SPI SLAVE INTERFACE

Digital design project

Done by

Saeed Maher Saeed & Sameh Mohammed Sameh

Table of Contents

Introduction

The design of the SPI Slave interface is divided into 3 parts the RAM design, SPI slave design and the top module that instantiate the RAM and SPI Slave and connect them together.

The report will cover each design with its testbench and the results of the testbench to check its functionality. So, the report will be divided into 3 parts a part for each design. In each part, you can find each design with all details and tests of the functionality.

You can find here a link with the codes and do file to run their testbenches each do file run and simulate the module testbench that is named after. The link.

RAM design:

The RTL code design:

```
module RAM(clk,rst_n,rx_valid,din,tx_valid,dout);
parameter MEM_DEPTH = 256;
parameter ADDR_SIZE = 8;
input clk,rst_n,rx_valid;
input [9:0] din;
output reg tx_valid;
output reg [7:0] dout;
reg [ADDR_SIZE-1:0]ADD_read,ADD_write;
reg [7:0] mem [MEM_DEPTH-1:0];

always@(posedge clk)
begin
    if(!rst_n) begin
        tx_valid<=0;
        dout<=0;
    end
    else begin
        if(rx_valid) begin
            case (din[9:8])
                2'b00: begin
                    tx_valid<=0;
                    ADD_write<=din[7:0];
                end
                2'b01: begin
                    tx_valid<=0;
                    mem[ADD_write]<=din[7:0];
                end
                2'b10: begin
                    tx_valid<=0;
                    ADD_read<=din[7:0];
                end
                2'b11: begin
                    tx_valid<=1;
                    dout<=mem[ADD_read];
                end
            endcase
        end
    end
end
endmodule
```

The testbench code:

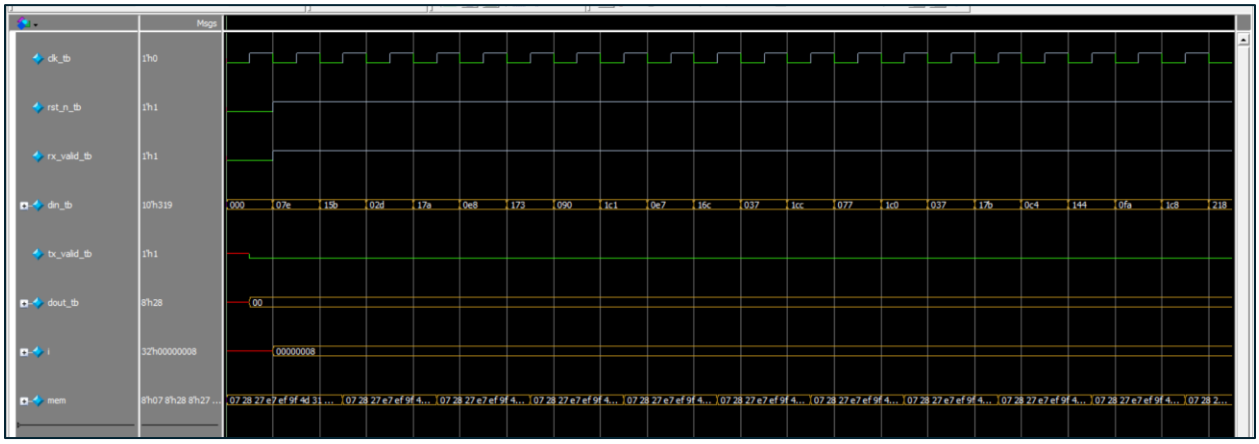
```
module RAM_tb();
reg clk_tb,rst_n_tb,rx_valid_tb;
reg [9:0] din_tb;
wire tx_valid_tb;
wire [7:0] dout_tb;

integer i;

RAM dut (clk_tb,rst_n_tb,rx_valid_tb,din_tb,tx_valid_tb,dout_tb);

initial begin
    clk_tb = 0;
    forever begin
        #10
        clk_tb = ~clk_tb;
    end
end

initial begin
    // initializing the RAM with zeros
    $readmemh("mem.dat",dut.mem);

    // check reset
    rst_n_tb=0;
    din_tb=0;
    rx_valid_tb=0;
    @(negedge clk_tb);

    rst_n_tb=1;
    rx_valid_tb=1;
    // check write with address and data
    repeat(10) begin
      // check writing the address
      din_tb[9:8] = 2'b00;
      for(i=0;i<8;i=i+1) begin
        din_tb[i]=$random;
      end
      @(negedge clk_tb);

      // check to write the data in previous address
      din_tb[9:8] = 2'b01;
      for(i=0;i<8;i=i+1) begin
        din_tb[i]=$random;
```

```
      end
      @(negedge clk_tb);
    end

    // check reading command
    repeat(10) begin
      // check reading the address
      din_tb[9:8] = 2'b10;
      for(i=0;i<8;i=i+1) begin
        din_tb[i]=$random;
      end
      @(negedge clk_tb);

      // check to read the data in previous address
      din_tb[9:8] = 2'b11;
      for(i=0;i<8;i=i+1) begin
        din_tb[i]=$random;
      end
      @(negedge clk_tb);
    end

    $stop;
end

endmodule
```

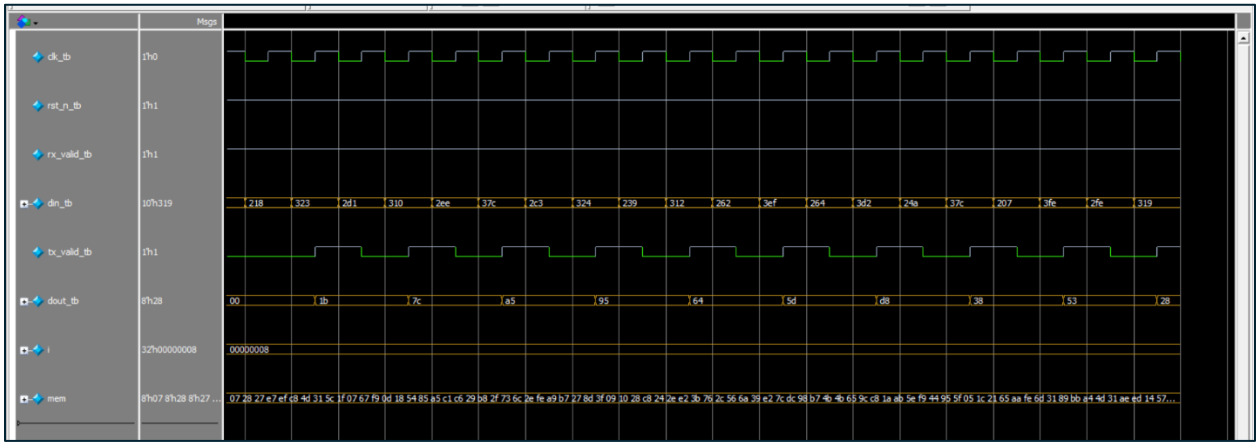
The simulation snippets:

○ waveform:

- waveform for writing process:

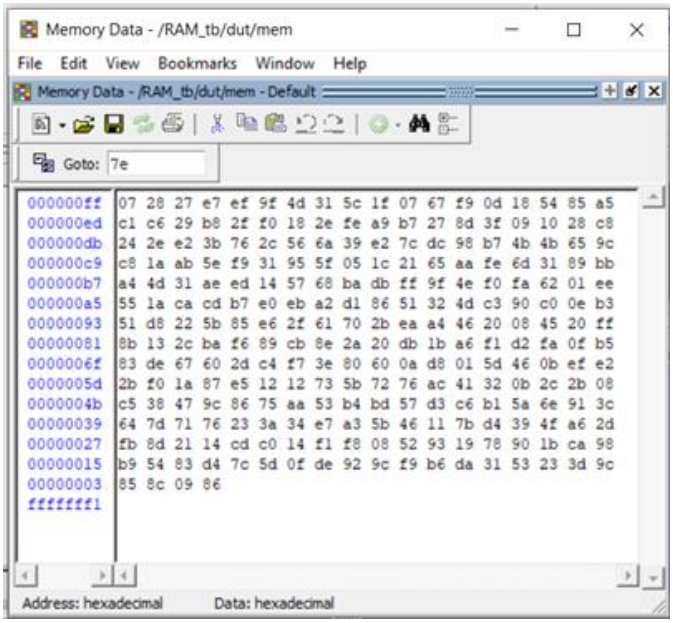


- waveform for reading process:

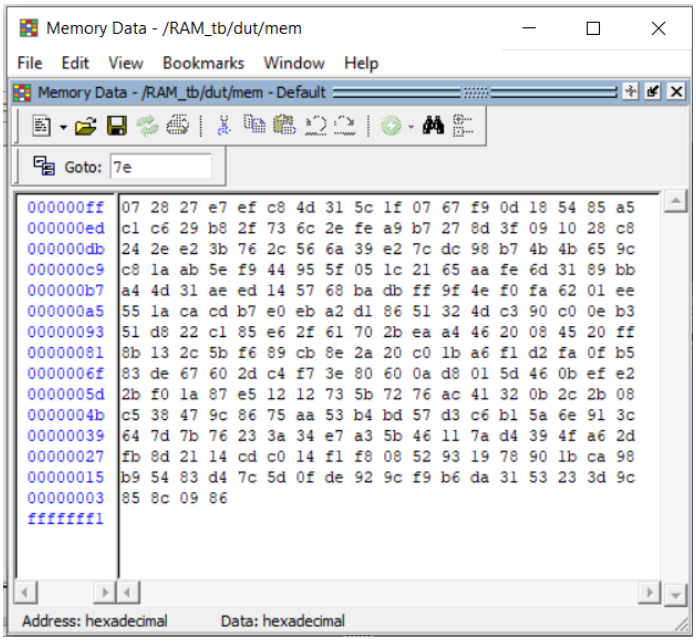


○ Memory values:

- memory initial values before running the simulation:



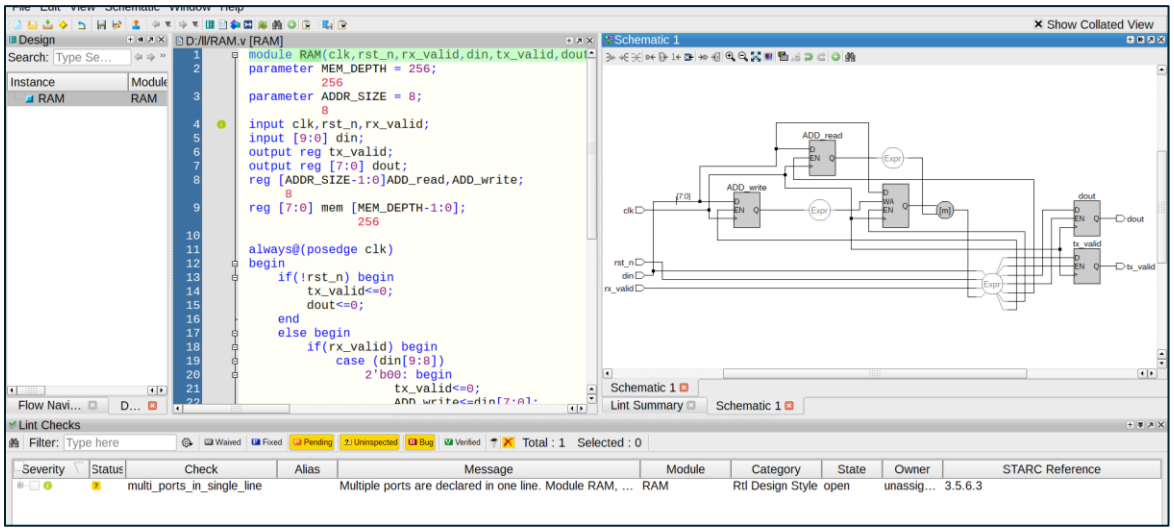
- memory values after running the simulation:



From the snippets, we can find that the data wrote in their dorrect address successfully and also read successfully.

The lint tool:

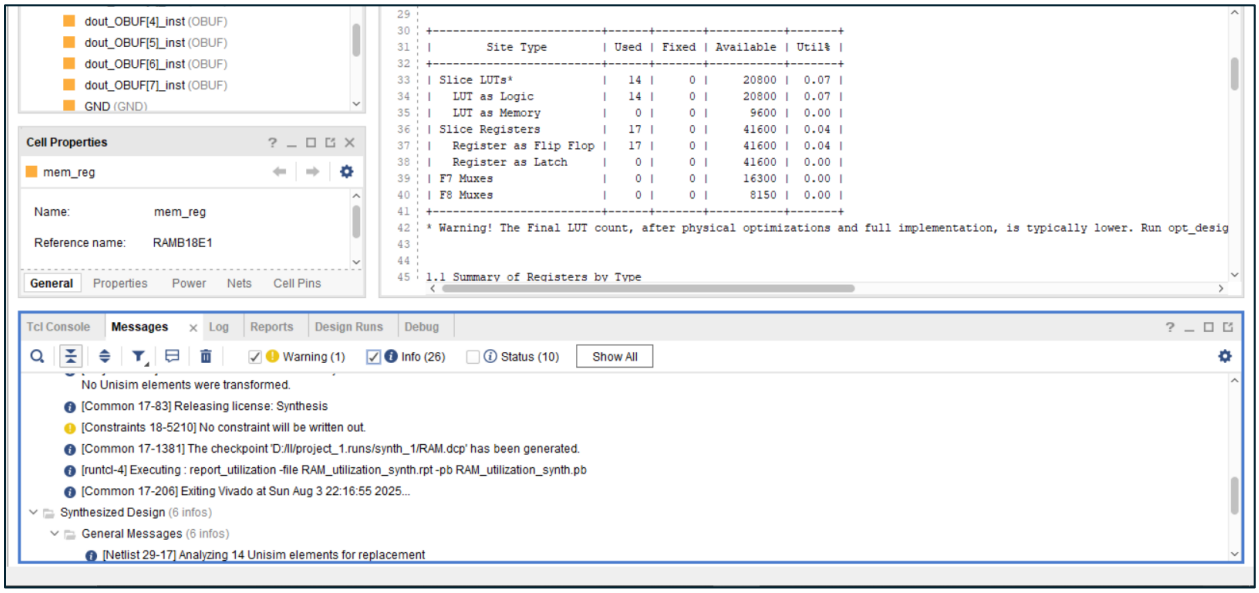
In this part, it checks that this module is synthesizable or not also if it shows the correct RTL schematic.



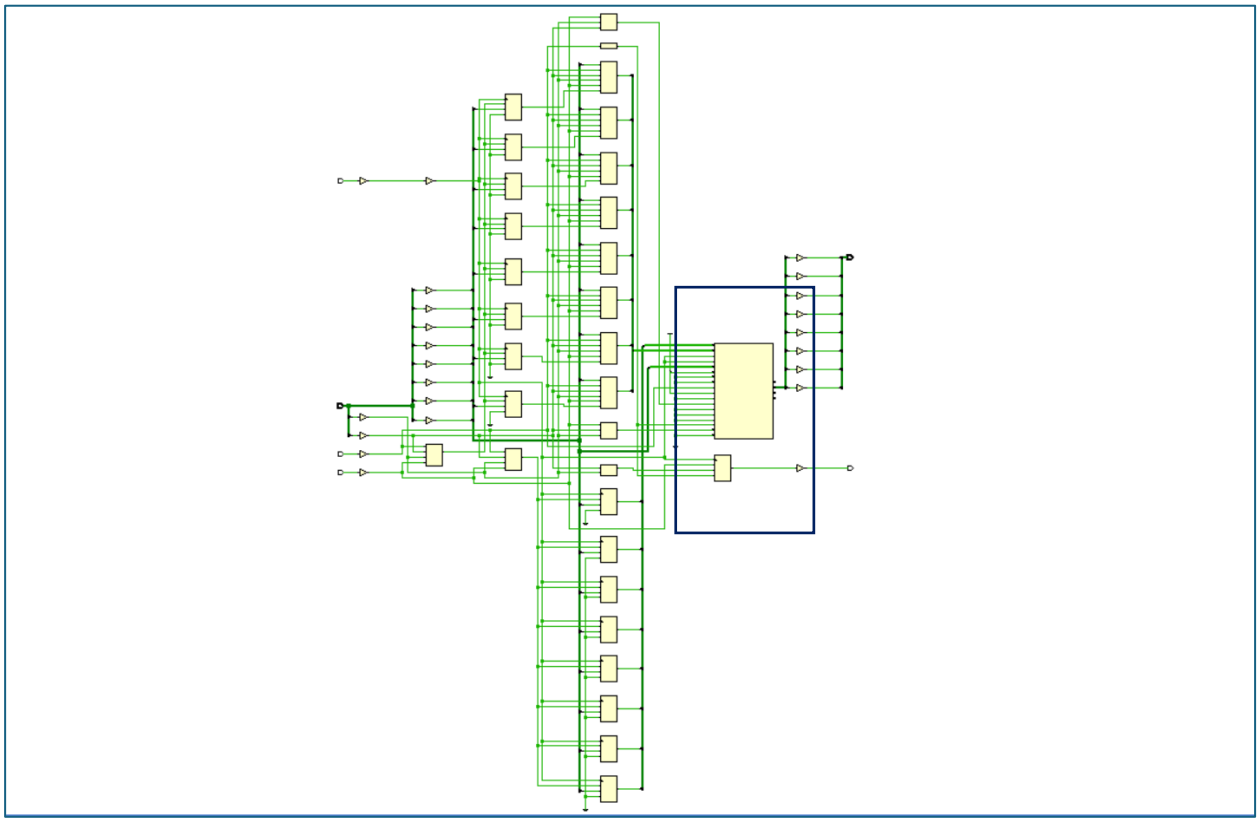
The synthesis tool:

In this part, we will see the synthesized schematic and check any error during synthesis would appear.

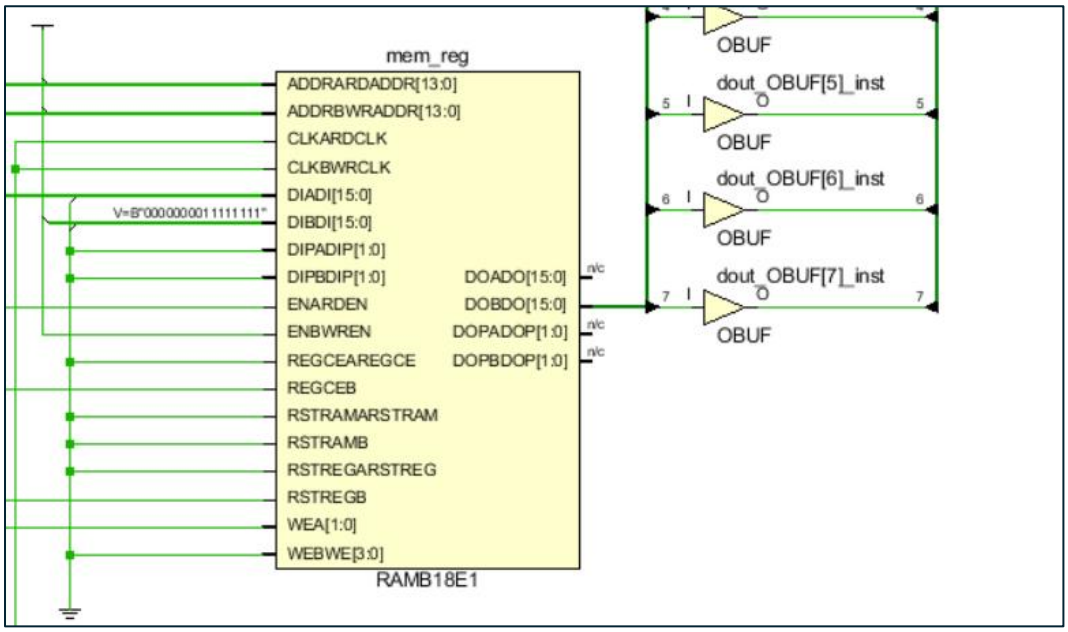
- **Check errors:**



- **The whole schematic:**



- **The zoomed in schematic “zoomed on the selected part on the picture”:**



SPI Slave design:

The RTL code design:

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

input MOSI,SS_n,clk,rst_n,tx_valid;
input [7:0]tx_data;
output reg MISO,rx_valid;
output reg [9:0]rx_data;
reg [2:0]cs,ns;
reg address_enable; // 0 means no address is written to read from while 1 means
there is address
reg [3:0]count;
localparam IDLE = 3'b000;
localparam CHK_CMD = 3'b001;
localparam WRITE = 3'b010;
localparam READ_ADD = 3'b011;
localparam READ_DATA = 3'b100;

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

always @(*) begin
    case(cs)
        IDLE: begin
          if(~SS_n) begin
            ns=CHK_CMD;
          end
          else begin
            ns=IDLE;
          end
        end

        CHK_CMD: begin
          if (SS_n) begin
            ns=IDLE;
          end
```

```
          else if (~MOSI) begin
            ns=WRITE;
          end
          else begin
            if (~address_enable) begin
                ns=READ_ADD;
            end else begin
                ns=READ_DATA;
            end
          end
        end

        WRITE: begin
          if(SS_n == 1) begin
            ns=IDLE;
          end
          else begin
            ns=WRITE;
          end
        end

        READ_ADD: begin
          if(SS_n == 1) begin
            ns=IDLE;
          end
          else begin
            ns=READ_ADD;
          end
        end

        READ_DATA: begin
          if(SS_n == 1) begin
            ns=IDLE;
          end
          else begin
            ns=READ_DATA;
          end
        end

    endcase
end

always @(posedge clk) begin
    if (~rst_n) begin
        MISO<=0;
```

```
        rx_data<=0;
        rx_valid<=0;
        count<=0;
        address_enable<=0;
    end
    else begin
        case (cs)

        IDLE: rx_valid<=0;

        CHK_CMD: count<=0;

        WRITE: begin
            if (count<10) begin
              rx_data[9-count]<=MOSI;
              count<=count+1;
            end
            else if (count == 10) begin
                rx_valid<=1;
            end
        end

        READ_ADD : begin
          if(count<10) begin
              rx_data[9-count]<=MOSI;
              count<=count+1;
          end
          else if (count==10) begin
                rx_valid<=1;
                address_enable<=1;
                count<=count+1;
          end
        end

        READ_DATA: begin
          if(rx_valid && ~tx_valid) begin
            count<=0;
          end
          else if(count<10 && ~tx_valid) begin
              rx_data[9-count]<=MOSI;
              count<=count+1;
          end
          else if (count==10 && ~tx_valid) begin
                rx_valid<=1;
          end
```

```
          else if (tx_valid && count<8) begin
            address_enable<=0;
            rx_valid<=0;
            MISO<=tx_data[7-count];
            count<=count+1;
          end
        end

        endcase
    end
end

endmodule
```

The testbench code:

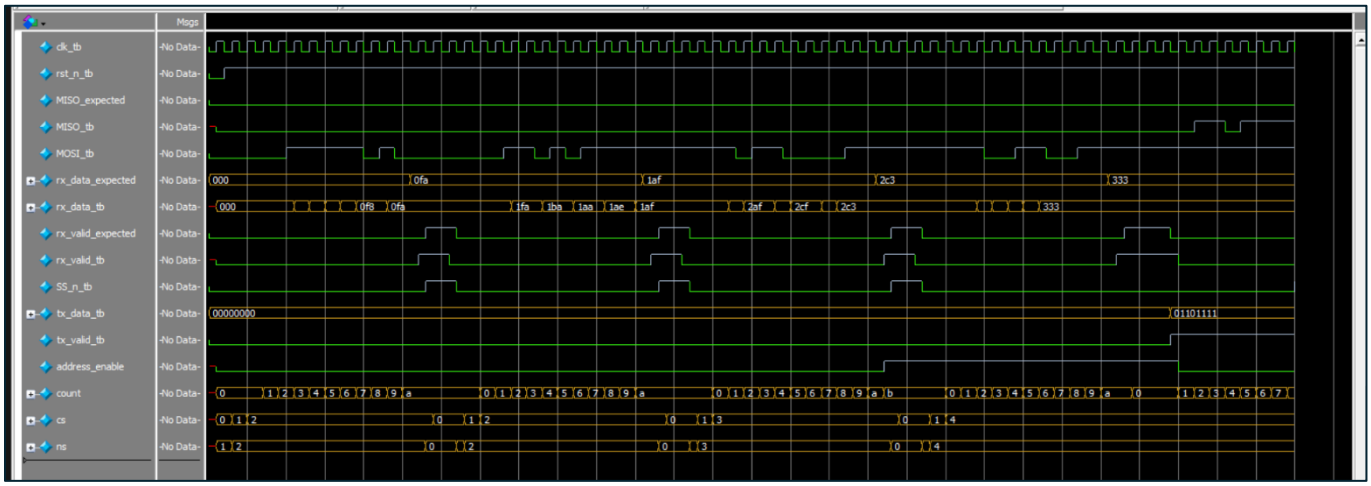
```
module SPI_slave_tb();
reg MOSI_tb,SS_n_tb,clk_tb,rst_n_tb,tx_valid_tb;
reg [7:0]tx_data_tb;
wire MISO_tb,rx_valid_tb;
wire [9:0]rx_data_tb;
reg MISO_expected,rx_valid_expected;
reg [9:0]rx_data_expected;

SPI_slave dut
(MOSI_tb,SS_n_tb,clk_tb,rst_n_tb,tx_valid_tb,tx_data_tb,MISO_tb,rx_valid_tb,rx_da
ta_tb);

initial begin
    clk_tb = 0;
    forever begin
        #10
        clk_tb = ~clk_tb;
    end
end

// used this task to sent 10 bits in serial to SPI
// the task made the test easier
task spi_send_10bits(input [9:0] data_in);
  integer i;
  begin
    for (i = 9; i >= 0; i = i - 1) begin
      MOSI_tb = data_in[i];
      @(negedge clk_tb);
```

```
    end
  end
endtask

initial begin
    // check reset
    rst_n_tb=0;
    MOSI_tb=0;
    SS_n_tb=0;
    tx_valid_tb=0;
    tx_data_tb=0;
    MISO_expected=0;
    rx_data_expected=0;
    rx_valid_expected=0;
    @(negedge clk_tb);
    if (MISO_tb != MISO_expected || rx_data_expected != rx_data_tb ||
rx_valid_expected != rx_valid_tb) begin
        $display("error in reset");
        $stop;
    end

    // CHECK giving address to write in
    rst_n_tb=1;
    repeat(2) @(negedge clk_tb); // goes to WRITE state
    spi_send_10bits(10'b00_1111_1010);
    rx_data_expected=10'b00_1111_1010;
    @(negedge clk_tb);
    rx_valid_expected=1;
    SS_n_tb=1;
    if (MISO_tb != MISO_expected || rx_data_expected != rx_data_tb ||
rx_valid_expected != rx_valid_tb) begin
        $display("error in write address command");
        $stop;
    end
    @(negedge clk_tb); // enter the IDLE state
    @(negedge clk_tb); // give the rx_valid output to execute in IDLE
    rx_valid_expected=0;
    if (rx_valid_expected != rx_valid_tb) begin
        $display("error in backing to IDLE from WRITE address");
        $stop;
    end


    // check writing data
    SS_n_tb=0;
```

```
    MOSI_tb=0;
    repeat(2) @(negedge clk_tb); // goes to WRITE state
    spi_send_10bits(10'b01_1010_1111);
    rx_data_expected=10'b01_1010_1111;
    @(negedge clk_tb);
    rx_valid_expected=1;
    SS_n_tb=1;
    if (MISO_tb != MISO_expected || rx_data_expected != rx_data_tb ||
rx_valid_expected != rx_valid_tb) begin
        $display("error in write data command");
        $stop;
    end
    @(negedge clk_tb); // enter the IDLE state
    @(negedge clk_tb); // give the rx_valid output to execute in IDLE
    rx_valid_expected=0;
    if (rx_valid_expected != rx_valid_tb) begin
        $display("error in backing to IDLE from WRTIE data");
        $stop;
    end

    // check giving reading address
    SS_n_tb=0;
    MOSI_tb=1;
    repeat(2) @(negedge clk_tb); // goes to READ_ADD state
    spi_send_10bits(10'b10_1100_0011);
    rx_data_expected=10'b10_1100_0011;
    @(negedge clk_tb);
    rx_valid_expected=1;
    SS_n_tb=1;
    if (MISO_tb != MISO_expected || rx_data_expected != rx_data_tb ||
rx_valid_expected != rx_valid_tb) begin
        $display("error in read address command");
        $stop;
    end
    @(negedge clk_tb); // enter the IDLE state
    @(negedge clk_tb); // give the rx_valid output to execute in IDLE
    rx_valid_expected=0;
    if (rx_valid_expected != rx_valid_tb) begin
        $display("error in backing to IDLE from READ ADD");
        $stop;
    end

