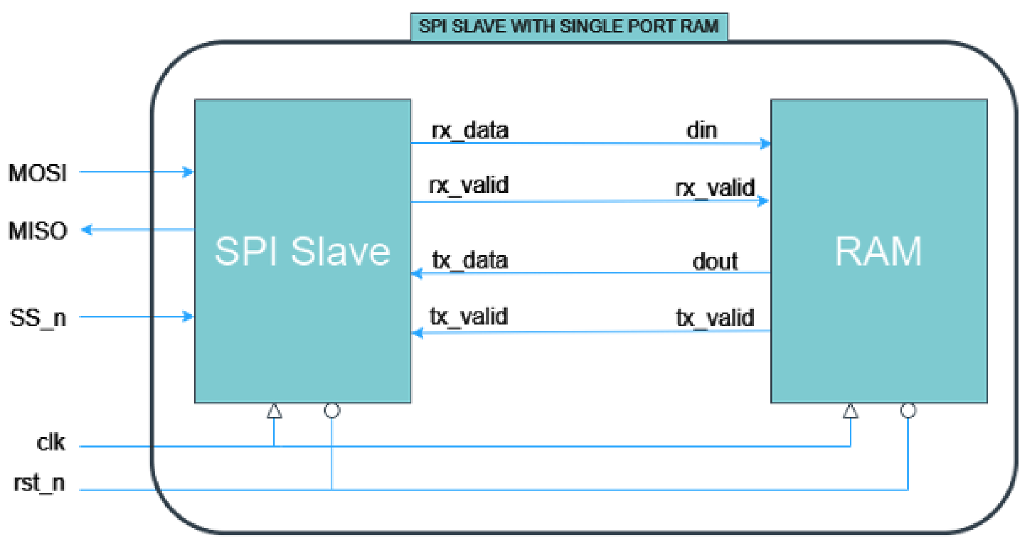
Digital Design Diploma

Project 2 :

SPI Slave with Single Port RAM



By: Kareem Hassan Atif

Submitted to: Eng. Kareem Waseem

Contents:

1. SPI Slave System Design ……………………………………………………………………3

1.1 Main Inputs and Outputs ……………………………………………………………3

1.2 Internal Structure of the RTL Code ……………………………………………………4

1.3 FSM STATES AND TRANSITIONS……………………………………………………4

1.4 Key Signal Behaviors …………………………………………………………………6

1.5 Design Highlights ……………………………………………………………………7

1.6 Summary ………………………………………………………………………………7

2. Digital Design (Verilog) ………………………………………………………………8

2.1 Codes …………………………………………………………………………………8

2.2 Testbench ……………………………………………………………………………10

2.3 Linting ………………………………………………………………………………10

2.4 DO File ………………………………………………………………………………10

2.5 Wave Snippets ………………………………………………………………………11

3. Implementation ………………………………………………………………………16

3.1 Gray Encoding ………………………………………………………………………16

3.1.1 RTL Schematic ……………………………………………………………16

3.1.2 Synthesis Schematic ………………………………………………………17

3.1.3 Device Schematic …………………………………………………………18

3.1.4 Encoding Report …………………………………………………………18

3.1.5 Synthesis Reports ……………………………………………………….......19

3.1.6 Implementation Reports …………………………………………………………20

3.1.7 Messages…………………..………………………………………………………..21

3.2 Sequential Encoding ……………………………………………………………….22

3.2.1 RTL Schematic …………………………………………………………...……22

3.2.2 Synthesis Schematic ……………………………………………………………...23

3.2.3 Encoding Report …………………………………………………………………23

3.2.4 Device Schematic ………………………………………………………………24

3.2.5 Messages…………………..………………………………………………………..24

3.2.6 Synthesis Reports ……………………………………………………….......25

3.2.7 Implementation Reports …………………………………………………………26

3.3 One-Hot Encoding …………………………………………………………………27

3.3.1 RTL Schematic …………………………………………………………...……27

3.3.2 Synthesis Schematic ……………………………………………………………...28

3.3.3 Encoding Report …………………………………………………………………28

3.3.4 Device Schematic ………………………………………………………………29

3.3.5 Synthesis Reports ……………………………………………………….......30

3.3.6 Implementation Reports …………………………………………………………31

3.2.7 Implementation After Debugging…………………..……………………………….32

3.3.8 Messages…………………..………………………………………………………..34

1. SPI Slave System Design:

This SPI Slave receives data from the SPI Master via MOSI and responds via MISO. It:

- Converts **serial input** (from MOSI) into **parallel 10-bit commands** (rx_data)
- Signals a successful reception with a **1-cycle pulse** on rx_valid
- In the read operation, it **sends 8-bit data** back to the master on MISO
- Uses a **state machine** with a **clock cycles counter** to manage precise timing

---

1.1 Main Inputs and Outputs:

Signal	Direction	Description
Clk	Input	System clock
rst_n	Input	Active-low synchronous reset
ss_n	Input	Active-low Slave Select (from master)
MOSI	Input	Serial data in
tx_valid	Input	Indicates RAM has valid data to send
tx_data	Input	8-bit data from RAM to master
MISO	Output	Serial data out
rx_valid	Output	One-cycle pulse when a 10-bit packet is received (RAM Enable)
rx_data	Output	10-bit received data for the RAM

1.2 Internal Structure of the RTL Code:

- cs, ns: current and next state of the FSM
- COUNTER: counts clock cycles to track bit positions (begins with 0)
- add: tracks whether we've already received a read address (for distinguishing **READ_ADD** and **READ_DATA**)

1.3 FSM STATES AND TRANSITIONS:

IDLE — Reset state & Waiting for SPI command

- Happens when ss_n = 1
- If ss_n = 0, transition to **CHK_CMD**

Time spent: 1 cycle (for starting the communication)

CHK_CMD — Detecting command type

- Based on MOSI value and add flag:
  - MOSI = 0 → WRITE
  - MOSI = 1 and add = 0 → READ_ADD
  - MOSI = 1 and add = 1 → READ_DATA

Time spent: 1 cycle (for checking which operation we're doing)

WRITE — Receiving a write command

- Receives 10 bits: 2 control bits + 8 address/data
- Bits stored in rx_data [9:0], MSB first (COUNTER counts from 0 to 9)
- After bit 9 is received: rx_valid = 1 for 1 cycle (Let the RAM read), and the next state is **IDLE**

Time spent: 10 cycles for bits + 1 cycle for rx_valid and to return to **IDLE** = 11 cycles

Total Cycles from start to end: 13 cycles

READ_ADD — Receiving a read address

- Same as **WRITE**, but after 10 bits, sets add = 1 (so next read goes to READ_DATA)

Time spent: 10 cycles for bits + 1 cycle for rx_valid and to return to **IDLE** = 11 cycles

Total Cycles from start to end: 13 cycles

READ_DATA — Receiving read data command and sending back data

- First 10 bits from master received as usual
- rx_valid = 1 asserted for 1 cycle to let RAM read (COUNTER = 10)
- Then, over the next 8 clock cycles (COUNTER 11:18):
    - Sends tx_data (MSB first) to master on MISO
    - Only happens if tx_valid = 1
- Returns to IDLE at the end (COUNTER = 19)

Time spent: 20 cycles = 19 for read operation + 1 for return:

- 10 cycles to receive data
- 1 cycle for rx_valid and let RAM read
- 8 cycles transmit on MISO
- 1 cycle Return to **IDLE**

Total Cycles from start to end: 22 cycles

1.4 Key Signal Behaviors

- rx_data [9:0]

- Filled serially during WRITE, READ_ADD, READ_DATA using:

  rx_data [9 - COUNTER] <= MOSI;   //First in MSB

- rx_valid

- Asserted (=1) when 10 bits have been fully received
- Helps RAM know when to read rx_data

- add

- add = 1 after READ_ADD: tells slave that the next MOSI stream is data, not address
- add = 0 after READ_DATA: reset tracking after sending

- MISO

- Only used in READ_DATA, during COUNTER 11–18

  MISO <= tx_data [18 - COUNTER]; //MSB First out

1.5 Design Highlights:

- **Modular**: Each state does exactly one job
- **Precise Timing**: COUNTER controls transitions and output assertions
- **State Machine Logic**: Clean separation of state memory (sequential), next state (combinational), and output logic (sequential)
- **Synchronous**: Counter and all outputs updated on clock edges based on the current state we were on before the new clock edge came (sequential behavioural)

1.6 Summary:

The SPI Slave handles communication using a well-defined sequence:

1. Waits for command via ss_n
2. Detects the operation type (**WRITE**, **READ_ADD**, **READ_DATA**)
3. Receives 10-bit frames serially via MOSI
4. Converts them to parallel for RAM (rx_data, rx_valid)
5. If it's a read, sends 8-bit data back via MISO
6. Transitions back to **IDLE** — ready for the next command

2. Digital Design (Verilog):

```verilog
module SPI_SLAVE (input clk,rst_n,ss_n,MOSI,tx_valid, input[7:0] tx_data,
output reg MISO,rx_valid, output reg [9:0] rx_data);
localparam IDLE=3'b000, CHK_CMD=3'b001, WRITE=3'b010, READ_ADD=3'b011, READ_DATA=3'b100;
(*fsm_encoding = "one_hot"*)
reg [2:0] cs,ns;
reg add;                  //Register Memorize if the read address is recieved or not (READ_ADD or READ_DATA)
reg [4:0] COUNTER;        //5-bits-Counter for counting clock cycles for proper operation for each state.

/* === State Memory and Register and COUNTER Logic === */
always @(posedge clk) begin
    if (!rst_n)             {cs, COUNTER, add} <= {IDLE, 5'b0, 1'b0};
    else begin
        cs <= ns;

        // COUNTER LOGIC
/* Counter increment every clk. It counts from 0 to 9 to assign MOSI in rx_data, from 0 to 8  ns = cs, rx_valid=0.
but in the ninth count, ns = cs and rx_valid=1, as the data is complete. we need another cycle after the 10 cycles
to keep rx_valid=1 for the RAM, and to determine the ns=IDLE. */
        if (cs == WRITE || cs == READ_ADD || cs == READ_DATA)      COUNTER <= COUNTER + 1;
        else                                                        COUNTER <= 0;

        // ADDRESS TRACKING LOGIC FOR READ FLOW
/* After 10 cycles, the data is complete, then we have an address or we done using it, depending on which state
we are in. If we are in READ_ADD, then we have an address, so we set add=1.
If we are in READ_DATA, then we used the previous address, so we release it by setting add=0.  */
        if      (cs == READ_ADD && COUNTER == 9)                    add <= 1;
        else if (cs == READ_DATA && COUNTER == 9)                   add <= 0;
    end
end

/* === Next State Logic === */
always @(*) begin
    case (cs)
/* One clock cycle for starting the communication*/
        IDLE: begin
            if (!ss_n)            ns = CHK_CMD;
                else              ns = IDLE;
        end

/* One clock cycle for checking which operation we're doing*/
        CHK_CMD: begin
            if      (!ss_n && !MOSI)              ns = WRITE;
            else if (!ss_n && MOSI && !add)       ns = READ_ADD;
            else if (!ss_n && MOSI && add)        ns = READ_DATA;
            else                                  ns = IDLE;
        end

/* 10 clock cycles to recieve the data, and another One clock cycle for returning to IDLE, and rx_valid=1 to let the RAM read.
        (total clock cycles from beginning to end = 13) */
        WRITE: begin
            if (COUNTER < 10 && !ss_n)            ns = WRITE;
            else                                  ns = IDLE;
        end

/*Same as WRITE, but here we have ADD*/
        READ_ADD: begin
            if (COUNTER < 10 && !ss_n)            ns = READ_ADD;
            else                                  ns = IDLE;
        end

/* 10 clock cycles to recieve the data, and another One clock cycle for rx_valid=1 to let the RAM read. Another 8 clock cycles
to send the data to the master, and another One clock cycle for returning to IDLE.
        (total clock cycles from beginning to end = 22) */
        READ_DATA: begin
            if (COUNTER < 19 && !ss_n)            ns = READ_DATA;
            else                                  ns = IDLE;
        end

        default: ns = IDLE;
    endcase
end

/* === Output Sequential Logic for proper operation and synchronization === */
always @(posedge clk) begin
    if (!rst_n)             {rx_valid, rx_data, MISO} <= 0;
    else begin
        rx_valid <= 0;       // Default low, pulse for 1 cycle only
        case (cs)
            WRITE: begin
                 //Writing data from most significant bit (rx_data [9]) to least significant (rx_data [0]) bit for 10 clock cycles
                if (COUNTER <= 9 && !ss_n)       rx_data [9 - COUNTER] <= MOSI;
                if (COUNTER == 9)                rx_valid <= 1;      //rx_valid=1 to let the RAM read the data after receiving it
            end

            READ_ADD: begin
                if (COUNTER <= 9 && !ss_n)       rx_data [9 - COUNTER] <= MOSI;
                if (COUNTER == 9)                rx_valid <= 1;
            end

            READ_DATA: begin
                if (COUNTER <= 9 && !ss_n)     rx_data [9 - COUNTER] <= MOSI;
                if (COUNTER == 9)              rx_valid <= 1;
                //Wait for a clock cycle (COUNTER = 10) to let the RAM read the data, then send it to the master
                //Recieving data from most significant bit (tx_data [7]) to least significant (tx_data [7]) bit for 8 clock cycles.
                if (COUNTER >= 11 && COUNTER <= 18 && tx_valid)   MISO <= tx_data [18 - COUNTER];
                else      MISO <= 0;
            end
        endcase
    end
end
endmodule
```

Figure 1: SPI Slave Code

```verilog
module RAM #(parameter MEM_DEPTH=256,     //Memory Depth
ADDR_SIZE=8  //address Bus Size and Memory Width
)
(input [ADDR_SIZE+1:0] din, input clk,rst_n,rx_valid, output reg [ADDR_SIZE-1:0] dout, output reg tx_valid);

reg [ADDR_SIZE-1:0] mem [MEM_DEPTH-1:0];
reg [ADDR_SIZE-1:0] WRITE_address, READ_address;        //Internal address Lines

always @(posedge clk) begin
    if(!rst_n)              {dout,tx_valid,WRITE_address,READ_address} <= 0;
    else if (rx_valid) begin
        case( din [ADDR_SIZE+1 : ADDR_SIZE] )
            2'b00:          WRITE_address <= din[ADDR_SIZE-1:0];
            2'b01:          mem [WRITE_address] <= din[ADDR_SIZE-1:0];
            2'b10:          READ_address <= din[ADDR_SIZE-1:0];
            2'b11:          {dout,tx_valid} <= {mem [READ_address], 1'b1};
        endcase
    end
end
endmodule
```

Figure 2: RAM Code

```verilog
module SPI_Wrapper (input clk,rst_n,ss_n,MOSI, output MISO);
parameter MEM_DEPTH=256, ADDR_SIZE=8;
wire [ADDR_SIZE+1:0] w1;
wire [ADDR_SIZE-1:0] w3;
wire w2,w4;

/*module SPI_SLAVE (input clk,rst_n,ss_n,MOSI,tx_valid, input[7:0] tx_data,
output reg MISO,rx_valid, output reg [9:0] rx_data);*/

SPI_SLAVE DUT_SPI(.clk(clk),.rst_n(rst_n),.ss_n(ss_n),.MOSI(MOSI),.tx_valid(w4),.tx_data(w3),.MISO(MISO),.rx_valid(w2),.rx_data(w1));

/*module RAM #(parameter MEM_DEPTH=256, ADDR_SIZE=8)
(input [ADDR_SIZE+1:0] din, input clk,rst_n,rx_valid, output reg [ADDR_SIZE-1:0] dout, output reg tx_valid);*/

RAM #(.MEM_DEPTH(MEM_DEPTH),.ADDR_SIZE(ADDR_SIZE)) DUT_RAM(.din(w1),.clk(clk),.rst_n(rst_n),.rx_valid(w2), .dout(w3),.tx_valid(w4));

endmodule
```

