SPI Slave with Single Port RAM

FPGA Flow

Created By : Yousef Gamal

Table of Contents

1. SPI overview

1.1. Description

SPI (Serial Peripheral Interface) is a synchronous serial communication protocol **used primarily for short-distance communication**, typically between microcontrollers and peripherals such as sensors and SD cards. It was developed by Motorola in the 1980s and is widely used in embedded systems due to its simplicity and high speed.

SPI is implemented in different ways, but the same basic functionality holds for each implementation.

1.2. How SPI Works

SPI is a **synchronous three-wire serial communications interface** based on a master/slave relationship. The master and slave both **contain serial shift registers** that are **connected to form a circular shift buffer**. The master supplies the clock which is used to shift data out of the master and into the slave and simultaneously out of the slave and into the master.

1.3. Key Features of SPI

- **Synchronous Communication**

SPI is a synchronous protocol, meaning data is transferred with respect to a clock signal.

- **Full-Duplex Communication**

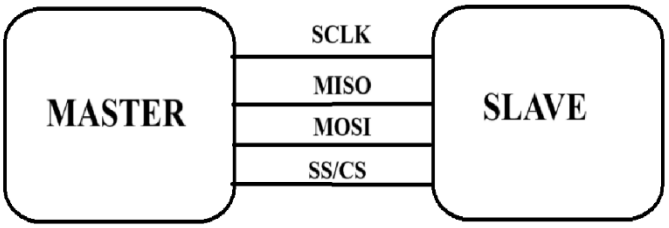
Data can be transmitted and received simultaneously using separate data lines.

- **Master-Slave Architecture**

One device acts as the master, which controls the clock and initiates communication. The other devices are slaves, which respond to the master's commands.

- **Four Primary Signals:**

➢ **SCLK** (Serial Clock): Generated by the master to synchronize data transmission.
➢ **MOSI** (Master Out Slave In): Data line for data sent from the master to the slave.
➢ **MISO** (Master In Slave Out): Data line for data sent from the slave to the master.
➢ **SS/CS** (Slave Select/Chip Select): **Active low** signal used by the master to select the specific slave device for communication.



1.4. Advantages of SPI

- **High Speed:**

SPI can operate at much higher speeds compared to other protocols like I2C because it is less complex operates in full duplex.

- **Simple Hardware Interface**:

Only four wires are needed to connect the master and slave devices, and additional slaves can be added using more SS/CS lines.

- **No Addressing Overhead:**

Unlike I2C, SPI does not require device addressing, which reduces overhead and increases speed.

- **Flexibility:**

SPI can be used in various applications, from simple data acquisition systems to complex multi-device setups.

1.5. Disadvantages of SPI

- **No Standard Acknowledgment Mechanism:**

SPI does not have an acknowledgment mechanism to confirm receipt of data, which can be a disadvantage in some applications.

- **More Pins Required:**

For each additional slave device, a separate SS/CS line is required, increasing the number of pins needed on the master device.

- **Short Distance Communication**

SPI is typically used for short-distance communication due to its lack of inherent error-checking and noise immunity.

1.6. Typical Use Cases

- **Microcontroller to Peripheral Communication**:

  - Connecting microcontrollers to sensors, displays, memory devices, and other peripherals.

- **Data Acquisition Systems**:

  - Reading data from analog-to-digital converters (ADCs) or writing data to digital-to-analog converters (DACs).

- **Interfacing with Memory**:

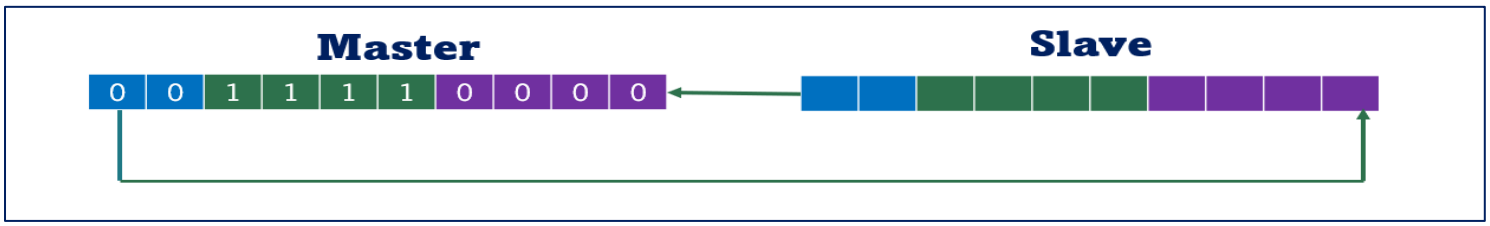
  - Accessing data from EEPROMs, flash memory, and SD cards.

1.7. Our SPI configurations

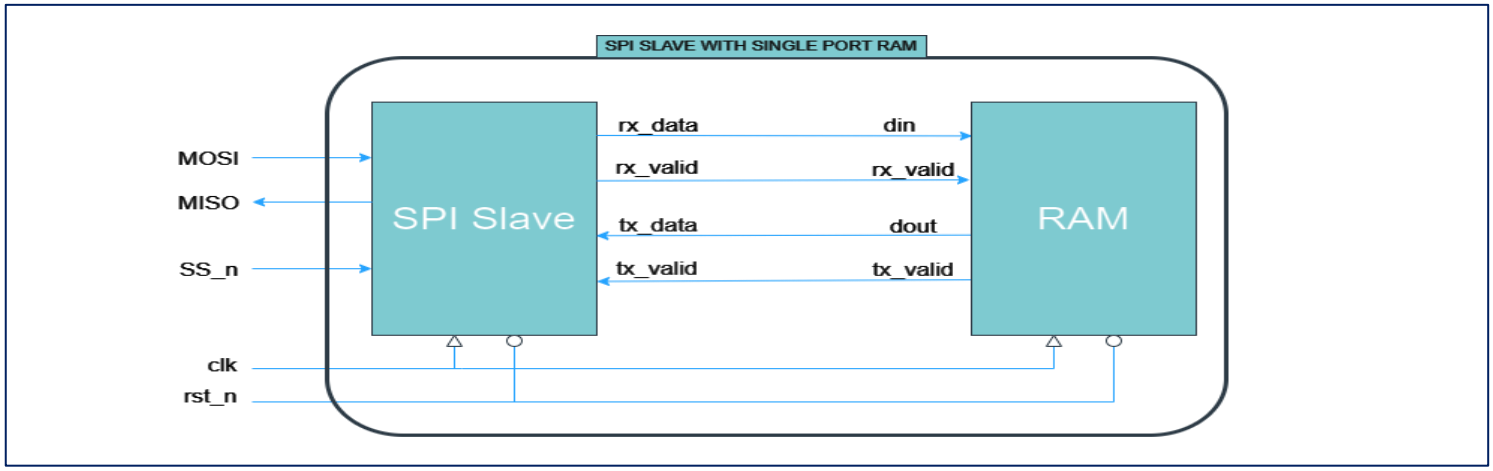
The master initiates data transmission beginning with the most significant bit (MSB).