    // check read data of the address given
    SS_n_tb=0;
    MOSI_tb=1;
```

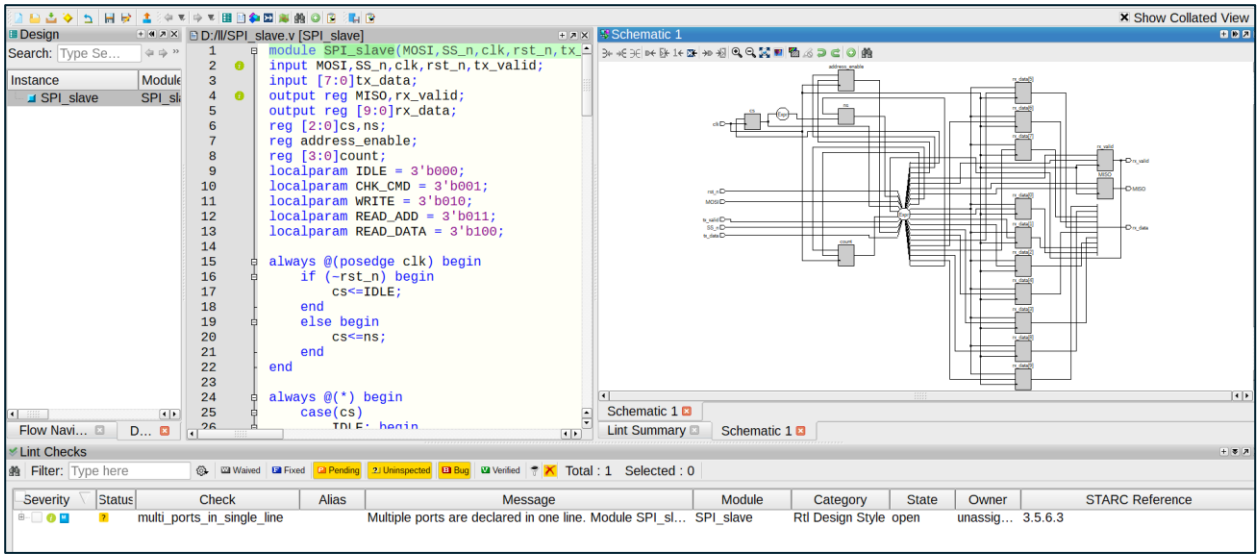
```
    repeat(2) @(negedge clk_tb); // goes to READ_DATA state
    spi_send_10bits(10'b11_0011_0011);
    rx_data_expected=10'b11_0011_0011;
    @(negedge clk_tb);
    rx_valid_expected=1;
    @(negedge clk_tb);
    if (MISO_tb != MISO_expected || rx_data_expected != rx_data_tb ||
rx_valid_expected != rx_valid_tb) begin
        $display("error in read data command");
        $stop;
    end
    repeat(2) @(negedge clk_tb); // equivalent to the number of cycles till RAM
gives tx_valid "1"
    tx_valid_tb=1;
    tx_data_tb=8'b0110_1111;
    rx_valid_expected=0;
    repeat(8) @(negedge clk_tb);

    SS_n_tb=1; // back to idle

    $stop;
end
endmodule
```

The simulation snippets:



The lint tool:

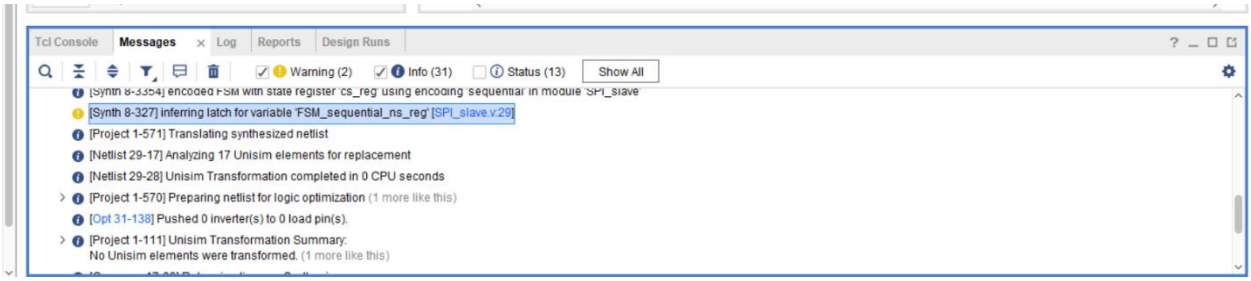
In this part, it checks that this module is synthesizable or not also if it shows the correct RTL schematic.



The synthesis tool:

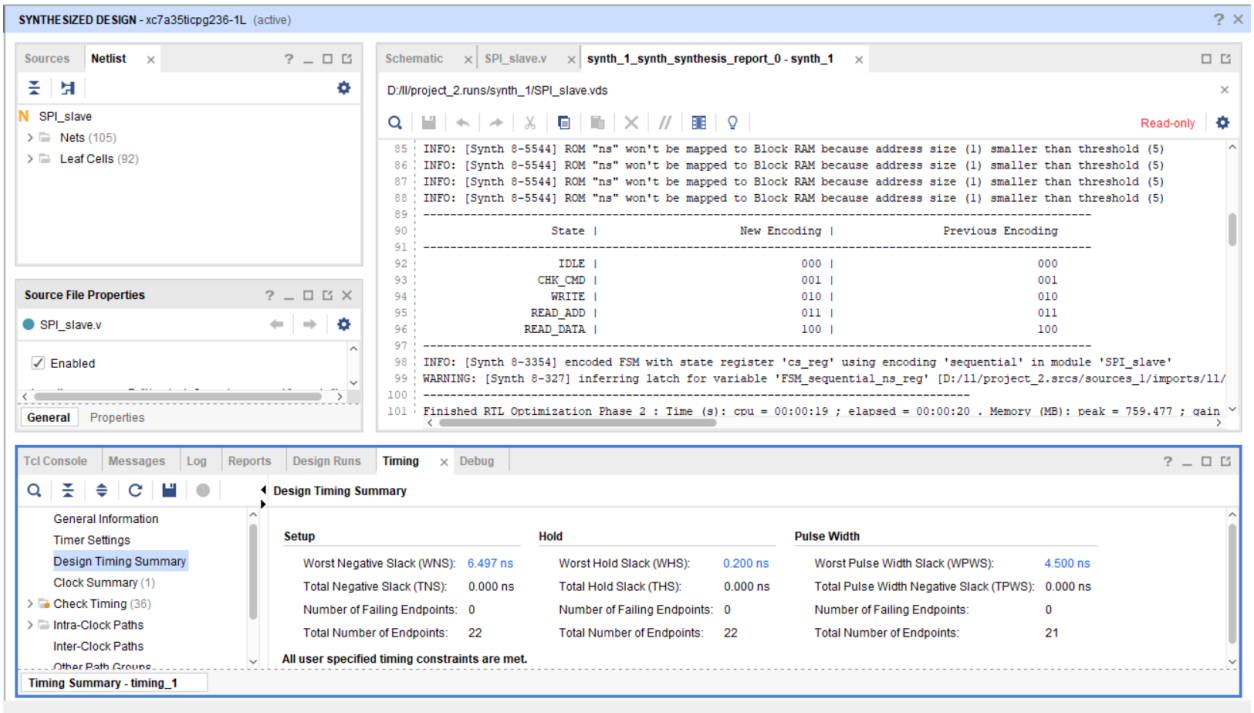
In this part, we will see the synthesized schematic and check any error during synthesis would appear. Also, we can see the timing report of each state encoding type to decide which one has highest frequency.

- **Check errors:**

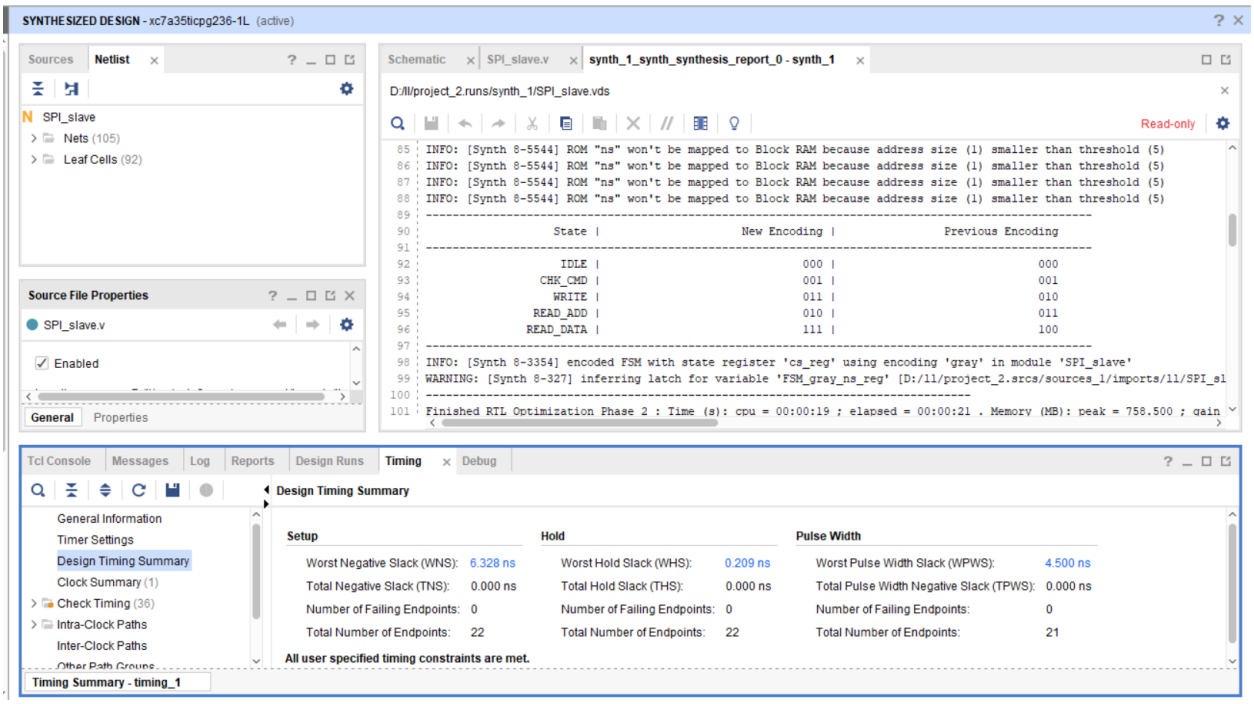


○ timing reports:

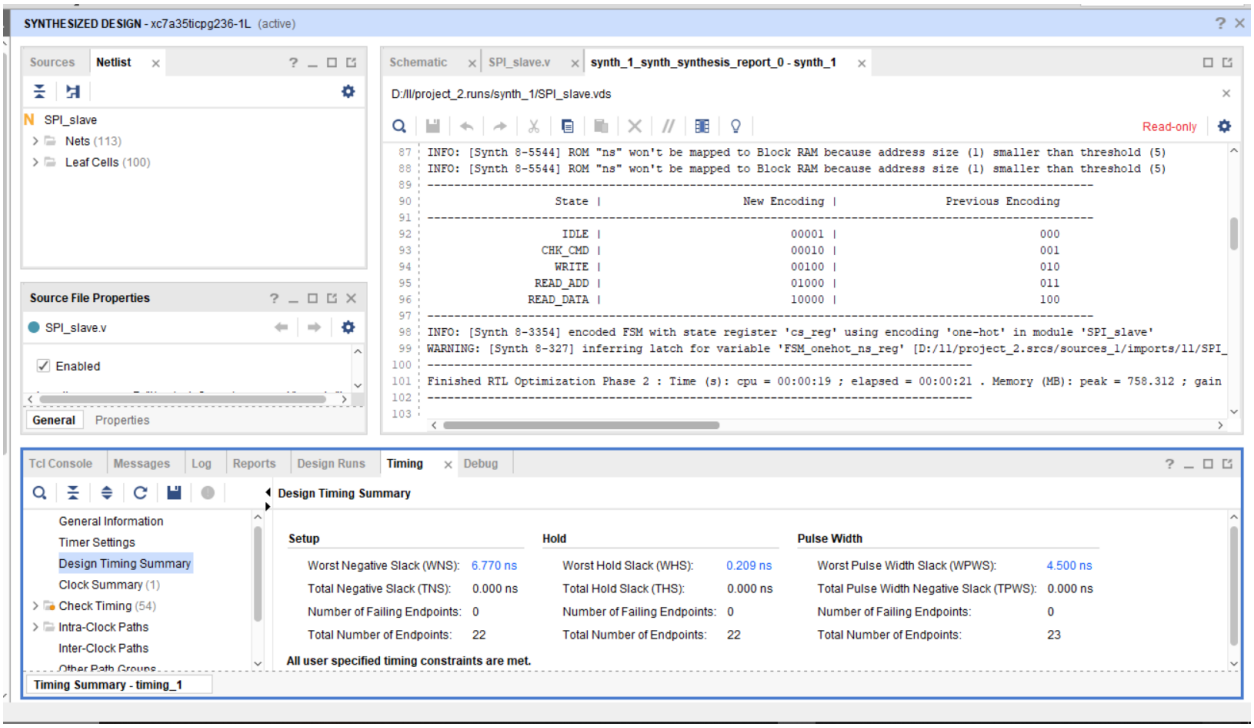
○ sequential encoding:



○ gray encoding:

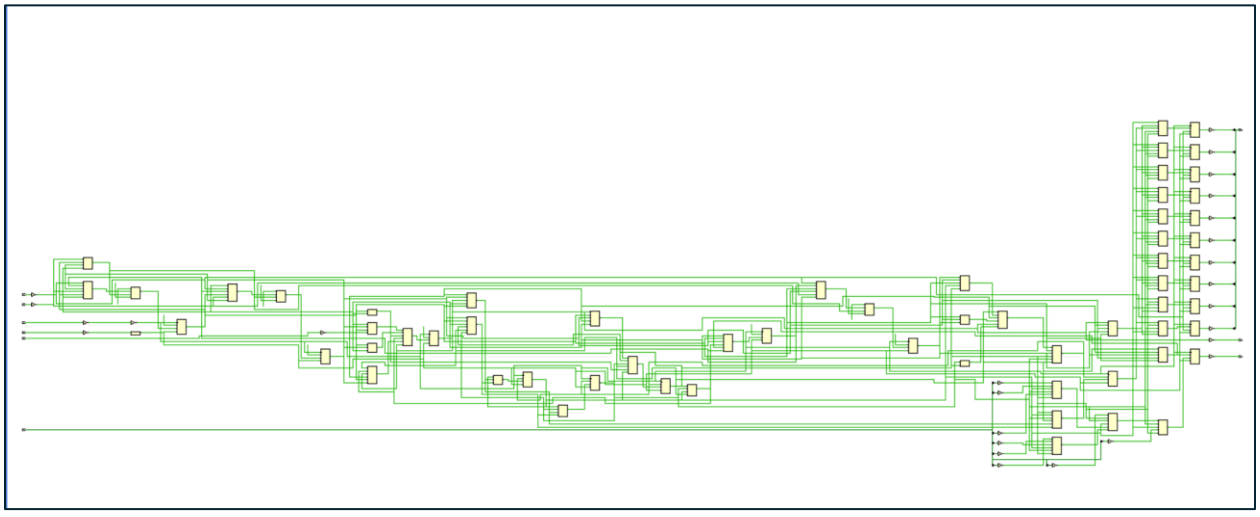


- one hot encoding:



From above figures, we can see that the one hot encoding gives highest frequency as it gives the highest setup slack timing.

- **Schematic:**



Top module:

The RTL code design:

```
module top_module_SPI(MOSI,MISO,SS_n,clk,rst_n);
input clk,rst_n,SS_n,MOSI;
output MISO;
wire tx_valid,rx_valid;
wire [9:0]rx_data;
wire [7:0]tx_data;

RAM RAM_block (clk,rst_n,rx_valid,rx_data,tx_valid,tx_data);
SPI_slave SPI_block (MOSI,SS_n,clk,rst_n,tx_valid,tx_data,MISO,rx_valid,rx_data);

endmodule
```

The testbench code:

The testbench is similar to the one of the SPI Slave but without forcing some input values that the RAM will make them like “tx_valid & tx_data”.

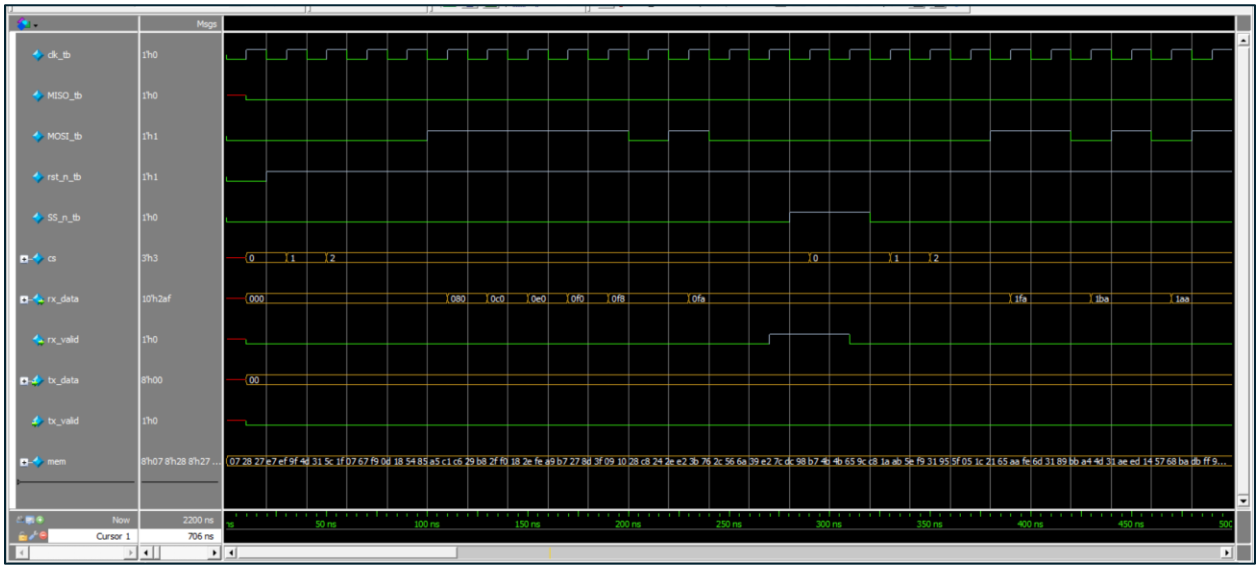
```
module top_module_SPI_tb();
reg clk_tb,rst_n_tb,SS_n_tb,MOSI_tb;
wire MISO_tb;

top_module_SPI dut (MOSI_tb,MISO_tb,SS_n_tb,clk_tb,rst_n_tb);

initial begin
    clk_tb = 0;
    forever begin
        #10
        clk_tb = ~clk_tb;
    end
end

task spi_send_10bits(input [9:0] data_in);
  integer i;
  begin
    for (i = 9; i >= 0; i = i - 1) begin
      MOSI_tb = data_in[i];
      @(negedge clk_tb);
    end
  end
```

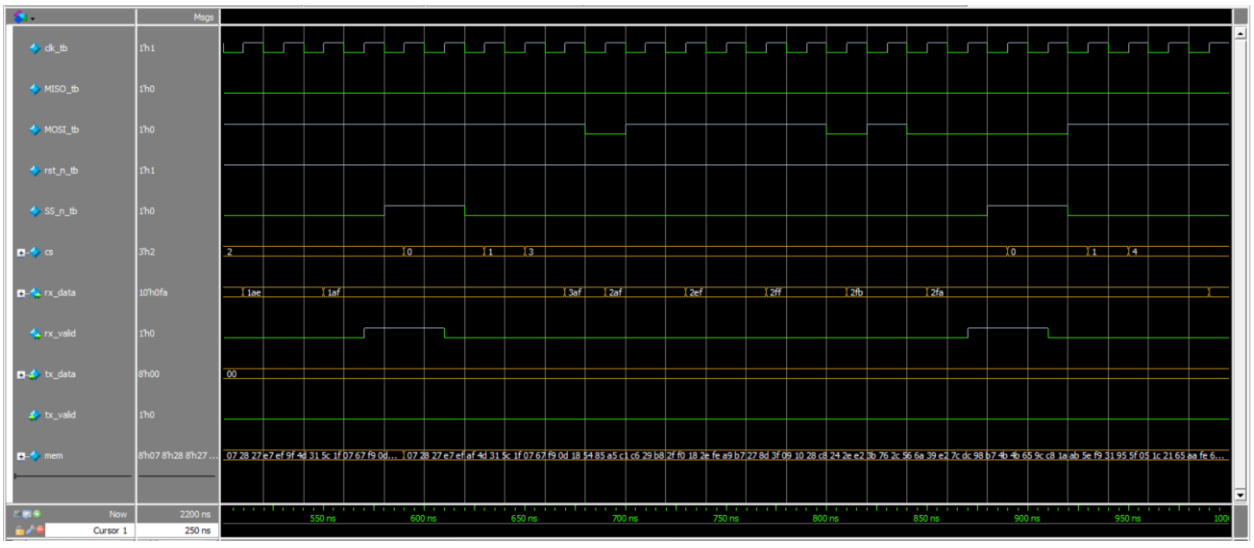
```
endtask

initial begin
    // intializing the RAM with zeros
    $readmemh("mem.dat",dut.RAM_block.mem);