Figure 3: SPI Wrapper Code

```verilog
module SPI_Wrapper_tb();
reg clk,rst_n,ss_n,MOSI;
wire MISO;

//module SPI_Wrapper (input clk,rst_n,ss_n,MOSI, output MISO);
SPI_Wrapper DUT(clk,rst_n,ss_n,MOSI, MISO);

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

integer i;
initial begin
    $readmemh ("mem.dat", DUT.DUT_RAM.mem);
    rst_n=0; ss_n=1; MOSI=1;
    @(negedge clk);
    rst_n=1;

    //WRITE in address(1) (13 clock cycles)
    ss_n=0;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);
    for(i=0;i<9;i=i+1) begin
        MOSI=0;
        @(negedge clk);
    end
    MOSI=1;
    @(negedge clk);
    @(negedge clk);

    //WRITE data(11110001) (13 clock cycles)
    ss_n=0;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    for(i=0;i<4;i=i+1) begin
        MOSI=1;
        @(negedge clk);
    end
    for(i=0;i<3;i=i+1) begin
        MOSI=0;
        @(negedge clk);
    end
    MOSI=1;
    @(negedge clk);
    @(negedge clk);

    //read address(1) (13 clock cycles)
    ss_n=0;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    for(i=0;i<8;i=i+1) begin
        MOSI=0;
        @(negedge clk);
    end
    MOSI=1;
    @(negedge clk);
    @(negedge clk);

    //read data (22 clock cycles)
    ss_n=0;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    for(i=0;i<8;i=i+1) begin
        MOSI=$random;
        @(negedge clk);
    end
    @(negedge clk);

    repeat (9) @(negedge clk);
    #10 $stop;
end
endmodule
```

Figure 4: SPI Wrapper Testbench Code

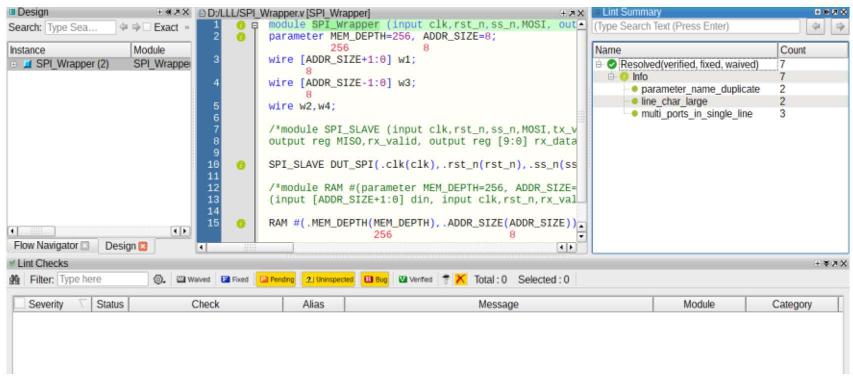


Figure 5: Linting

```
vlib work
vlog RAM.v SPI_SLAVE.v SPI_Wrapper.v SPI_Wrapper_tb.v
vsim -voptargs=+acc SPI_Wrapper_tb
add wave *
run -all
#quit -sim
```

Figure 6: DO File

Wave Snippets:

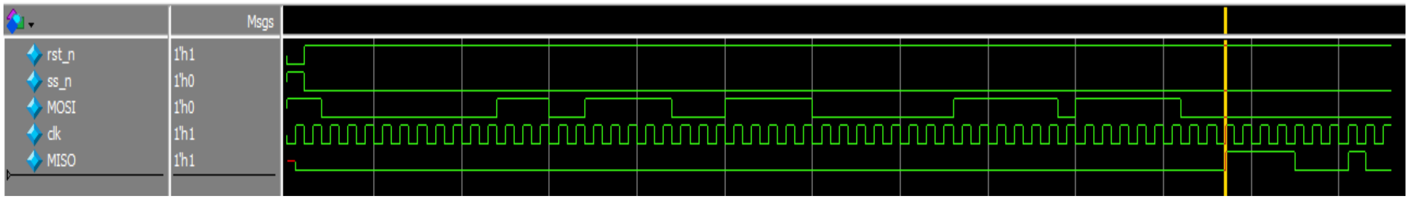


Figure 7: SPI Wrapper Wave

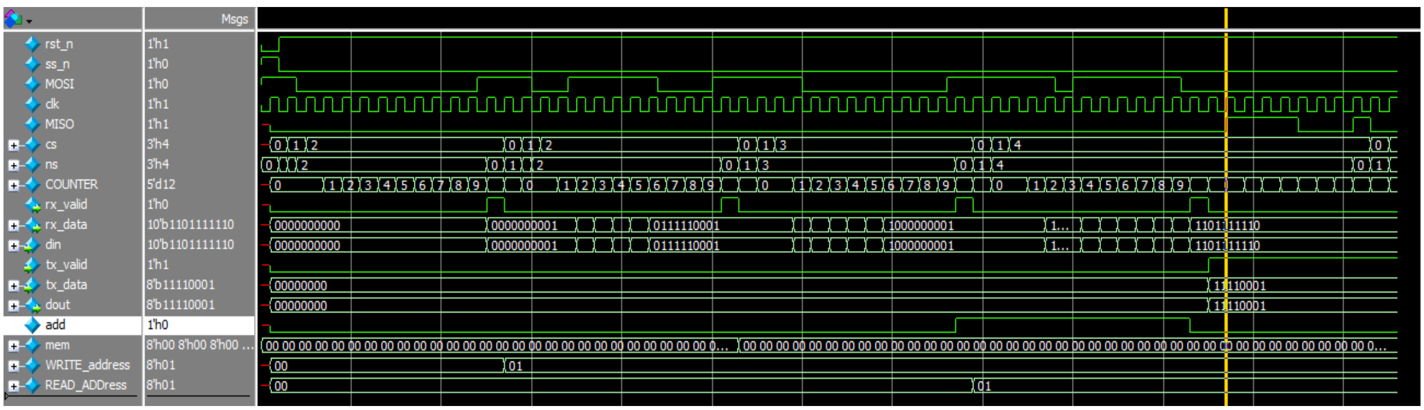


Figure 8: Detailed SPI Wrapper Wave

Detailed Wave Snippets:

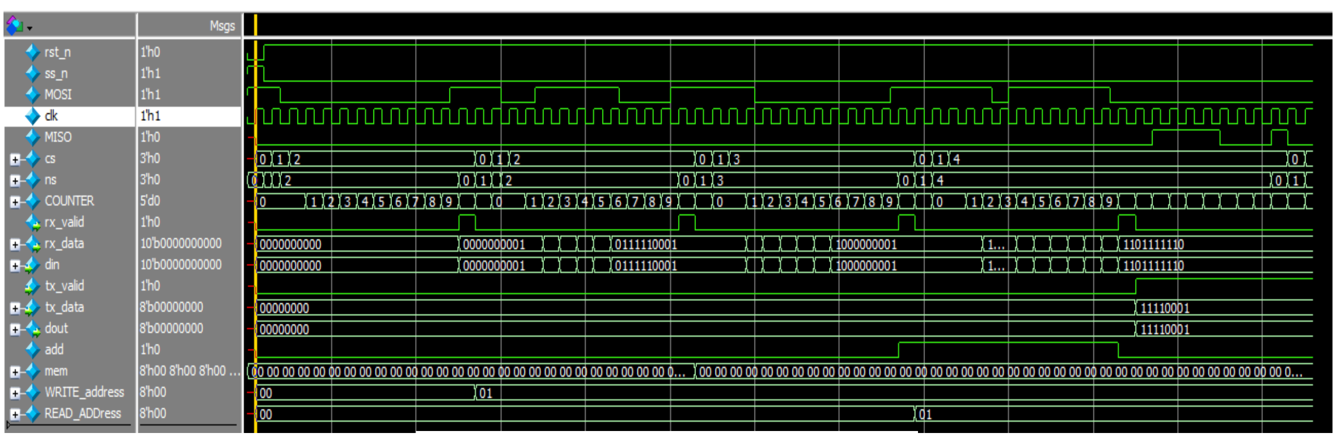


Figure 9: Synch Reset Operation

With the first clock edge, all outputs are reset to 0 due to the active-low synchronous reset signal.

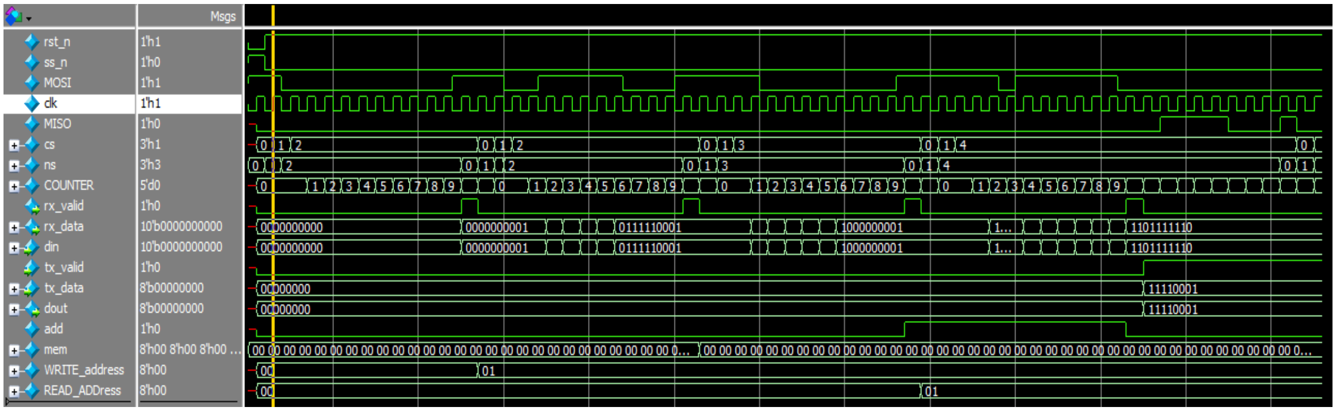


Figure 10: One clock cycle to start communication

After the reset, the next state is **CHK_CMD**, but it won't be assigned until the next clock edge arrives. When it arrives, the current state becomes **CHK_CMD**.

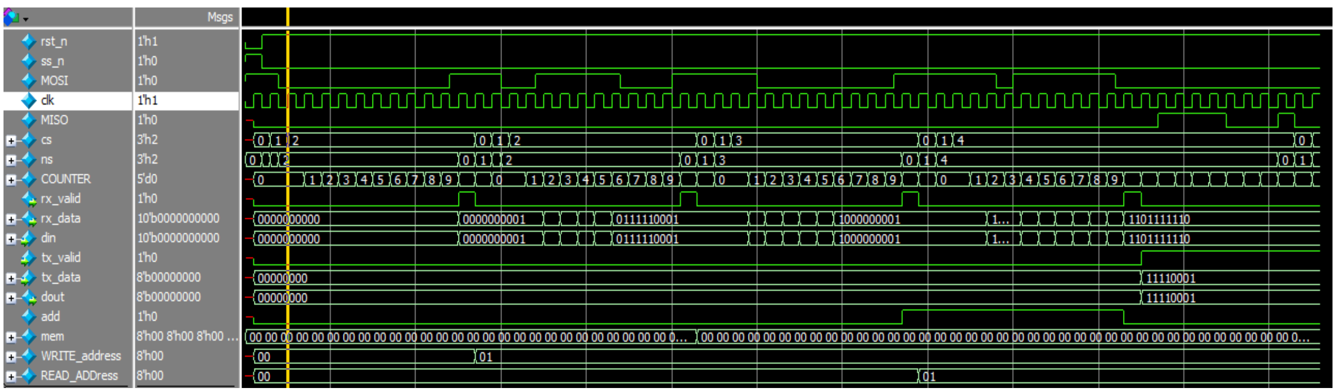


Figure 11: One clock cycle to check command

We wait a clock cycle to determine which operation we'll be doing. When we arrive at the **WRITE** state on a clock edge, the next-state logic (combinational always block) executes immediately to determine the upcoming state. However, the counter and output logic (sequential always blocks) do not update instantly. At this moment, they still consider the current state to be **CHK_CMD**. Therefore, data reception or counter start won't happen immediately upon entering **WRITE**; they will occur on the following clock edge due to sequential behavior.

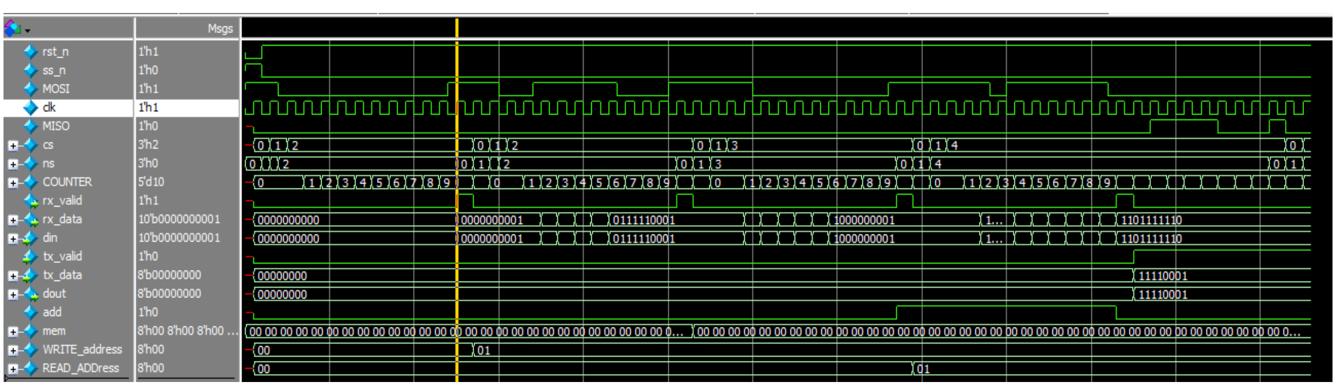


Figure 12: Ten clock cycles to receive address (1)

Cycle after cycle, we receive serial data — this continues for ten clock cycles. On the last clock cycle, we receive the final bit, and rx_valid is asserted (set to 1). However, the RAM hasn't sensed the input yet; it will register it on the next clock edge due to sequential behavior. And we go to **IDLE**.