To maintain correct data ordering, the slave receives the least significant bit (LSB) first and subsequently shifts the data with each incoming bit.

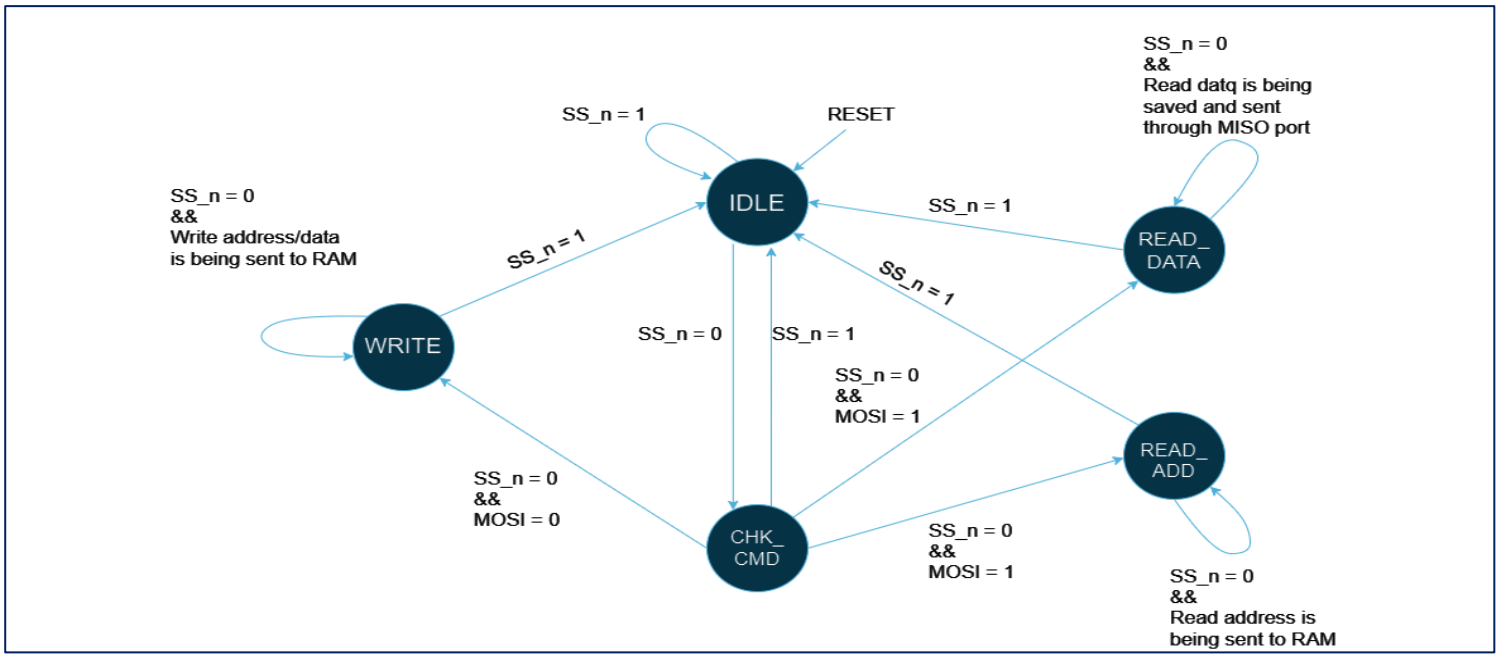
Similarly, the slave transmits its data starting with the MSB, and the master, to ensure proper data alignment, receives it starting with the LSB, shifting the data with each successive bit.



1.8. SPI wrapper



1.9. SPI FSM



2. RTL Code

2.1. Single Port Synchronous RAM

```
module SPI_Single_Port_RAM
(
    input clk , rst_n ,
    input [9:0] din ,
    input rx_valid ,
    output reg [7:0] dout ,
    output reg tx_valid
);

localparam MEM_DEPTH = 256 ;
localparam MEM_WIDTH = 8   ;
reg [MEM_WIDTH-1 : 0] RAM [MEM_DEPTH-1 : 0] ;
reg [7:0] temp_address ;
integer i ;
always @(posedge clk)
begin
    if(!rst_n)
        begin
            dout <= 8'h00;
            tx_valid <= 1'b0;
            temp_address <= 8'h00;
            for(i=0 ; i<MEM_DEPTH ; i=i+1)
               begin
                    RAM[i] <= { MEM_WIDTH{1'b0} } ;
               end

        end
    else
        begin
            if (rx_valid)
            begin
                case (din[9:8])
                    2'b00:
                        temp_address <= din[7:0];

                    2'b01:
                        RAM[temp_address] <= din[7:0];

                    2'b10:
                        temp_address <= din[7:0];

                    2'b11:
                        dout <= RAM[temp_address];
                endcase
            end
            tx_valid <= (din[9:8] == 2'b11) ? 1'b1 : 1'b0;
        end
end
endmodule
```

