SPI Slave with Single Port RAM

Abdelrahman Hatem Nasr

Table of Contents

Overview

Objective

To design and implement an SPI communication protocol in Verilog HDL. The system features an SPI master, an integrated SPI wrapper module that combines the SPI slave and RAM functionalities, and a testbench to validate the entire setup.

Components

1. **SPI Master**: A Verilog module responsible for generating the SPI clock, initiating communication, and handling the MOSI (Master Out Slave In), MISO (Master In Slave Out), and SS_n (Slave Select) lines.
2. **Integrated SPI Wrapper**: A Verilog module that encapsulates both SPI slave functionality and RAM access, bridging communication between the SPI master and memory storage.
3. **Testbench**: A Verilog module to simulate and validate the SPI master, integrated SPI wrapper, and overall system behavior.

System Design

1. **Integrated SPI Wrapper Module**:
    - **Function**: Combines SPI slave functionality with RAM access. It handles SPI communication, interacts with the RAM module, and manages data operations.
    - **Verilog Code**:
        - **SPI Slave Functionality**: Interface with the SPI master to receive commands and data.
        - **RAM Integration**: Implement RAM access within the wrapper, allowing the SPI slave to perform read and write operations.
        - **Data Flow Management**: Ensure proper synchronization and data transfer between the SPI master and RAM through the wrapper.
2. **RAM Module**:
    - **Function**: Provides memory storage for the SPI communication system, allowing the integrated SPI wrapper to store and retrieve data.
    - **Verilog Code**:
        - Design a memory array and address decoding logic.
        - Implement read and write operations with appropriate timing and control signals.
3. **Testbench Module**:
    - **Function**: Simulates the SPI communication system to verify that all modules work correctly together.
    - **Verilog Code**:
        - Generate stimulus to test SPI communication, including sending commands and transferring data.
        - Monitor and verify the behavior of the SPI communication, RAM operations, and overall system functionality.

- Implement assertions to check for correct operation and detect any errors.

Implementation Steps

1. **Design Modules**:
   - Write the Verilog code for the SPI slave,Ram and the integrated SPI wrapper module.
   - Incorporate the RAM functionality within the SPI wrapper to provide memory access.
2. **Simulation and Testing**:
   - Develop a testbench to simulate interactions between the SPI master and the integrated SPI wrapper.
   - Perform simulations to validate the SPI communication protocol, data integrity, and RAM operations.
3. **Verification**:
   - Ensure that the SPI master correctly communicates with the integrated SPI wrapper.
   - Verify that the integrated SPI wrapper correctly handles SPI commands, manages data transfer, and interacts with the RAM.
   - Check the overall system for expected performance and correctness.
4. **Debugging and Optimization**:
   - Analyze simulation results to identify and fix any issues or inconsistencies.
   - Optimize the design for performance, ensuring efficient data handling and minimal latency.

RTL Codes

1.slave:

```
//============================================================
// Project:     SPI Slave with Single Port RAM
// File:        slave.v
// Author:      Abdelrahman Hatem Nasr
// Date:        2024-08-05
// Description: Implementation of an SPI slave module for
//              serial communication. The module handles
//              data reception and transmission with an SPI
//              master, including state management and data
//              handling logic.
//============================================================

module SLAVE (MOSI, MISO, SS_n, clk, rst_n, rx_data, rx_valid, tx_data,
tx_valid);

//============================================================
// Parameter Declarations
//============================================================

parameter IDLE      = 0;
parameter CHK_CMD   = 1;
parameter WRITE     = 2;
parameter READ_ADD  = 3;
parameter READ_DATA = 4;

//============================================================
// Port Declarations
//============================================================

input MOSI, SS_n, clk, rst_n, tx_valid;
input [7:0] tx_data;
output reg rx_valid, MISO;
output reg [9:0] rx_data;

//============================================================
// Internal Signal Declarations
// counter : Counter for bit shifts and data handling
// allow_memorize : Flag to determine if data should be memorized
// ns, cs : Next state and current state for state machine
//============================================================
```

```
reg [3:0] counter;
reg allow_memorize;
(*fsm_encoding="gray"*)
reg [2:0] ns, cs;

//============================================================
// State Memory
//============================================================

always @(posedge clk) begin
    if (~rst_n) begin
        cs <= IDLE;
    end else begin
        cs <= ns;
    end
end

//============================================================
// Next State Logic
//============================================================

always @(*) begin
    case(cs)
    IDLE: begin
        if (SS_n) begin
            ns = IDLE;
        end else begin
            ns = CHK_CMD;
        end
    end
    CHK_CMD: begin
        if (SS_n) begin
            ns = IDLE;
        end else if (MOSI == 0) begin
            ns = WRITE;
        end else if (MOSI == 1 && allow_memorize == 0) begin
            ns = READ_ADD;
        end else if (MOSI == 1 && allow_memorize == 1) begin
            ns = READ_DATA;
        end
    end
    WRITE: begin
        if (SS_n) begin
            ns = IDLE;
        end else begin
            ns = WRITE;
        end
    end
```

```
    READ_ADD: begin
        if (SS_n) begin
            ns = IDLE;
        end else begin
            ns = READ_ADD;
        end
    end
    READ_DATA: begin
        if (SS_n) begin
            ns = IDLE;
        end else begin
            ns = READ_DATA;
        end
    end
    default : ns = IDLE;
    endcase
end

//===============================================================
// Output Logic
//===============================================================

always @(posedge clk) begin
    if (~rst_n) begin
        counter <= 0;
        rx_valid <= 0;
        rx_data <= 0;
        MISO <= 0;
        allow_memorize <= 0;
    end else begin
        case (cs)
            IDLE: begin
                rx_valid <= 0;
                MISO <= 0;
                counter <= 0;
            end
            WRITE: begin
                if (counter < 10) begin
                    rx_data <= {rx_data[8:0], MOSI};
                    rx_valid<=0;
                    counter <= counter + 1;
                end
                if (counter == 10) begin
                    rx_valid <= 1;
                end
            end
            READ_ADD: begin
                if (counter < 10) begin
```

```
                    rx_data <= {rx_data[8:0], MOSI};
                    rx_valid<=0;
                    counter <= counter + 1;
                end
                if (counter == 10) begin
                    rx_valid <= 1;
                    allow_memorize <= 1;
                end
            end
            READ_DATA: begin
                if (~tx_valid) begin
                    if (counter < 10) begin
                        rx_data <= {rx_data[8:0], MOSI};
                        rx_valid<=0;
                        counter <= counter + 1;
                    end
                    if (counter == 10) begin
                        rx_valid <= 1;
                        allow_memorize <= 0;
                    end
                end
                if (tx_valid) begin
                    if(counter >= 3)begin
                        MISO <= tx_data[counter - 3];
                        counter <= counter - 1;
                    end
                end
            end
        endcase
    end
end

endmodule
```

2.Ram:

```
//============================================================
// Project:     SPI Slave with Single Port RAM
// File:        ram.v
// Author:      Abdelrahman Hatem Nasr
// Date:        2024-08-05
// Description: Implementation of a simple RAM module with
//              basic read and write operations. The module
//              stores data in memory and provides output based
//              on command signals.
//============================================================

module RAM (din,clk, rx_valid, rst_n,dout,tx_valid);

//============================================================
// Parameter Declarations
//============================================================

parameter MEM_DEPTH = 256;
parameter ADDR_SIZE = 8;

//============================================================
// Port Declarations
//============================================================

input  [9:0] din;
input  clk, rx_valid, rst_n;
output reg [7:0] dout;
output reg tx_valid;

//============================================================
// Internal Signal Declarations
// mem : Memory array for storage
// stored_command : Command for read or write operations
//============================================================

reg [ADDR_SIZE-1:0] mem [MEM_DEPTH-1:0];
reg [9:0] stored_command;

//============================================================
// Main Functionality
//============================================================

always @(posedge clk) begin
    if (~rst_n) begin
```

```
        // Reset logic: clear outputs and stored command
        tx_valid <= 0;
        dout <= 0;
        stored_command<=0;
    end else begin
        if(rx_valid)begin
            // Command decoding based on the higher 2 bits of din
            case(din[9:8])
            2'b00: begin
                 // Store the command in stored_command
                stored_command<=din;
                tx_valid<=0;
            end
            2'b01:begin
                // Write data to memory at the address specified by
stored_command
                mem[stored_command[7:0]]<=din[7:0];
                tx_valid<=0;
            end
            2'b10:begin
                 // Update the stored command
                stored_command<=din;
                tx_valid<=0;
            end
            2'b11:begin
                // Read data from memory at the address specified by
stored_command
                dout<=mem[stored_command[7:0]];
                tx_valid<=1;
            end
            default:tx_valid<=0;
        endcase
        end
    end
end

endmodule
```

3.SPI Wrapper:

```
//============================================================
// Project:     SPI Slave with Single Port RAM
// File:        spi_wrapper.v
// Author:      Abdelrahman Hatem Nasr
// Date:        2024-08-05
// Description: Wrapper module for interfacing with SPI.
//              It instantiates the RAM and SPI Slave modules
//              to handle data transfer and memory operations.
//============================================================

module SPI_Wrapper(MOSI, MISO, SS_n, rst_n, clk);

//============================================================
// Port Declarations
//============================================================

input MOSI, SS_n, clk, rst_n;  // SPI input signals and reset
output MISO;                   // SPI output signal

//============================================================
// Internal Signal Declarations
//============================================================

wire [9:0] rx_data;  // Data received from SPI slave
wire rx_valid;       // Valid signal for received data
wire tx_valid;       // Valid signal for transmitted data
wire [7:0] tx_data;  // Data to be transmitted via SPI

//============================================================
// Module Instantiations
//============================================================

RAM #(.MEM_DEPTH(256), .ADDR_SIZE(8)) m1 (
    .din(rx_data),        // Data input to RAM
    .clk(clk),            // Clock signal
    .rx_valid(rx_valid),  // Valid signal for received data
    .rst_n(rst_n),        // Reset signal
    .dout(tx_data),       // Data output from RAM
    .tx_valid(tx_valid)   // Valid signal for transmitted data
);

SLAVE s1 (
    .MOSI(MOSI),         // SPI Master Out Slave In
    .MISO(MISO),         // SPI Master In Slave Out
```

```
    .SS_n(SS_n),          // SPI Slave Select
    .clk(clk),            // Clock signal
    .rst_n(rst_n),        // Reset signal
    .rx_data(rx_data),    // Data received from SPI master
    .rx_valid(rx_valid), // Valid signal for received data
    .tx_data(tx_data),    // Data to be transmitted to SPI master
    .tx_valid(tx_valid)  // Valid signal for transmitted data
);

endmodule
```