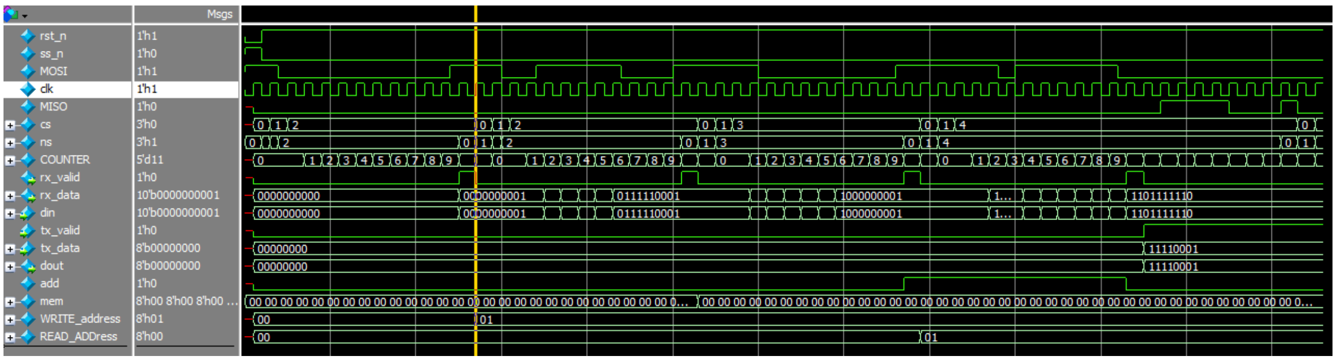


Figure 13: One clock for RAM and returning to IDLE

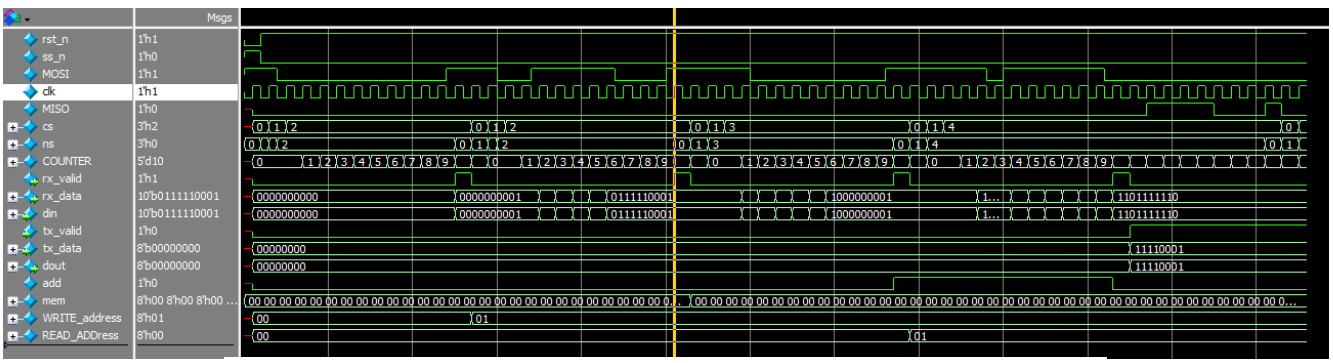


Figure 14: Twelve clocks to repeat from the start to receiving all data

We repeat the same technique to receive the data that will be inserted in RAM.

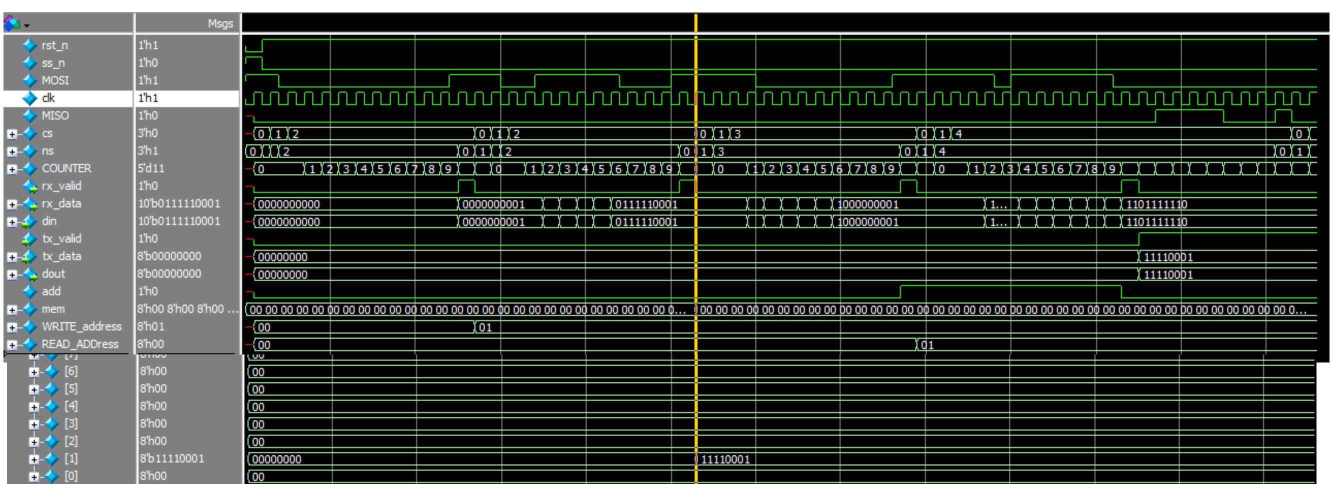


Figure 15: One clock for RAM to insert data in address (1), and to go to IDLE

On the thirteenth clock cycle, RAM will insert data in it and we finish command and go to **IDLE**.

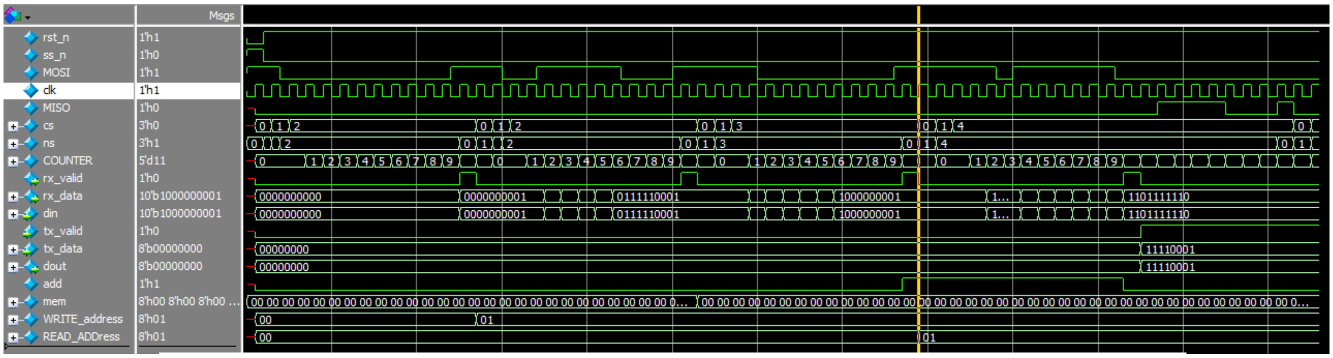


Figure 16: Thirteen clock cycles to read address (1)

Exactly the same with **READ_ADD**, but we have register "add" to memorize having an address.

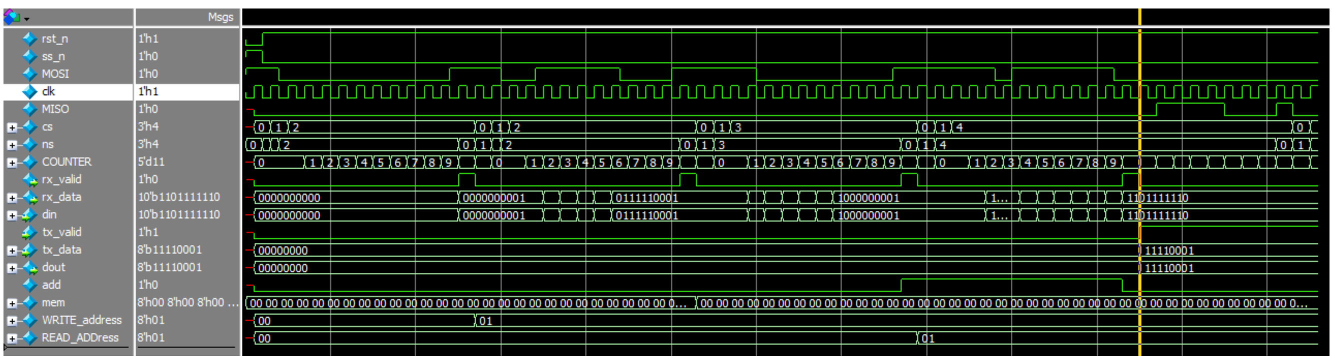


Figure 17: Thirteen clock cycles to read data from RAM

Exactly the same with **READ_DATA** to read the data from RAM, and we reset add. Then, we wait for eight clock cycles to display data on MISO. Finally, one clock cycle to end communication.

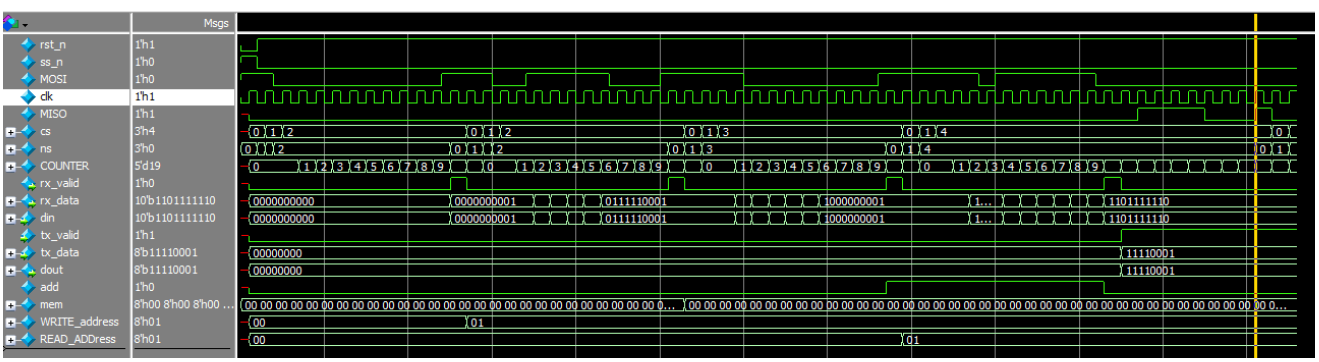


Figure 18: Eight clock cycles to display data on MISO

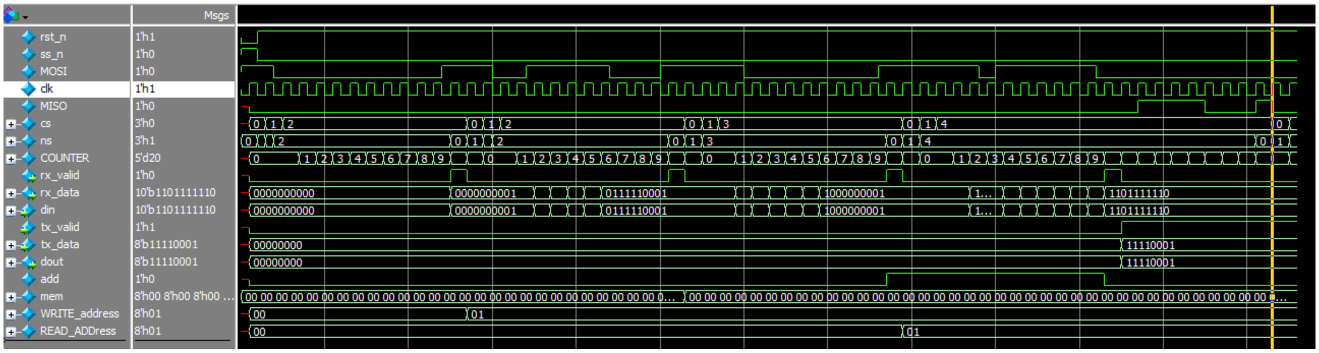


Figure 19: One clock cycle to end communication

The key idea is to understand the distinction between combinational and sequential logic, and how timing affects when each action occurs.

- The next-state logic (combinational always block) determines the next state immediately, based on the current state and inputs. It also implicitly defines how many clock cycles we spend in each state to complete the full operation.
- The counter and output logic (sequential always blocks) operate on each clock edge. The counter increments on every clock cycle spent in a state, and the outputs typically depend on the current state and the counter value.

So even if the next state is decided instantly, the outputs and state memory don't reflect that change until the next clock edge, which highlights the sequential nature of those blocks.

Think of the **combinational block** as a decision-maker that *plans* what to do next, and the **sequential blocks** as *doers* that wait for a clock tick before actually doing it.

Now we go to synthesis and implementation. We'll try different types of FSM encodings and choose the best one for timing.

3. Implementation:

3.1 Gray Encoding:

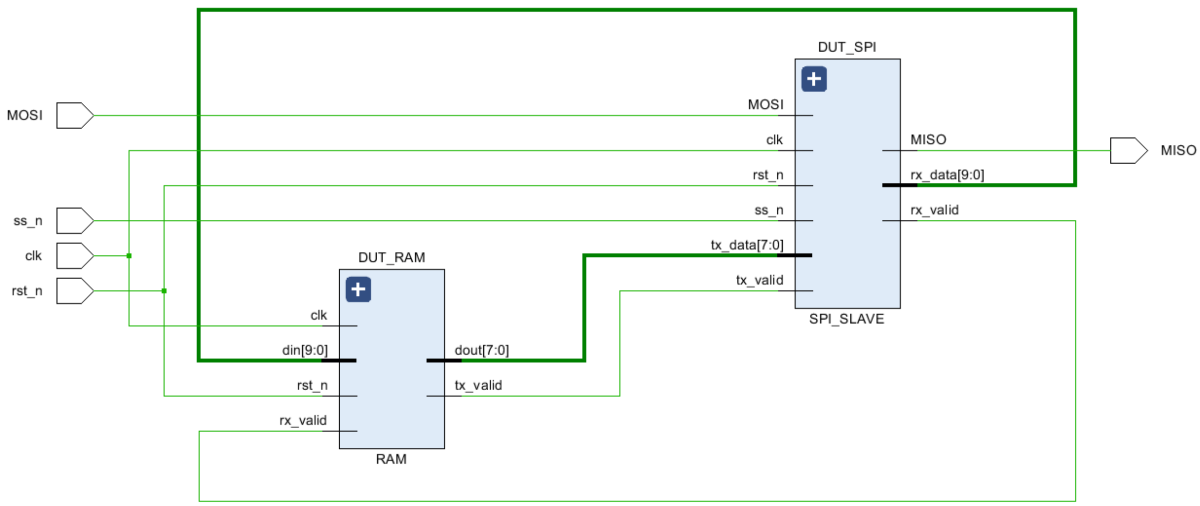


Figure 20: RTL (Gray)

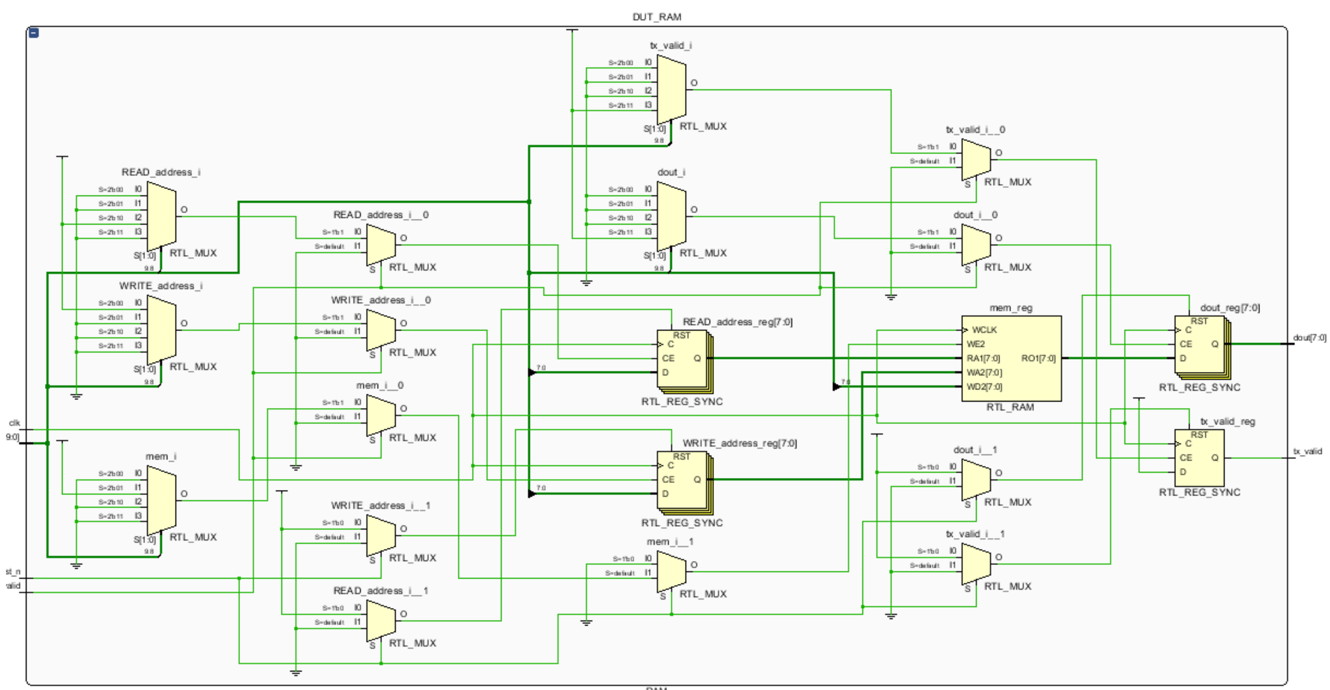


Figure 21: RAM RTL (Gray)