2.2. SPI Slave

```
module SPI_Slave
(
    input clk , rst_n  ,        // active low asynchronous reset
    input MOSI ,                // Master out slave in
    input SS_n ,               //Active low signal used by the master
                              // to select the specific slave device for communication.
    input tx_valid ,          // control for inpit data
    input [7 : 0] tx_data , // input data for SPI slave

    output reg MISO ,              // Data output
    output reg rx_valid ,         // control for output data
    output reg [9 : 0] rx_data  // output data from SPI slave
);


 reg [2:0] CS , NS ;
 reg [3:0] counter_1 ;
// counts from zero to 9 based on #of received bits (from series to parallel)
 reg [2:0] counter_2 ;
// counts from zero to 7 based on #of transmitted bits (from parallel to series)
 reg Is_the_address_sent = 1'b0; // to go to read address first

// storage data from serial to parallel
 reg [9 : 0] SPI_Slave_register ;   //{   2 selectio bits  ,   8 bit  Data    }
                                   // {       First        ,      second      }

 localparam IDLE      = 3'b000  ;
 localparam CHK_CMD   = 3'b001  ;
 localparam WRITE     = 3'b010  ;
 localparam READ_ADD  = 3'b011  ;
 localparam READ_DATA = 3'b100  ;

/* Gray Encoding for FSM */

(* fsm_encoding = "Gray"  *);

//state memory
always @(posedge clk or negedge rst_n)
begin
        if( ! rst_n )

                CS <= IDLE ;
        else
                CS <= NS ;

end
```

```
// next state logic
    always @(*)
    begin
        case (CS)
            IDLE:
                begin
                    if( SS_n || !rst_n )
                        NS = IDLE ;
                    else
                        NS = CHK_CMD ;
                end

            CHK_CMD:
                    begin
                        if(SS_n)
                            NS = IDLE;

                        else if( (SS_n == 0) && (MOSI == 0) )
                            NS = WRITE; // still in write to receive the Data

                        else if((SS_n == 0) && (MOSI == 1))
                            begin
                                if(Is_the_address_sent)
                                    NS = READ_DATA;
                            // read the data of address written before

                                else
                                    NS = READ_ADD;
                            end
                        else
                            NS = CHK_CMD;
                    // terminate with else to avoid the inferring latch
                    end

            WRITE:
                begin
                    if(SS_n)
                        NS = IDLE;
                    else
                        NS = WRITE;
                   // stay in write to receive the data then write the data in RAM
                    end

            READ_ADD:
                begin
                    if(SS_n)
                        NS = IDLE;
                    else
                        NS = READ_ADD;
                   // stay in read to receive the address then go to memory
                    End
```

```
            READ_DATA:
                begin
                    if(SS_n)
                        NS = IDLE;
                    else
                        NS = READ_DATA;
            // stay in read to receive the Data then go to memory
                end

            default: NS = IDLE; // to avoid inferring latch

        endcase
    end


// output logic  sensitive to clock as the MOSI is sent bit by bit with Clock
always @(posedge clk)
begin
        if(rst_n) // Output appears when the reset is de-asserted.
        begin
            case (CS)
                IDLE: /*zero outputs*/
                    begin
                        MISO <= 0;
                        counter_1 <= 4'b0000 ;
                        counter_2 <= 3'b000 ;
                        rx_valid  <= 1'b0 ;
                        rx_data   <= 10'b0 ;
                    end

                CHK_CMD:
                        begin
                            MISO <= 0;
                            rx_data <= 0;
                            rx_valid <= 0;
                        end

                WRITE:
                    begin
                            if (counter_1 < 10)
                                begin
                                SPI_Slave_register <= (SPI_Slave_register<<1) | MOSI  ;
                                  // convert from series to parallel
                                    counter_1 <= counter_1 + 1;
                                    rx_valid  <= 1'b0;
                                end
                            else
                                begin
                                    counter_1 <= 4'h0;
                                    rx_data  <= SPI_Slave_register;
                                    rx_valid <= 1'b1;
                                    SPI_Slave_register <= 10'd0;
                                end
                    end
```

```
            READ_ADD:
                    begin

                          Is_the_address_sent <= 1'b1 ;
                         // to make the upcoming state is read data

                        if (counter_1 < 10)
                            begin
                            SPI_Slave_register <= (SPI_Slave_register<<1) | MOSI  ;
                                // convert from series to parallel
                                counter_1 <= counter_1 + 1;
                                rx_valid  <= 1'b0;
                            end
                        else
                            begin
                                counter_1 <= 4'h0;
                                rx_data  <= SPI_Slave_register;
                                rx_valid <= 1'b1;
                                SPI_Slave_register <= 10'd0;
                            end
                    end

                READ_DATA:
                        begin
                            Is_the_address_sent <= 1'b0;
                         // to make the upcoming state is to read address

                            // Receiving data logic
                            if (counter_1 < 10)
                                begin
                                 SPI_Slave_register <= (SPI_Slave_register << 1) | MOSI;
                                        // convert from series to parallel
                                        counter_1 <= counter_1 + 1;
                                        rx_valid  <= 1'b0;
                                end
                            else
                                begin
                                    counter_1 <= 4'h0;
                                    rx_data  <= SPI_Slave_register;
                                    rx_valid <= 1'b1;
                                    SPI_Slave_register <= 10'd0;
                                end

                            // Transmitting data on MISO
                            if ( tx_valid )
                                begin
                                    MISO <= tx_data[7 - counter_2];
                                    counter_2 <= counter_2 + 1;
                                end
```

```
                                else
                                    begin
                                        MISO <= 0;
                                  // Ensure MISO is low if tx_valid is not asserted
                                    end
                            end

                default:
                    begin
                         MISO <= 0;
                        counter_1 <= 4'b0000 ;
                        counter_2 <= 3'b000 ;
                        Is_the_address_sent <= 1'b0 ;
                        rx_valid <= 1'b0 ;
                        rx_data <= 10'b0 ;
                    end

            endcase
        end

end
endmodule
```

2.3. SPI wrapper

```
module SPI_Wrapper_top_module
(
    input  wire clk , rst_n ,
    input  wire MOSI , SS_n ,
    output wire MISO
);

wire [9:0] RAM_data_in ;
wire [7:0] RAM_data_out ;
wire rx , tx ;

SPI_Single_Port_RAM mem (
                        .clk(clk) ,
                        .rst_n(rst_n) ,
                        .rx_valid(rx) ,
                        .tx_valid(tx) ,
                        .din(RAM_data_in) ,
                        .dout(RAM_data_out)
                       ) ;

SPI_Slave DUT (
                  .clk(clk)             ,
                  .SS_n(SS_n)           ,
                  .MOSI(MOSI)           ,
                  .MISO(MISO)           ,
                  .rst_n(rst_n)         ,
                  .tx_valid(tx)         ,
                  .rx_valid(rx)         ,
```

```
                .rx_data(RAM_data_in) ,
                .tx_data(RAM_data_out)
            ) ;



endmodule
```