    // check reset
    rst_n_tb=0;
    MOSI_tb=0;
    SS_n_tb=0;
    @(negedge clk_tb);


    // CHECK giving address to write in
    rst_n_tb=1;
    repeat(2) @(negedge clk_tb);
    spi_send_10bits(10'b00_1111_1010);
    @(negedge clk_tb);
    SS_n_tb=1;
    @(negedge clk_tb);
    @(negedge clk_tb);

    // check writing data
    SS_n_tb=0;
    MOSI_tb=0;
    repeat(2) @(negedge clk_tb);
    spi_send_10bits(10'b01_1010_1111);
    @(negedge clk_tb);
    SS_n_tb=1;
    @(negedge clk_tb);
    @(negedge clk_tb);

    // check giving read address
    SS_n_tb=0;
    MOSI_tb=1;
    repeat(2) @(negedge clk_tb);
    spi_send_10bits(10'b10_1111_1010);
    @(negedge clk_tb);
    SS_n_tb=1;
    @(negedge clk_tb);
    @(negedge clk_tb);

    // check to read the written data before
    SS_n_tb=0;
    MOSI_tb=1;
```

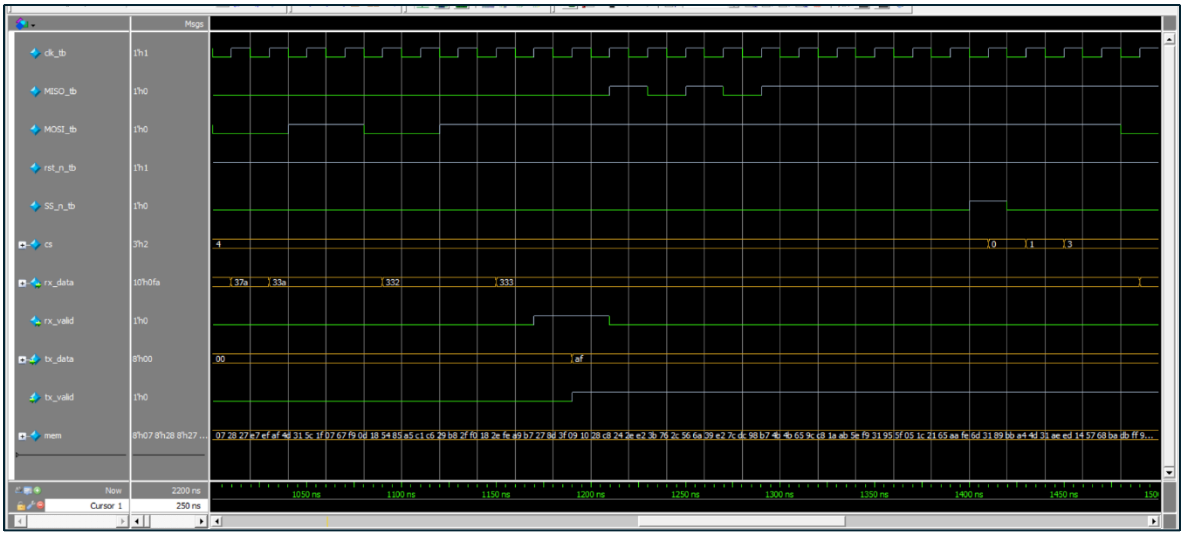
```
    repeat(2) @(negedge clk_tb);
    spi_send_10bits(10'b11_0011_0011);
    @(negedge clk_tb);
    @(negedge clk_tb);
    repeat(2) @(negedge clk_tb);
    repeat(8) @(negedge clk_tb);

    SS_n_tb=1;
    @(negedge clk_tb);

    // check to read another data saved before in memory
    // first read address state
    SS_n_tb=0;
    MOSI_tb=1;
    repeat(2) @(negedge clk_tb);
    spi_send_10bits(10'b10_0011_0011);
    @(negedge clk_tb);
    SS_n_tb=1;
    @(negedge clk_tb);
    @(negedge clk_tb);

    // second read data state
    SS_n_tb=0;
    MOSI_tb=1;
    repeat(2) @(negedge clk_tb);
    spi_send_10bits(10'b11_0011_0011);
    @(negedge clk_tb);
    @(negedge clk_tb);
    repeat(2) @(negedge clk_tb);
    repeat(8) @(negedge clk_tb);