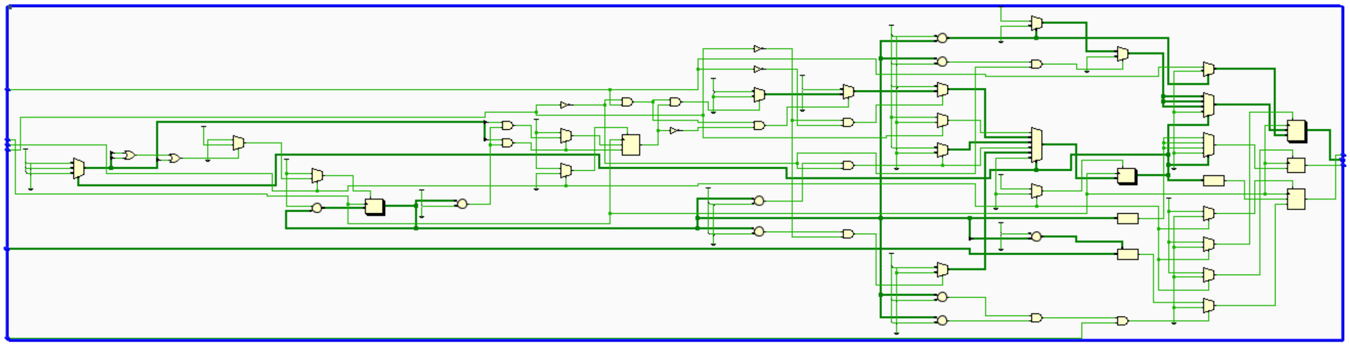


Figure 22: SPI Slave RTL (Gray)

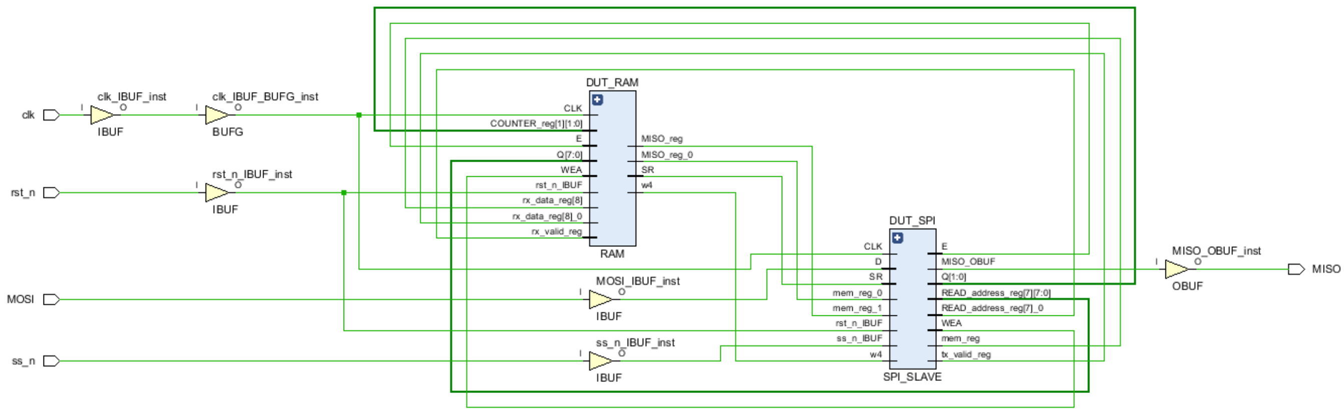


Figure 23: Synthesis (Gray)

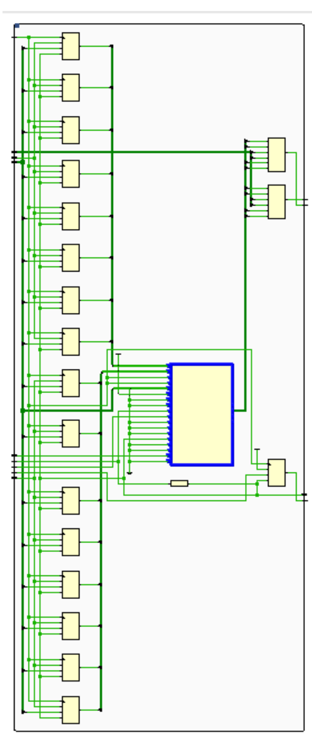


Figure 24: RAM Synthesis (Gray)

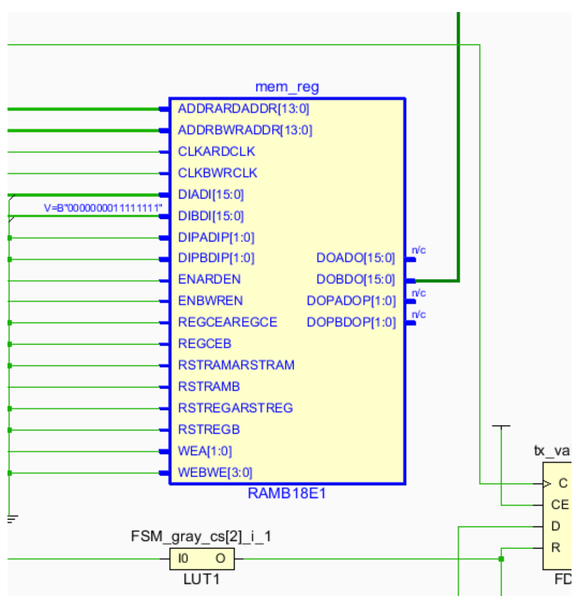


Figure 25: Closer RAM Synthesis (Gray)

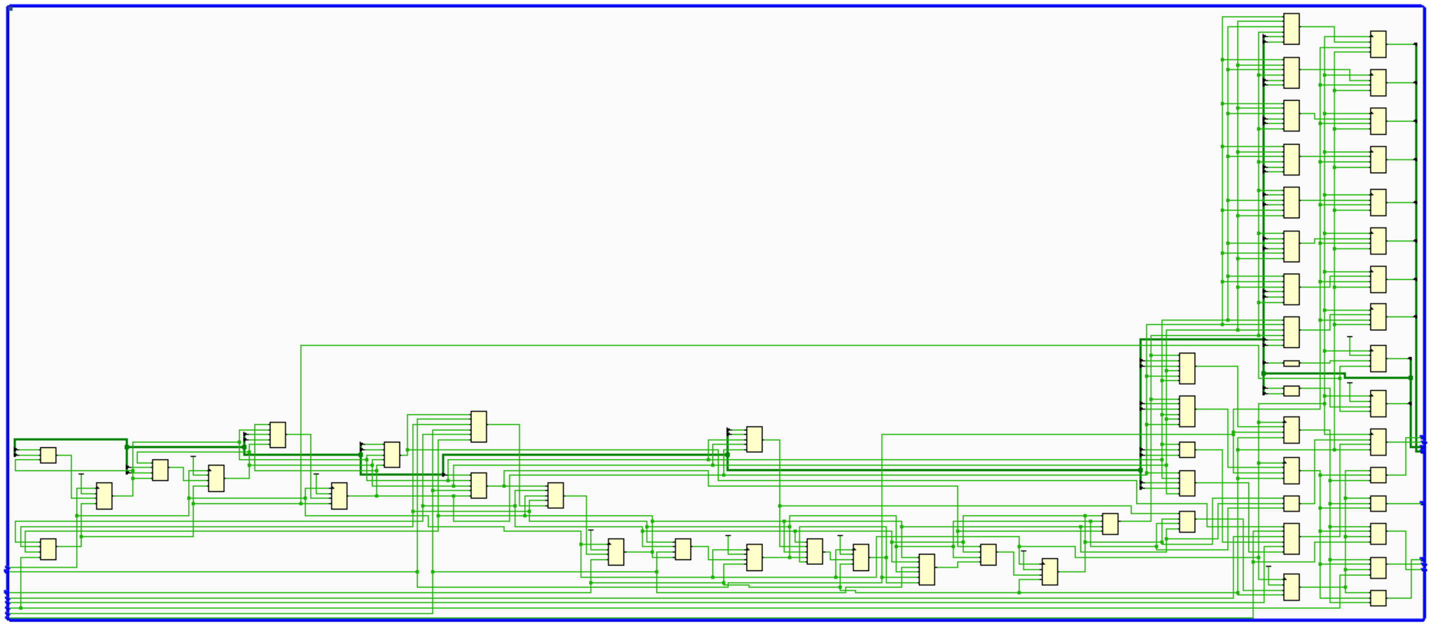


Figure 26: SPI Slave Synthesis (Gray)

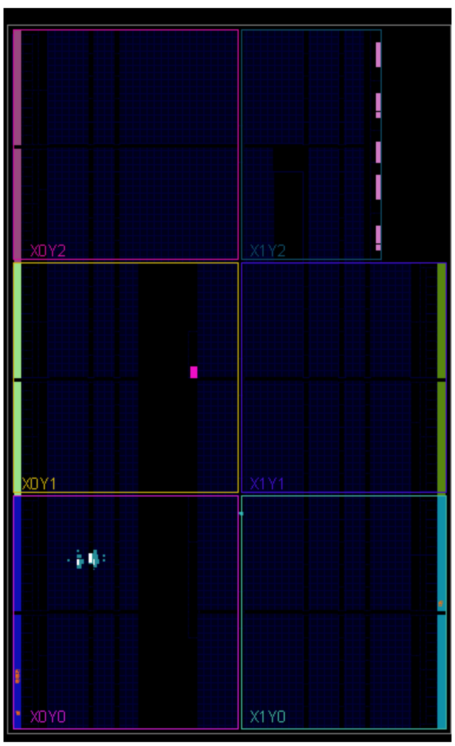


Figure 27: Device (Gray)

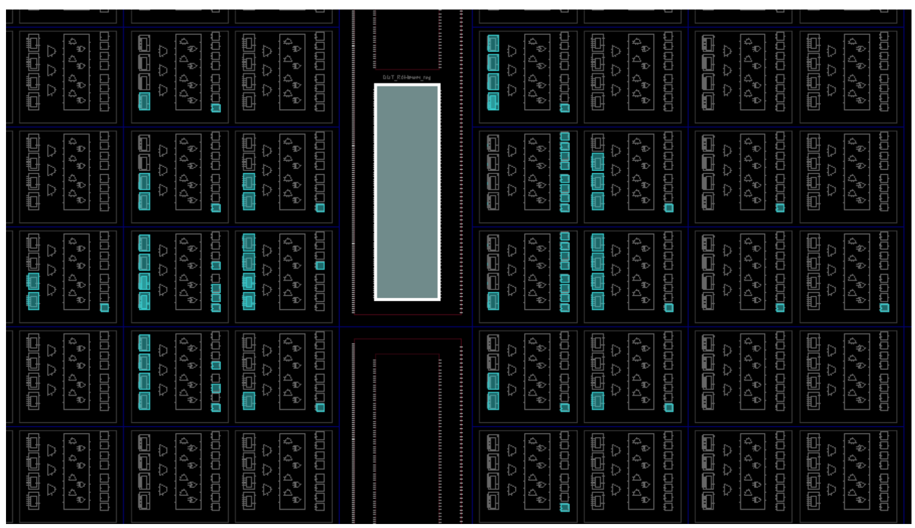


Figure 28: Closer Device (Gray)

```
---------------------------------------------------------------------------------------------------
                   State |                     New Encoding |                Previous Encoding
---------------------------------------------------------------------------------------------------
                    IDLE |                              000 |                              000
                 CHK_CMD |                              111 |                              001
                   WRITE |                              010 |                              010
                READ_ADD |                              011 |                              011
               READ_DATA |                              001 |                              100
---------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'cs_reg' using encoding 'gray' in module 'SPI_SLAVE'
------------------------
```

Figure 29: Encoding Report (Gray)

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.682 ns	Worst Hold Slack (WHS):	0.139 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:	87	Total Number of Endpoints:	87	Total Number of Endpoints:	41

All user specified timing constraints are met.

Figure 30: Synthesis Timing Report (Gray)

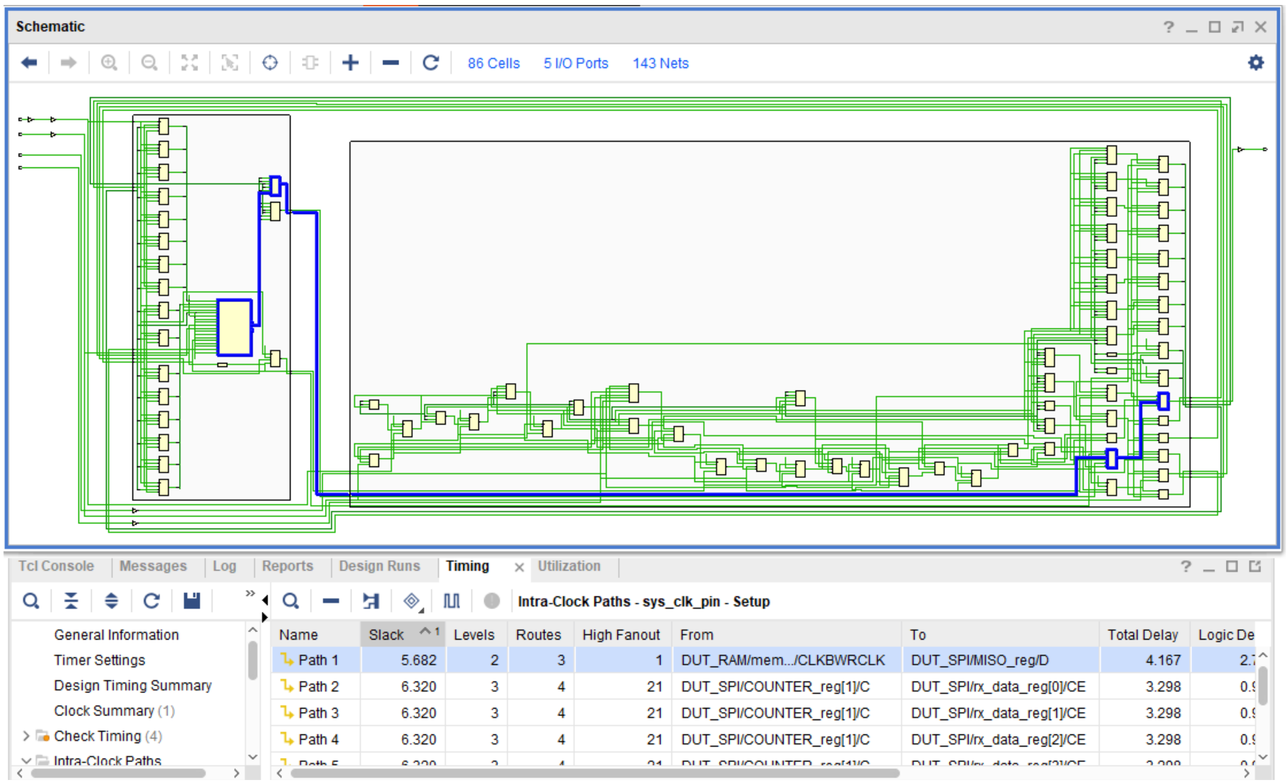


Figure 31: Synthesis Critical Path (Gray)

Name	Slice LUTs (20800)	Slice Registers (41600)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
SPI_Wrapper	34	38	0.5	5	1
DUT_RAM (RAM)	3	17	0.5	0	0
DUT_SPI (SPI_SLAVE)	31	21	0	0	0

Figure 32: Synthesis Utilization Report (Gray)

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.202 ns	Worst Hold Slack (WHS):	0.078 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:	88	Total Number of Endpoints:	88	Total Number of Endpoints:	41

All user specified timing constraints are met.