3. SPI testbench

```
module SPI_testbench ;

    reg CLK , rst_n ;
    reg MOSI , SS_n ;

    wire MISO  ;

    reg [9:0] combined_data ;
    integer i ;

reg[7:0] received_data_at_Master  = 8'h00 ; // register to check the correct
functionality

SPI_Wrapper_top_module DUT (.*) ;


//Generate the Clock
localparam T_CLK = 10 ;
always
begin
    CLK = 0 ;
    #(T_CLK / 2) ;

    CLK = 1 ;
    #(T_CLK / 2) ;
end

initial
begin

$display("---------Strat simulation----------");


/*-------------reset functionaity-------------*/
rst_n = 1'b0 ;

repeat(2)@(negedge CLK);
if(1'b0 != MISO)
begin
    $display("Error in reset");
    $stop;
end
```

```
@(negedge CLK) ;
rst_n = 1 ;


/*-----------------------write address----------------------*/
@(negedge CLK) ;
SS_n = 0 ; // start communication && Go to CHK_CMD state

@(negedge CLK) ;
MOSI = 0 ; SS_n = 0 ; // Go to write state

/*send the selection bits = 2'b00 to write address*/

/*choose the address 0XFF */

combined_data = 10'b00_1111_1111 ;

        for(i=0; i<10; i=i+1)
            begin
                @(negedge CLK);
                MOSI = combined_data[9-i];
            end

@(negedge CLK) ;
SS_n = 1'b1 ; // stop communication

#(2*T_CLK) ;  // Ensure data is stable

/*-----------------------write Data----------------------*/

@(negedge CLK) ;
SS_n = 1'b0 ; // start communication

@(negedge CLK) ;
MOSI = 0 ; SS_n = 0 ; // Go to write state

/*send the selection bits = 2'b01 to write address*/

/*Data 1111_1100 (FC) */

combined_data = 10'b01_1111_1100 ;

        for(i=0; i<10; i=i+1)
        begin
            @(negedge CLK);
            MOSI = combined_data[9-i];
        end

@(negedge CLK) ;
SS_n = 1'b1 ; // stop communication

#(2*T_CLK) ;  // Ensure data is stable
```

```
$display("Check address = 0XFF contains Data = 0XFC in RAM");

/*------------------------read address--------------------------*/

@(negedge CLK) ;
SS_n = 1'b0 ; // start communication

@(negedge CLK) ;
MOSI = 1 ; SS_n = 0 ; // Go to read address state

/*send the selection bits = 2'b10 to read address*/

/*address is 0XFF "1111_1111" */

combined_data = 10'b10_1111_1111 ;

        for(i=0; i<10; i=i+1)
        begin
            @(negedge CLK);
            MOSI = combined_data[9-i];
        end

@(negedge CLK) ;
SS_n = 1'b1 ; // stop communication

#(2*T_CLK) ;  // Ensure data is stable

/*------------------------------read data-------------------------------*/

@(negedge CLK) ;
SS_n = 1'b0 ; // start communication

@(negedge CLK) ;
MOSI = 1 ; SS_n = 0 ; // Go to read data

/*send the selection bits = 2'b11 to read data*/

/*we will read the data 0XFC which is written before */

combined_data = 10'b11_00000000 ;

        for(i=0; i<10; i=i+1)
        begin
            @(negedge CLK);
            MOSI = combined_data[9-i];
        end

    @(negedge CLK) ; // wait the memory to accept its inputs

    #(T_CLK) ; // wait the memory to deliver the outputs and activate the tx_valid

    #(T_CLK) ; // wait clock after tx_valid = 1 --> could be done using [
wait(DUT.mem.tx_valid) ]
```

```
    #(T_CLK) ;

     for(i=0 ; i<8 ; i=i+1)
        begin
            @(negedge CLK) ; // as mosi is changed with +Ve edge
            received_data_at_Master = ( received_data_at_Master<<1 ) | MISO ;
        end

if(received_data_at_Master != 8'hFC)
begin
       $display("Error in receiveing MISO ");
       $display("received_data_at_Master = 0h%0h" , received_data_at_Master);
       $stop;
end

@(negedge CLK) ;
SS_n = 1'b1 ; // stop communication

    $display("The testbench is done successfully :) ");
    $display("-------------------------------------");
    $stop;

end
endmodule
```

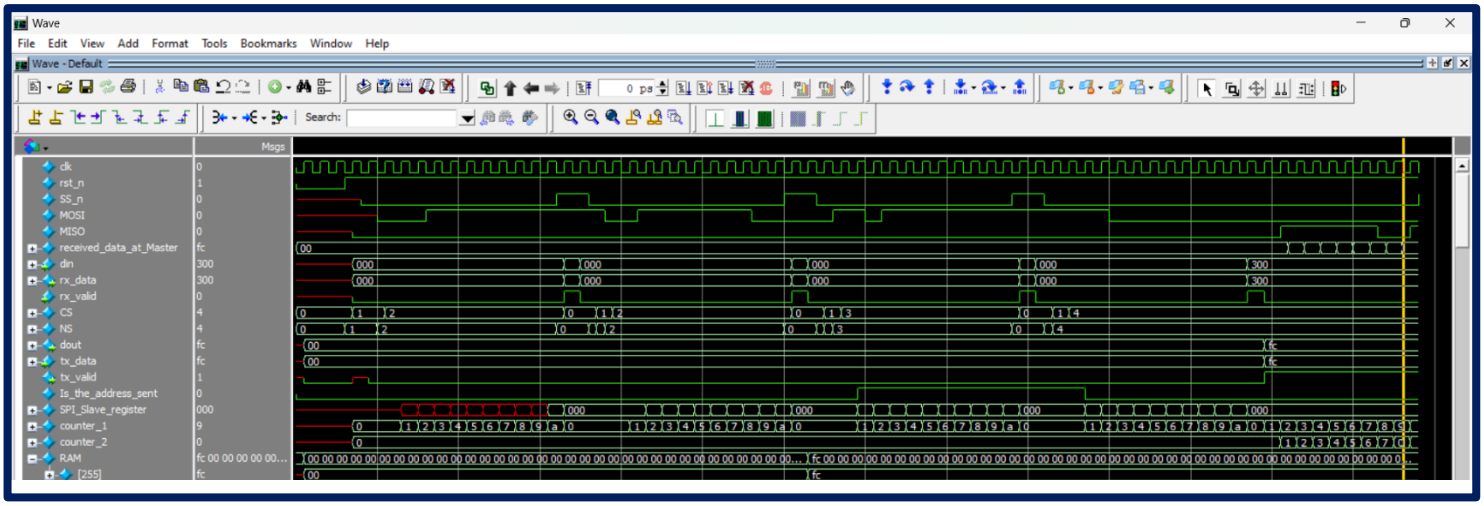
4. constraint file

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name clk -period 10.00 -waveform {0 5} [get_ports clk]


## 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]
```

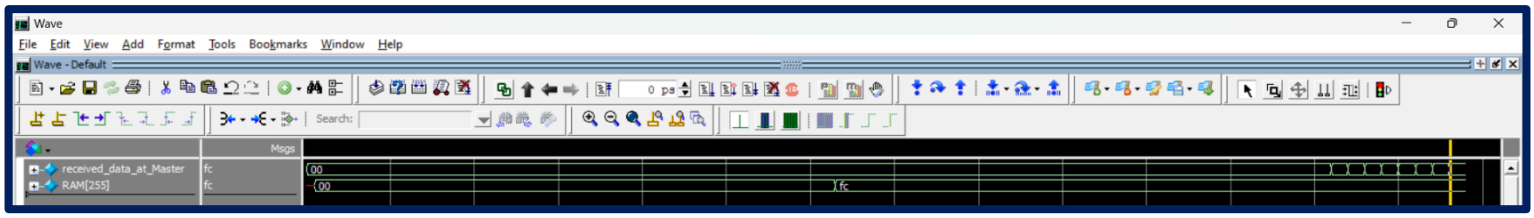
5. Do file

```
# open work for projec
vlib work

# compile the files
vlog SPI_Single_Port_RAM.v  SPI_Slave.v SPI_Wrapper.v SPI_testbench.v

# simulate testbench
vsim -voptargs="+acc" work.SPI_testbench

# Add signals to wave
add wave -position insertpoint \
sim:/SPI_testbench/clk \
sim:/SPI_testbench/rst_n \
sim:/SPI_testbench/SS_n \
sim:/SPI_testbench/MOSI \
sim:/SPI_testbench/MISO \
sim:/SPI_testbench/received_data_at_Master \
sim:/SPI_testbench/DUT/mem/din \
sim:/SPI_testbench/DUT/DUT/rx_data \
sim:/SPI_testbench/DUT/mem/rx_valid \
sim:/SPI_testbench/DUT/DUT/CS \
sim:/SPI_testbench/DUT/DUT/NS \
sim:/SPI_testbench/DUT/mem/dout \
sim:/SPI_testbench/DUT/DUT/tx_data \
sim:/SPI_testbench/DUT/mem/tx_valid \
sim:/SPI_testbench/DUT/DUT/Is_the_address_sent \
sim:/SPI_testbench/DUT/DUT/SPI_Slave_register \
sim:/SPI_testbench/DUT/mem/RAM \
sim:/SPI_testbench/DUT/DUT/counter_1 \
sim:/SPI_testbench/DUT/DUT/counter_2

