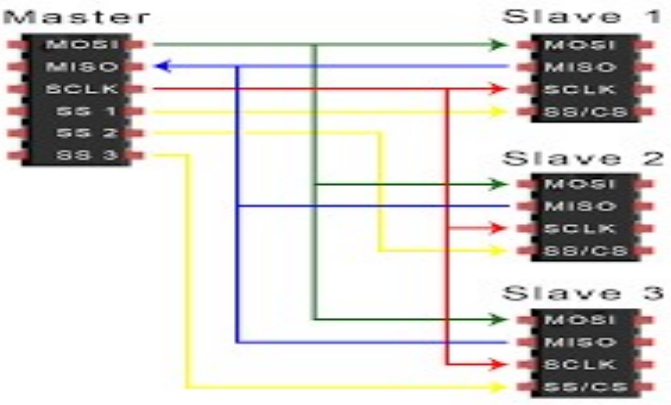
Mini Project

SPI Slave with Single Port RAM]

By Akshay Pawar

SPI INTERFACE



Serial peripheral interface (SPI) is one of the most widely used interfaces between microcontroller and peripheral ICs such as sensors, ADCs, DACs, shift registers, SRAM, and others. This article provides a brief description of the SPI interface followed by an introduction to Analog Devices' SPI enabled switches and muxes, and how they help reduce the number of digital GPIOs in system board design.SPI is a synchronous, full duplex main-subnode-based interface. The data from the main or the subnode is synchronized on the rising or falling clock edge. Both main and subnode can transmit data at the same time. The SPI interface can be either 3-wire or 4-wire.

4-wire SPI devices have four signals:

- Clock (SPI CLK, SCLK)
- Chip select (CS)
- main out, subnode in (MOSI)
- main in, subnode out (MISO)

RTL CODE

RAM:

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

    parameter MEM_DEPTH = 256;
    parameter ADDR_SIZE = 8;


// Creating the memory array
reg [ADDR_SIZE-1:0] memory [MEM_DEPTH-1:0];

reg [7:0] addr_wr; // Write address
reg [7:0] addr_re; // Read address


always @(posedge clk) begin
    if (!rst_n) begin
        dout <= 8'b0;
        tx_valid <= 1'b0;
        addr_wr <= 8'b0;
        addr_re <= 8'b0;
 end else begin
     if (rx_valid) begin
         case (din[9:8])
             2'b00: begin
                 addr_wr <= din[7:0]; // Write address
                 tx_valid <= 1'b0;
             end
             2'b01: begin
                 memory[addr_wr] <= din[7:0]; // Write data
                 tx_valid <= 1'b0;
             end
             2'b10: begin
                 addr_re <= din[7:0]; // Read address
                 tx_valid <= 1'b0;
             end
             2'b11: begin // Read data
                 dout <= memory[addr_re];
                 tx_valid <= 1'b1;
             end
         endcase
     end
    end
end
endmodule
```

SPI SLAVE:

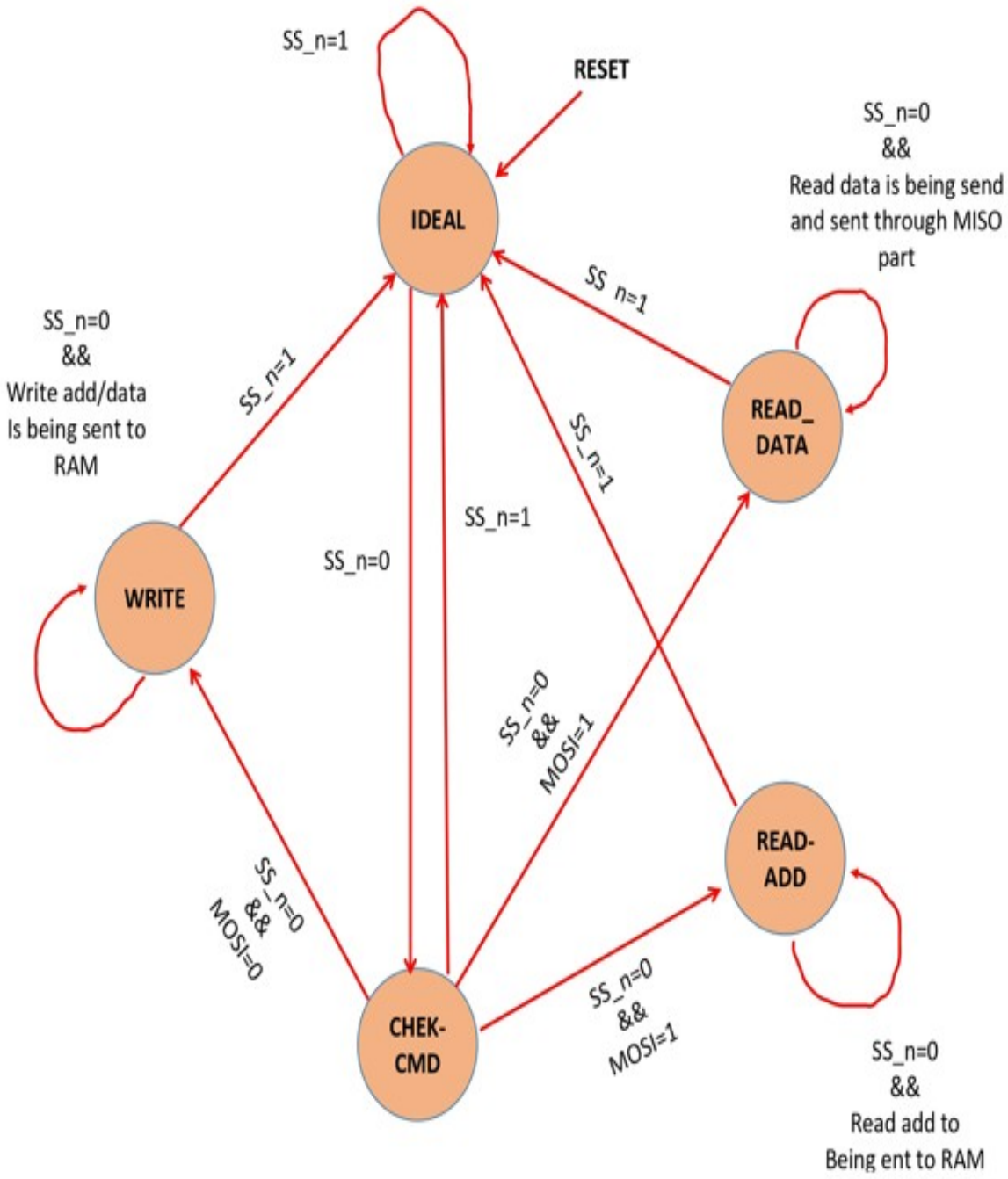
```
module SPI_SLAVE(
    input MOSI , tx_valid , clk , rst_n , ss_n ,
    input [7:0] tx_data ,
    output reg  MISO , rx_valid ,
    output reg [9:0] rx_data
);
localparam IDLE = 3'b000,
           CHK_CMD = 3'b001,
           WRITE = 3'b010,
           READ_ADD = 3'b011,
           READ_DATA = 3'b100;

reg [2:0] cs , ns ; ///current state and next state
reg ADD_DATA_checker ;
reg [3:0] counter1 ; //counter to fill the rx_data
reg [2:0] counter2 ; // counter for the READ_DATA
reg [9:0] bus ;

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

//next state logic
always @(*) begin
    ns = cs ;
    case(cs)
        IDLE : begin
            if(ss_n)
                ns = IDLE;
            else
                ns = CHK_CMD;
        end
        CHK_CMD : begin
            if(ss_n)
                ns = IDLE;
            else begin
                if((~ss_n) && (MOSI == 0))
                    ns = WRITE;
```

```
                else if ((~ss_n) && (MOSI == 1) && (ADD_DATA_checker == 1))
                    ns = READ_ADD;
                else if ((~ss_n) && (MOSI == 1) && (ADD_DATA_checker == 0))
                    ns = READ_DATA;
            end
        end
        WRITE : begin
            if(ss_n || counter1 == 4'b1111)
                ns = IDLE;
            else
                ns = WRITE;
        end
        READ_ADD : begin
            if(ss_n || counter1 == 4'b1111)
                ns = IDLE;
            else
                ns = READ_ADD;
        end
        READ_DATA : begin
            if(ss_n)
                ns = IDLE;
            else
                ns = READ_DATA;
        end
    endcase
end

//output logic
always @(posedge clk or negedge rst_n) begin
    if (~rst_n) begin
        counter1 <= 9; //as the first bit entered will be the MSB
        counter2 <= 7; // as the first bit outted will be the MSB
        ADD_DATA_checker <= 1;
        bus <= 0;
        rx_data <= 0;
        rx_valid <= 0;
        MISO  <= 0;
    end
```

```
    end
    //IDLE state
    else begin
        if(cs == IDLE) begin
            rx_valid <= 0;
            counter1 <= 9 ;
            counter2 <= 7 ;
        end
        //WRITE state
        else if(cs == WRITE) begin
            if (counter1 >= 0)begin
                bus[counter1] <= MOSI;
                counter1 <= counter1 - 1;   //decrement the counter
            end
            if(counter1 == 4'b1111) begin
                rx_vali
                rx_data                                  data to the RAM
            end
        end
        //READ_ADD state
        else if (cs == READ_ADD) begin
            if (counter1 >= 0)begin
                bus[counter1] <= MOSI;
                counter1 <= counter1 - 1;   //decrement the counter
            end
            if(counter1 == 4'b1111) begin
                rx_valid <= 1;
                rx_data <= bus ; //sending the parallel data to the RAM
                ADD_DATA_checker <= 0;
            end
        end

         //READ_DATA state
         else if (cs == READ_DATA) begin
             if (counter1 >= 0)begin
                 bus[counter1] <= MOSI;
                 counter1 <= counter1 - 1;   //decrement the counter
             end
             if(counter1 == 4'b1111) begin
                 rx_valid <= 1;
                 rx_data <= bus ; //sending the parallel data to the RAM
                 counter1 <= 9 ;
             end
             if(rx_valid  == 1) rx_valid <= 0;
             if(tx_valid==1 && counter2 >=0)begin
                 MISO <= tx_data[counter2] ;
                 counter2 <= counter2 - 1 ;
             end
             if(counter2 == 3'b111)begin
                 ADD_DATA_checker <= 1;
             end
         end
     end

end
endmodule
```

/SPI_Wrapper_tb/dut/SPI/counter1
No_Data

FSM DIAGRAM:


SS_n=1
RESET
IDEAL
SS_n=0
&&
Read data is being send
and sent through MISO
part
SS_n=0
&&
Write add/data
Is being sent to
RAM
SS_n=1
SS_n=1
READ_
DATA
SS_n=1
SS_n=1
SS_n=0
WRITE
SS_n=0
&&
MOSI=1
READ-
ADD
SS_n=0
&&
MOSI=0
CHEK-
CMD
SS_n=0
&&
MOSI=1
SS_n=0
&&
Read add to
Being ent to RAM

SPI WRAPPER:

```
module SPI_Wrapper #(parameter MEM_DEPTH = 256 , parameter ADDR_SIZE = 8)(
    input clk , ss_n , MOSI , rst_n ,
    output MISO
);
//this module is to connect between the SPI_SLAVE and RAM modules
wire [9:0] rxdata ;
wire [7:0] txdata ;
wire rx_valid , tx_valid ;

SPI_SLAVE SPI(
    .MOSI(MOSI),
    .MISO(MISO),
    .clk(clk),
    .ss_n(ss_n),
    .rst_n(rst_n),
    .rx_data(rxdata),
    .tx_data(txdata),
    .rx_valid(rx_valid),
    .tx_valid(tx_valid)
);

RAM #(MEM_DEPTH,ADDR_SIZE) Ram (
    .din(rxdata),
    .dout(txdata),
    .clk(clk),
    .rx_valid(rx_valid),
    .tx_valid(tx_valid),
    .rst_n(rst_n)
);
endmodule
```

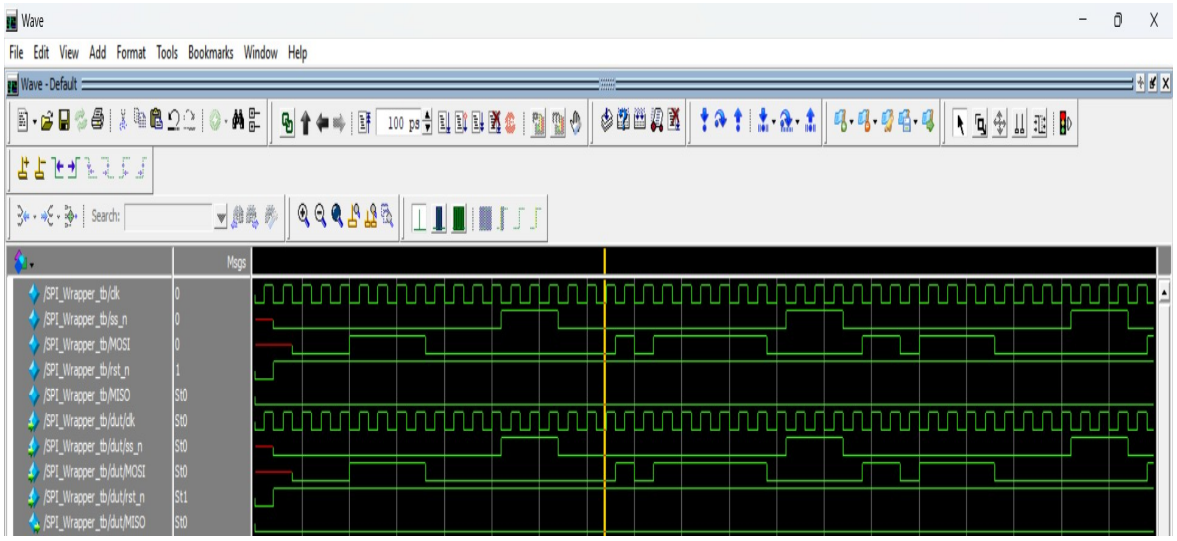
TESTBENCH:

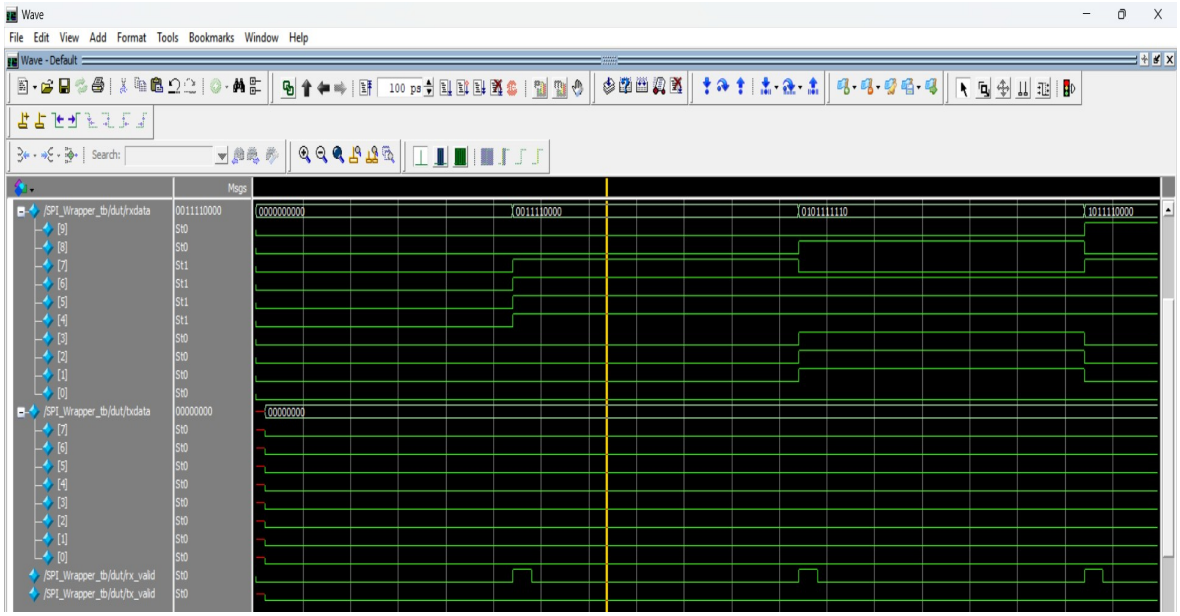
```
module SPI_Wrapper_tb();
parameter MEM_DEPTH = 256 ;
parameter ADDR_SIZE = 8 ;
reg clk , ss_n , MOSI , rst_n ;
wire MISO;


SPI_Wrapper #(.MEM_DEPTH(MEM_DEPTH),.ADDR_SIZE(ADDR_SIZE)) dut (clk, ss_n, MOSI , rst_n , MISO);
initial begin
    clk = 0;
    forever #10 clk = ~clk; // 20ns period clock
end
initial begin

//********************************** first scenario ******************************\\
    rst_n = 0;  //reset first so that the first state is IDLE
    @(negedge clk);
    rst_n = 1;
    ss_n = 0; //go to CHK_CMD state
    //first the write address process
    #20 MOSI = 0; //to decide it's a write process
    #20 MOSI = 0;
    #20 MOSI = 0; //the first two bits are zeroes

    //we will write in the address : 8'b1111_0000
    #20 MOSI = 1;
    #20 MOSI = 1;
    #20 MOSI = 1;
    #20 MOSI = 1;
    #20 MOSI = 0;
    #20 MOSI = 0;
    #20 MOSI = 0;
    #20 MOSI = 0;
    //address finished
    #20 ss_n = 1; //end protocol
    #60 ; //time to wait until the next process comes


    ss_n = 0; //go to CHK_CMD state
    //second comes write data process
    #20 MOSI = 0; //to decide it's a write process
    #20 MOSI = 0;
    #20 MOSI = 1; //the first two bits are 2'b10
```

```
repeat(8) begin
    #20 MOSI = $random; //randomize the data
end
#20 ss_n = 1; //end protocol
#60 ; //time to wait until the next process comes


ss_n = 0; //go to CHK_CMD state
//third comes read address process
#20 MOSI = 1; //to decide it's a read process
#20 MOSI = 1;
#20 MOSI = 0; //the first two bits are 2'b10

#20 MOSI = 1;
#20 MOSI = 1;
#20 MOSI = 1;
#20 MOSI = 1;
#20 MOSI = 0;
#20 MOSI = 0;
#20 MOSI = 0;
#20 MOSI = 0;
//address finished
#20 ss_n = 1; //end protocol
#60 ; //time to wait until the next process comes


ss_n = 0; //go to CHK_CMD state
//fourth comes read data process
#20 MOSI = 1; //to decide it's a read process
#20 MOSI = 1;
#20 MOSI = 1; //the first two bits are 2'b11


repeat(8) begin
    #20 MOSI = $random; //randomize the data as we won't use it (dummy data)
end
#200 ss_n = 1;
```

```
rst_n = 0;  //reset again to start a new process
#20 rst_n = 1;
    ss_n = 0;

#20 MOSI = 0; //to decide it's a write process
#20 MOSI = 0;
#20 MOSI = 0; //the first two bits are zeroes

//we will write in the address : 8'b0000_1111
#20 MOSI = 0;
#20 MOSI = 0;
#20 MOSI = 0;
#20 MOSI = 0;
#20 MOSI = 1;
#20 MOSI = 1;
#20 MOSI = 1;
#20 MOSI = 1;
//address finished
#20 ss_n = 1; //end protocol
#60 ; //time to wait until the next process comes


ss_n = 0; //go to CHK_CMD state
//second comes write data process
#20 MOSI = 0;
#20 MOSI = 0;
#20 MOSI = 1;
repeat(8) begin
    #20 MOSI = 1; //write data which is all ones
end
#20 ss_n = 1; //end protocol
#60 ; //time to wait until the next process comes

   ss_n = 0; //go to CHK_CMD state
   //third comes read address process
   #20 MOSI = 1;
   #20 MOSI = 1;
   #20 MOSI = 0;


   #20 MOSI = 0;
   #20 MOSI = 0;
   #20 MOSI = 0;
   #20 MOSI = 0;
   #20 MOSI = 1;
   #20 MOSI = 1;
   #20 MOSI = 1;
   #20 MOSI = 1;
   //address finished
   #20 ss_n = 1; //end protocol
   #60 ; //time to wait until the next process comes


   ss_n = 0; //go to CHK_CMD state
   //fourth comes read data process
   #20 MOSI = 1; //to decide it's a read process
   #20 MOSI = 1;
   #20 MOSI = 1; //the first two bits are 2'b11