Testbench Code

```
//============================================================
// Project:     SPI Slave with Single Port RAM
// File:        spi_master_tb.v
// Author:      Abdelrahman Hatem Nasr
// Date:        2024-08-05
// Description: Testbench for the SPI_Wrapper module.
//              Tests various RAM read and write operations
//              by sending commands via SPI and verifying
//              the memory contents.
//============================================================

module SPI_Master_tb();

//============================================================
// Testbench Signal Declarations
//============================================================

reg MOSI, SS_n, clk, rst_n;       // SPI interface signals and reset
wire MISO;                        // SPI MISO line

reg [7:0] expected_written_data_1; // Expected data for address 2
reg [7:0] expected_written_data_2; // Expected data for address 100

// Instantiate the SPI_Wrapper module
SPI_Wrapper SPI1 (
    .MOSI(MOSI),
    .MISO(MISO),
    .SS_n(SS_n),
    .rst_n(rst_n),
    .clk(clk)
);

//============================================================
// Clock Generation
//============================================================

initial begin
    clk = 0;
    forever #5 clk = ~clk; // Clock period of 10 units
end

//============================================================
// Test Sequence
//============================================================

initial begin
```

```
    $readmemh("mem.dat", SPI1.m1.mem); // Initialize RAM memory from file
    rst_n = 0;
    SS_n = 1;
    MOSI = 0;
    @(negedge clk);
    rst_n = 1; // Release reset

    expected_written_data_1 = 8;
    expected_written_data_2 = 14;

    // Test RAM write command: Write address 2
    SS_n = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    repeat (8) begin
        MOSI = 0;
        @(negedge clk);
    end
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    SS_n = 1;
    @(negedge clk);

    // Test RAM write command: Write data 8 to address 2
    SS_n = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    repeat (4) begin
        MOSI = 0;
        @(negedge clk);
    end
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    SS_n = 1;
```

```
    repeat(5)@(negedge clk);
    if (SPI1.m1.mem[2] != expected_written_data_1) begin
        $display("Error in write data at address 2 in RAM!");
    end

    // Test RAM write command: Write address 100
    SS_n = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    repeat (3) begin
        MOSI = 0;
        @(negedge clk);
    end
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    SS_n = 1;
    @(negedge clk);

    // Test RAM write command: Write data 14 to address 100
    SS_n = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    repeat (4) begin
        MOSI = 0;
        @(negedge clk);
    end
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
```

```
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    SS_n = 1;
    repeat(5)@(negedge clk);
    if (SPI1.m1.mem[100] != expected_written_data_2) begin
        $display("Error in write data at address 100 in RAM!");
    end

    // Test RAM read command: Read address 2
    SS_n = 0;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    repeat (6) begin
        MOSI = 0;
        @(negedge clk);
    end
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    SS_n = 1;
    @(negedge clk);

    // Test RAM read command: Read data from address 2
    SS_n = 0;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    repeat (6) begin
        MOSI = 0;
        @(negedge clk);
    end
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    repeat(11)@(negedge clk);
    SS_n = 1;
```

```
    @(negedge clk);

    // Test RAM read command: Read address 100
    SS_n = 0;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    MOSI = 0;
    @(negedge clk);
    SS_n = 1;
    @(negedge clk);

    // Test RAM read command: Read data from address 100
    SS_n = 0;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    MOSI = 1;
    @(negedge clk);
    repeat (6) begin
        MOSI = 0;
        @(negedge clk);
    end
    MOSI = 1;
    @(negedge clk);
    MOSI = 0;
    repeat(11)@(negedge clk);
    SS_n = 1;
```

```
    @(negedge clk);

    $stop; // Stop simulation
end

//============================================================
// Signal Monitoring
//============================================================

initial begin
    $monitor("Time: %0t | MOSI: %b | MISO: %b | SS_n: %b | clk: %b | rst_n:
%b",
             $time, MOSI, MISO, SS_n, clk, rst_n);
end

endmodule
```

Do file

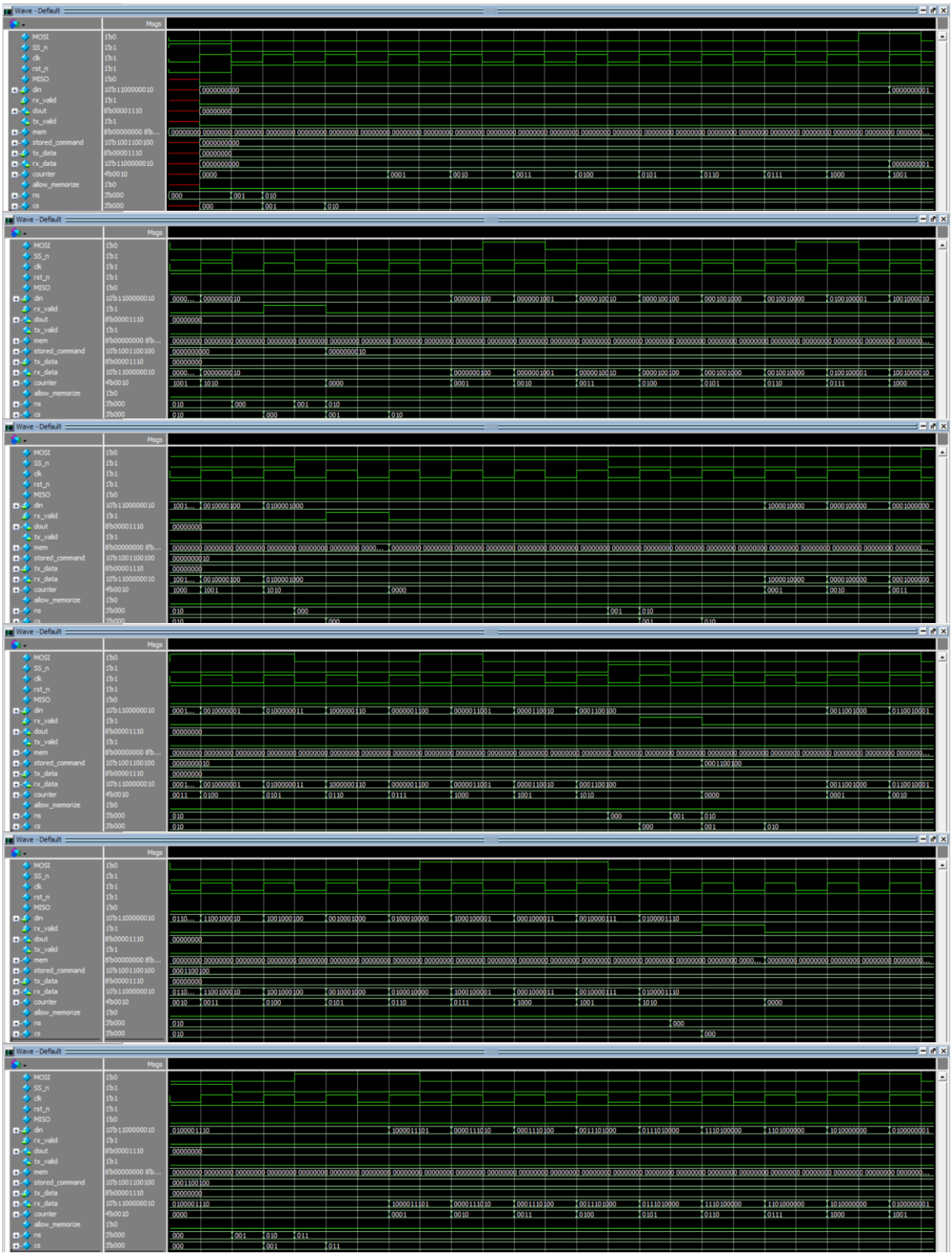
```
vlib work

vlog RAM.v SLAVE.v SPI_Wrapper.v SPI_Master_tb.v

vsim -voptargs=+acc work.SPI_Master_tb

add wave *

add wave -position insertpoint  \

sim:/SPI_Master_tb/SPI1/m1/din \

sim:/SPI_Master_tb/SPI1/m1/rx_valid \

sim:/SPI_Master_tb/SPI1/m1/dout \

sim:/SPI_Master_tb/SPI1/m1/tx_valid \

sim:/SPI_Master_tb/SPI1/m1/mem \

sim:/SPI_Master_tb/SPI1/m1/stored_command \

sim:/SPI_Master_tb/SPI1/s1/tx_data \

sim:/SPI_Master_tb/SPI1/s1/rx_data \

sim:/SPI_Master_tb/SPI1/s1/counter \

sim:/SPI_Master_tb/SPI1/s1/allow_memorize \

sim:/SPI_Master_tb/SPI1/s1/ns \

sim:/SPI_Master_tb/SPI1/s1/cs



run -all

#quit -sim
```

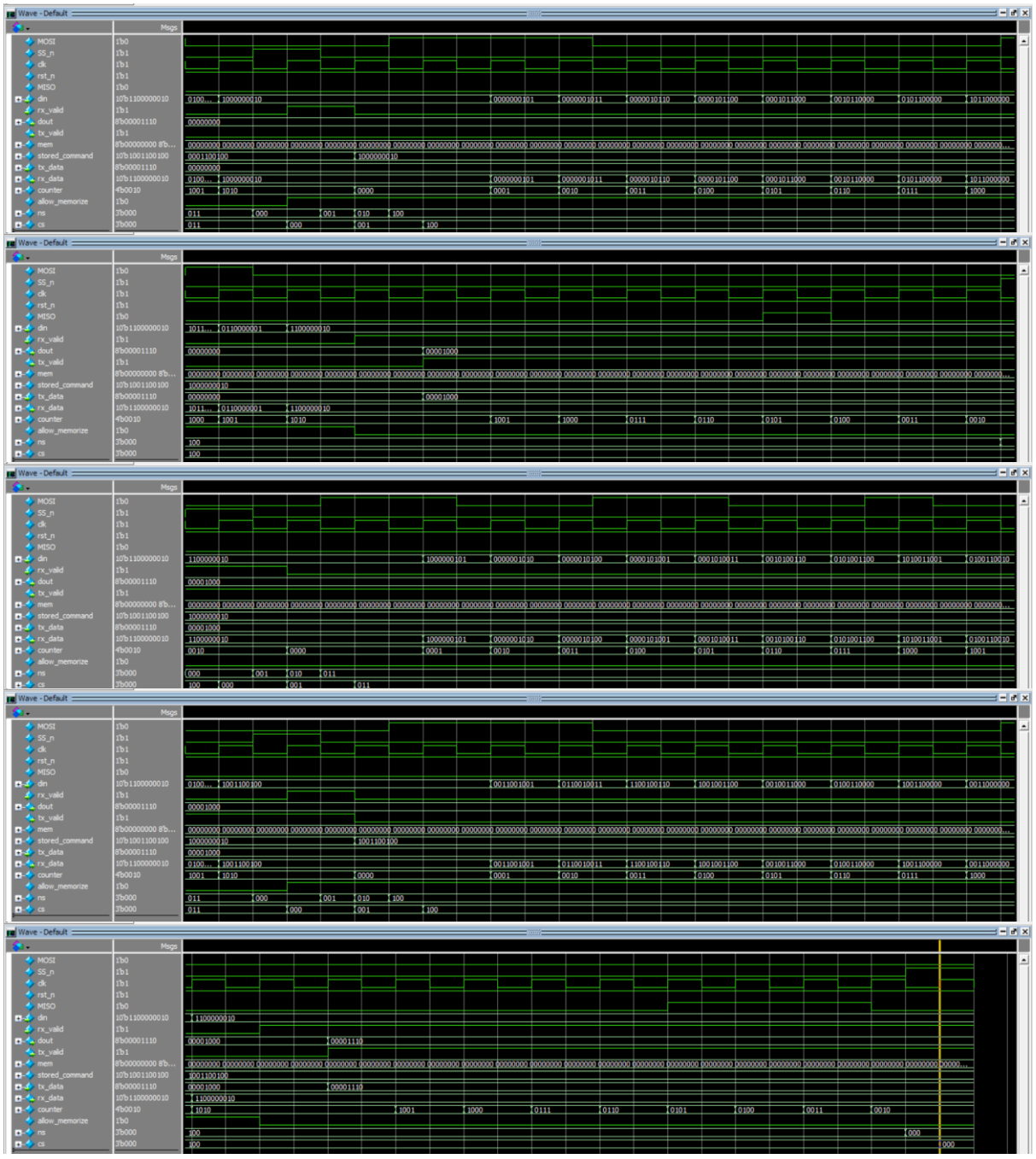
Constraints file

```
## Clock signal
set_property -dict {PACKAGE_PIN W5 IOSTANDARD LVCMOS33} [get_ports clk]
create_clock -period 10.000 -name sys_clk_pin -waveform {0.000 5.000} -add
[get_ports clk]


## Switches
set_property -dict {PACKAGE_PIN V17 IOSTANDARD LVCMOS33} [get_ports rst_n]
set_property -dict {PACKAGE_PIN V16 IOSTANDARD LVCMOS33} [get_ports MOSI]
set_property -dict {PACKAGE_PIN W16 IOSTANDARD LVCMOS33} [get_ports SS_n]


## LEDs
set_property -dict {PACKAGE_PIN U16 IOSTANDARD LVCMOS33} [get_ports MISO]


## Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]

## SPI configuration mode options for QSPI boot, can be used for all designs
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]


create_debug_core u_ila_0 ila
set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
set_property ALL_PROBE_SAME_MU_CNT 1 [get_debug_cores u_ila_0]
set_property C_ADV_TRIGGER false [get_debug_cores u_ila_0]
set_property C_DATA_DEPTH 1024 [get_debug_cores u_ila_0]
set_property C_EN_STRG_QUAL false [get_debug_cores u_ila_0]
set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ila_0]
set_property C_TRIGIN_EN false [get_debug_cores u_ila_0]
set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
set_property port_width 1 [get_debug_ports u_ila_0/clk]
connect_debug_port u_ila_0/clk [get_nets [list clk_IBUF_BUFG]]
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe0]
set_property port_width 1 [get_debug_ports u_ila_0/probe0]
connect_debug_port u_ila_0/probe0 [get_nets [list clk_IBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe1]
set_property port_width 1 [get_debug_ports u_ila_0/probe1]
connect_debug_port u_ila_0/probe1 [get_nets [list MISO_OBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe2]
```

```
set_property port_width 1 [get_debug_ports u_ila_0/probe2]
connect_debug_port u_ila_0/probe2 [get_nets [list MOSI_IBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe3]
set_property port_width 1 [get_debug_ports u_ila_0/probe3]
connect_debug_port u_ila_0/probe3 [get_nets [list rst_n_IBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe4]
set_property port_width 1 [get_debug_ports u_ila_0/probe4]
connect_debug_port u_ila_0/probe4 [get_nets [list SS_n_IBUF]]
set_property C_CLK_INPUT_FREQ_HZ 300000000 [get_debug_cores dbg_hub]
set_property C_ENABLE_CLK_DIVIDER false [get_debug_cores dbg_hub]
set_property C_USER_SCAN_CHAIN 1 [get_debug_cores dbg_hub]
connect_debug_port dbg_hub/clk [get_nets clk_IBUF_BUFG]
```