# run the simulation

 run -all
```

6. Waveform



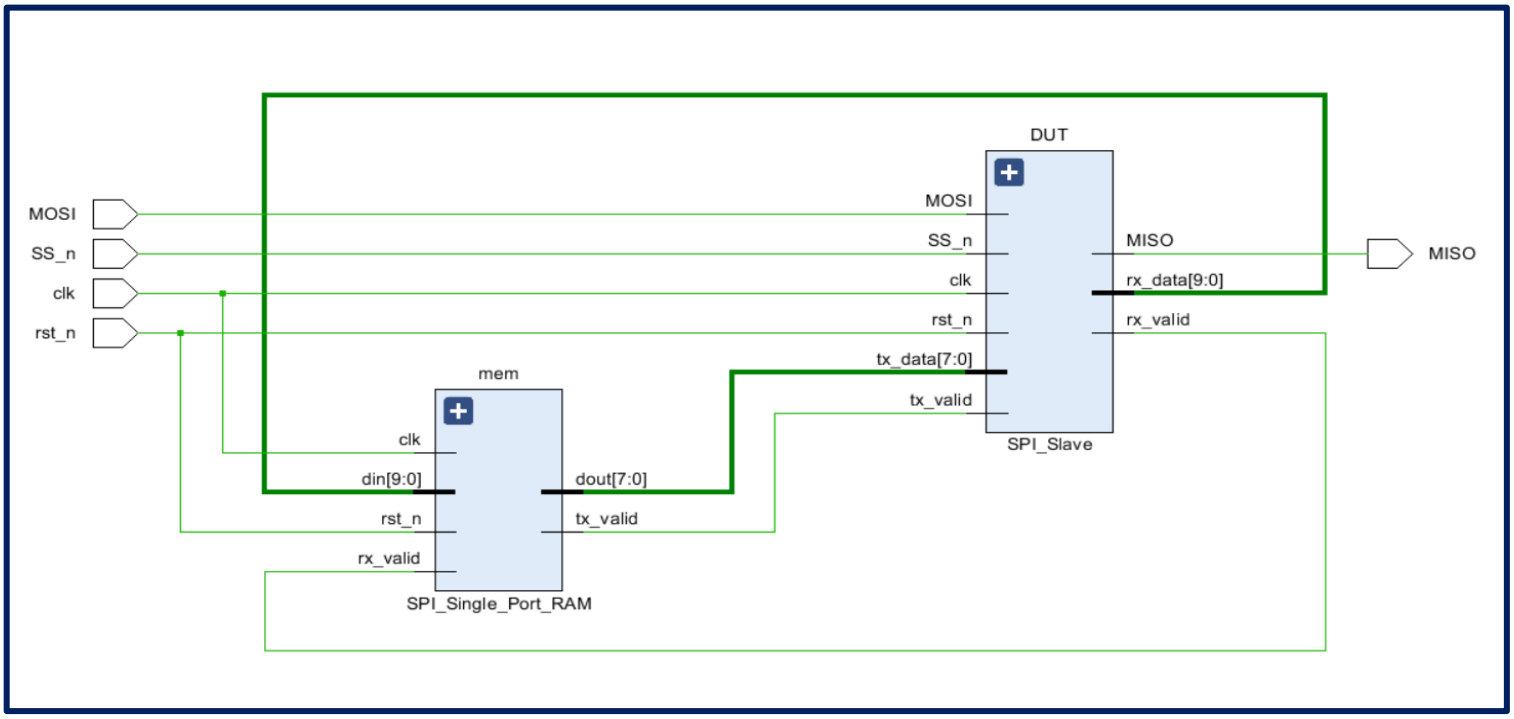
Ensure that the data received by the master matches the data stored in RAM.



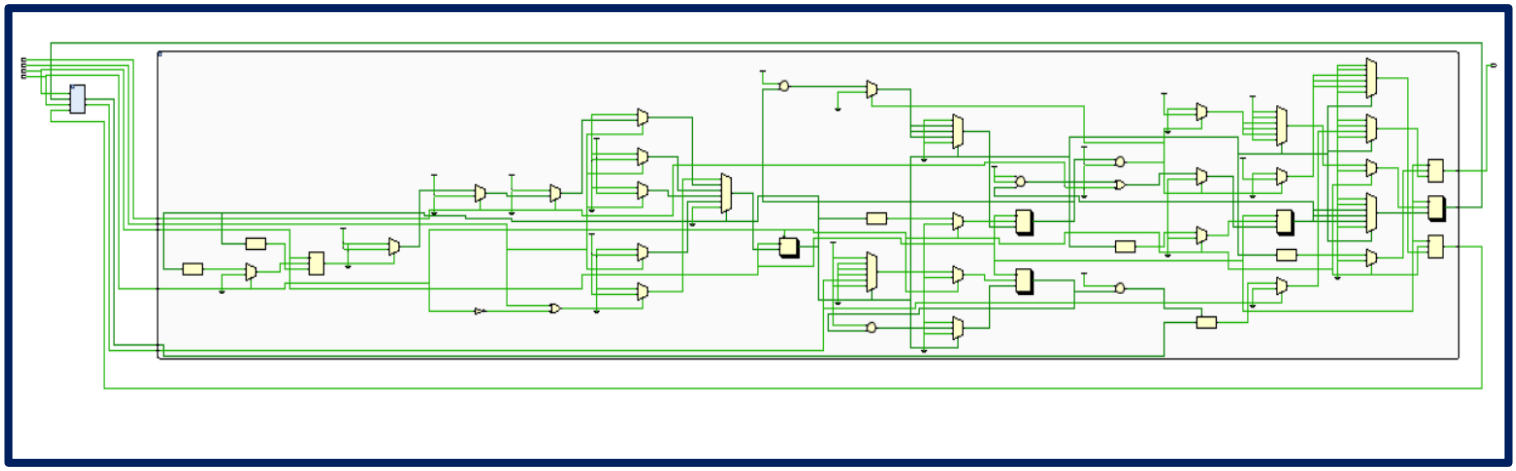
7. Transcript

8. Elaboration

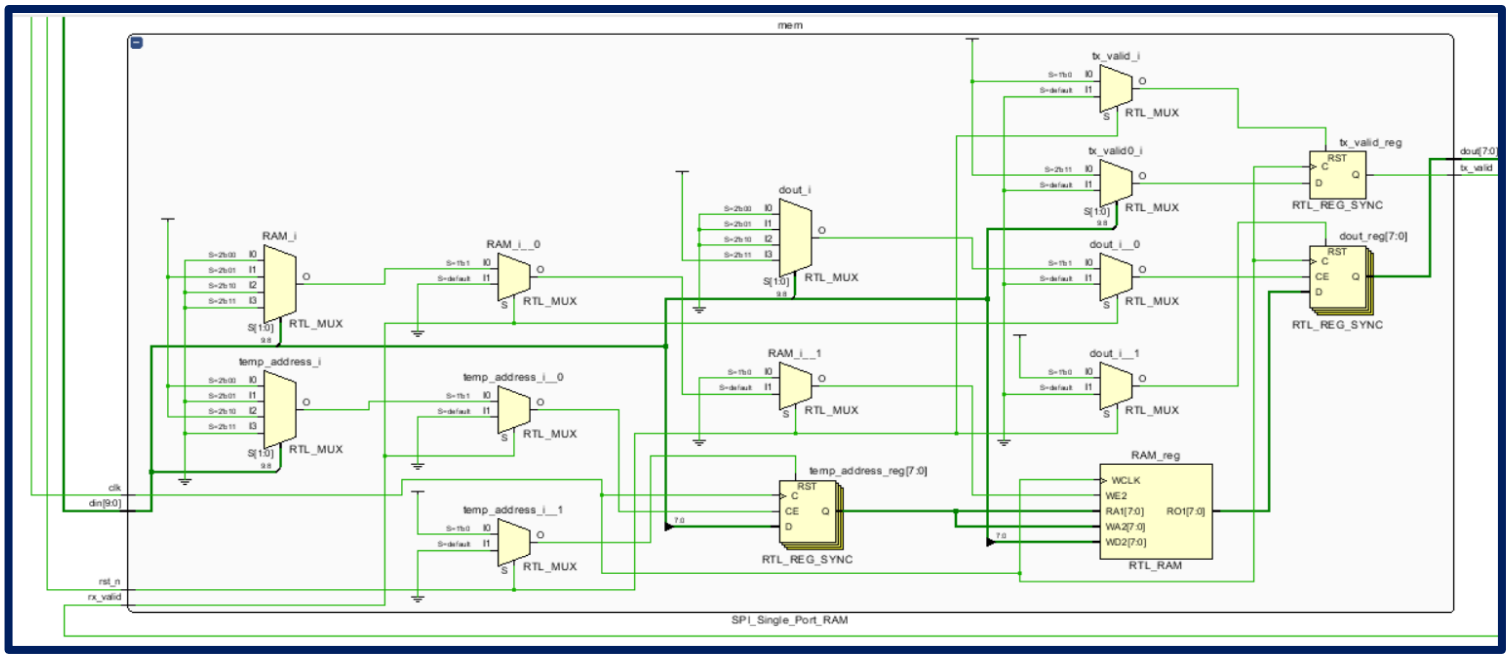
8.1. SPI wrapper Schematic