   //read data process will automatically read data from the address we chose in the read address state (which was : 8'b1111_0000)
   repeat(8) begin
       #20 MOSI = $random;
   end
   #200 ss_n = 1;
   #60 ; //time to wait until the next process comes

   $display("simulation finished");
   $stop;
end
endmodule
```

SIMULATION:

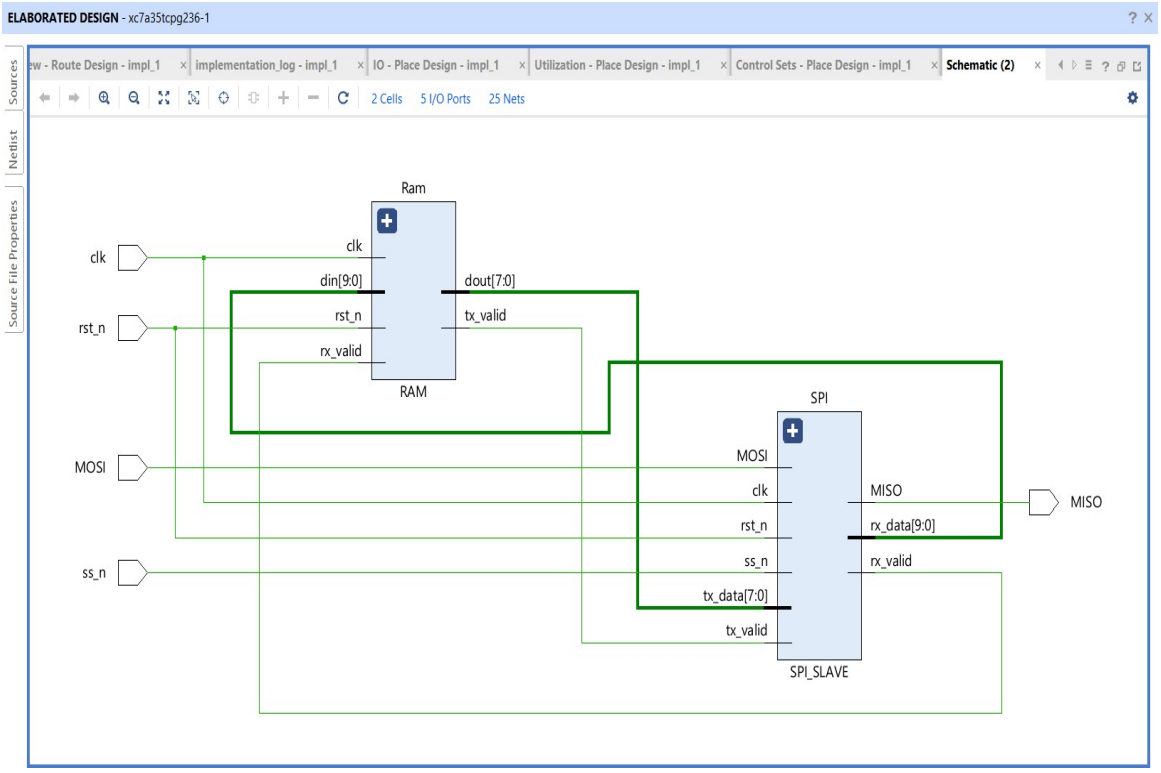

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
/SPI_Wrapper_tb/clk
/SPI_Wrapper_tb/ss_n
/SPI_Wrapper_tb/MOSI
/SPI_Wrapper_tb/rst_n
/SPI_Wrapper_tb/MISO
/SPI_Wrapper_tb/dut/clk
/SPI_Wrapper_tb/dut/ss_n
/SPI_Wrapper_tb/dut/MOSI
/SPI_Wrapper_tb/dut/rst_n
/SPI_Wrapper_tb/dut/MISO


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
/SPI_Wrapper_tb/dut/rxdata
/SPI_Wrapper_tb/dut/txdata
/SPI_Wrapper_tb/dut/rx_valid
/SPI_Wrapper_tb/dut/tx_valid

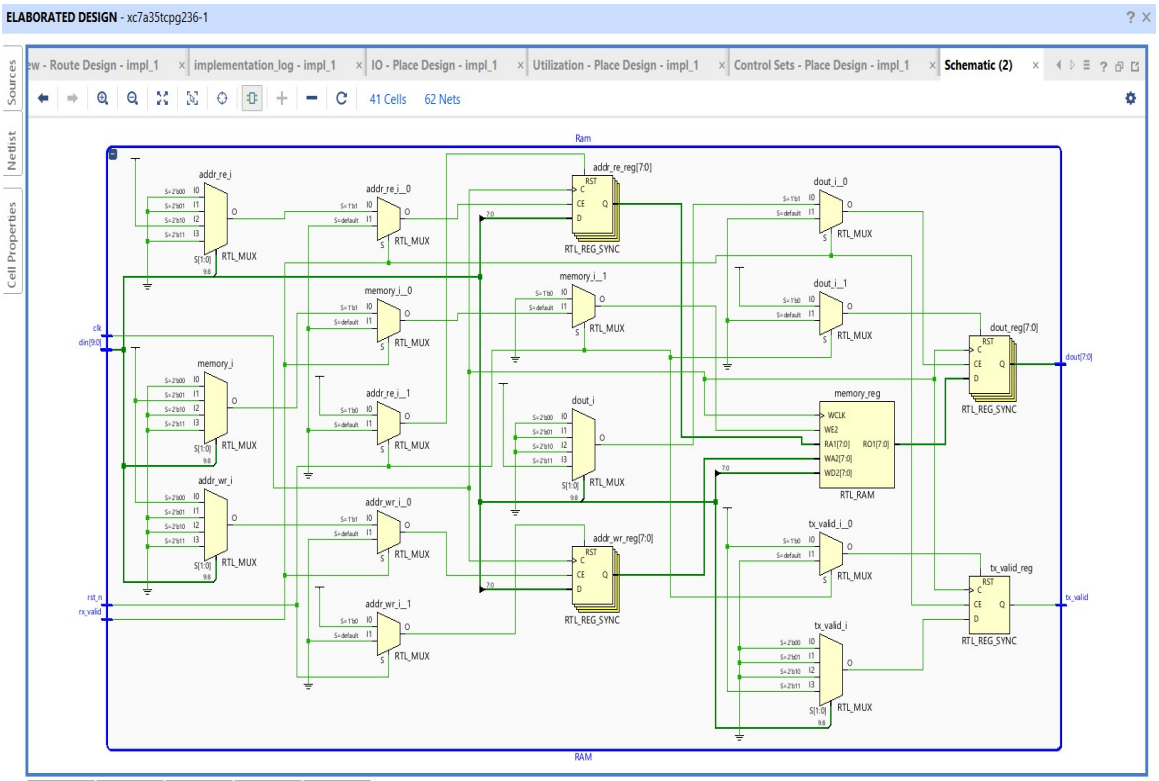
DO FILE:

```
onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate /SPI_Wrapper_tb/clk
add wave -noupdate /SPI_Wrapper_tb/ss_n
add wave -noupdate /SPI_Wrapper_tb/MOSI
add wave -noupdate /SPI_Wrapper_tb/rst_n
add wave -noupdate /SPI_Wrapper_tb/MISO
add wave -noupdate /SPI_Wrapper_tb/dut/clk
add wave -noupdate /SPI_Wrapper_tb/dut/ss_n
add wave -noupdate /SPI_Wrapper_tb/dut/MOSI
add wave -noupdate /SPI_Wrapper_tb/dut/rst_n
add wave -noupdate /SPI_Wrapper_tb/dut/MISO
add wave -noupdate -expand /SPI_Wrapper_tb/dut/rxdata
add wave -noupdate -expand /SPI_Wrapper_tb/dut/txdata
add wave -noupdate /SPI_Wrapper_tb/dut/rx_valid
add wave -noupdate /SPI_Wrapper_tb/dut/tx_valid
add wave -noupdate -format Analog-Step -height 500 -radix decimal /#vsim_capacity#/totals
add wave -noupdate /#vsim_capacity#/classes
add wave -noupdate /#vsim_capacity#/qdas
add wave -noupdate /#vsim_capacity#/assertions
add wave -noupdate /#vsim_capacity#/covergroups
add wave -noupdate /#vsim_capacity#/solver
add wave -noupdate /#vsim_capacity#/memories
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {369 ps} 0}
quietly wave cursor active 1
configure wave -namecolwidth 208
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 0
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 1
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ps