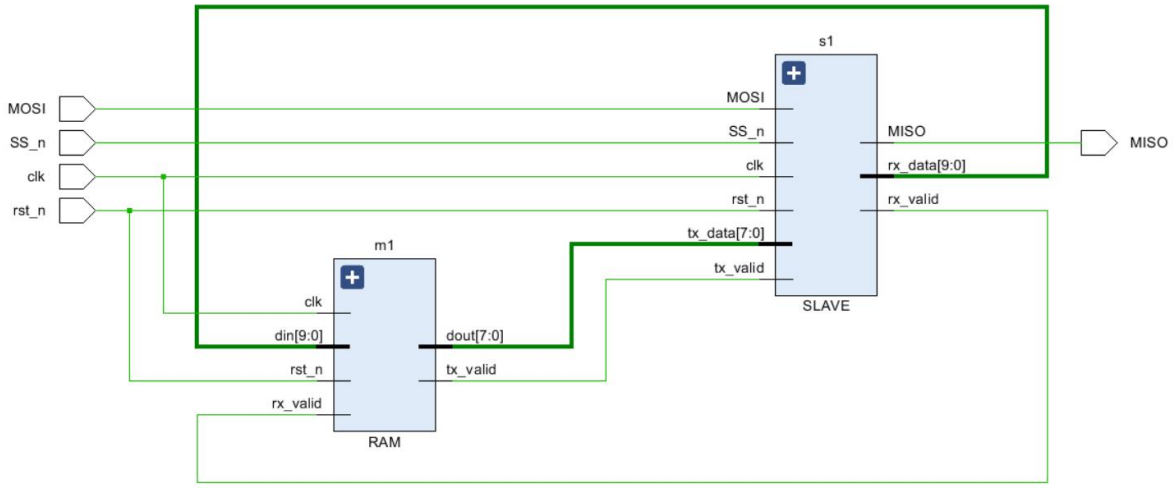
QuestaSim snippets



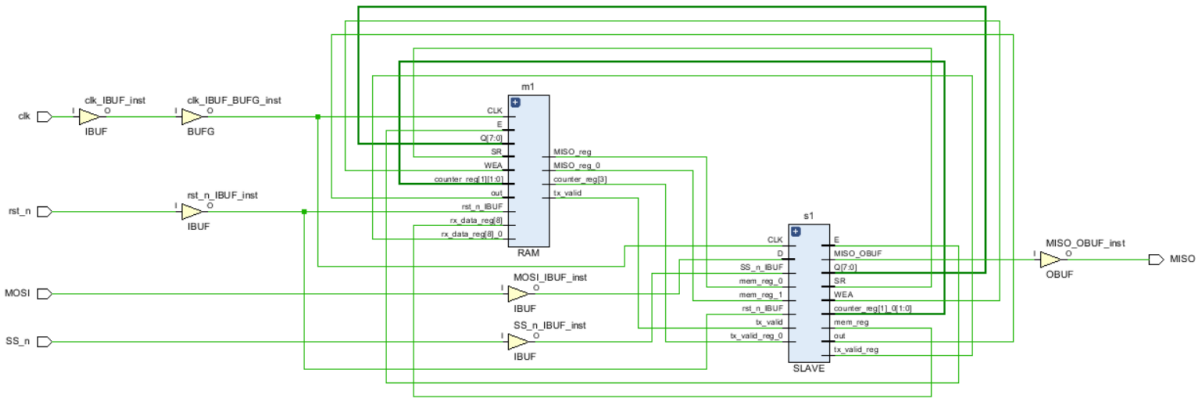

Wave - Default
Msgs
MOSI
SS_n
clk
rst_n
MISO
din
rx_valid
dout
tx_valid
mem
stored_command
tx_data
rx_data
counter
allow_memorize
ns
cs

Gray encoding

1.Elaboration schematic:



2.Synthesis schematic:



3.Encoding used:

State	New Encoding	Previous Encoding
IDLE	000	000
CHK_CMD	001	001
WRITE	011	010
READ_ADD	010	011
READ_DATA	111	100

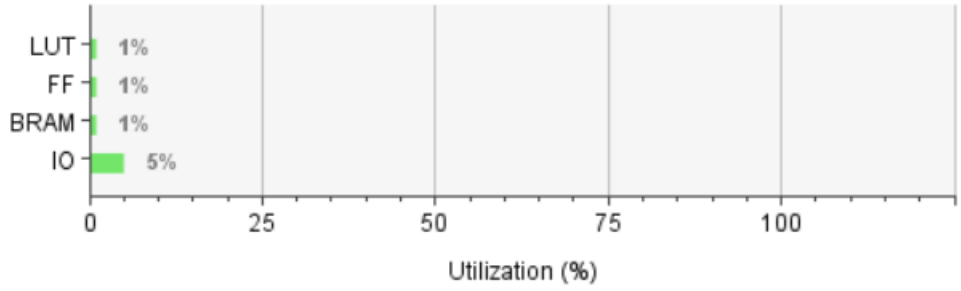
4.Synthesis timing report:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.898 ns	Worst Hold Slack (WHS):	0.149 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:	75	Total Number of Endpoints:	75	Total Number of Endpoints:	32

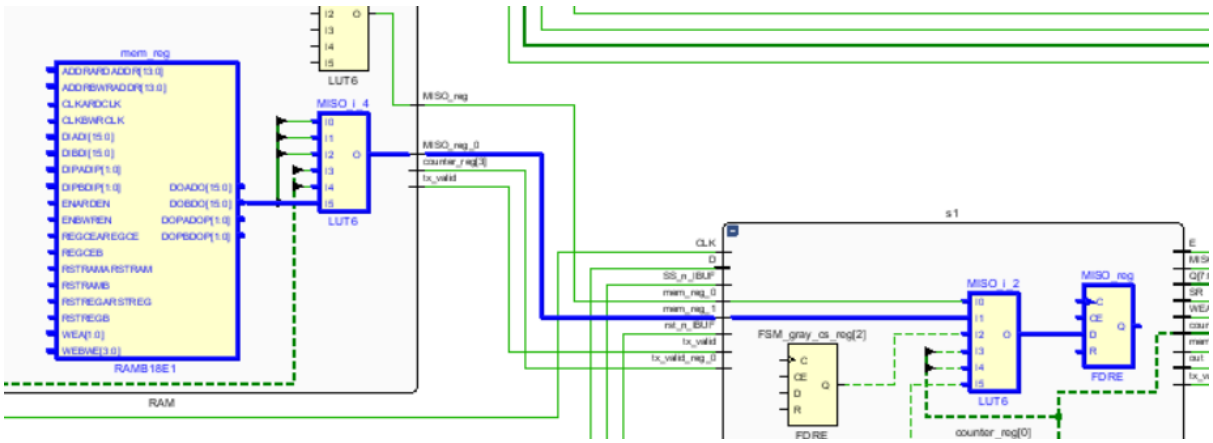
All user specified timing constraints are met.

5.Synthesis utilization report:

Resource	Utilization	Available	Utilization %
LUT	25	20800	0.12
FF	32	41600	0.08
BRAM	0.50	50	1.00
IO	5	106	4.72



6.Critical path:



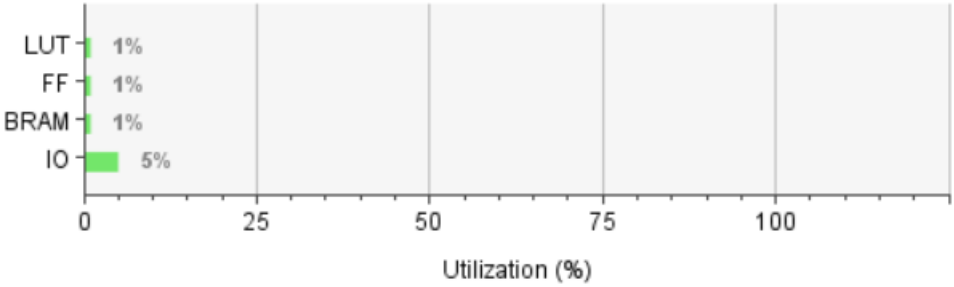
7.Implementation timing report:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.619 ns	Worst Hold Slack (WHS):	0.067 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:	76	Total Number of Endpoints:	76	Total Number of Endpoints:	32

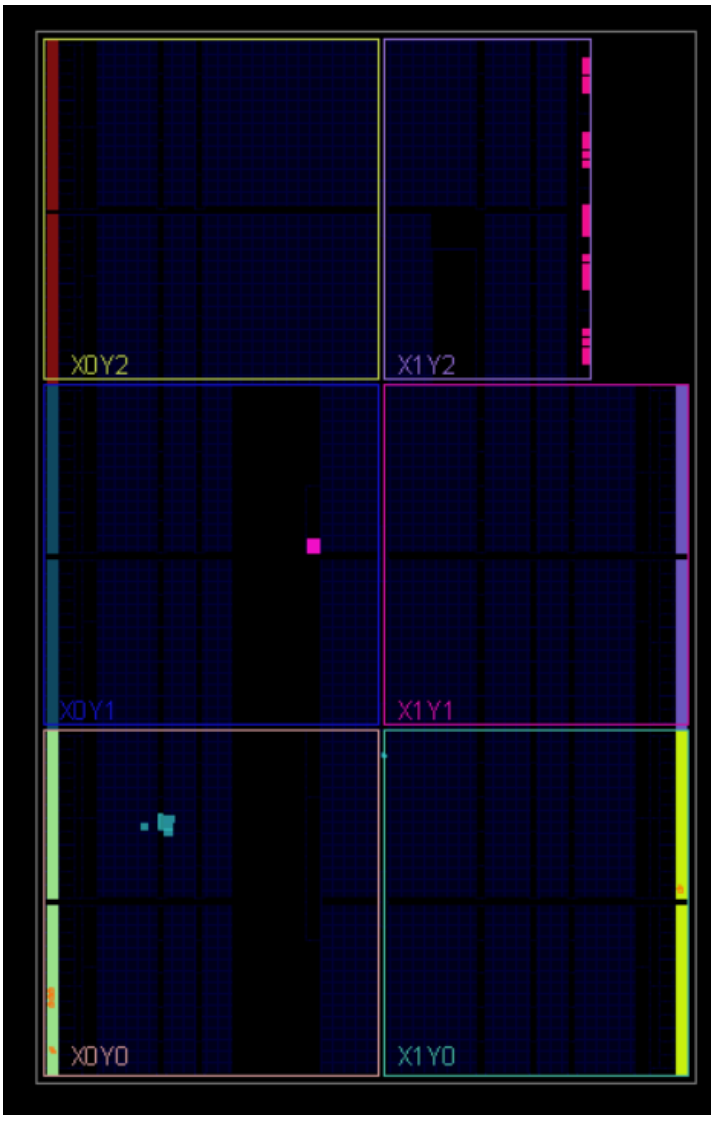
All user specified timing constraints are met.