    $stop;
end


endmodule
```

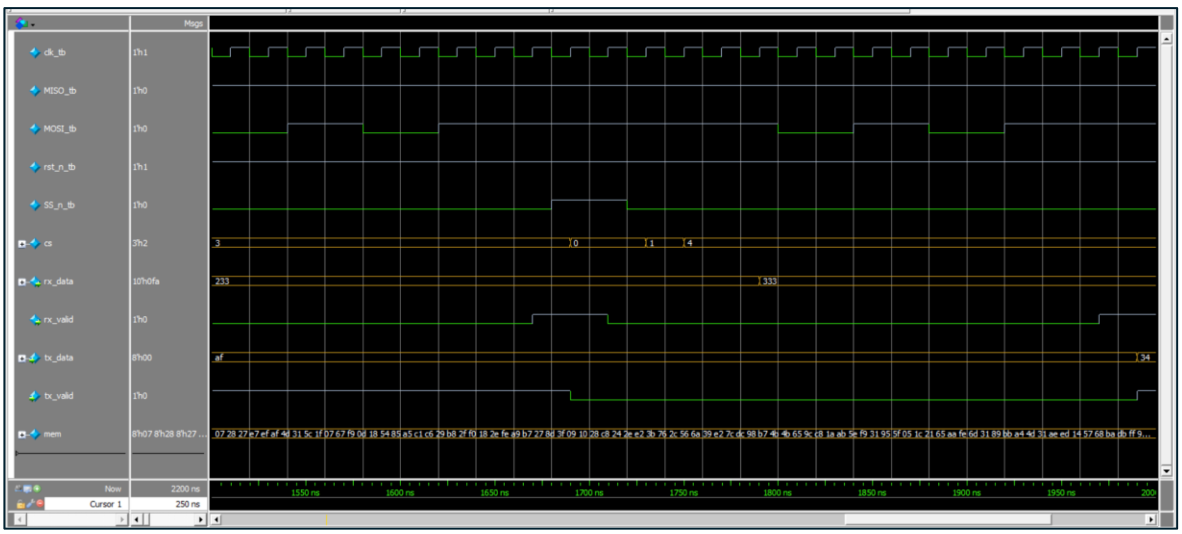
The simulation snippets:

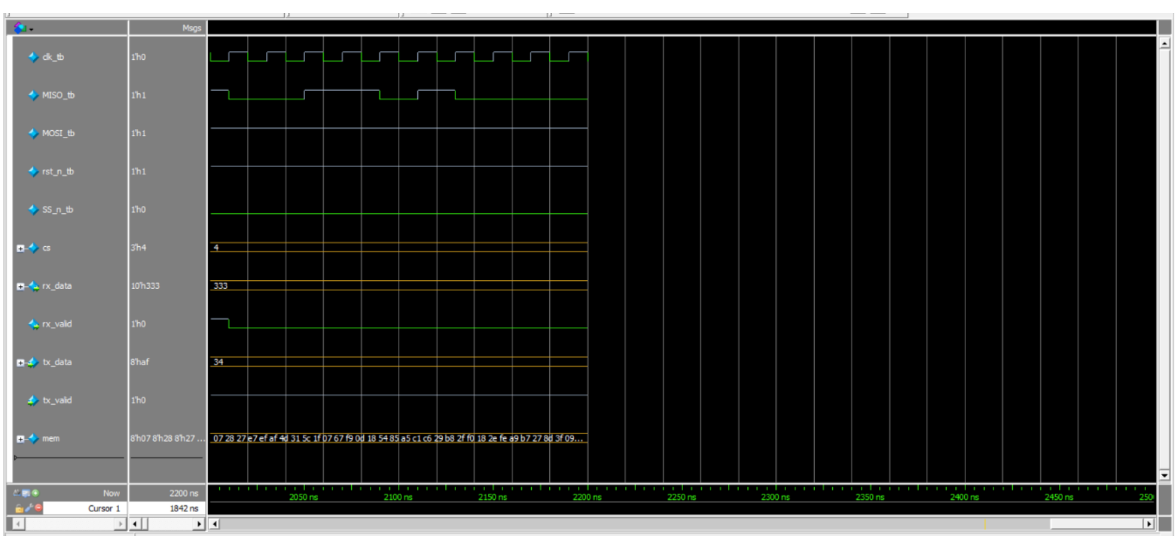
- **waveform:**





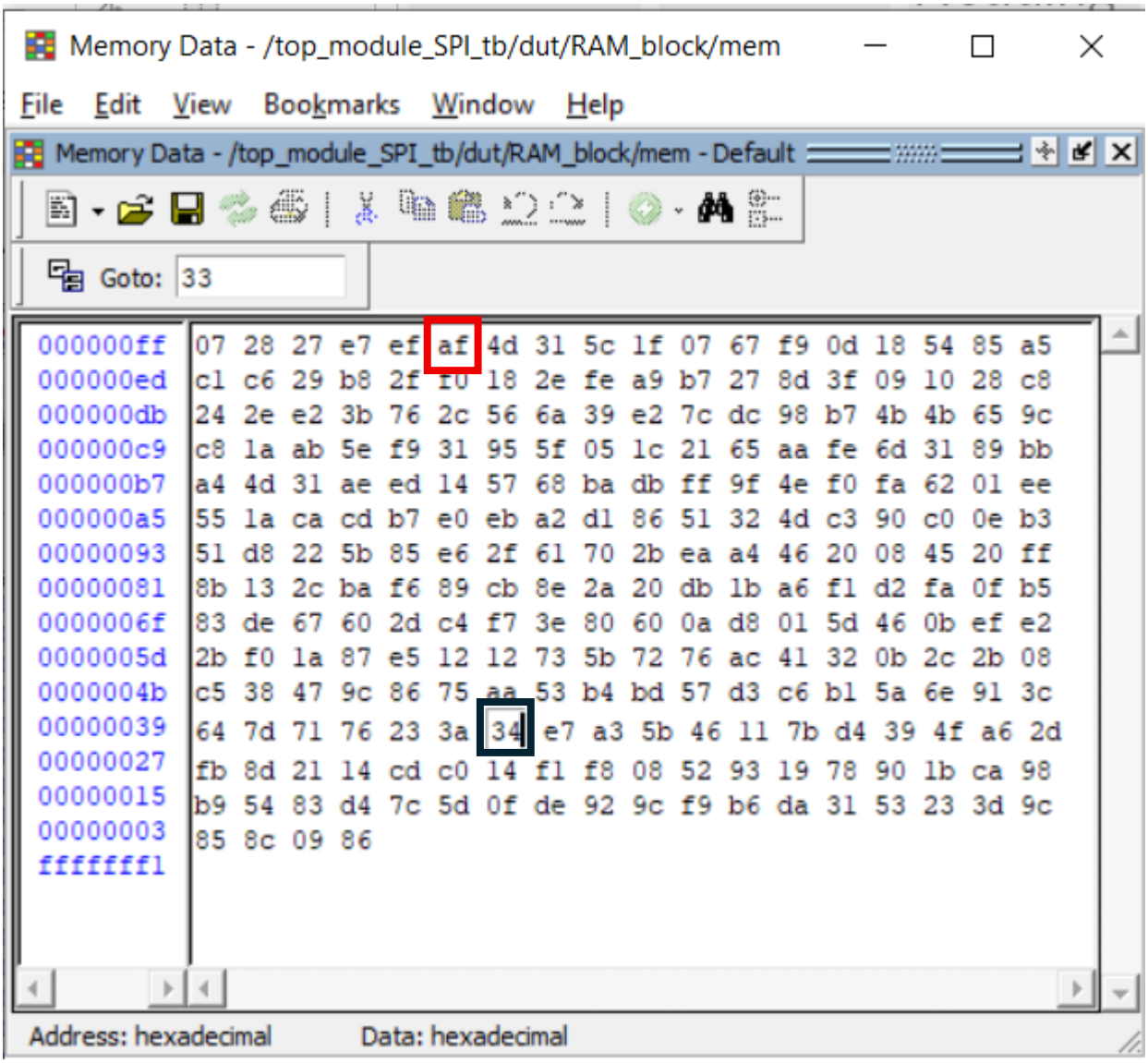

clk_tb
MISO_tb
MOSI_tb
rst_n_tb
SS_n_tb
cs
rx_data
rx_valid
tx_data
tx_valid
mem


clk_tb
MISO_tb
MOSI_tb
rst_n_tb
SS_n_tb
cs
rx_data
rx_valid
tx_data
tx_valid
mem


clk_tb
MISO_tb
MOSI_tb
rst_n_tb
SS_n_tb
cs
rx_data
rx_valid
tx_data
tx_valid
mem

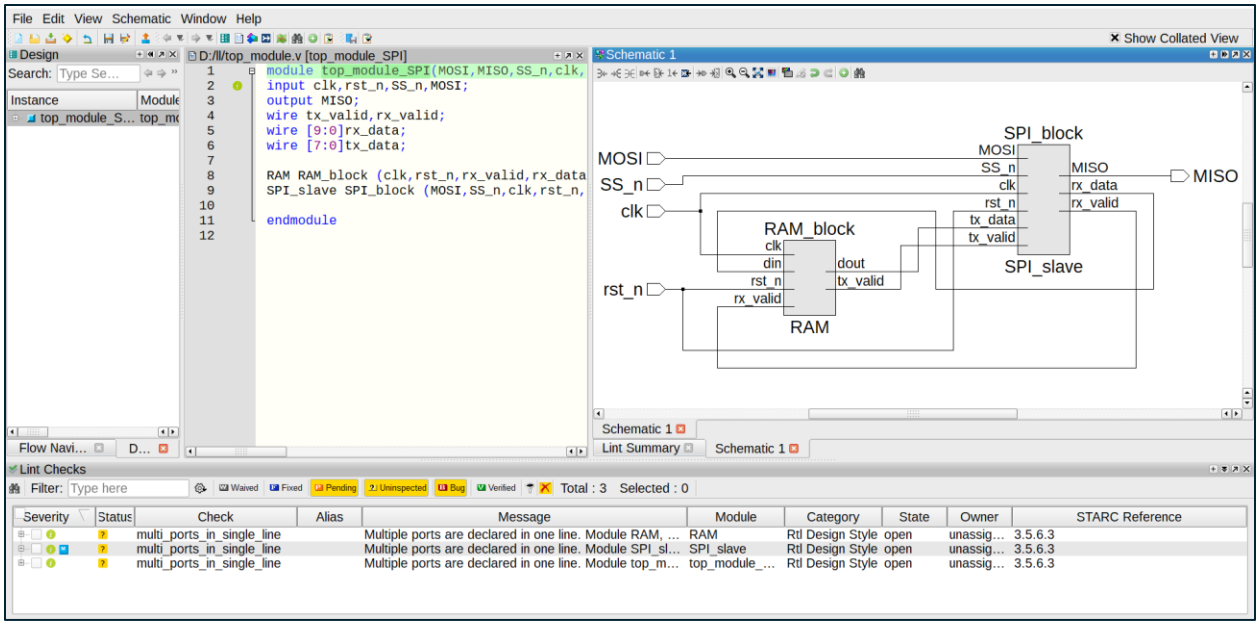
Note: the waveform is too long in width to be in one picture so I divided it into 4 and made each part showing 500nsecs if the sim.

- **Memory values:**