Figure 33: Implementation Timing Report (Gray)

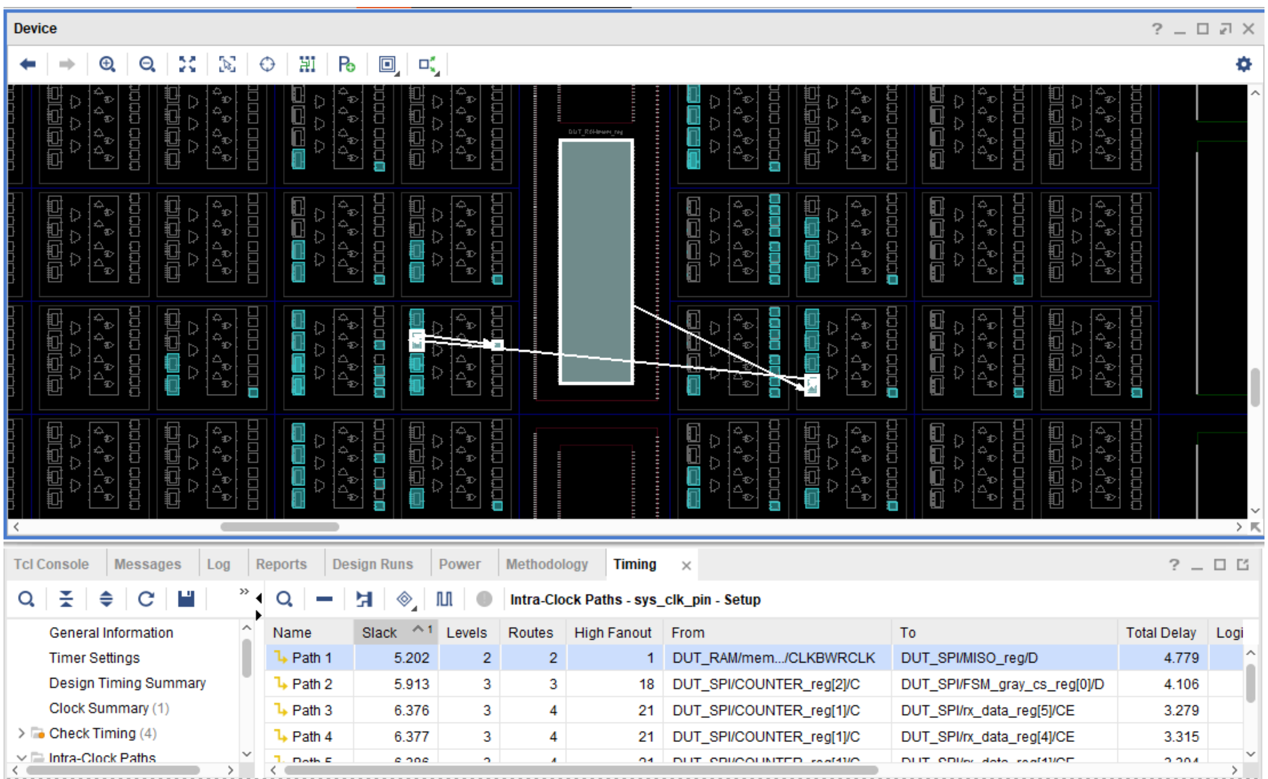


Figure 34: Implementation Critical Path (Gray)

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
SPI_Wrapper	35	38	19	35	10	0.5	5	1
DUT_RAM (RAM)	4	17	6	4	0	0.5	0	0
DUT_SPI (SPI_SLAVE)	31	21	18	31	9	0	0	0

Figure 35: Implementation Utilization Report (Gray)

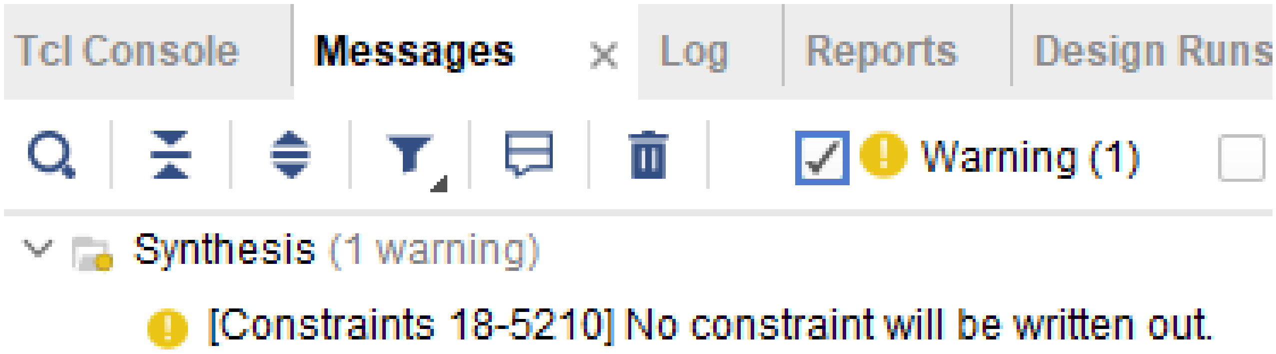


Figure 36: Messages after Implementation (Gray)

3.2 Sequential Encoding:

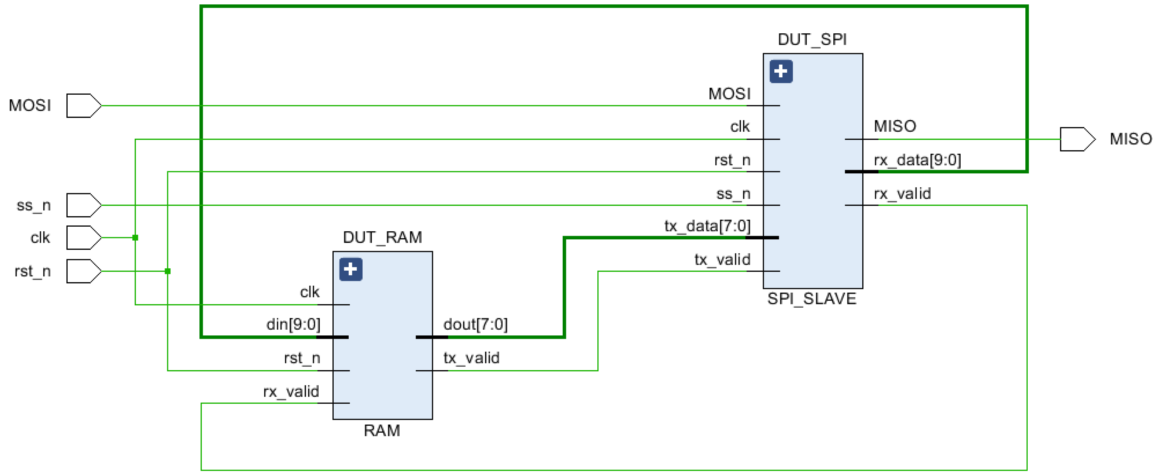


Figure 37: RTL (Sequential)

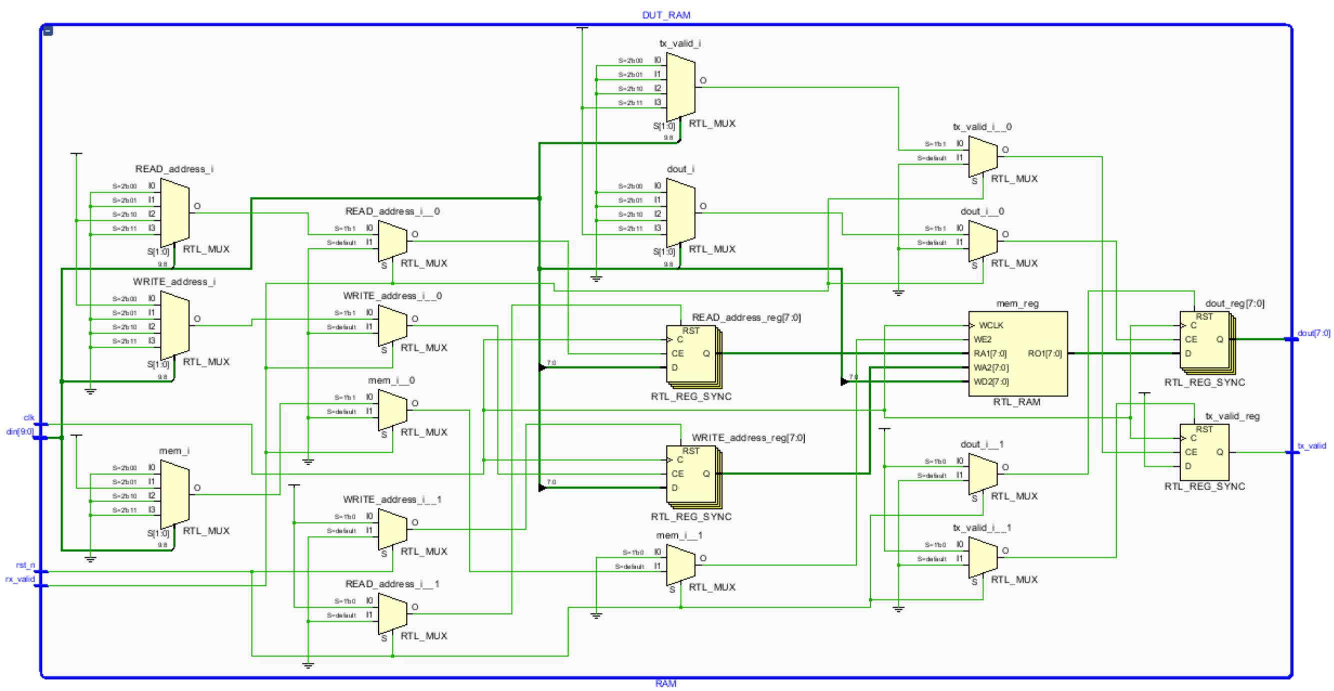


Figure 38: RAM RTL (Sequential)

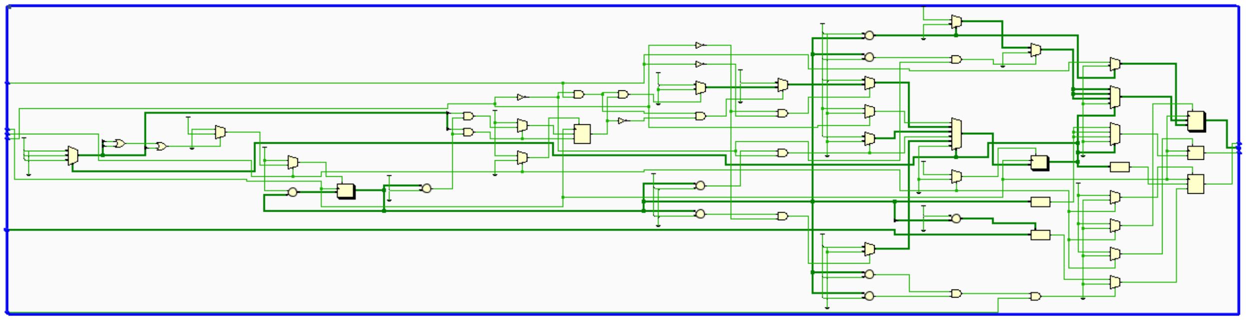


Figure 39: SPI Slave RTL (Sequential)

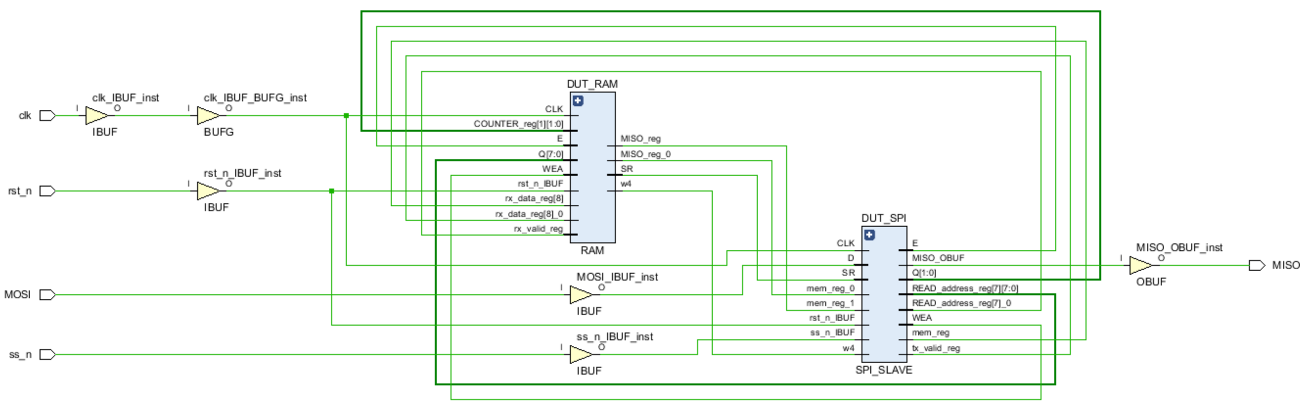
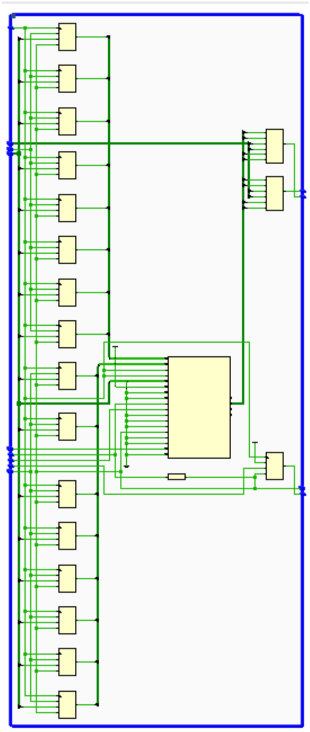


Figure 40: Synthesis (Sequential)



```
---------------------------------------------------------------------------------------------------
                   State |                     New Encoding |                 Previous Encoding
---------------------------------------------------------------------------------------------------
                    IDLE |                              000 |                               000
                 CHK_CMD |                              100 |                               001
                   WRITE |                              011 |                               010
                READ_ADD |                              010 |                               011
               READ_DATA |                              001 |                               100
---------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'cs_reg' using encoding 'sequential' in module 'SPI_SLAVE'
```

Figure 42: Encoding Report (Sequential)

Figure 41: RAM Synthesis (Sequential)