8.2. SPI Slave schematic



8.3. Memory schematic



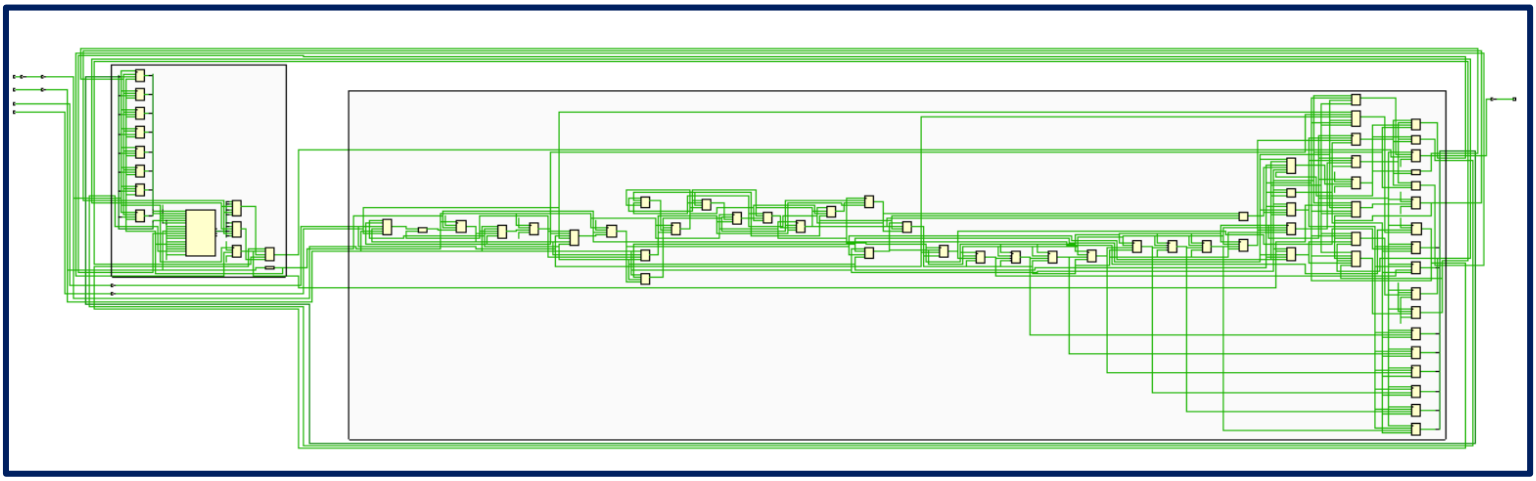
9. Synthesis

9.1. Sequential Encoding

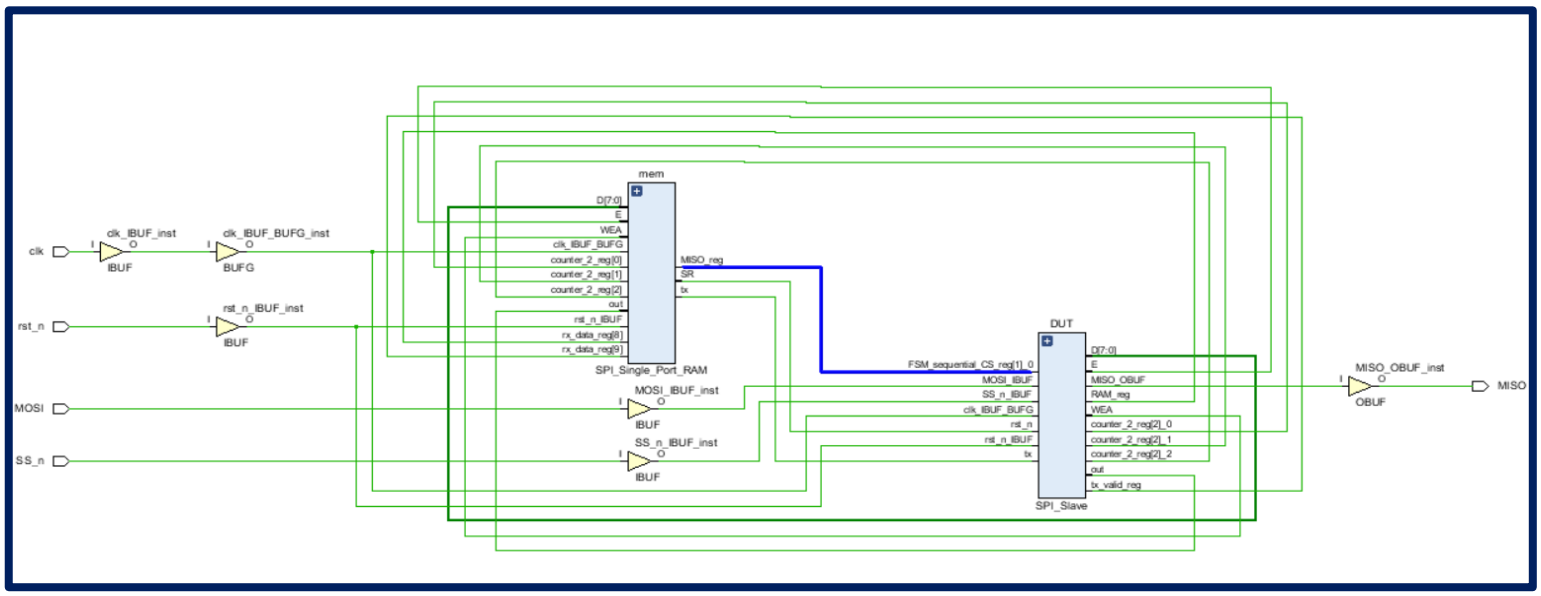
State	New Encoding	Previous Encoding
IDLE	000	000
CHK_CMD	001	001
WRITE	010	010
READ_DATA	011	100
READ_ADD	100	011