8.Implementation utilization report:

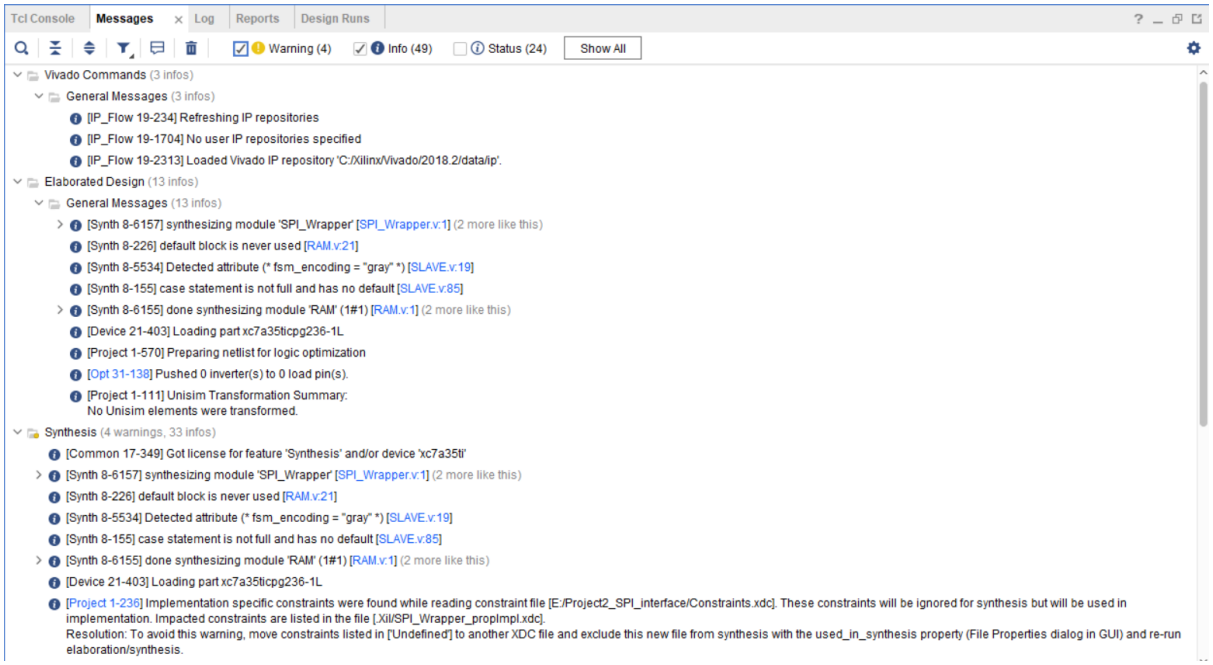
Resource	Utilization	Available	Utilization %
LUT	26	20800	0.13
FF	32	41600	0.08
BRAM	0.50	50	1.00
IO	5	106	4.72



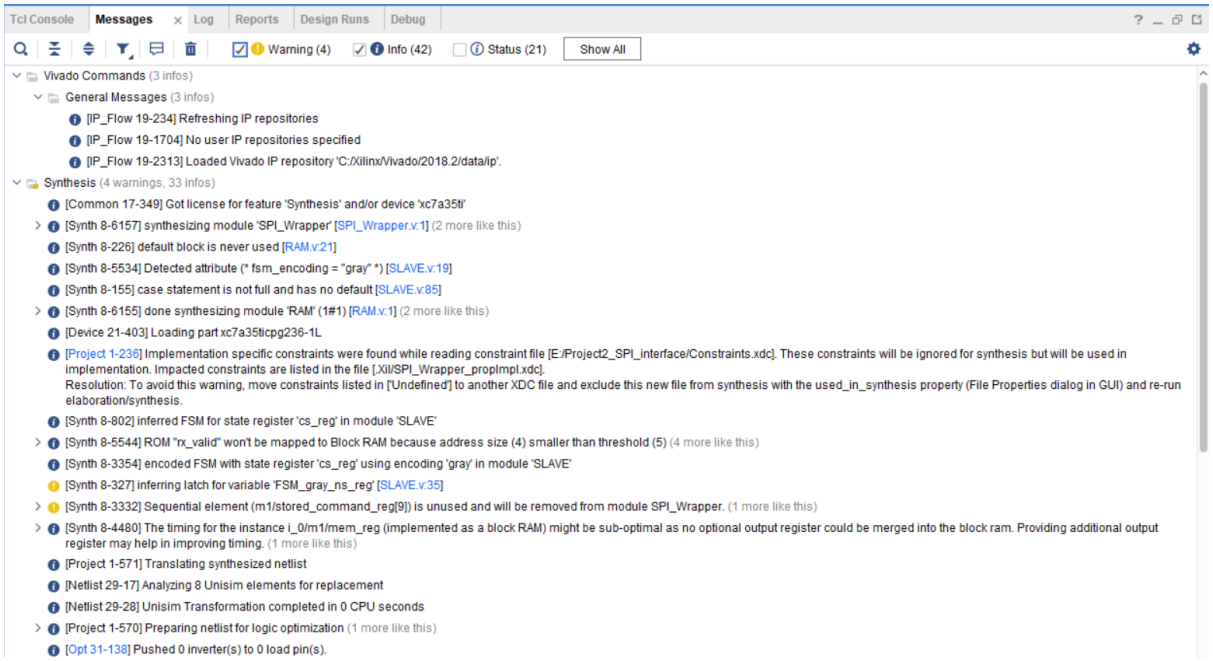
9.Implementation device:



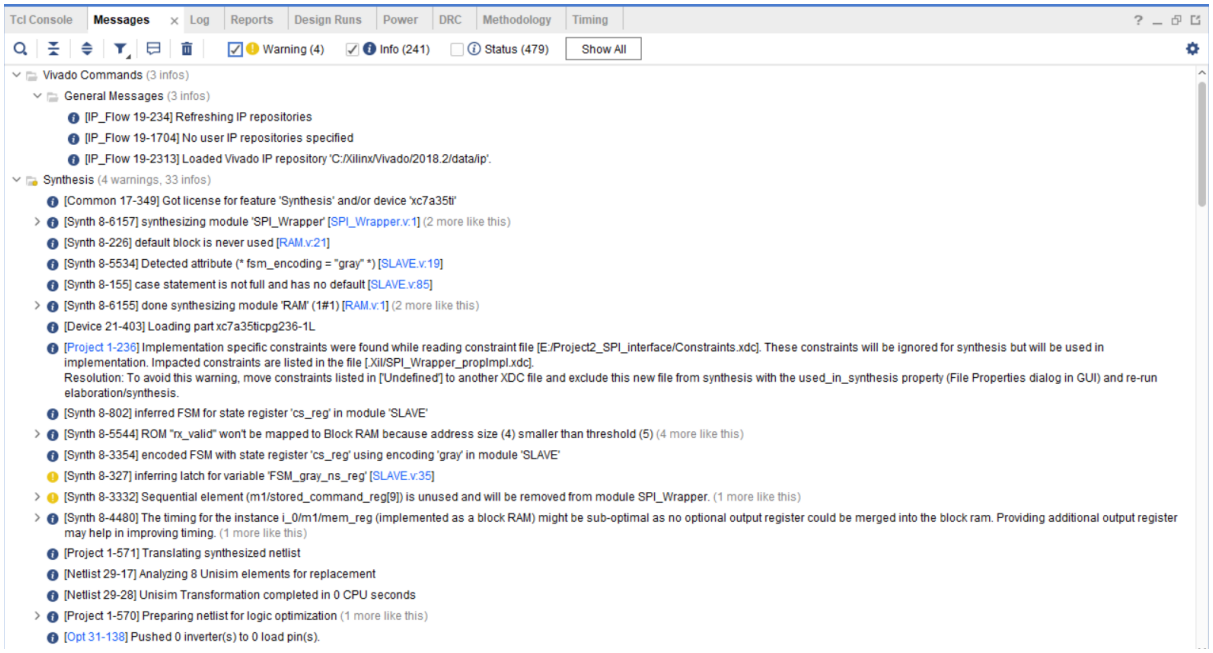
10.Elaboration messages:



11.Synthesis messages:

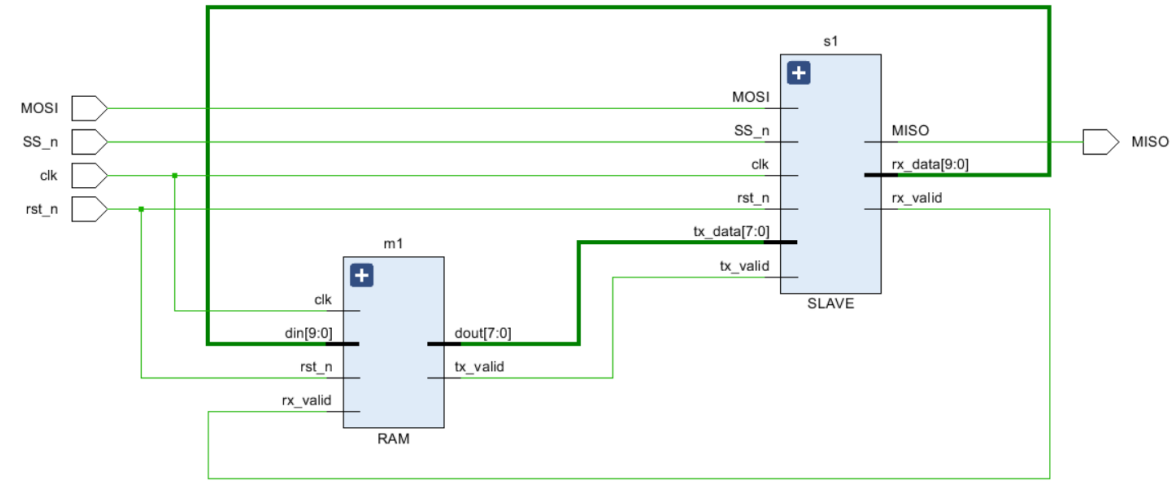


12.Implementation messages:

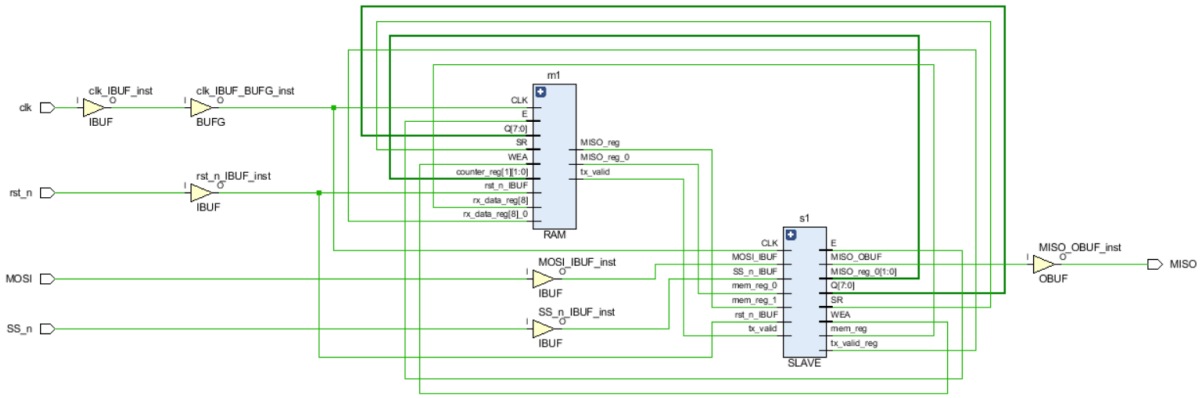


Sequential encoding

1.Elaboration schematic:



2.Synthesis schematic:



3.Encoding used:

State	New Encoding	Previous Encoding
IDLE	000	000
CHK_CMD	001	001
WRITE	010	010
READ_ADD	011	011
READ_DATA	100	100

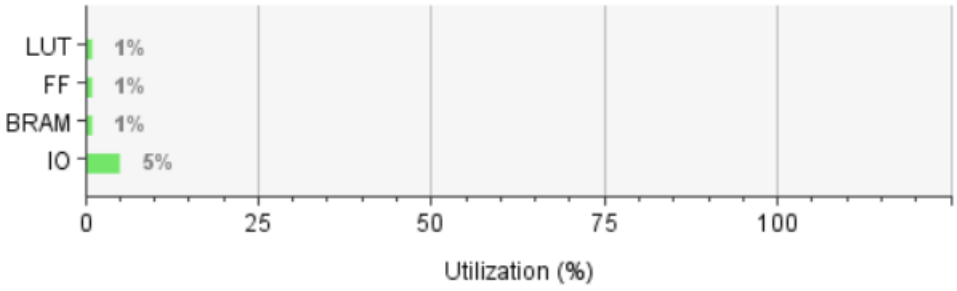
4.Synthesis timing report:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.898 ns	Worst Hold Slack (WHS):	0.151 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:	75	Total Number of Endpoints:	75	Total Number of Endpoints:	32

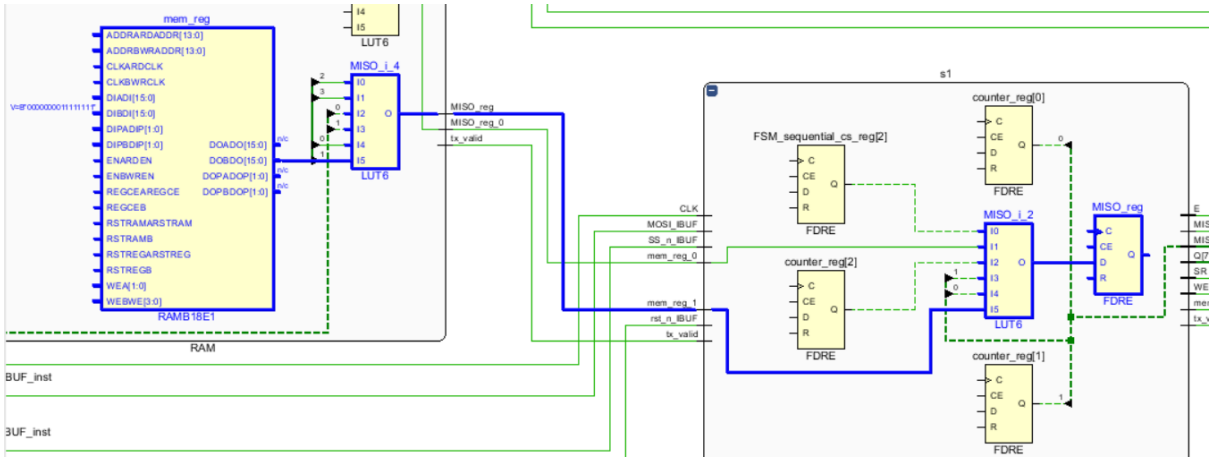
All user specified timing constraints are met.

5.Synthesis utilization report:

Resource	Utilization	Available	Utilization %
LUT	25	20800	0.12
FF	32	41600	0.08
BRAM	0.50	50	1.00
IO	5	106	4.72



6.Critical path:



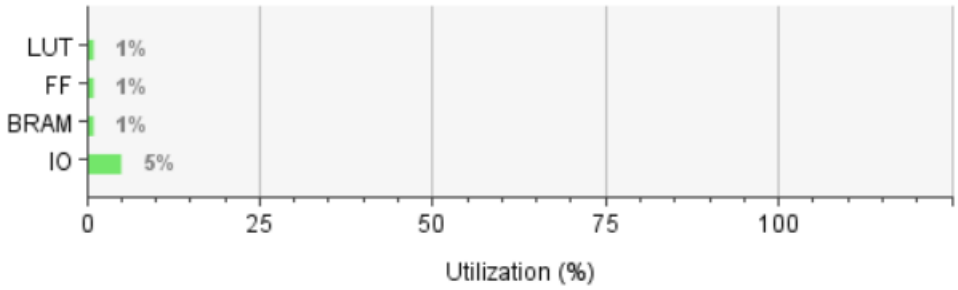
7.Implementation timing report:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.554 ns	Worst Hold Slack (WHS):	0.049 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:	76	Total Number of Endpoints:	76	Total Number of Endpoints:	32

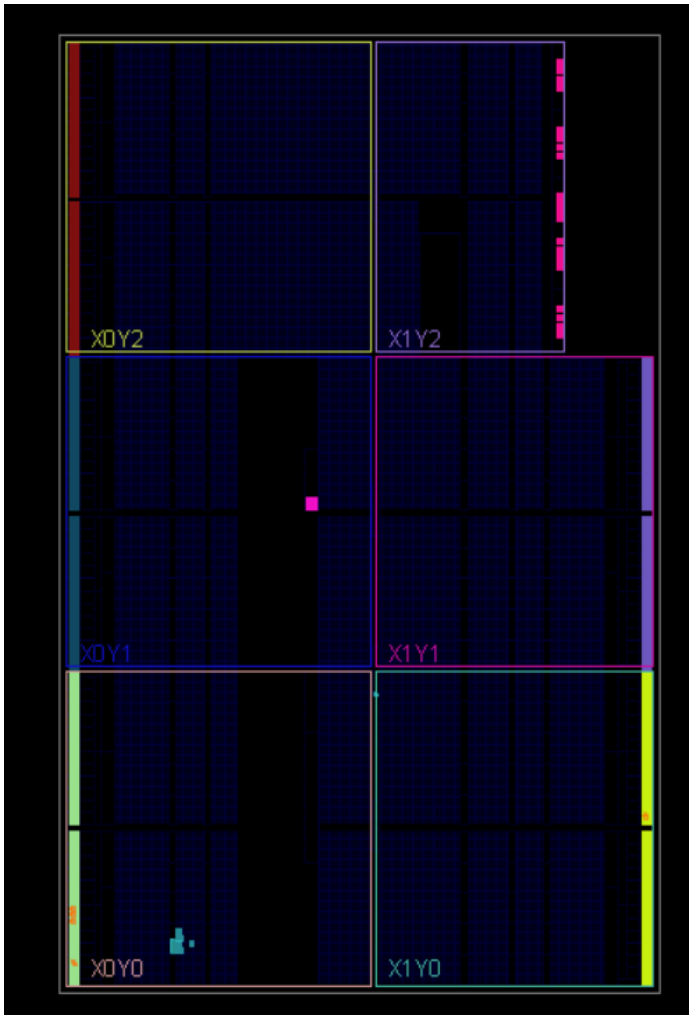
All user specified timing constraints are met.

8.Implementation utilization report:

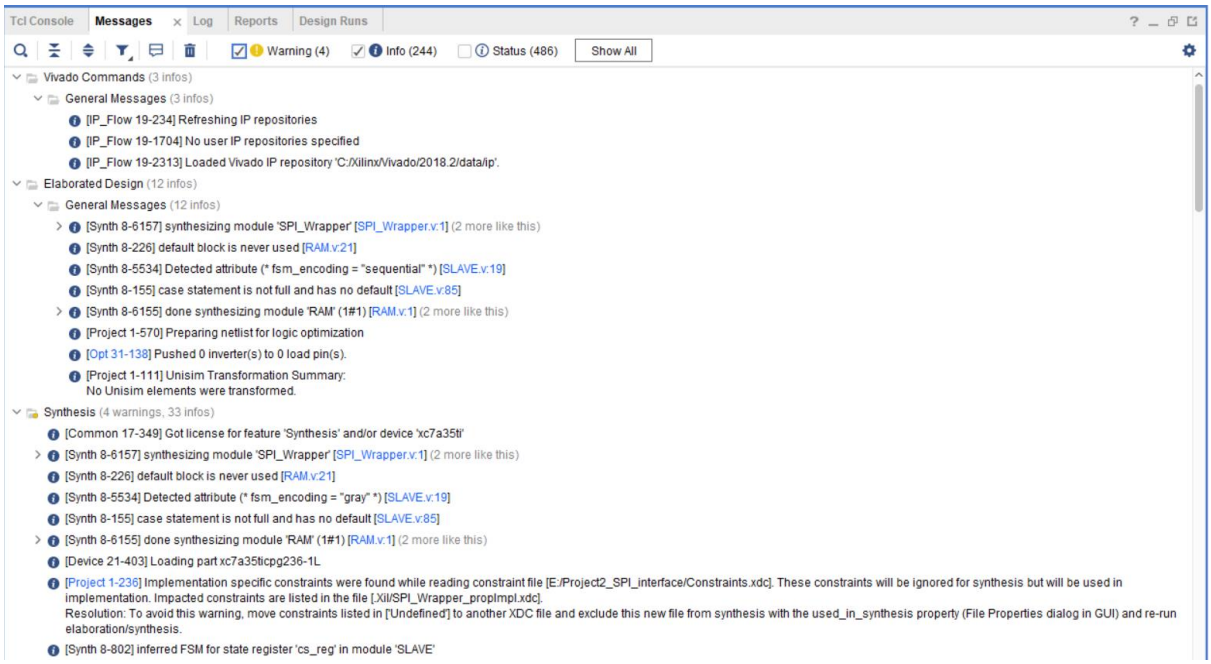
Resource	Utilization	Available	Utilization %
LUT	26	20800	0.13
FF	32	41600	0.08
BRAM	0.50	50	1.00
IO	5	106	4.72