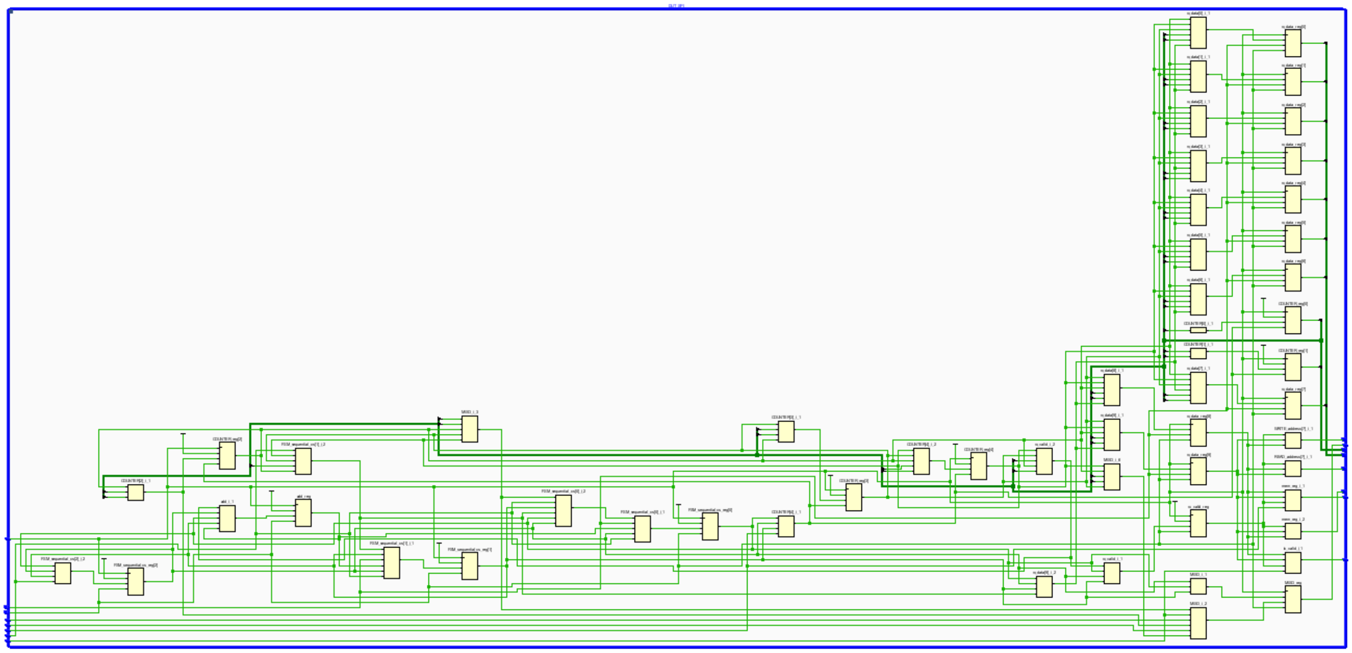


Figure 43: SPI Slave Synthesis (Sequential)

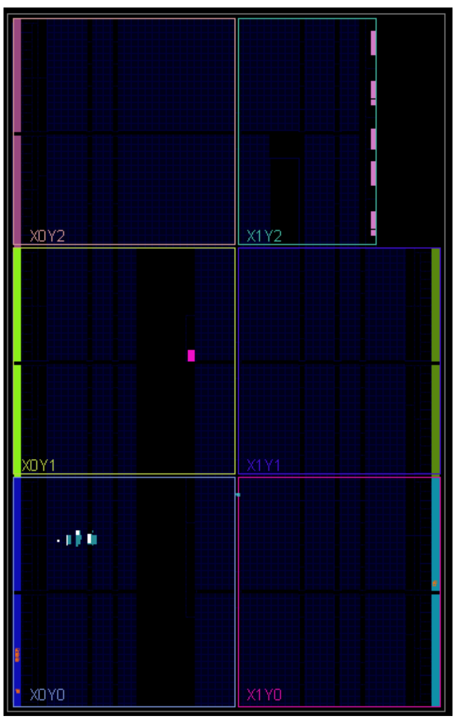


Figure 44: Device (Sequential)

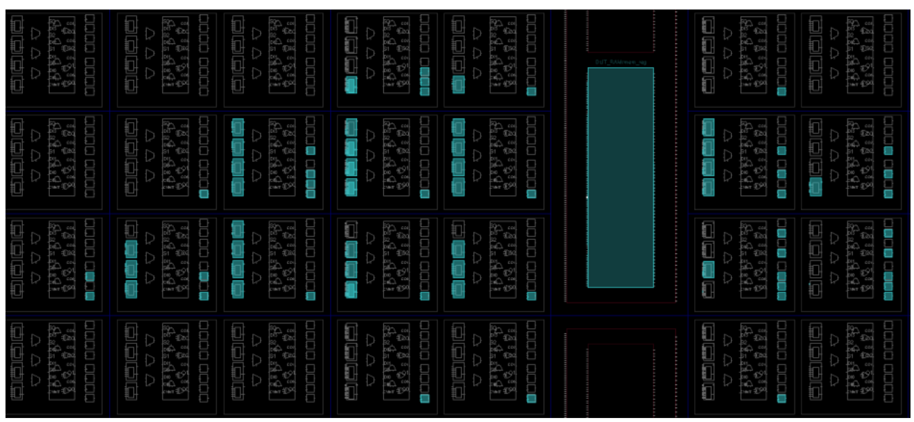


Figure 45: Closer Device (Sequential)

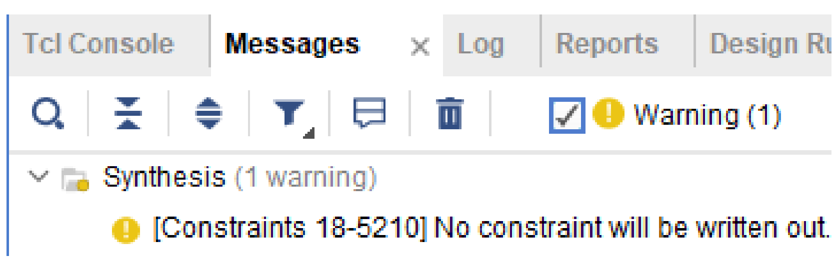


Figure 46: Messages after Implementation (Sequential)

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.682 ns	Worst Hold Slack (WHS):	0.139 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:	87	Total Number of Endpoints:	87	Total Number of Endpoints:	41

All user specified timing constraints are met.

Figure 47: Synthesis Timing Report (Sequential)

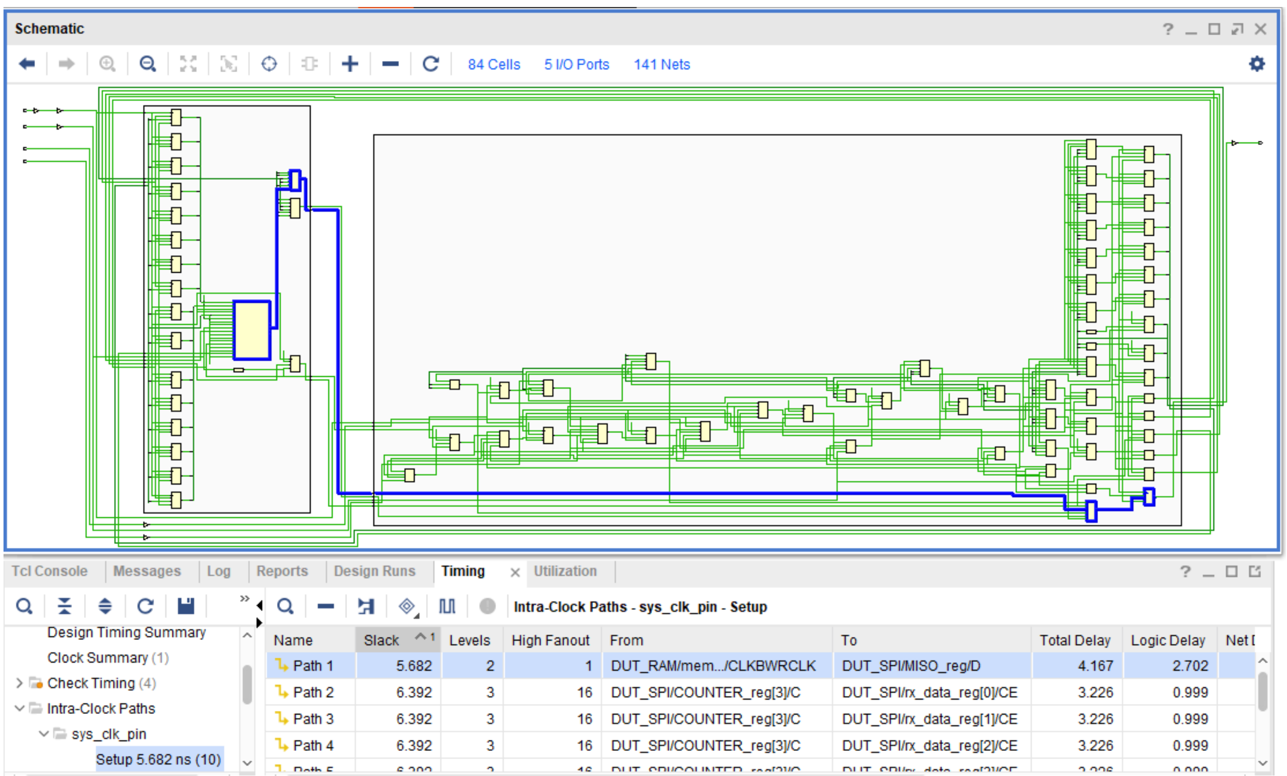


Figure 48: Synthesis Critical Path (Sequential)

Name	Slice LUTs (20800)	Slice Registers (41600)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
SPI_Wrapper	32	38	0.5	5	1
DUT_RAM (RAM)	3	17	0.5	0	0
DUT_SPI (SPI_SLAVE)	29	21	0	0	0

Figure 49: Synthesis Utilization Report (Sequential)

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.089 ns	Worst Hold Slack (WHS):	0.072 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:	88	Total Number of Endpoints:	88	Total Number of Endpoints:	41

All user specified timing constraints are met.

Figure 50: Implementation Timing Report (Sequential)

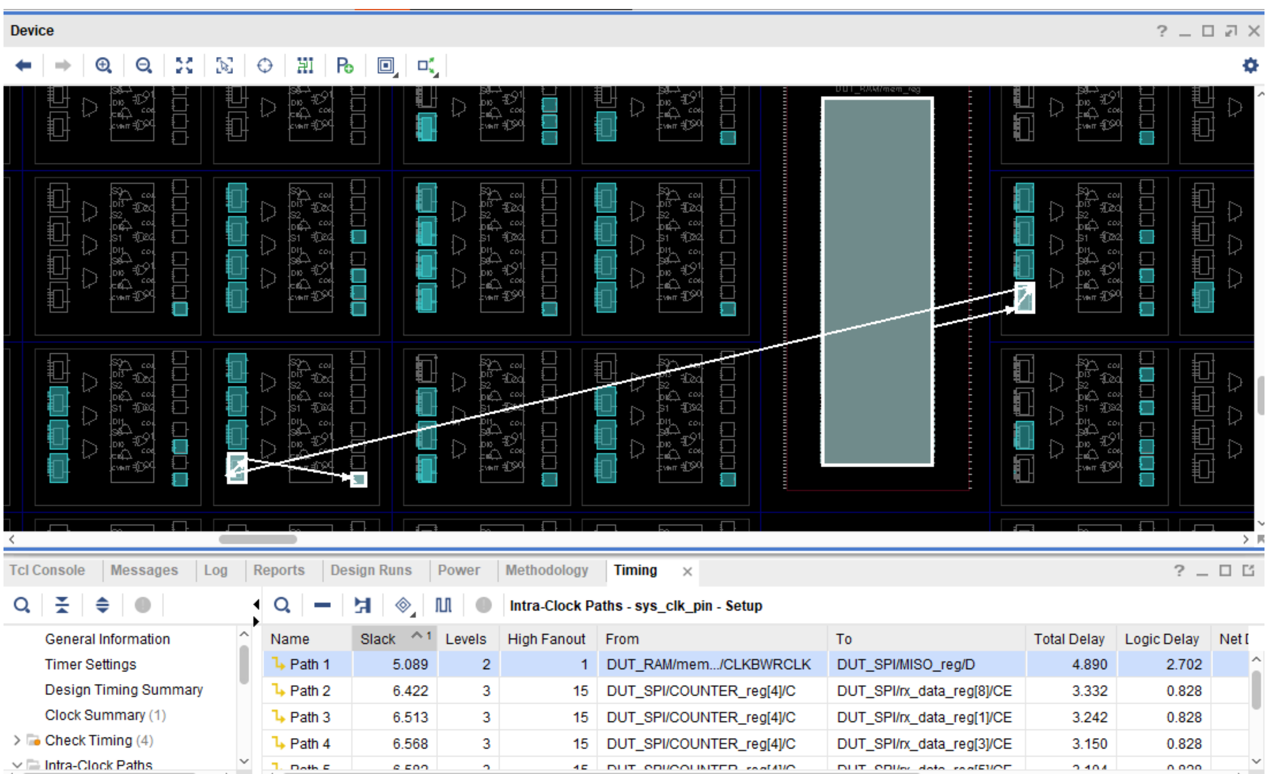


Figure 51: Implementation Critical Path (Sequential)

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
SPI_Wrapper	33	38	19	33	7	0.5	5	1
DUT_RAM (RAM)	4	17	6	4	0	0.5	0	0
DUT_SPI (SPI_SLAVE)	29	21	17	29	7	0	0	0

Figure 52: Implementation Utilization Report (Sequential)

3.3 One-Hot Encoding:

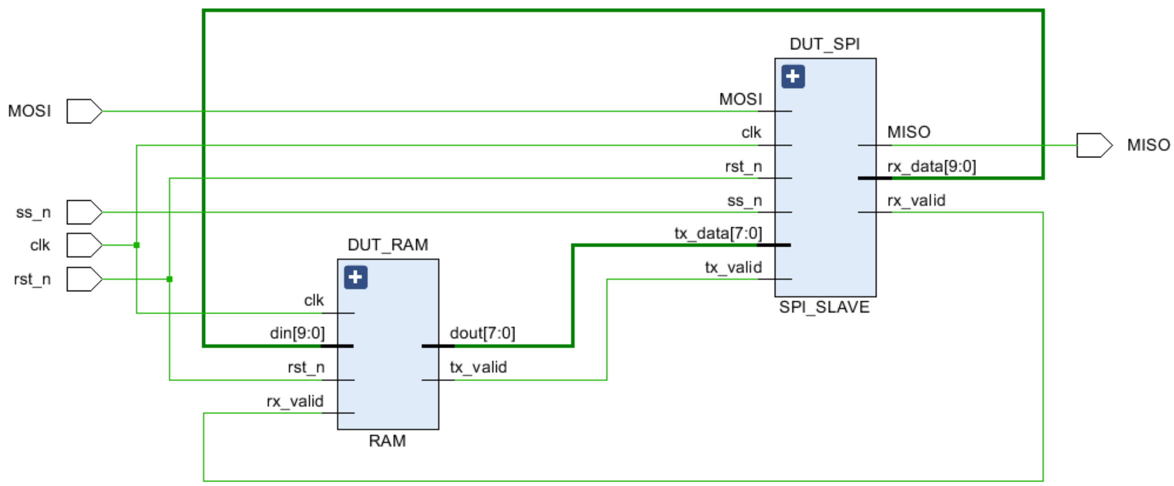


Figure 53: RTL (One-Hot)