update
WaveRestoreZoom {0 ps} {947 ps}
```

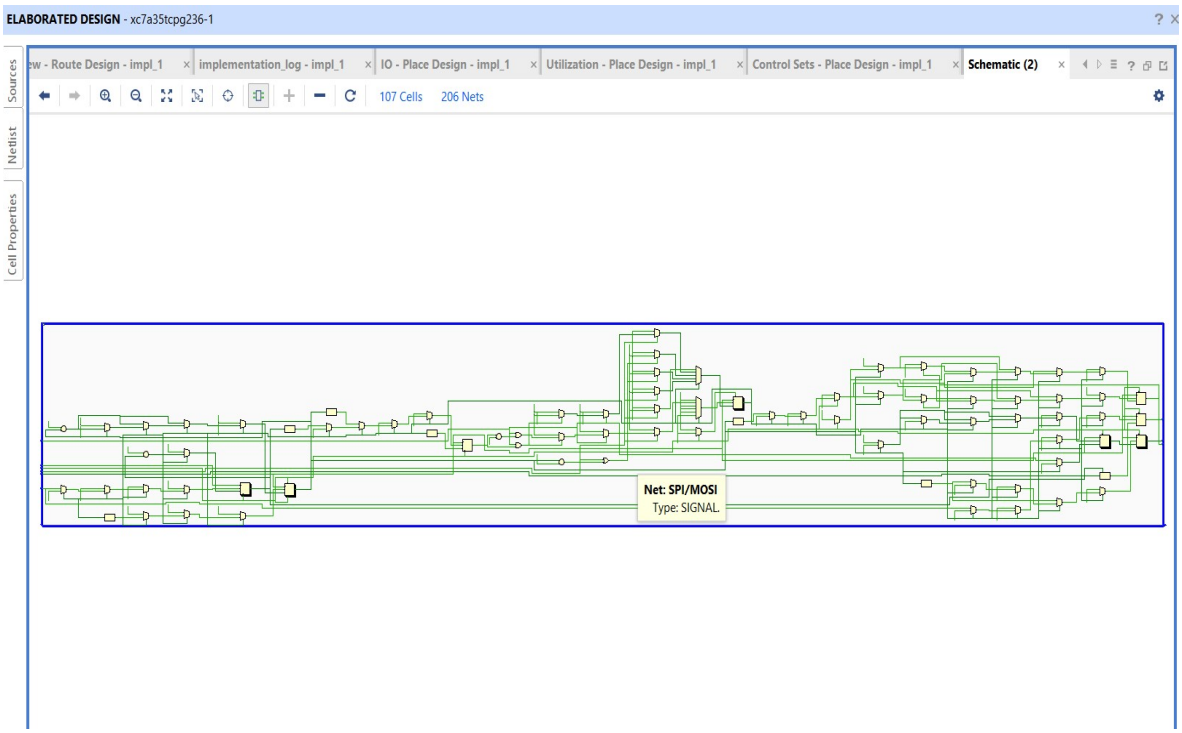
RTL DESIGN:



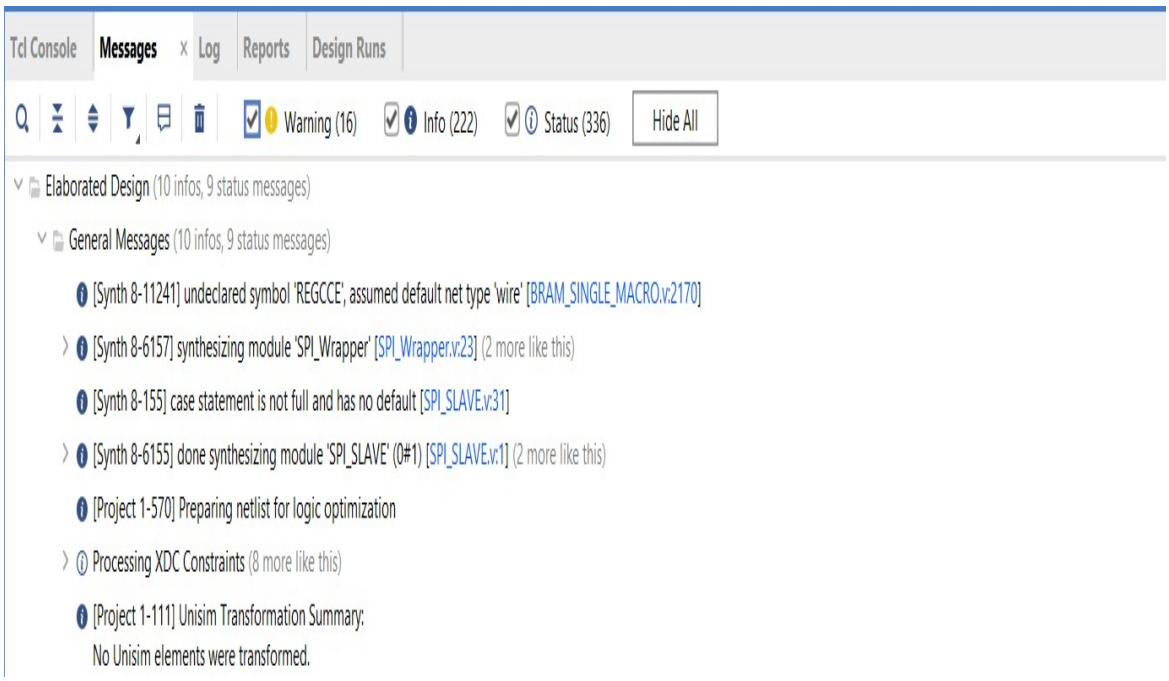
RAM:



SPL_SLAVE:

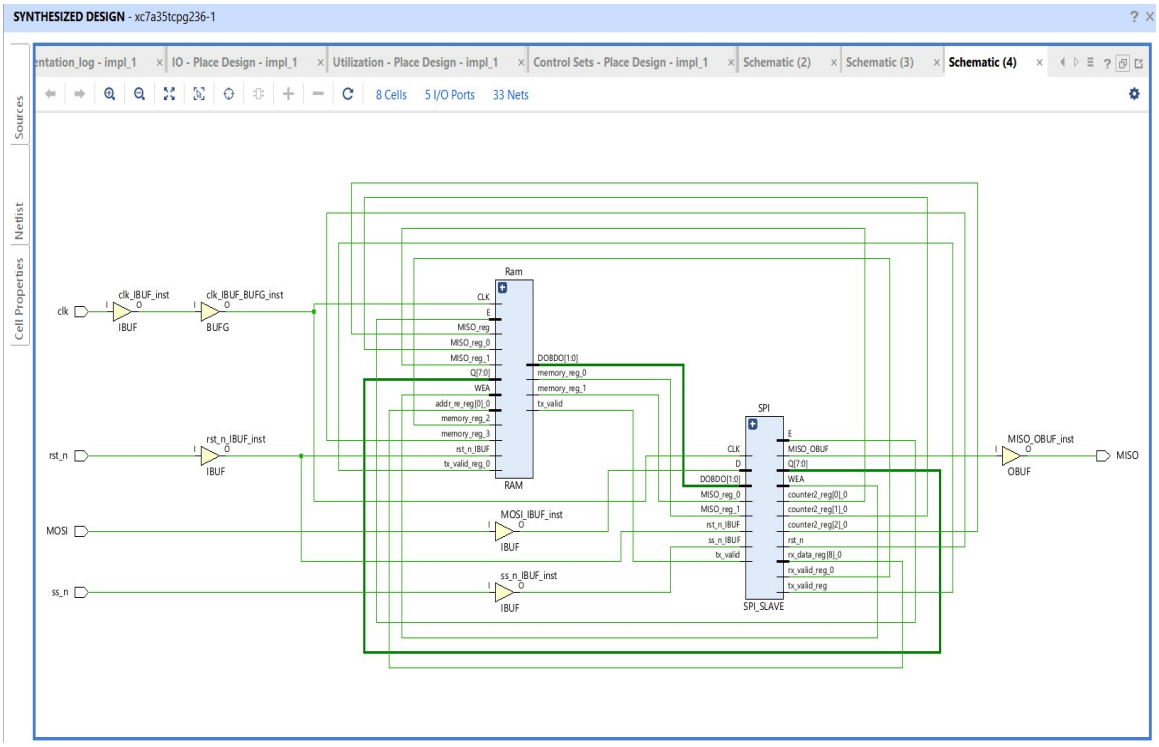


MASSAGE:

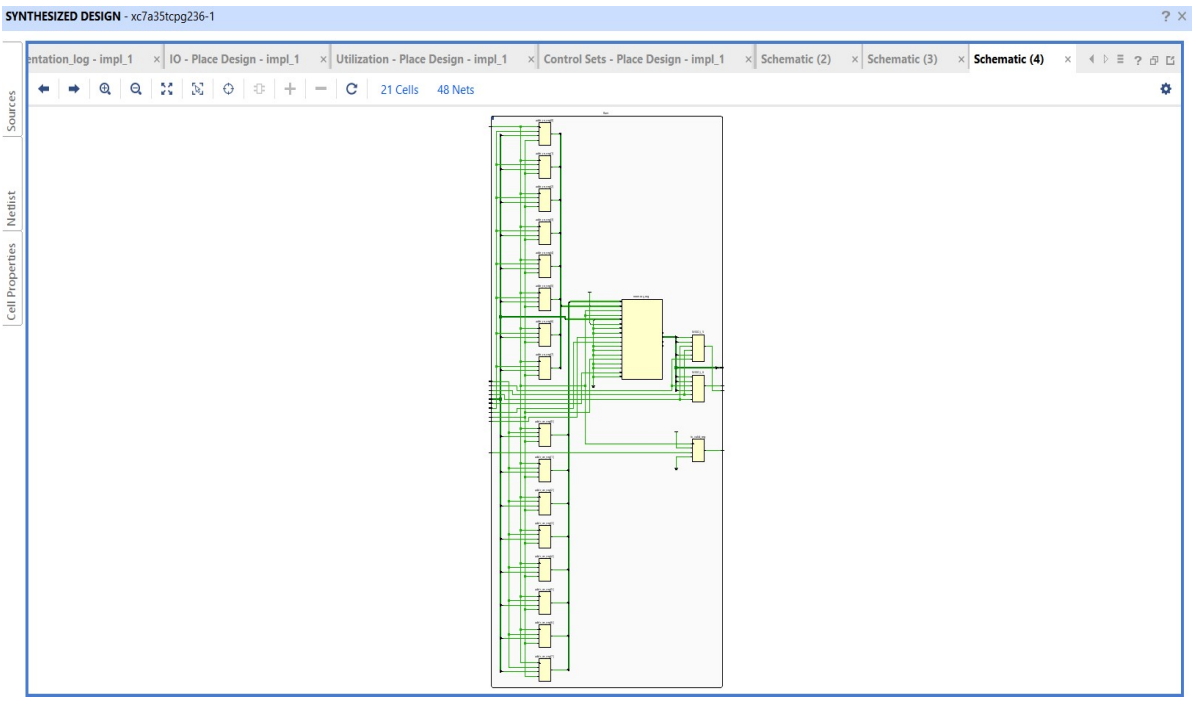


SYNTHESIS:

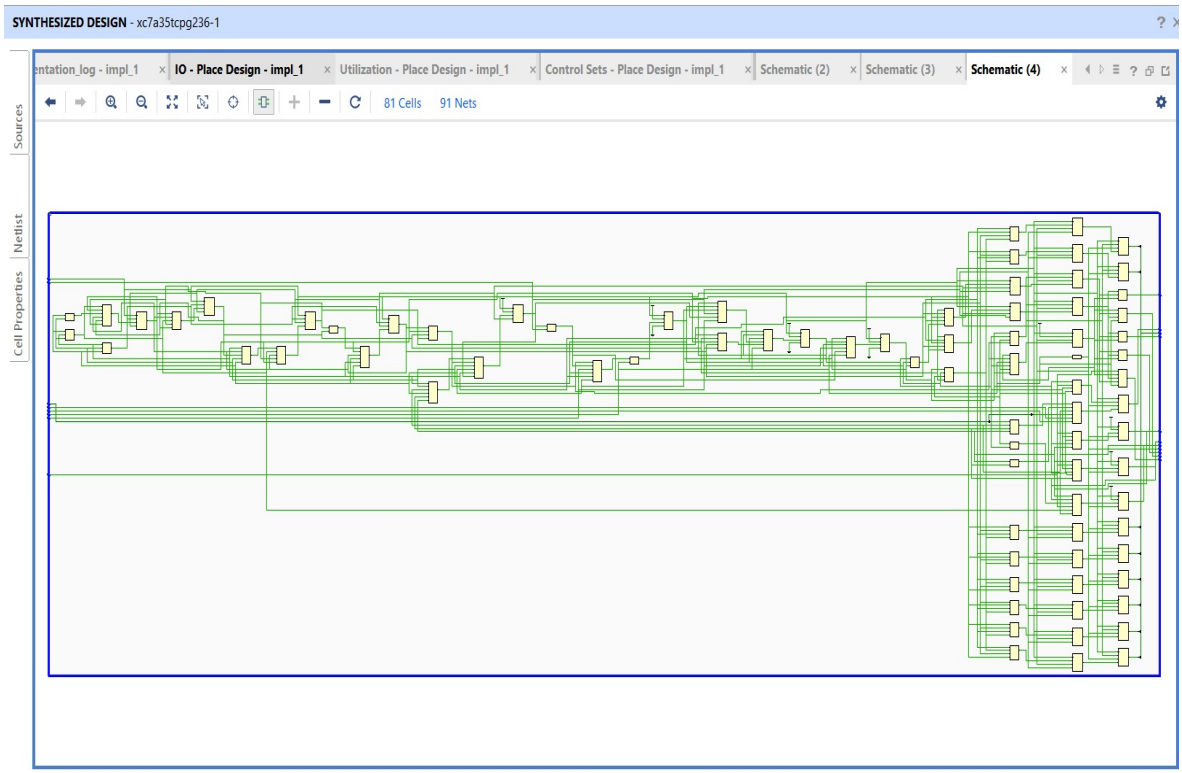
DESIGN:



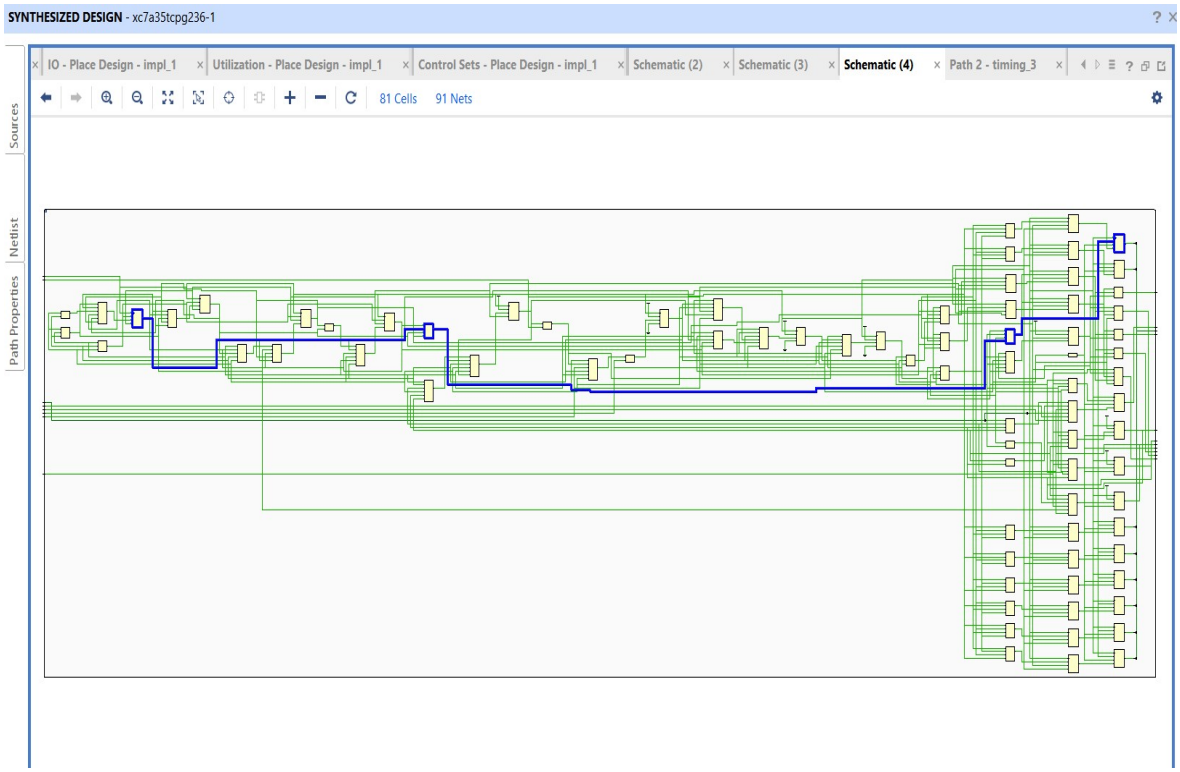
RAM BLOCK:



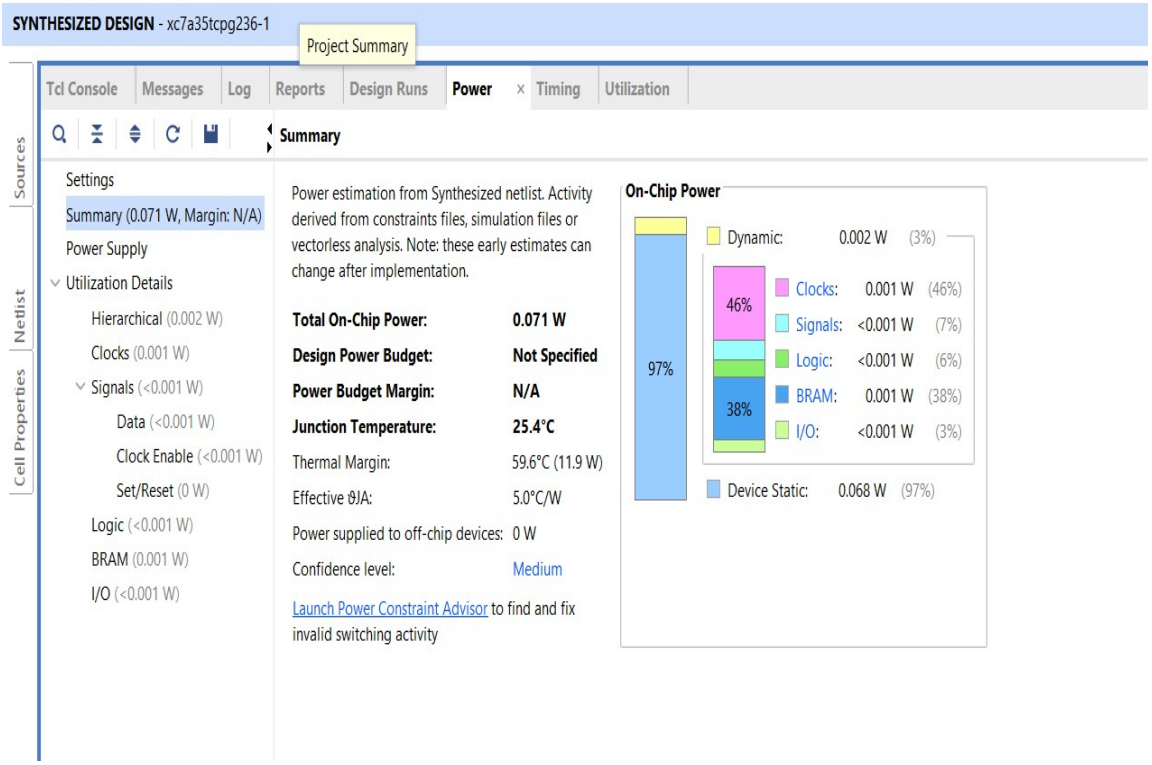
SPI_SLAVE BLOCK:



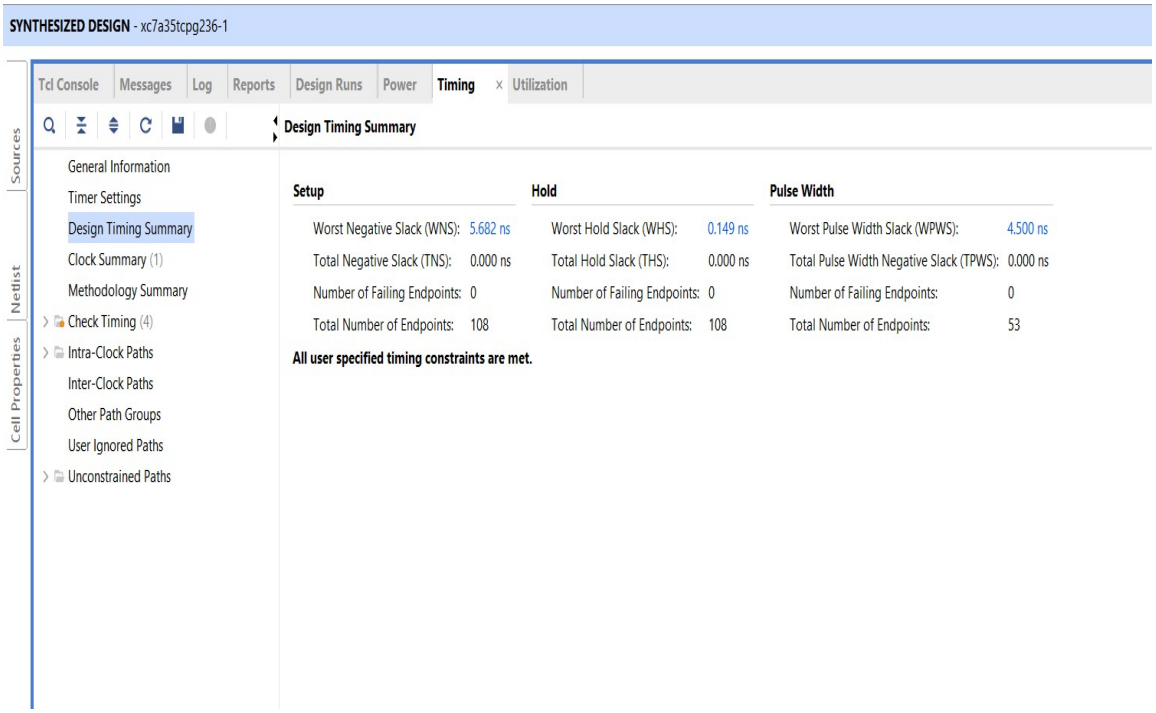
CRITICAL PATH:



POWER REPORT:



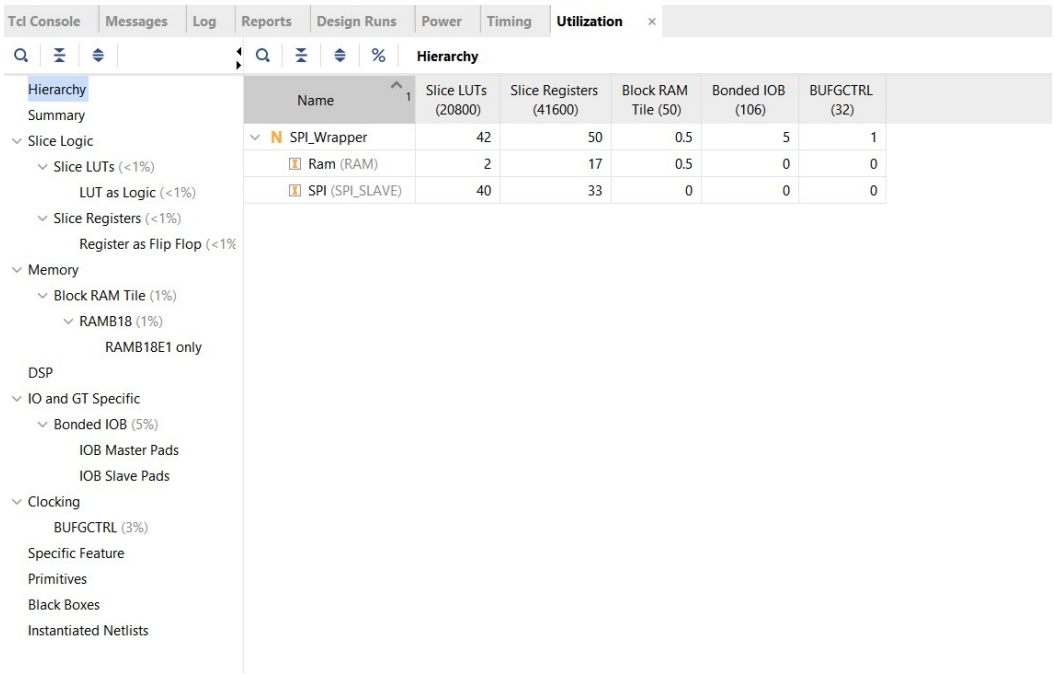
TIMING PORT:



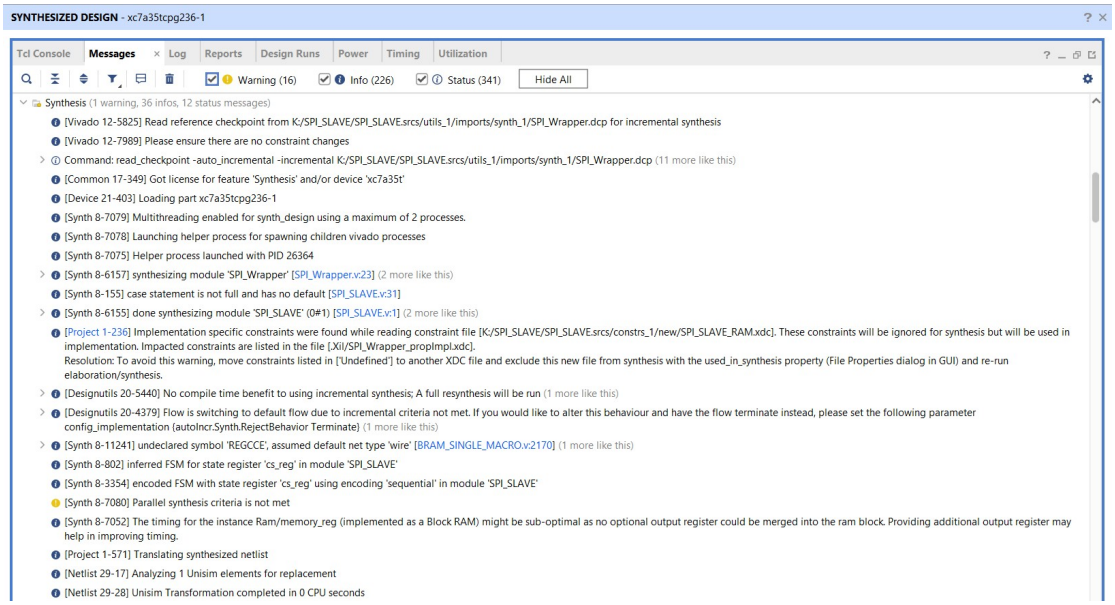
ENCODING:

```
---------------------------------------------------------------------------------
INFO: [Synth 8-802] inferred FSM for state register 'cs_reg' in module 'SPI_SLAVE'
---------------------------------------------------------------------------------------------------
                   State |                     New Encoding |                Previous Encoding
---------------------------------------------------------------------------------------------------
                    IDLE |                              000 |                             000
                  CHK_CMD |                              001 |                             001
                   WRITE |                              010 |                             010
                READ_ADD |                              011 |                             011
               READ_DATA |                              100 |                             100
---------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'cs_reg' using encoding 'sequential' in module 'SPI_SLAVE'
---------------------------------------------------------------------------------
```

UTILIZATION REPORT:

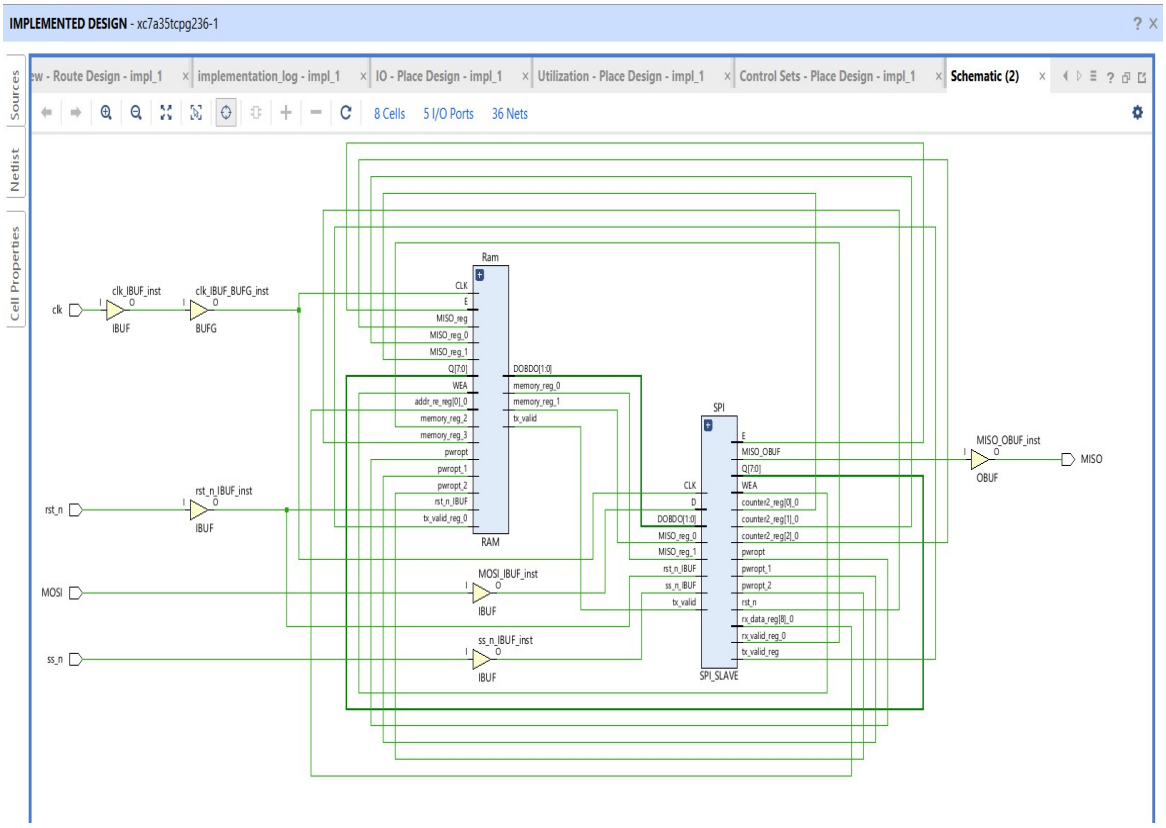


MASSAGE:

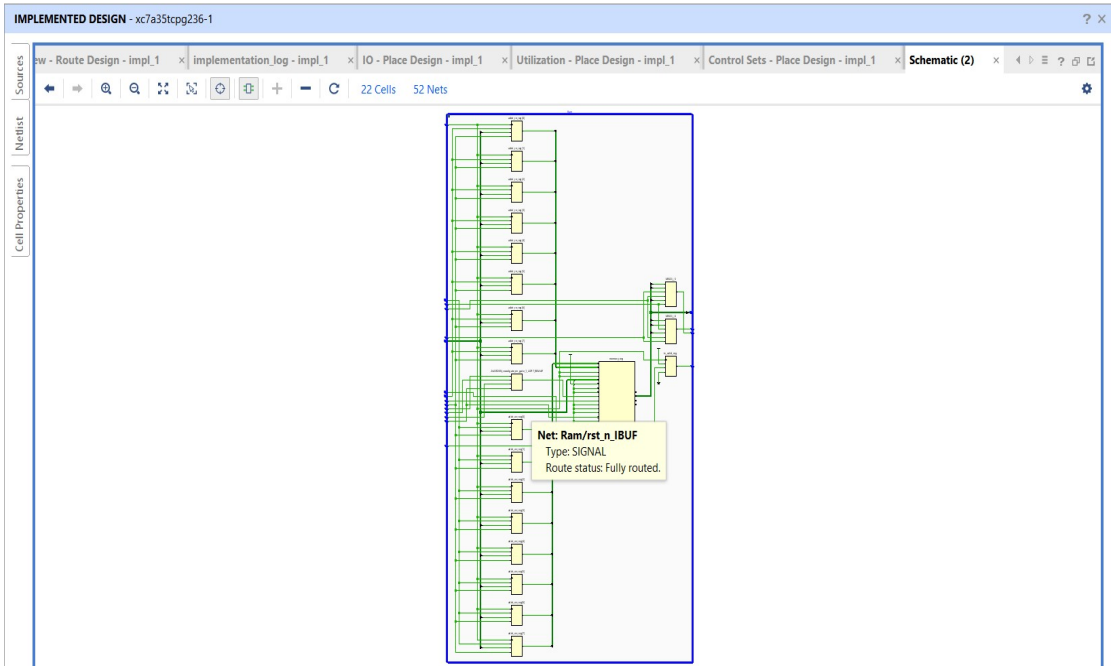


IMPLIMENTION:

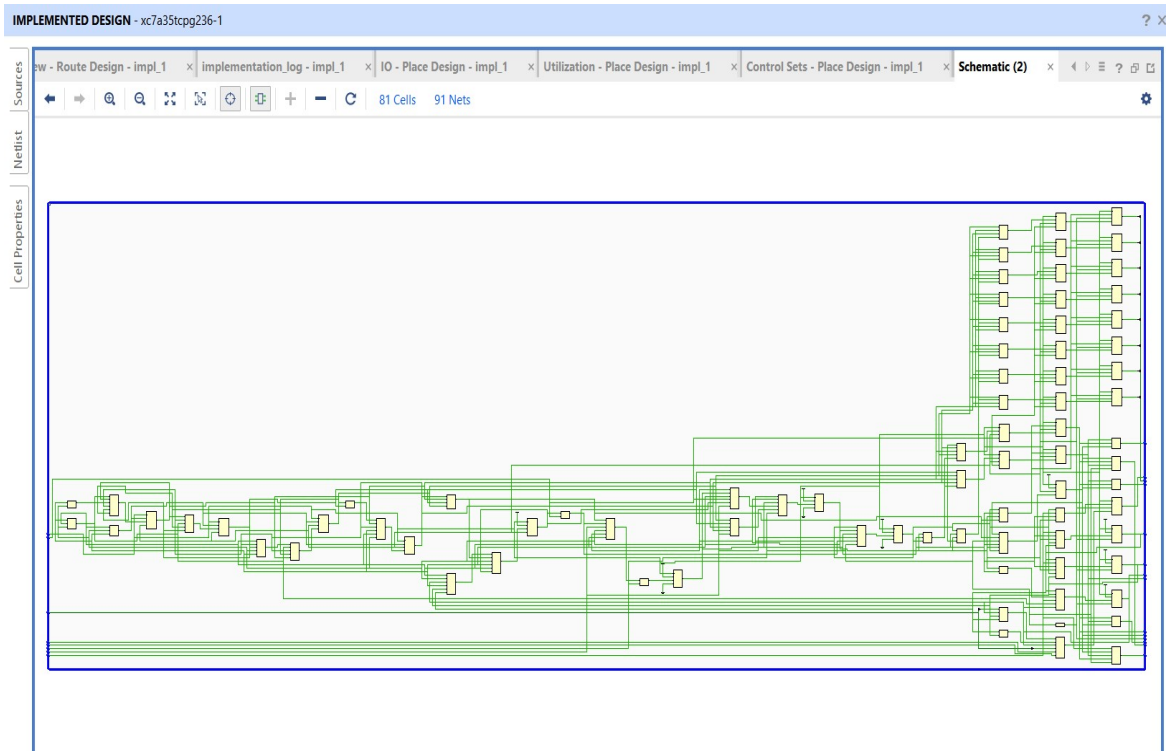
DESIGN:



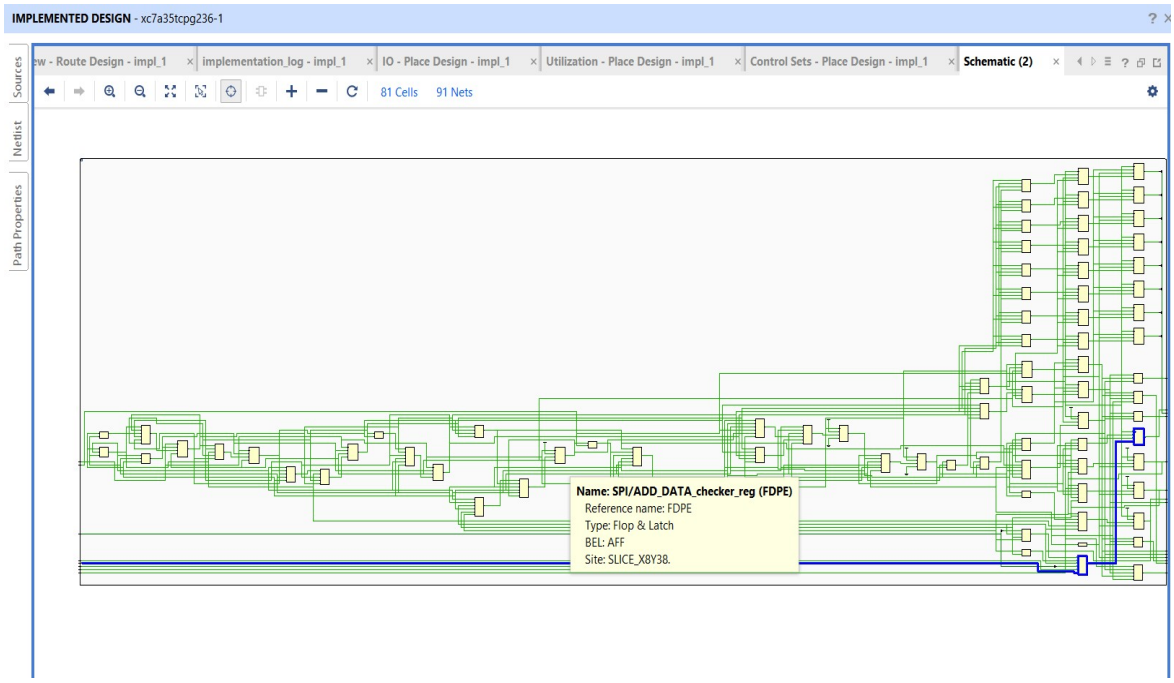
RAM BLOCK:



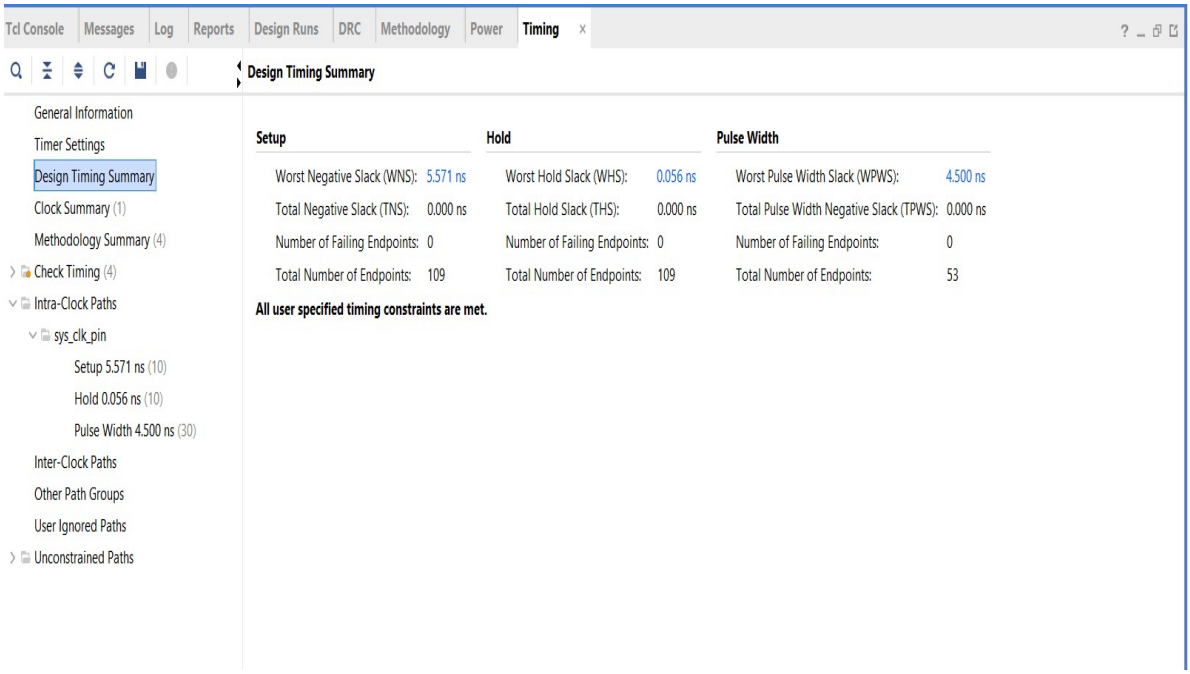
SPI SLAVE:



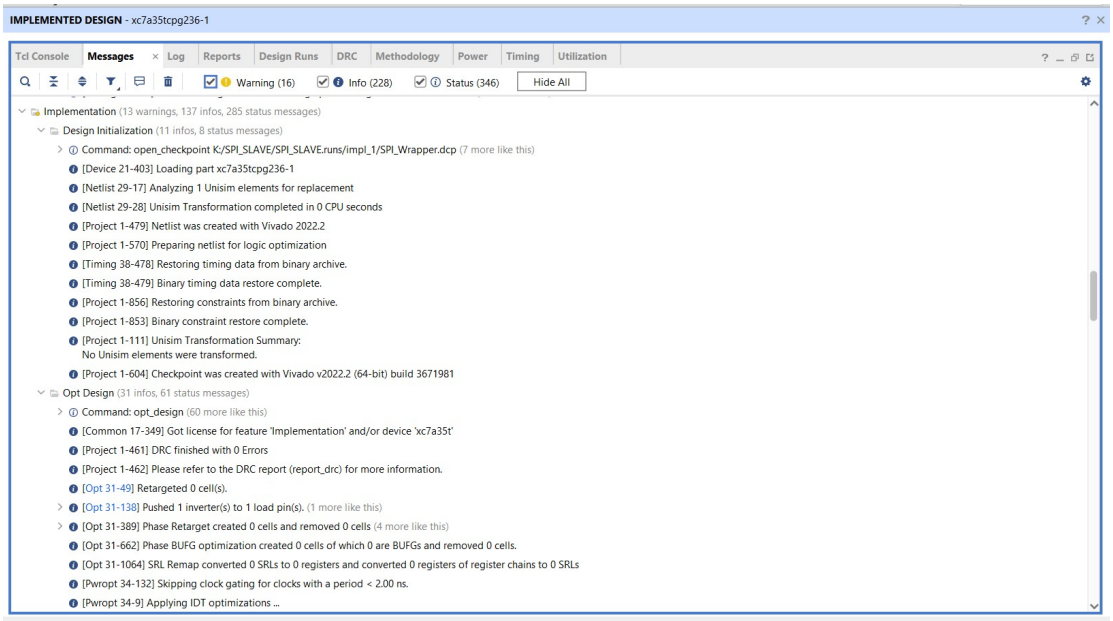
CRITICAL PATH:



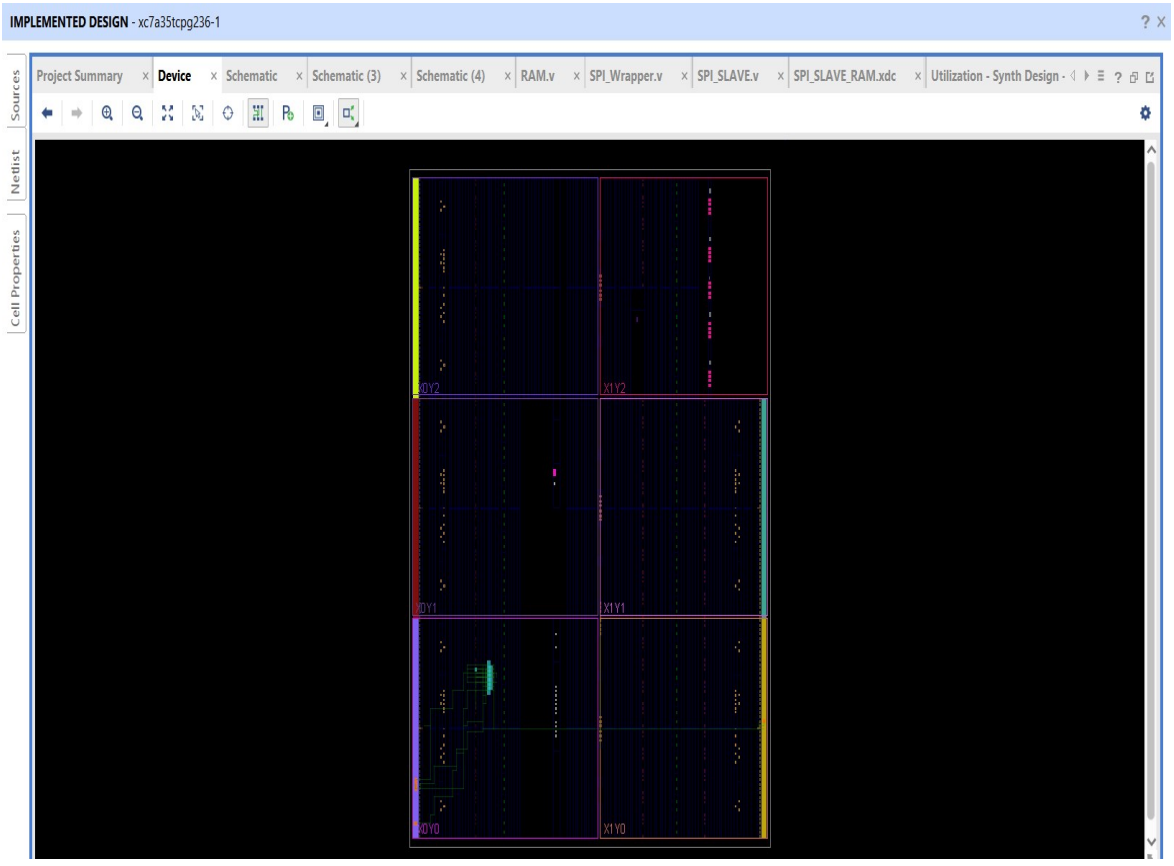
TIME REPORT:



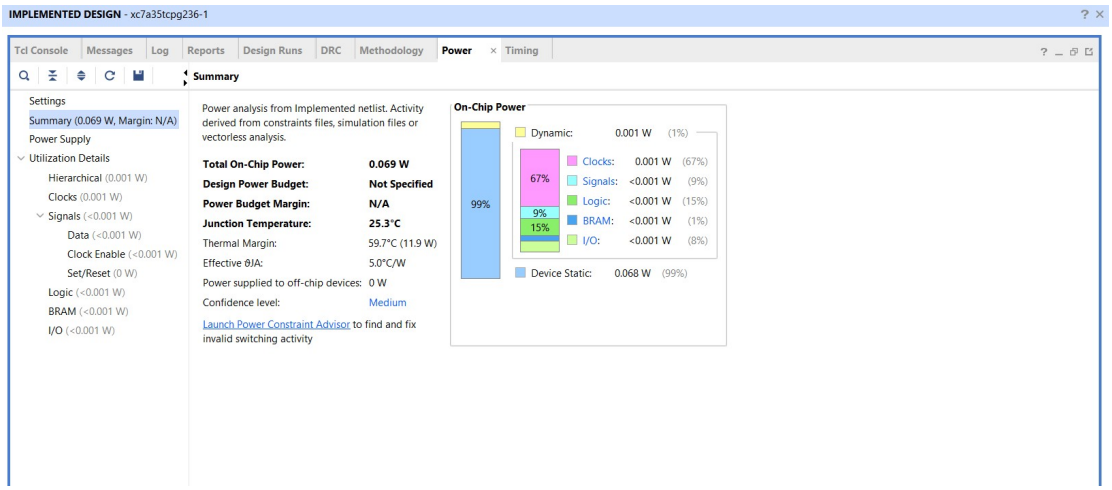
MASSSGE:



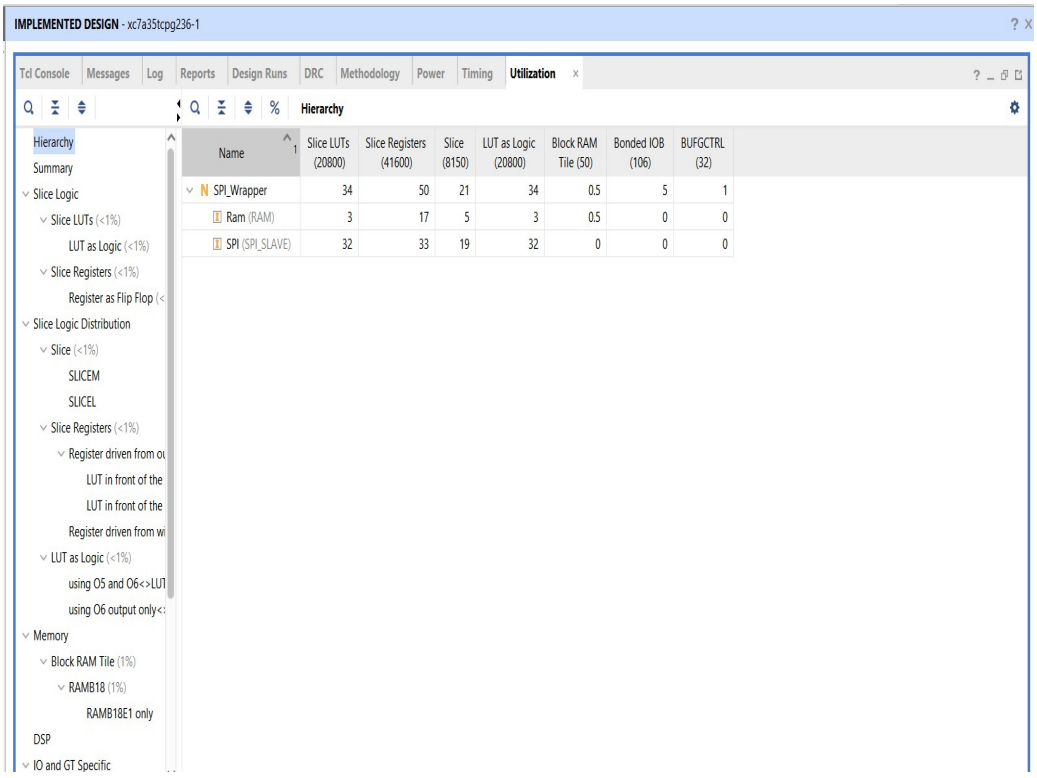
DEVICE:



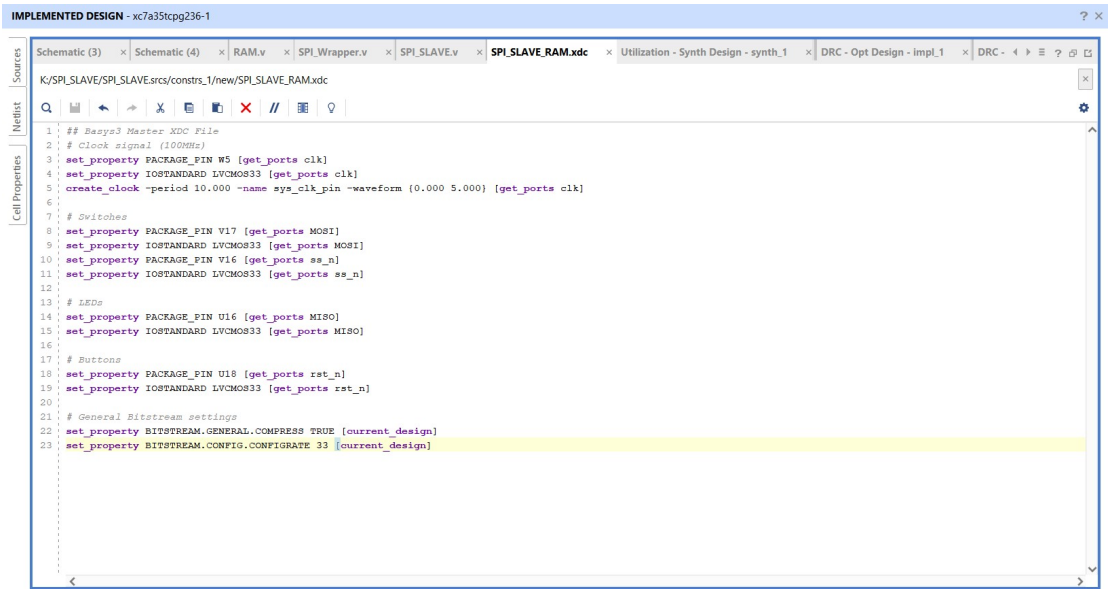
POWER REPORT:



UTILIZATION REPORT:



CONSTARIN FILE:



CONCLUSION:

The project is a robust, functioning, and resource-efficient implementation of an SPI-controlled RAM access system. The design meets its functional requirements, demonstrates correct simulation behavior, and achieves timing closure on the target FPGA, making it ready for integration into a larger system.

THANK YOU