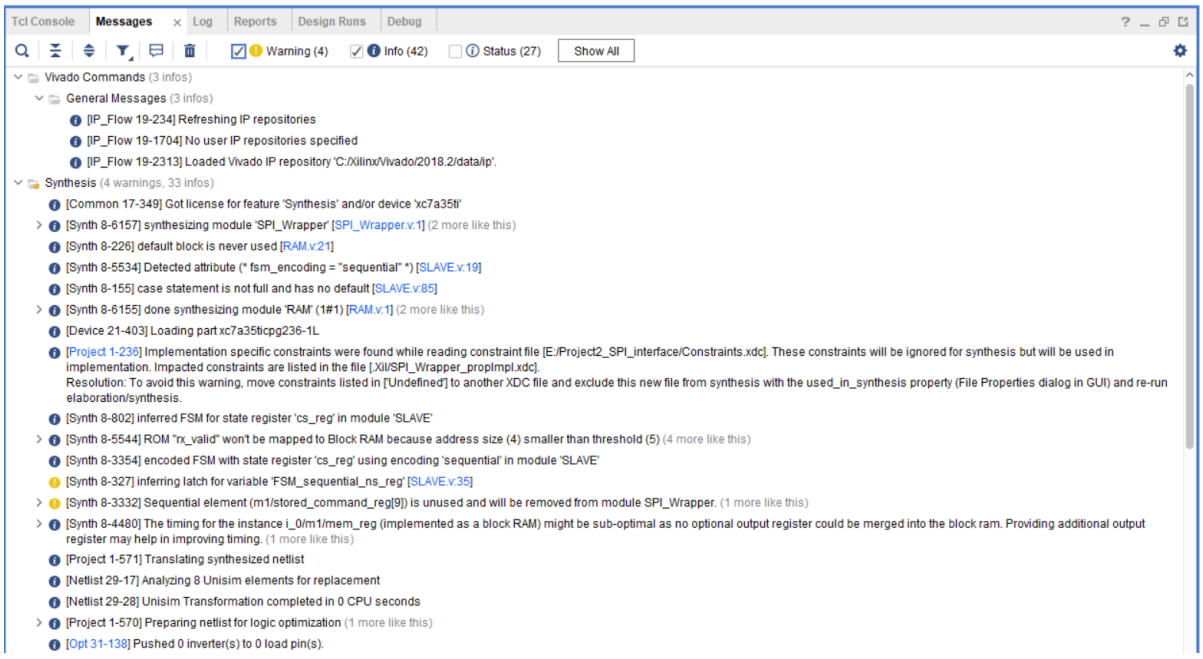
9.Implementation device:



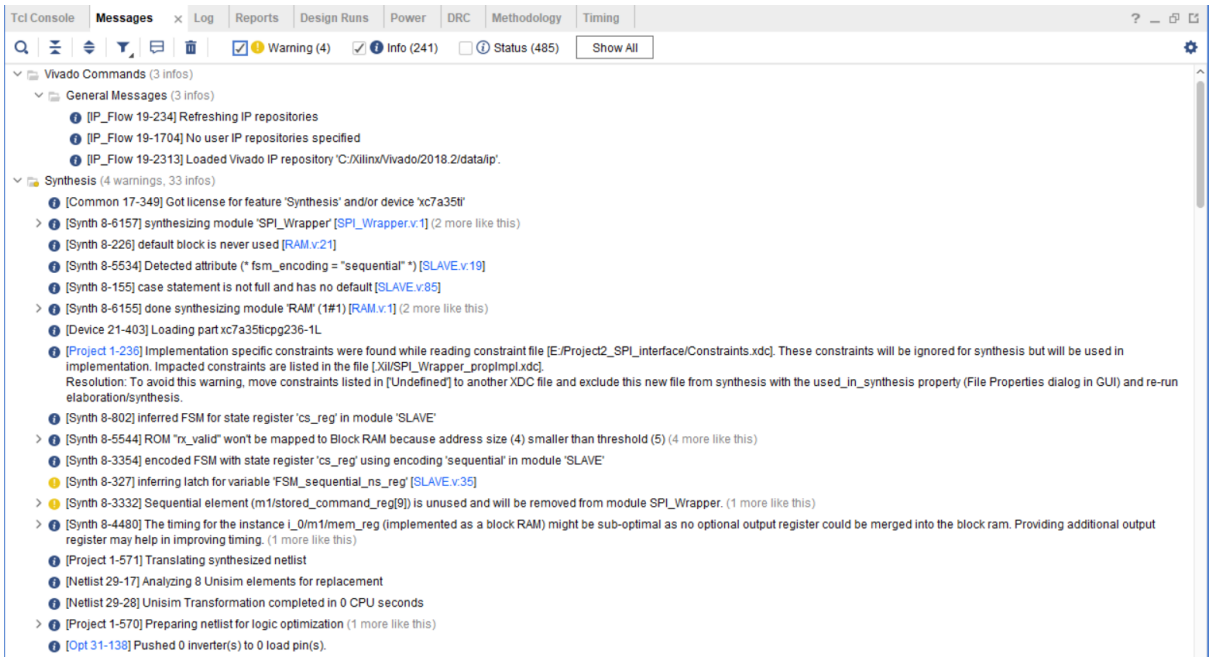
10.Elaboration messages:



11.Synthesis messages:

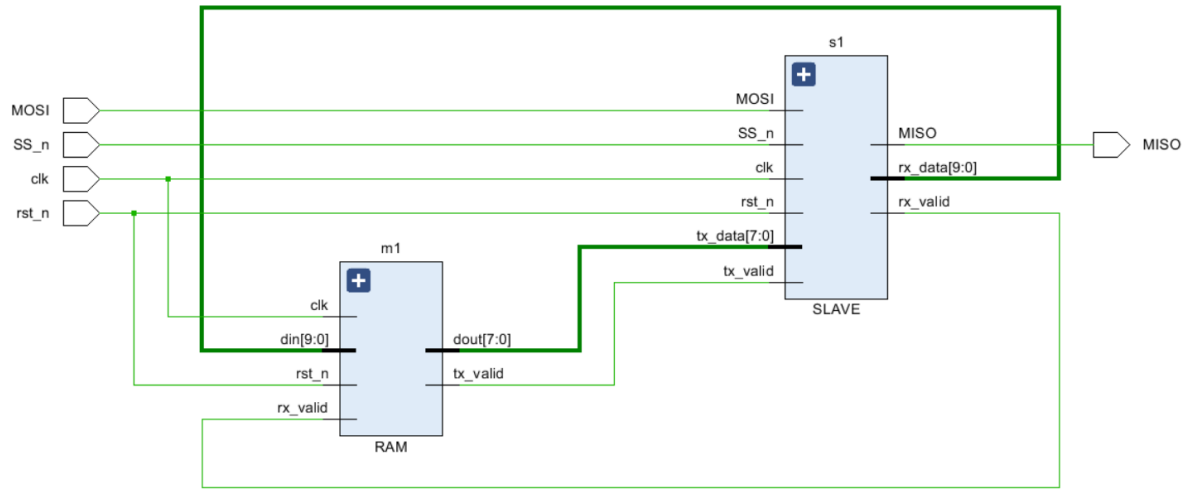


12.Implementation messages:

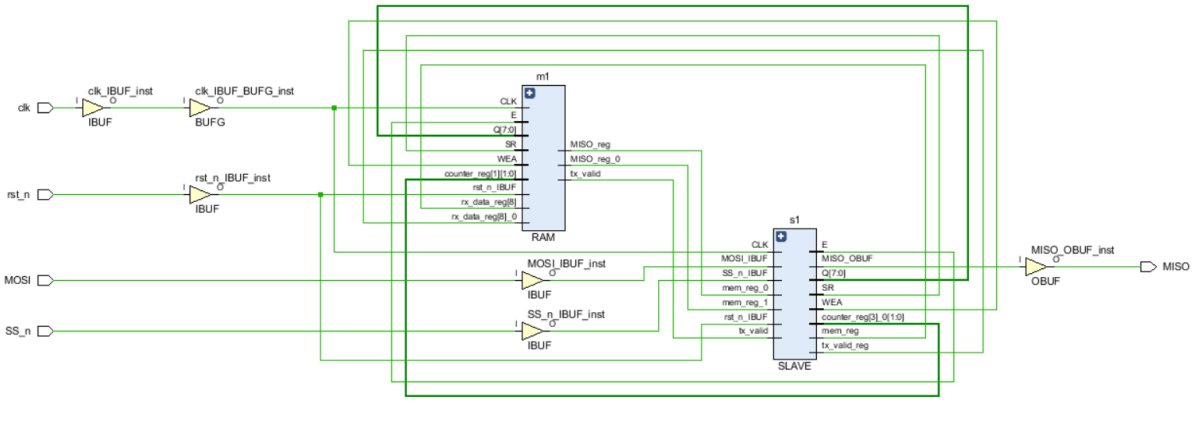


One hot encoding

1.Elaboration schematic:



2.Synthesis schematic:



3.Encoding used:

State	New Encoding	Previous Encoding
IDLE	00001	000
CHK_CMD	00010	001
WRITE	00100	010
READ_ADD	01000	011
READ_DATA	10000	100

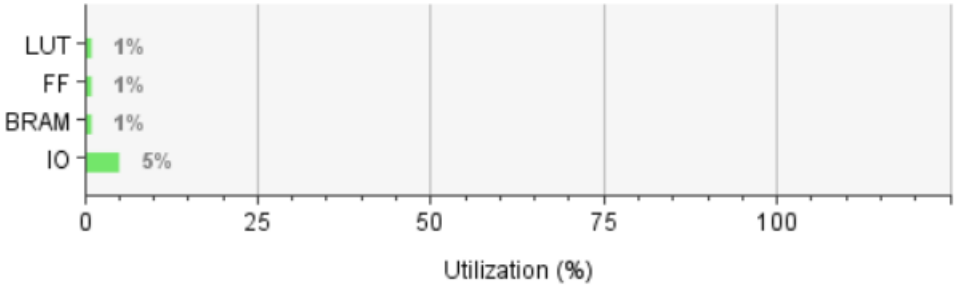
4.Synthesis timing report:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.898 ns	Worst Hold Slack (WHS):	0.148 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:	75	Total Number of Endpoints:	75	Total Number of Endpoints:	34

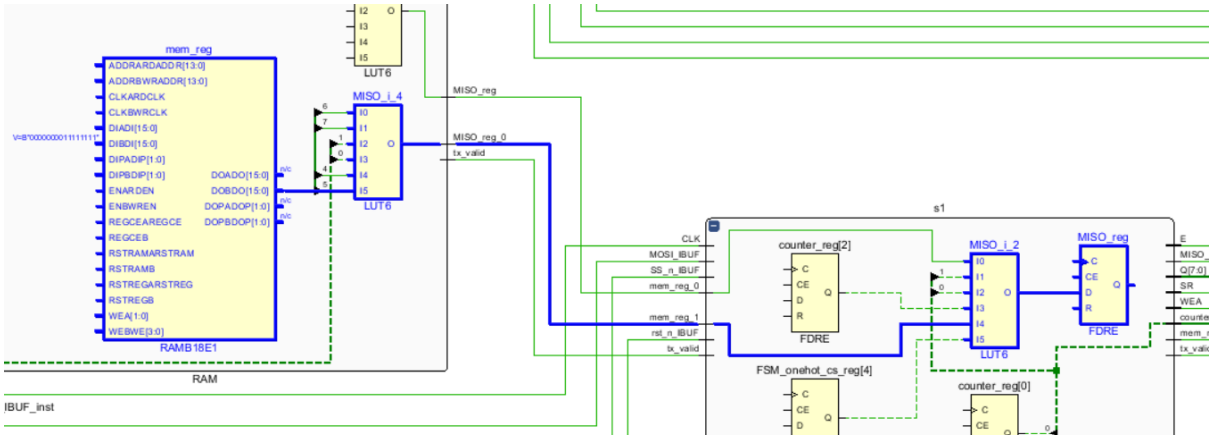
All user specified timing constraints are met.