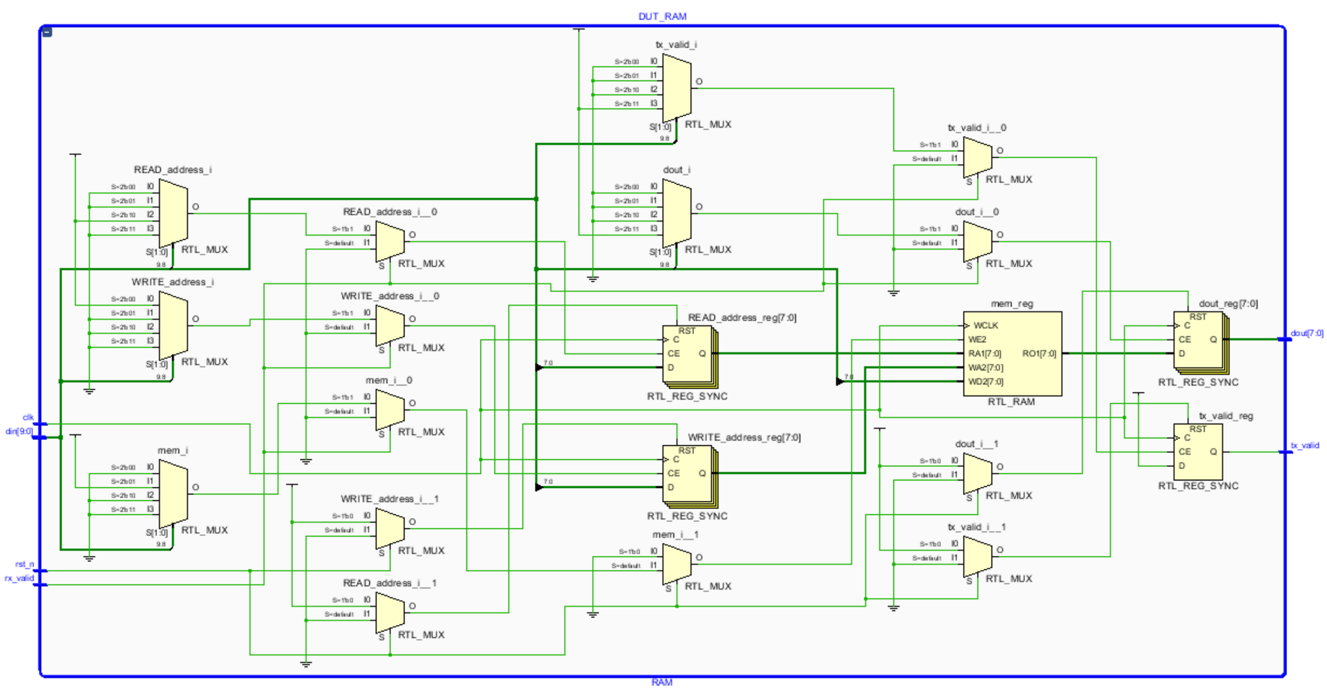


Figure 54: RAM RTL (One-Hot)

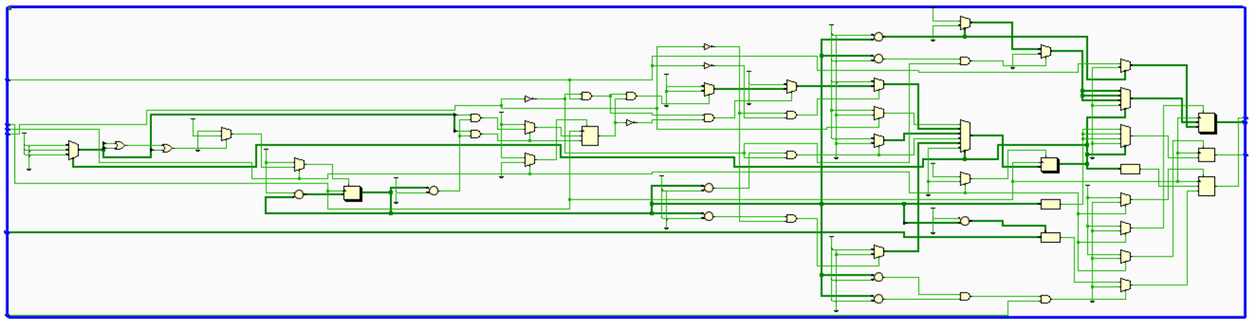


Figure 55: SPI Slave RTL (One-Hot)

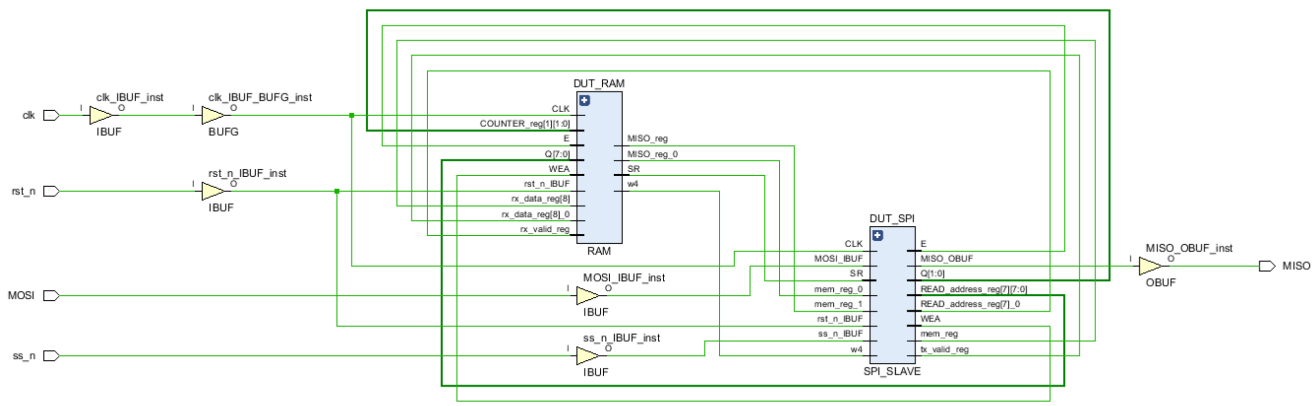


Figure 56: Synthesis (One-Hot)

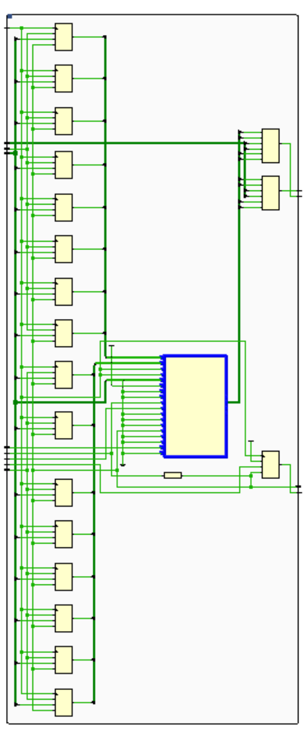


Figure 57: RAM Synthesis (One-Hot)

```
------------------------------------------------------------------------------------------------
               State |               New Encoding |              Previous Encoding
------------------------------------------------------------------------------------------------
                IDLE |                      00001 |                            000
             CHK_CMD |                      10000 |                            001
               WRITE |                      01000 |                            010
            READ_ADD |                      00100 |                            011
           READ_DATA |                      00010 |                            100
------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'cs_reg' using encoding 'one-hot' in module 'SPI_SLAVE'
------------------------------------------------------------------------------------------------
```

Figure 58: Encoding Report (One-Hot)

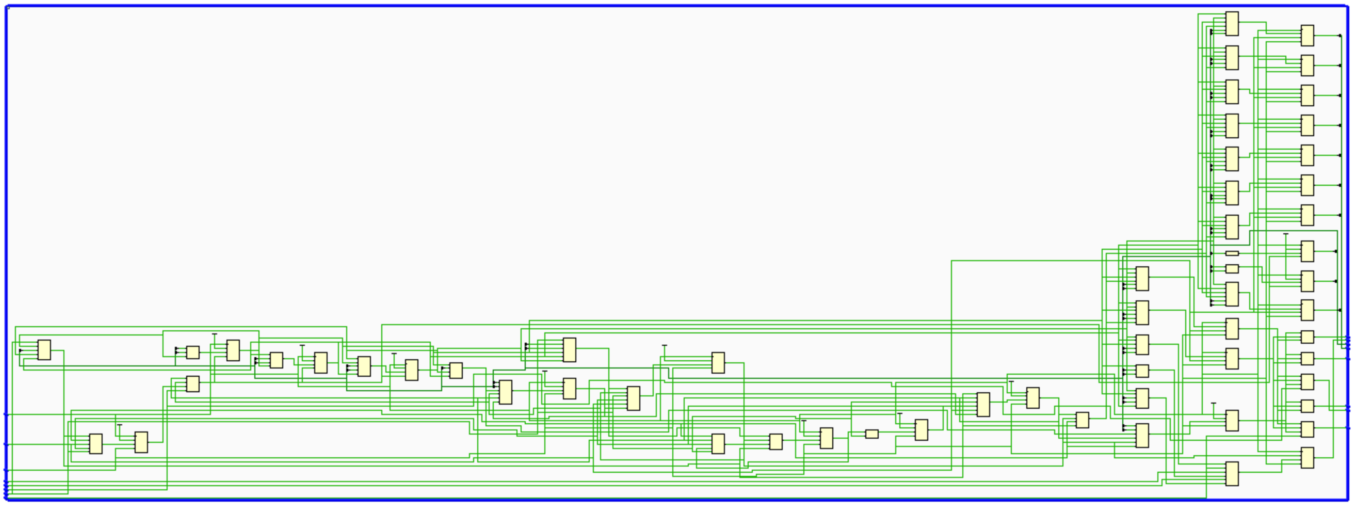


Figure 59: SPI Slave Synthesis (One-Hot)

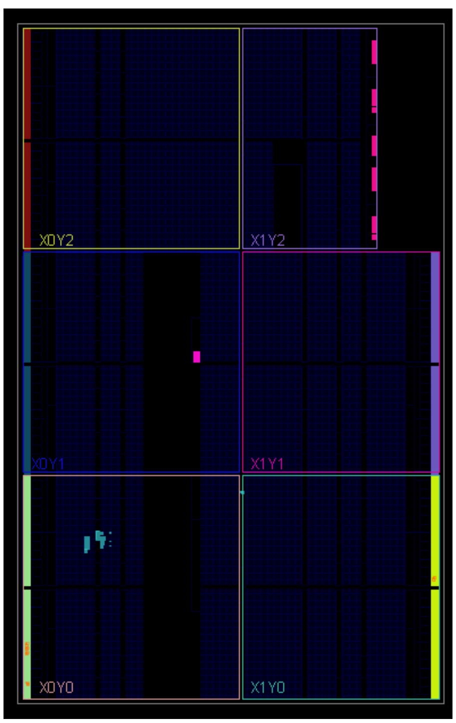


Figure 60: Device (One-Hot)

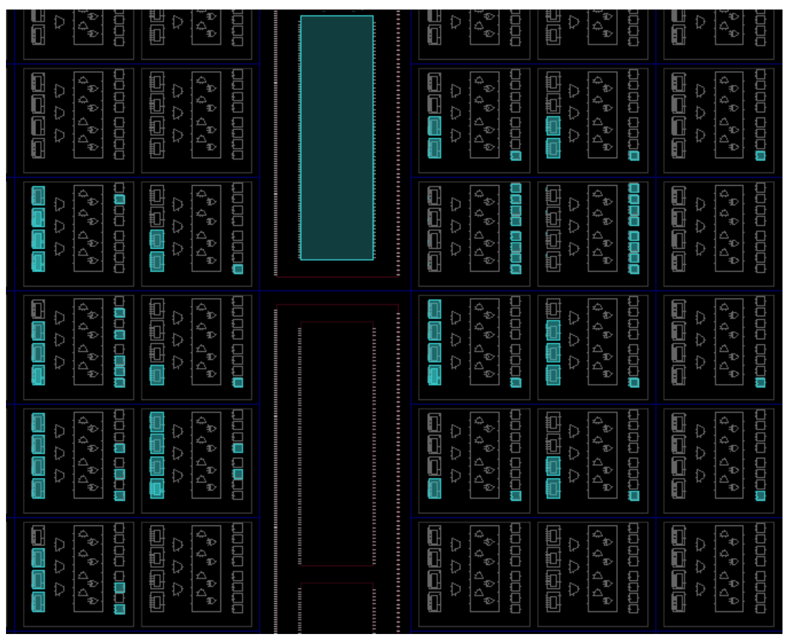


Figure 61: Closer Device (One-Hot)

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.682 ns	Worst Hold Slack (WHS):	0.139 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:	89	Total Number of Endpoints:	89	Total Number of Endpoints:	43

All user specified timing constraints are met.

Figure 62: Synthesis Timing Report (One-Hot)

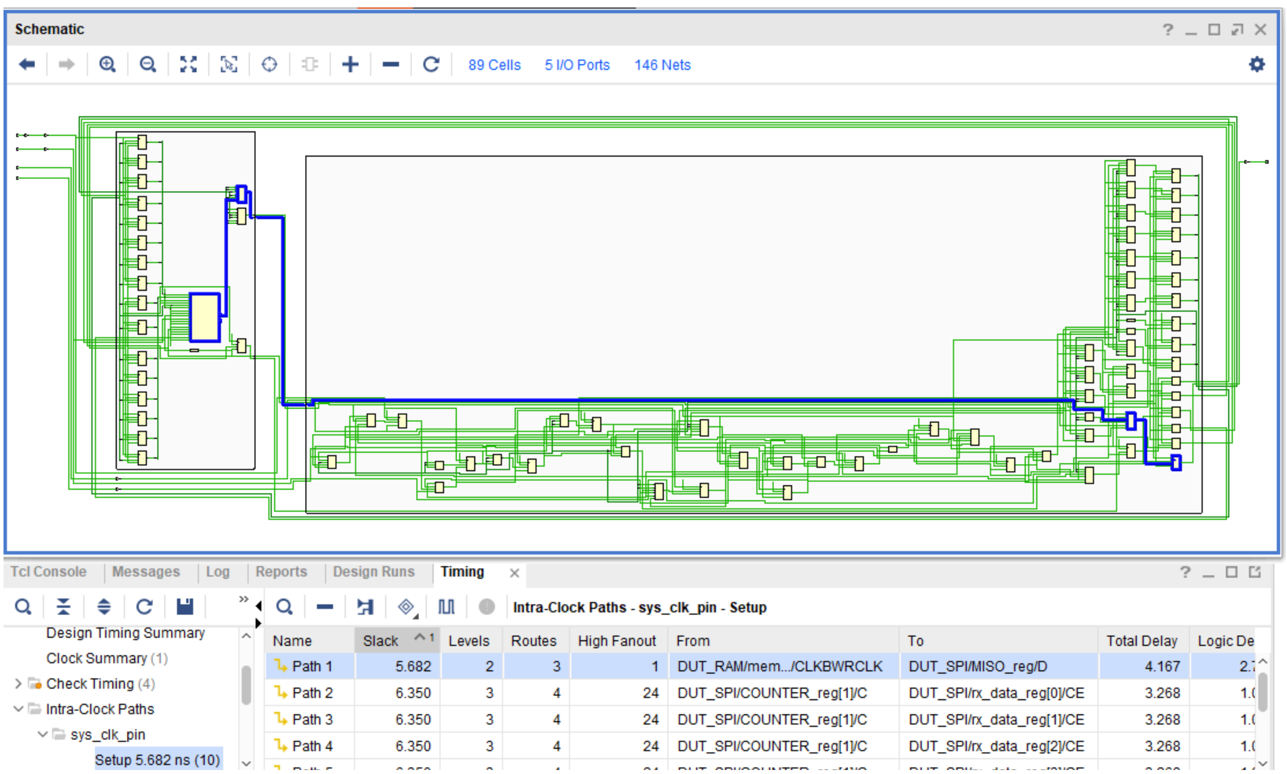


Figure 63: Synthesis Critical Path (One-Hot)

Name	Slice LUTs (20800)	Slice Registers (41600)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
SPI_Wrapper	34	40	0.5	5	1
DUT_RAM (RAM)	3	17	0.5	0	0
DUT_SPI (SPI_SLAVE)	31	23	0	0	0

Figure 64: Synthesis Utilization Report (One-Hot)

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.555 ns	Worst Hold Slack (WHS):	0.101 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:	90	Total Number of Endpoints:	90	Total Number of Endpoints:	43

All user specified timing constraints are met.