```
INFO: [Synth 8-3354] encoded FSM with state register 'CS_reg' using encoding 'sequential' in module 'SPI_Slave'
```

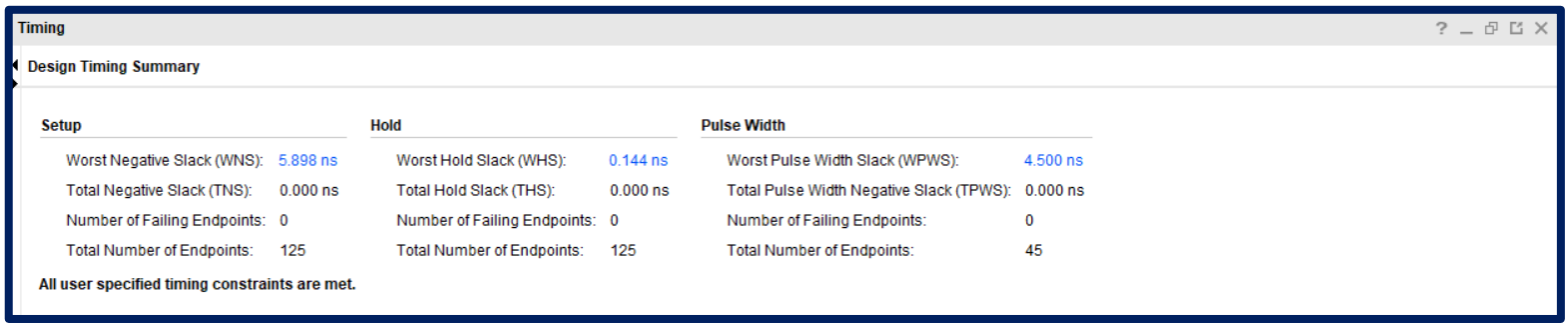
9.2. Schematic



9.3. Critical Path

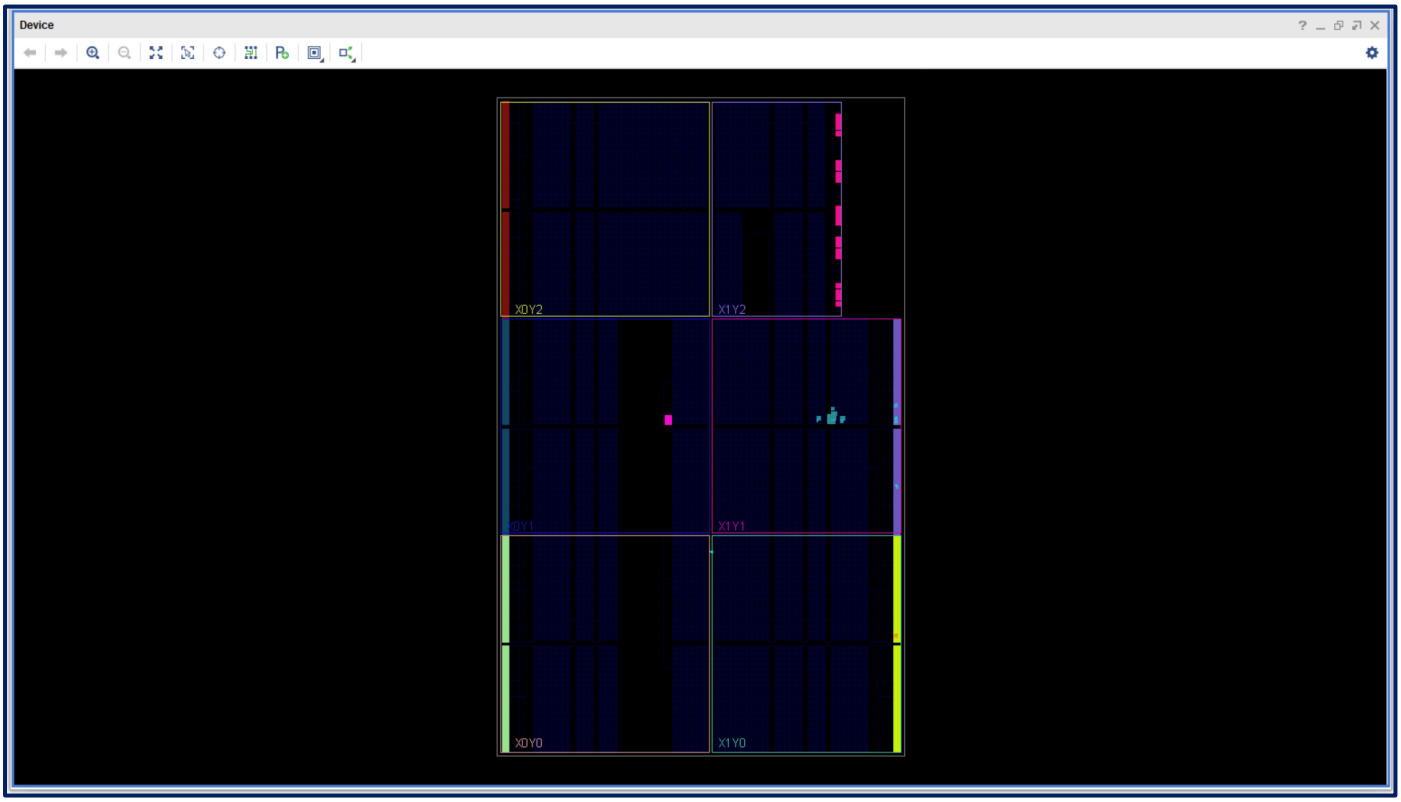


9.4. Report timing summary

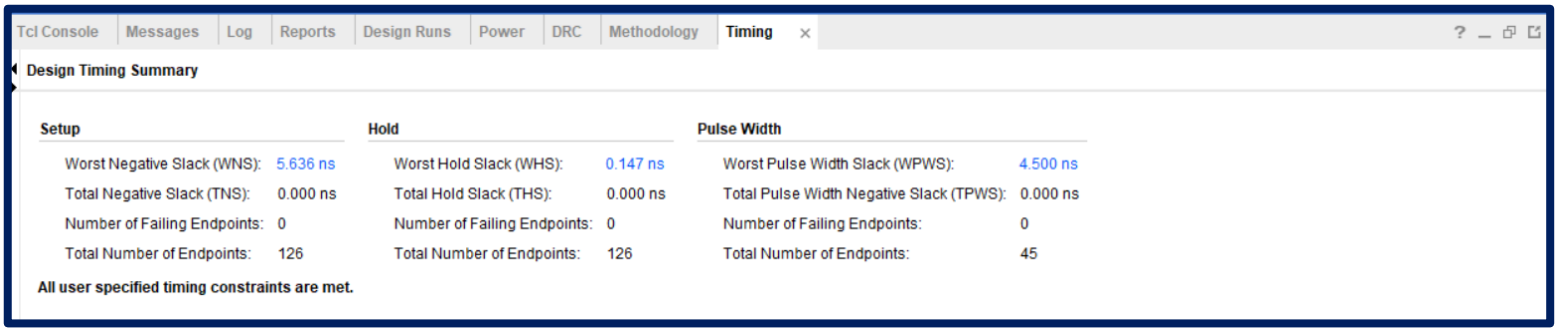


10. Implementation

10.1. Device

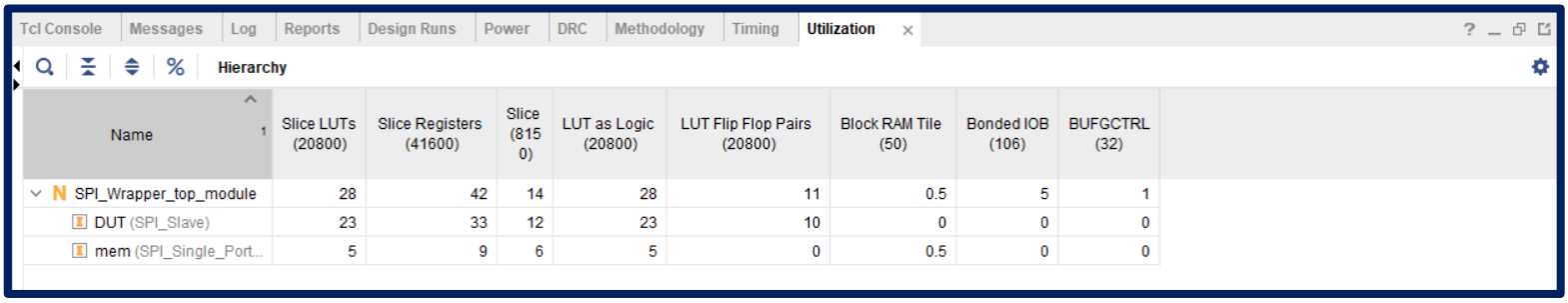