5.Synthesis utilization report:

Resource	Utilization	Available	Utilization %
LUT	28	20800	0.13
FF	36	41600	0.09
BRAM	0.50	50	1.00
IO	5	106	4.72



6.Critical path:



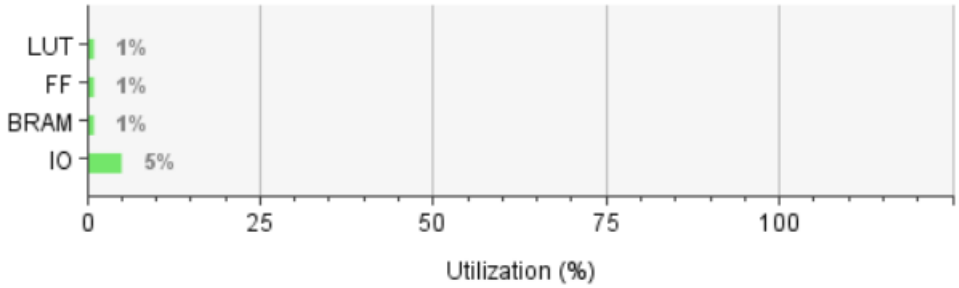
7.Implementation timing report:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.589 ns	Worst Hold Slack (WHS):	0.048 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:	76	Total Number of Endpoints:	76	Total Number of Endpoints:	34

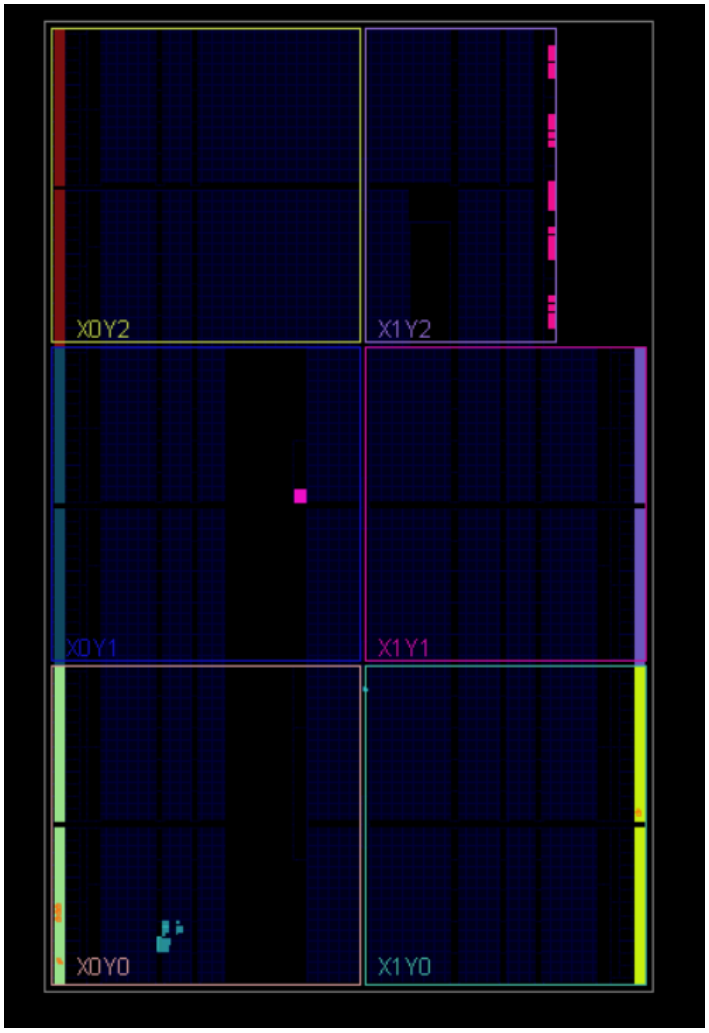
All user specified timing constraints are met.

8.Implementation utilization report:

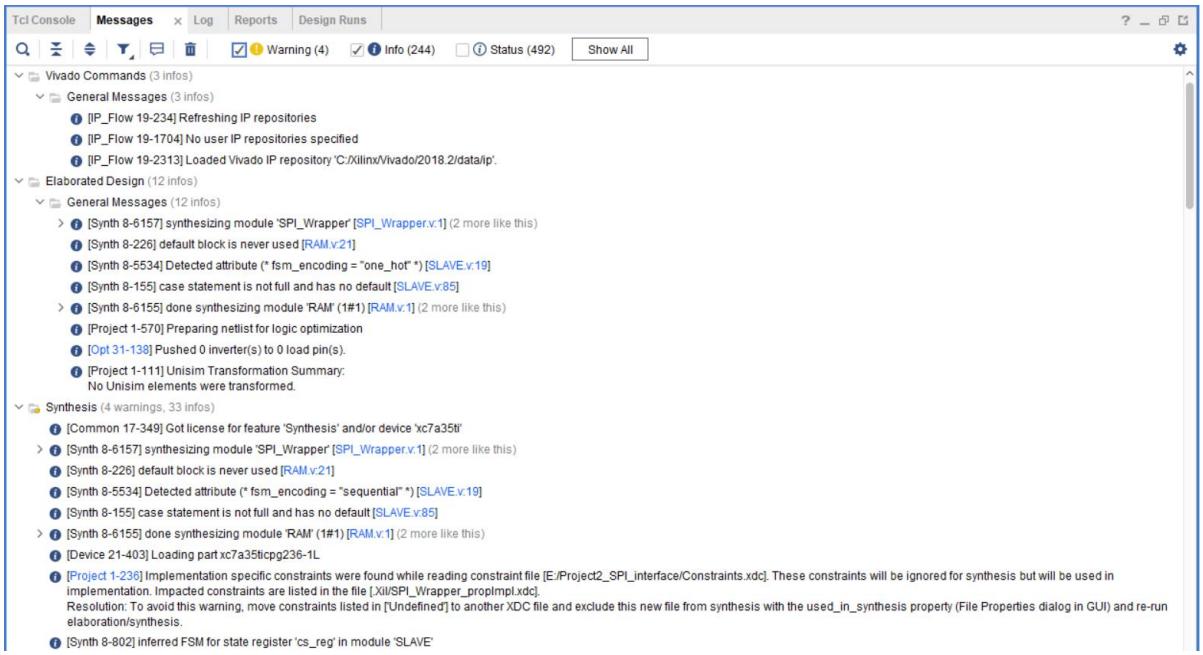
Resource	Utilization	Available	Utilization %
LUT	28	20800	0.13
FF	36	41600	0.09
BRAM	0.50	50	1.00
IO	5	106	4.72



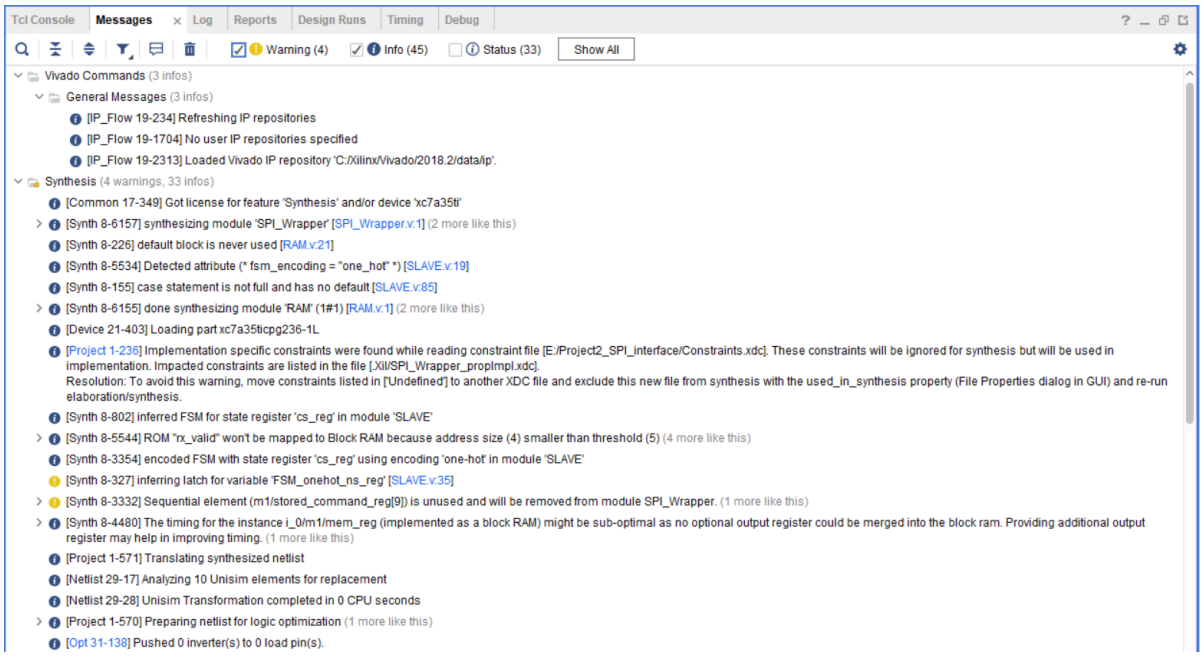
9.Implementation device:



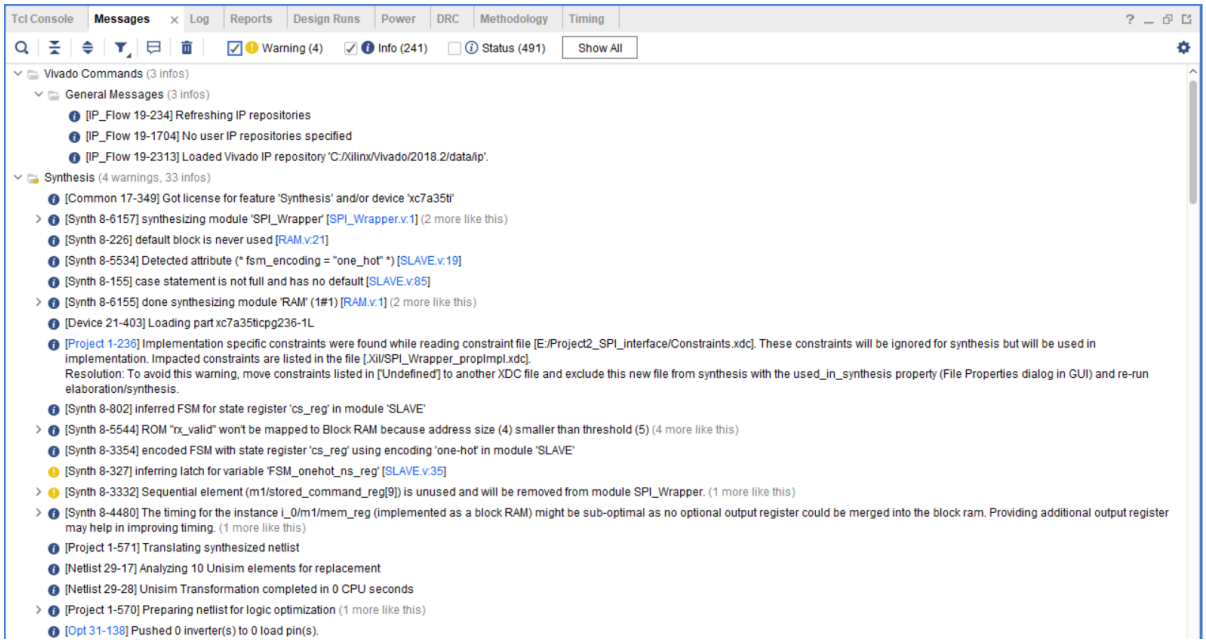
10.Elaboration messages:



11.Synthesis messages:



12.Implementation messages:



We wish to operate at the highest frequency possible based on the best timing report that gives the high setup/hold slack after implementation is gray encoding, so we adde debug cores to fsm gray encoding device and generate bitstream.