Figure 65: Implementation Timing Report (One-Hot)

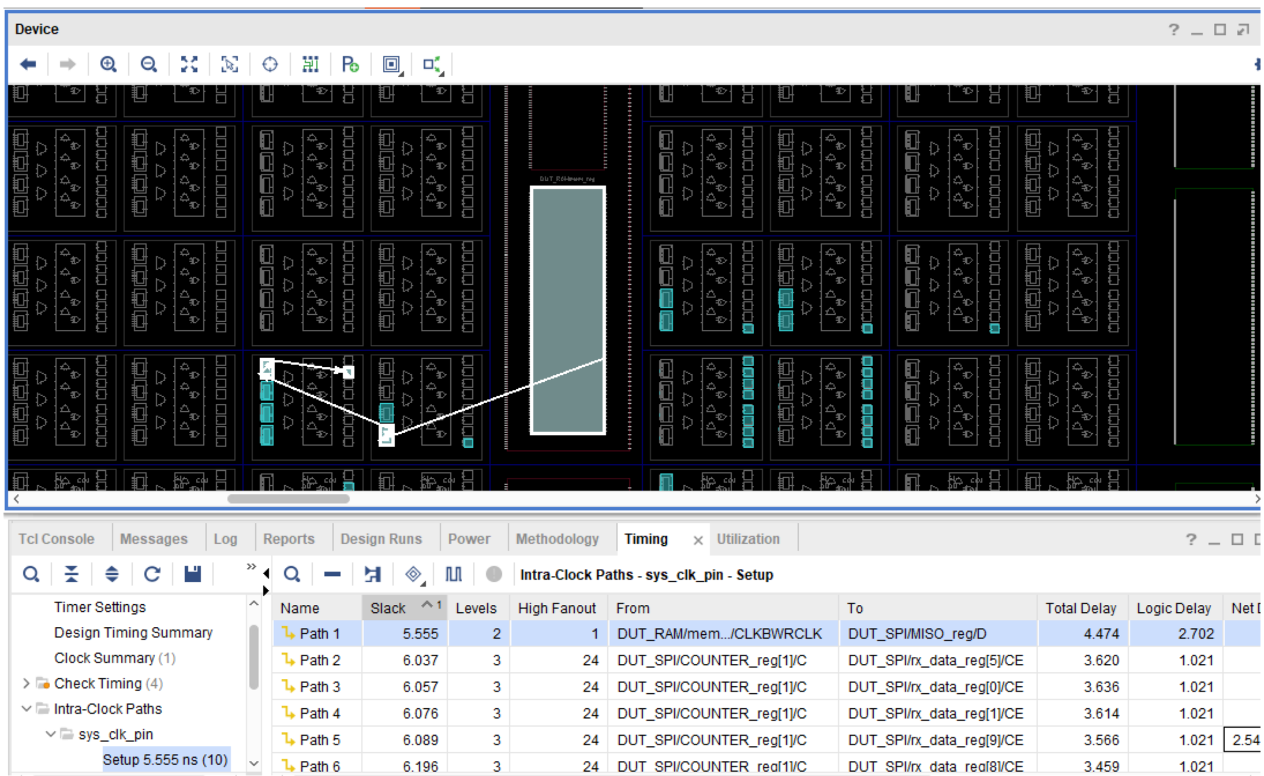


Figure 66: Implementation Critical Path (One-Hot)

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
SPI_Wrapper	35	40	18	35	10	0.5	5	1
DUT_RAM (RAM)	4	17	6	4	0	0.5	0	0
DUT_SPI (SPI_SLAVE)	31	23	16	31	9	0	0	0

Figure 67: Implementation Utilization Report (One-Hot)

We find out that **One-Hot** encoding has the best implementation timing report with the highest Setup/Hold Slack So, we choose **One-Hot** encoding in the design for higher frequency.

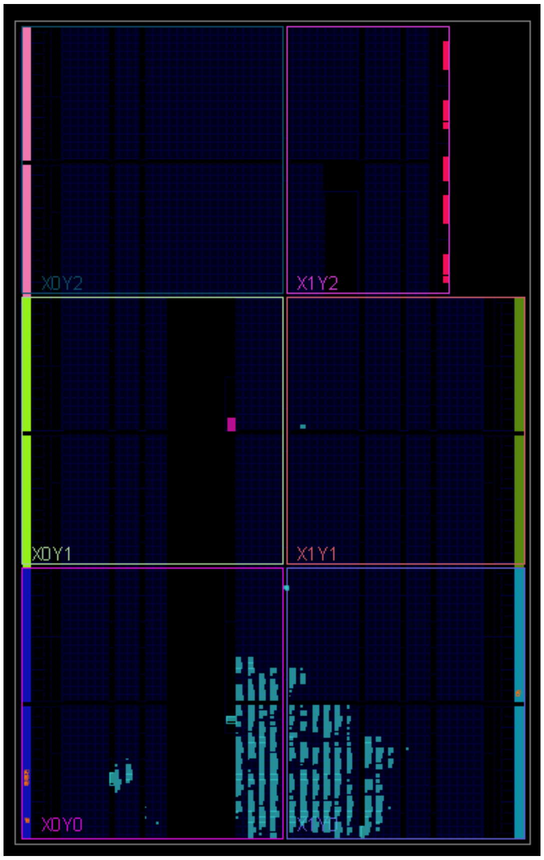


Figure 68: Device after Debugging (One-Hot)

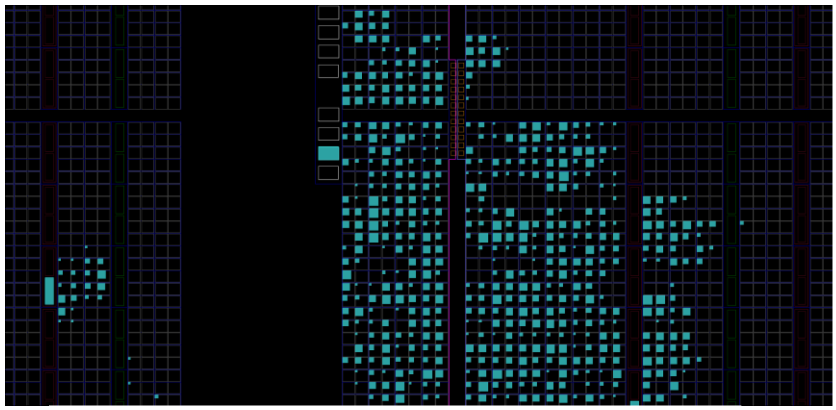


Figure 69: Closer Device after Debugging (One-Hot)

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	LUT Flip Flop Pairs (20800)
SPI_Wrapper	1107	1755	8	570	1019	88	611
dbg_hub (dbg_hub)	476	727	0	241	452	24	307
DUT_RAM (RAM)	3	17	0	6	3	0	0
DUT_SPI (SPI_SLAVE)	31	23	0	18	31	0	11
u_ila_0 (u_ila_0)	597	988	8	312	533	64	292

Figure 70: Implementation Utilization Report after Debugging (One-Hot)

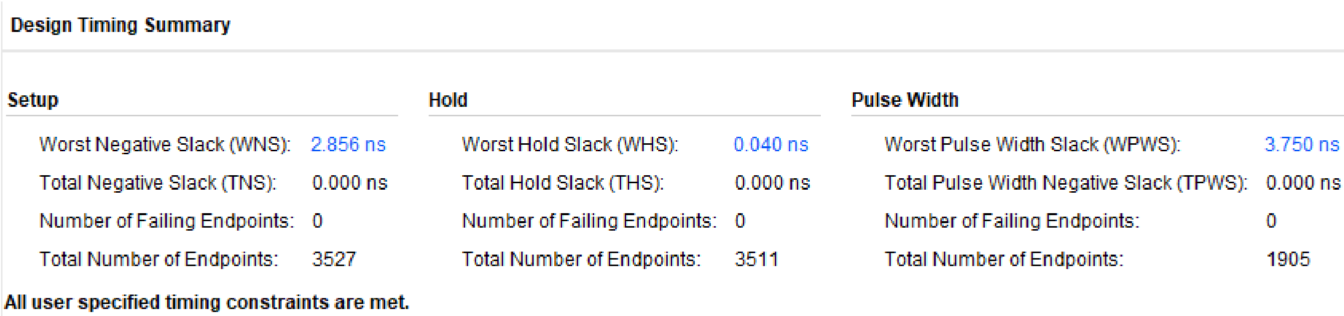


Figure 71: Implementation Timing Report after Debugging (One-Hot)

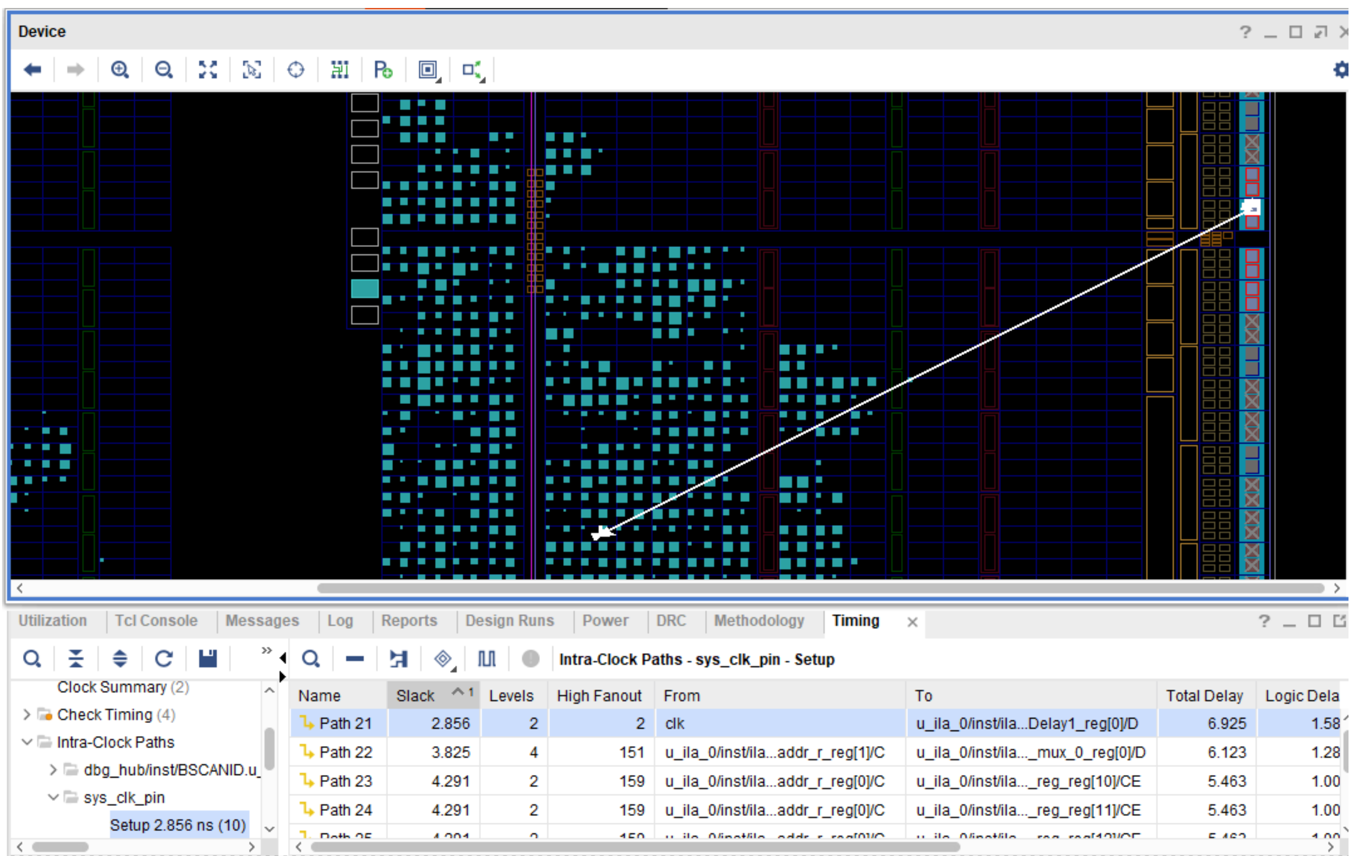


Figure 72: Implementation Critical Path after Debugging (One-Hot)

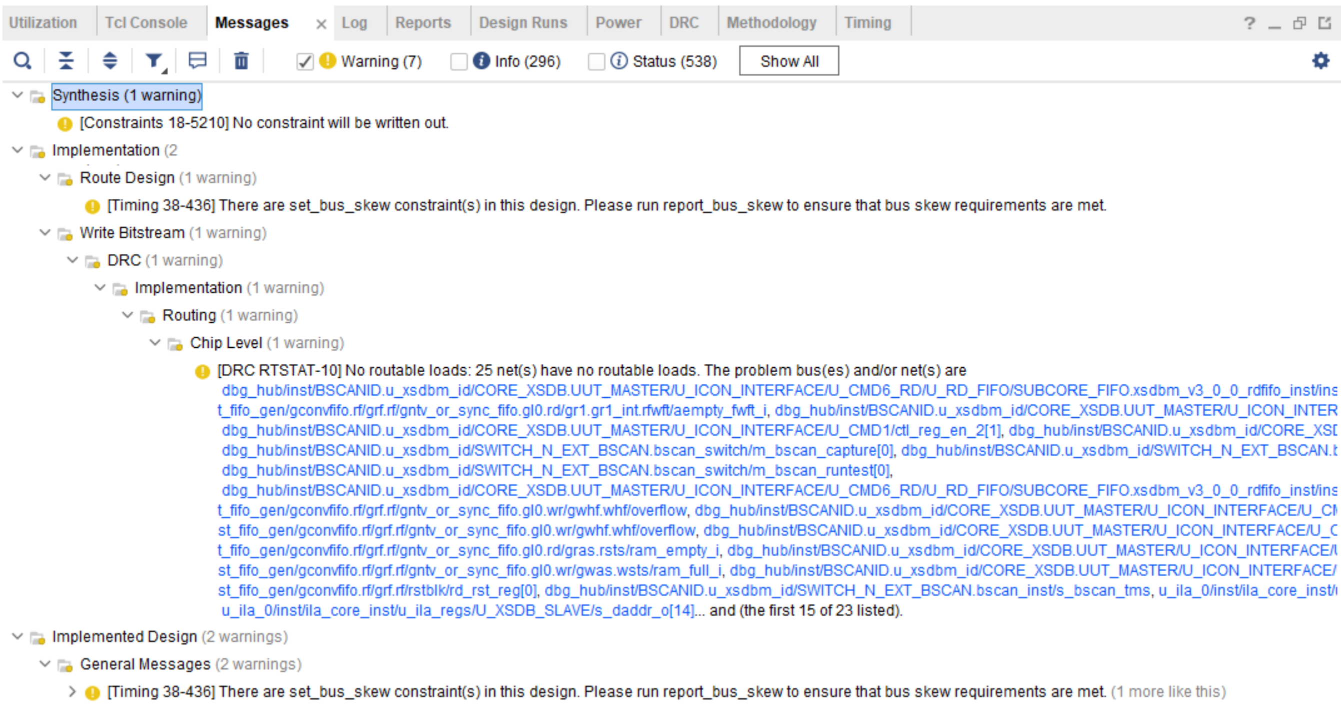


Figure 73: Messages after successful bitstream generation (One-Hot)