10.2. Design timing summary



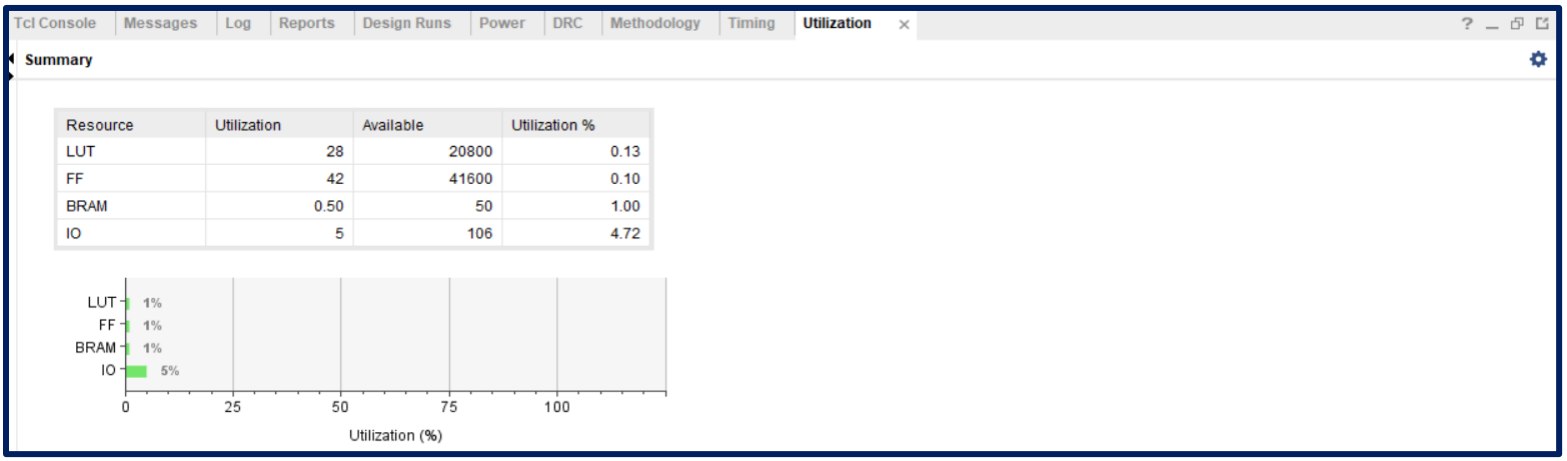
10.3. Utilization Report

10.3.1. Hierarchy



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_top_module	28	42	14	28	11	0.5	5	1
DUT (SPI_Slave)	23	33	12	23	10	0	0	0
mem (SPI_Single_Port...	5	9	6	5	0	0.5	0	0

10.3.2. Summary



Resource	Utilization	Available	Utilization %
LUT	28	20800	0.13
FF	42	41600	0.10
BRAM	0.50	50	1.00
IO	5	106	4.72