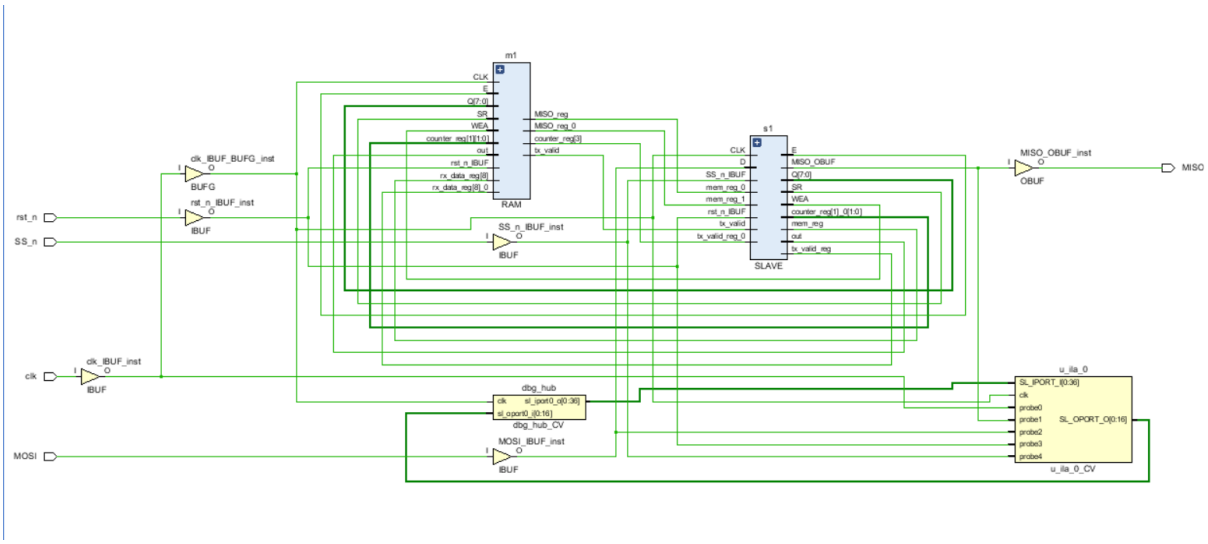
Netlist generation

project_sp_netlisti.v

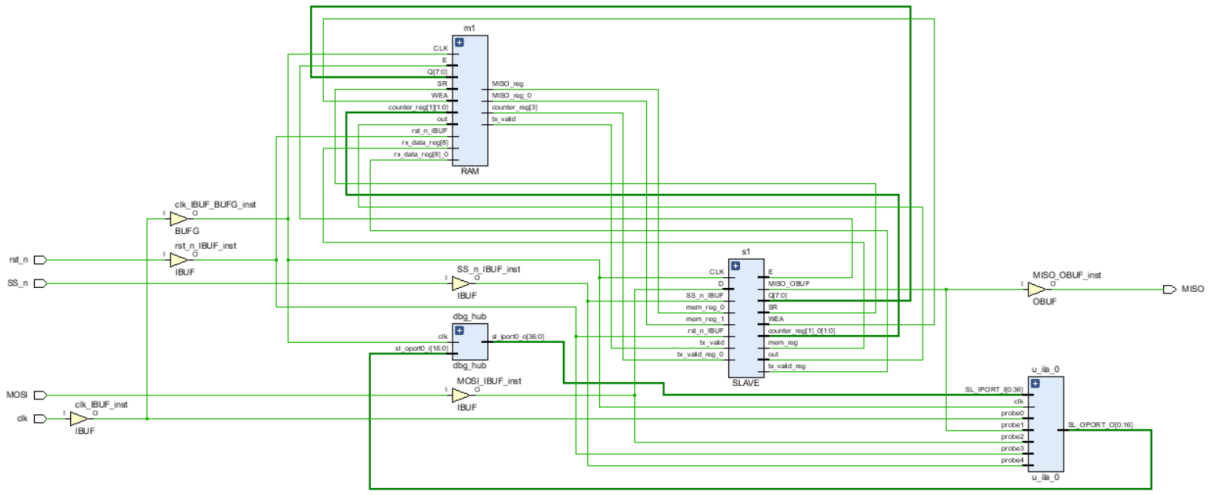
```
1  // Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
2  // ----------------------------------------------------------------------------
3  // Tool Version: Vivado v.2018.2 (win64) Build 2258646 Thu Jun 14 20:03:12 MDT 2018
4  // Date        : Sat Aug  3 16:54:19 2024
5  // Host        : DESKTOP-FOVHHBI running 64-bit major release  (build 9200)
6  // Command     : write_verilog E:/Project2_SPI_interface/project_sp_netlisti.v
7  // Design      : SPI_Wrapper
8  // Purpose     : This is a Verilog netlist of the current design or from a specific cell of the design. The output is an
9  //               IEEE 1364-2001 compliant Verilog HDL file that contains netlist information obtained from the input
10 //               design files.
11 // Device      : xc7a35ticpg236-1L
12 // ----------------------------------------------------------------------------
13 `timescale 1 ps / 1 ps
14
15 module blk_mem_gen_generic_cstr
16    (clk,
17     out,
18     s_dclk_o,
19     D,
20     DIADI,
21     Q,
22     \i_intcap.CAP_ADDR_O_reg[9] ,
23     \multiple_read_latency.read_enable_out_reg[3] );
24   input clk;
25   input out;
26   input s_dclk_o;
27   output [5:0]D;
28   input [5:0]DIADI;
29   input [9:0]Q;
30   input [9:0]\i_intcap.CAP_ADDR_O_reg[9] ;
31   input [0:0]\multiple_read_latency.read_enable_out_reg[3] ;
32
33   wire [5:0]D;
34   wire [5:0]DIADI;
35   wire [9:0]Q;
36   wire clk;
37   wire [9:0]\i_intcap.CAP_ADDR_O_reg[9] ;
38   wire [0:0]\multiple_read_latency.read_enable_out_reg[3] ;
```

Bitstream generation

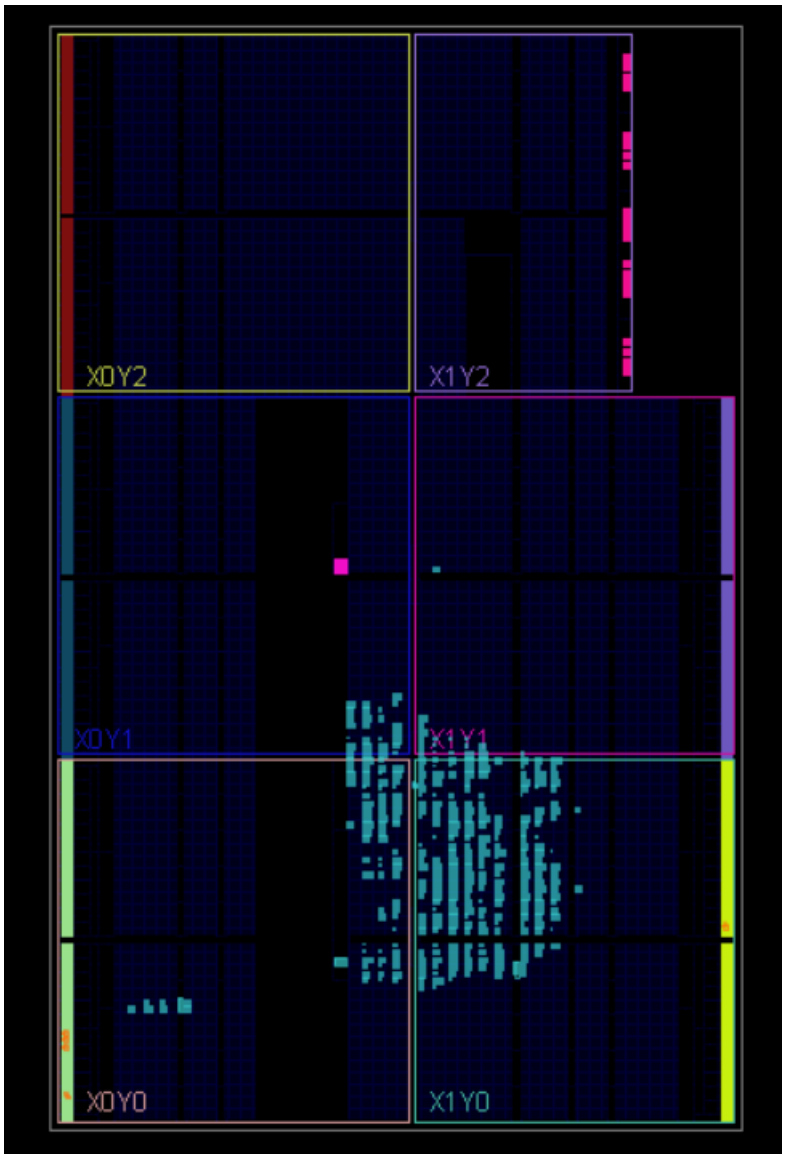
1.Synthesis schematic:



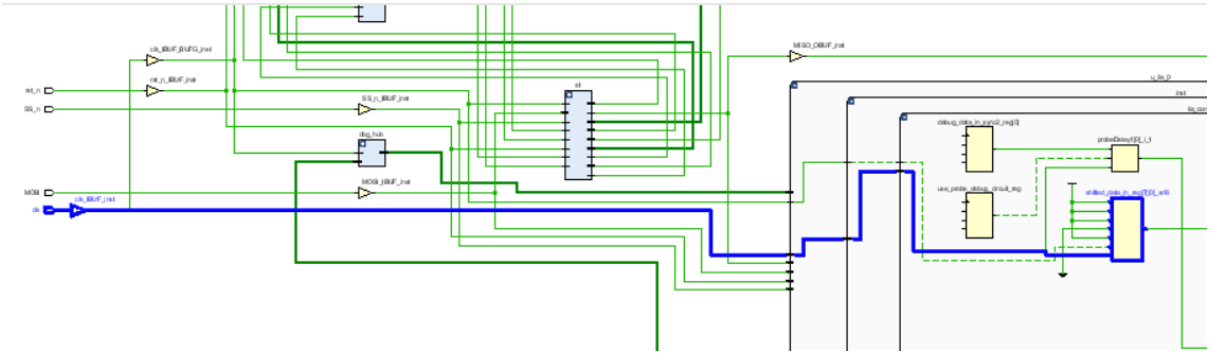
2.Implementation schematic:



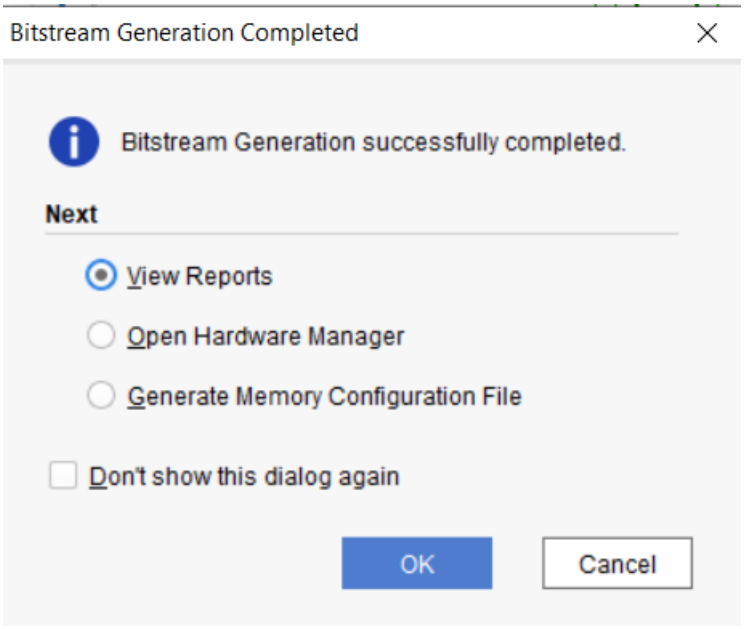
3.Device:



4.Critical path:



5.Message showing successful bitstream generation:



[Github Link]