Notice from the memory values that the write is successful and wrote in the memory as shown in the red square and read it then successfully and also it read the value in the black square successfully as shown in the waveform.

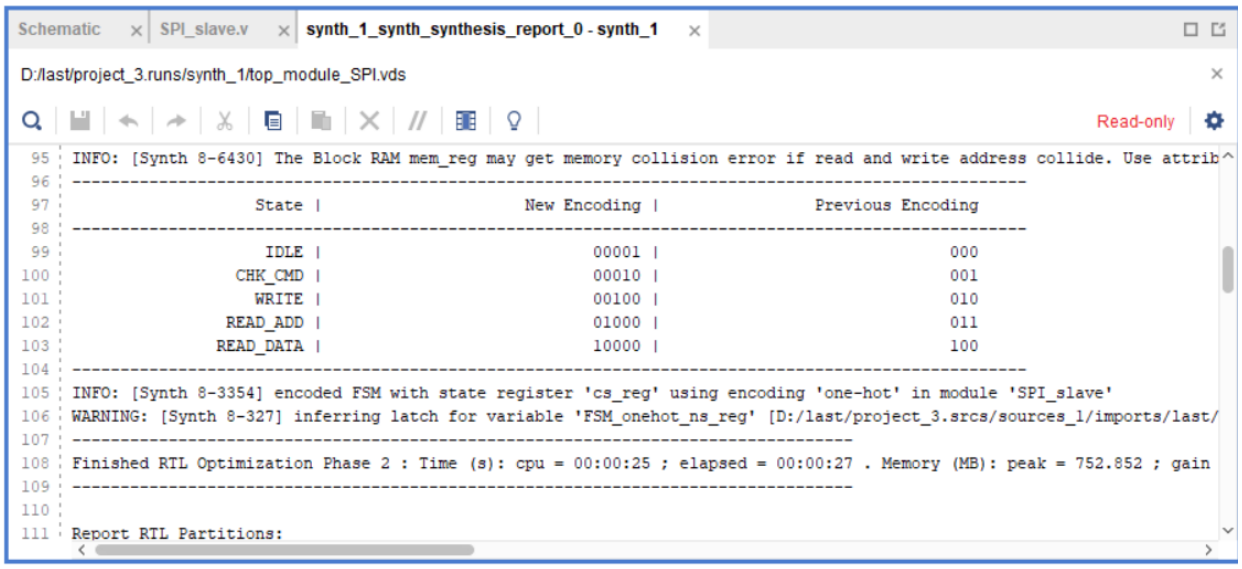
The lint tool:



The synthesis tool “Vavido”:

As we aim to work on the highest frequency possible we made it with the one hot encoding

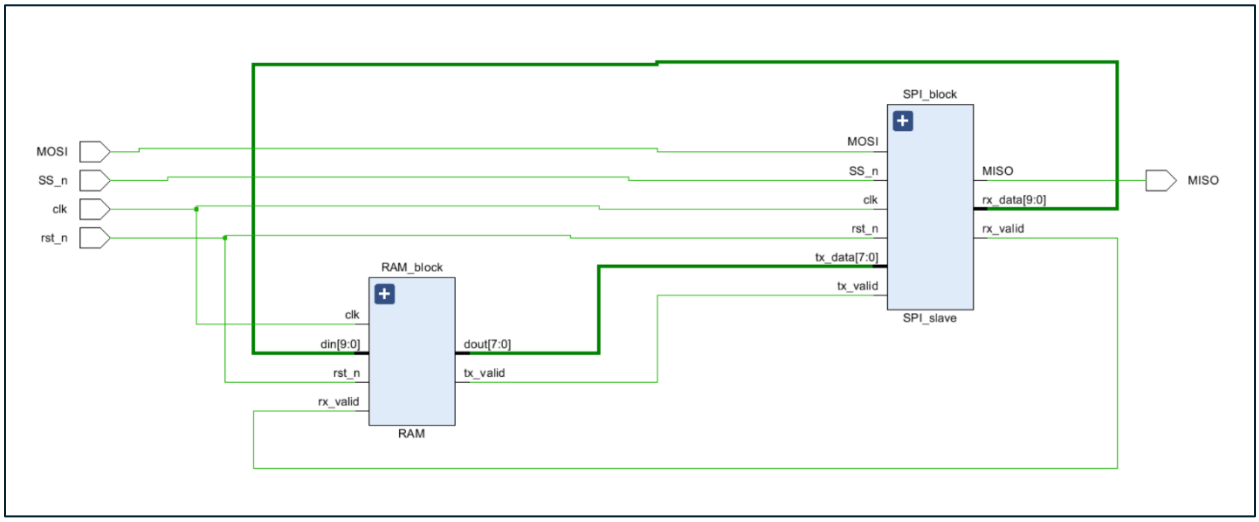
➢ For one hot code encoding:



1. Elaboration:

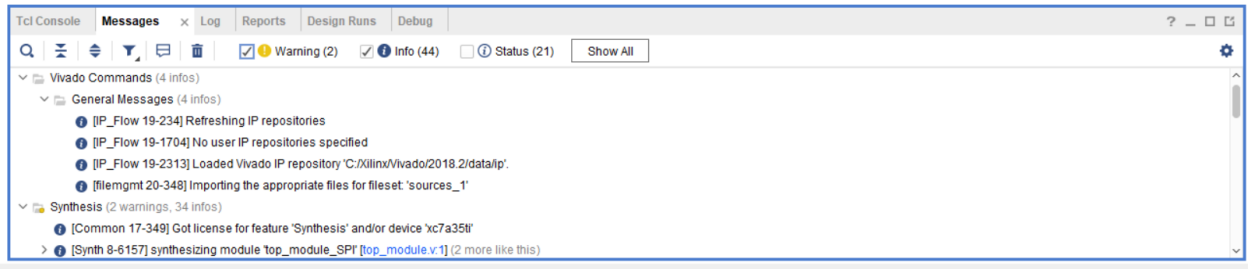
1.1 MSG of no errors:

1.2 Schematic:

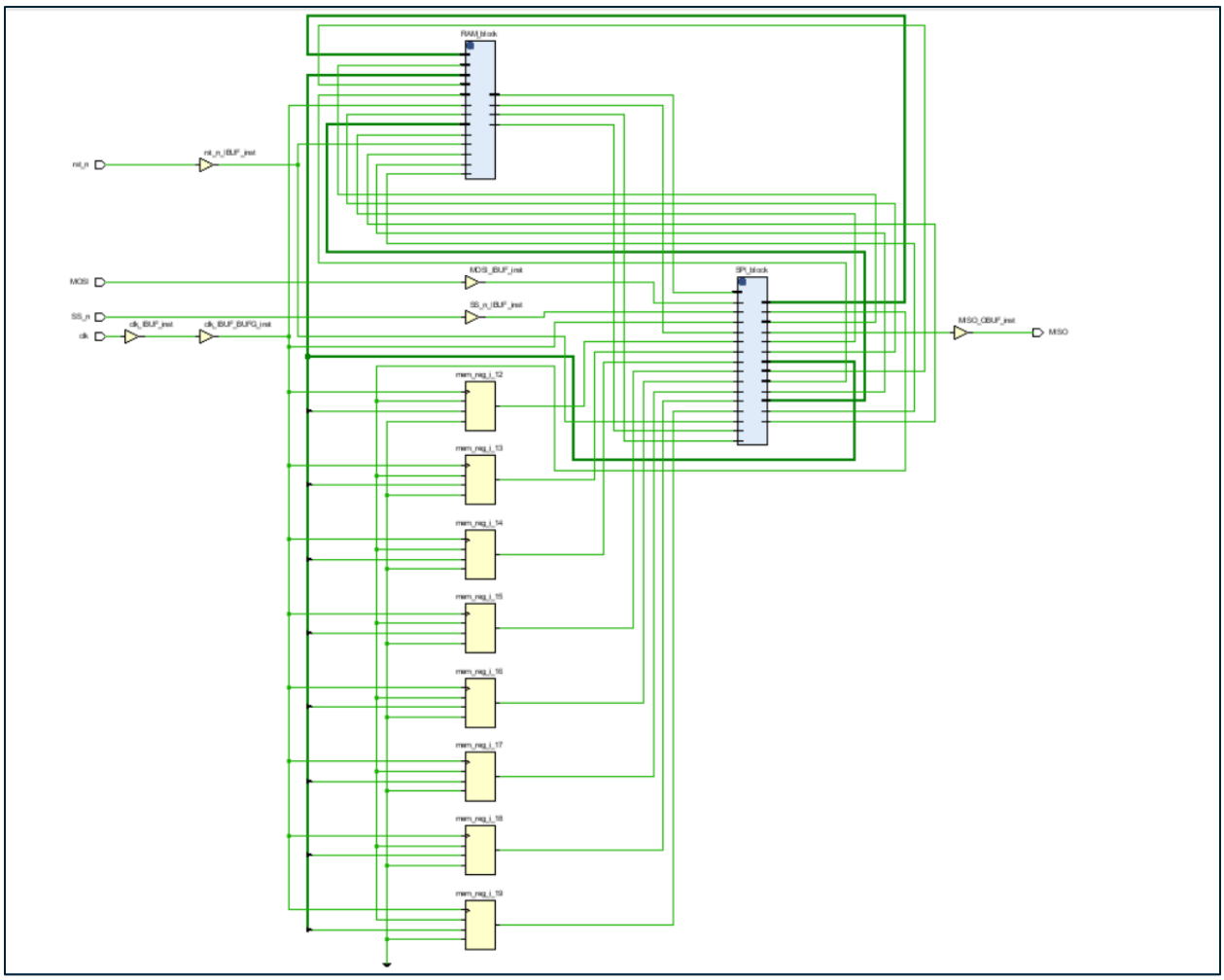


2. Synthesis:

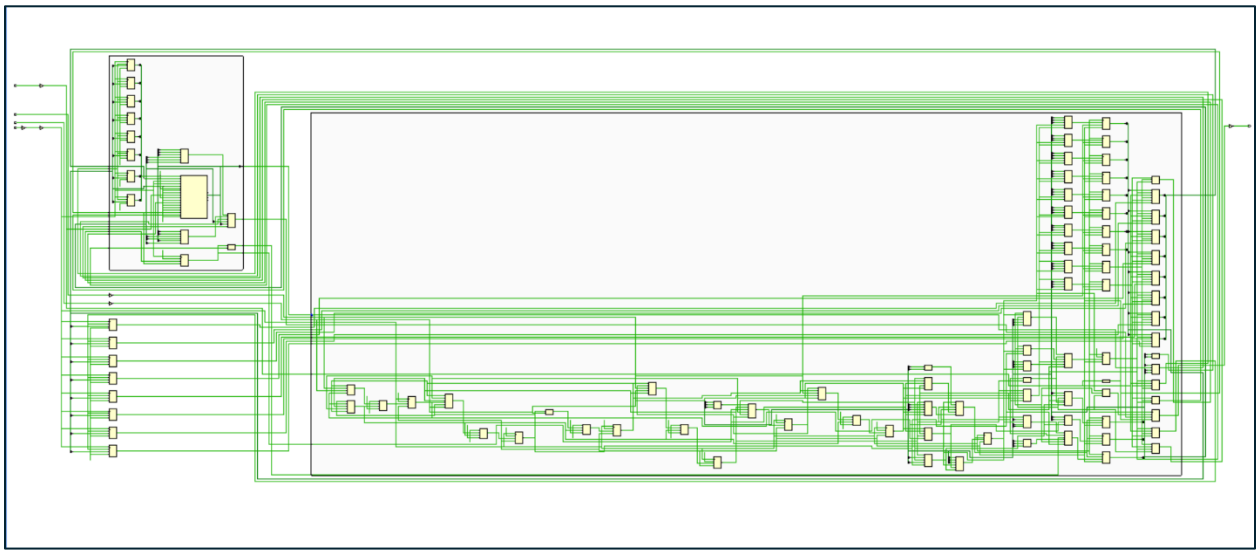
2.1 MSG of no errors:



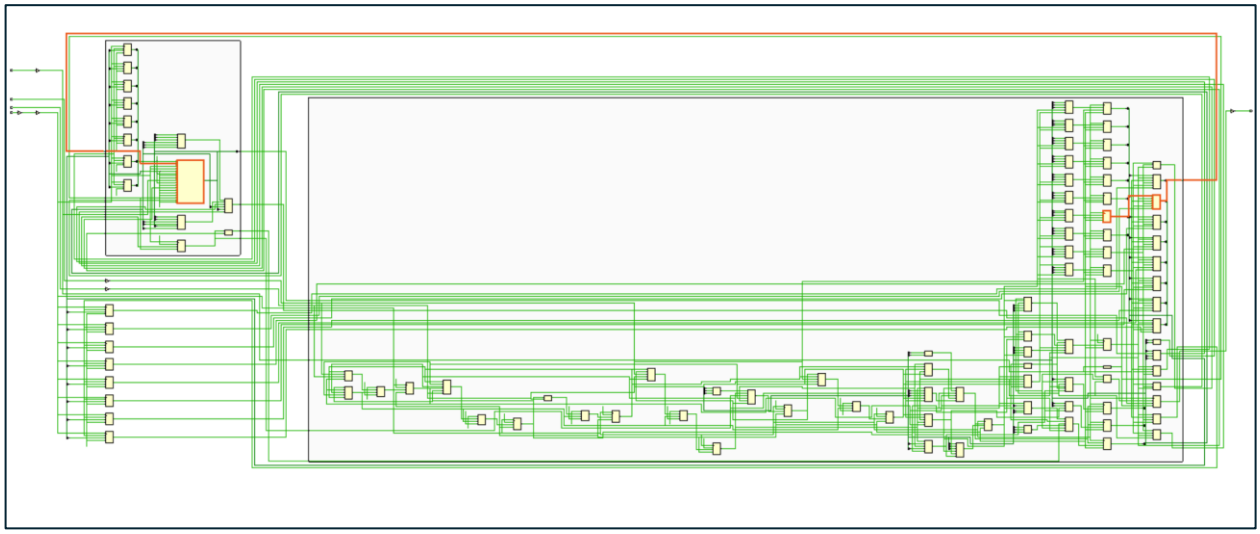
2.2 Schematic:



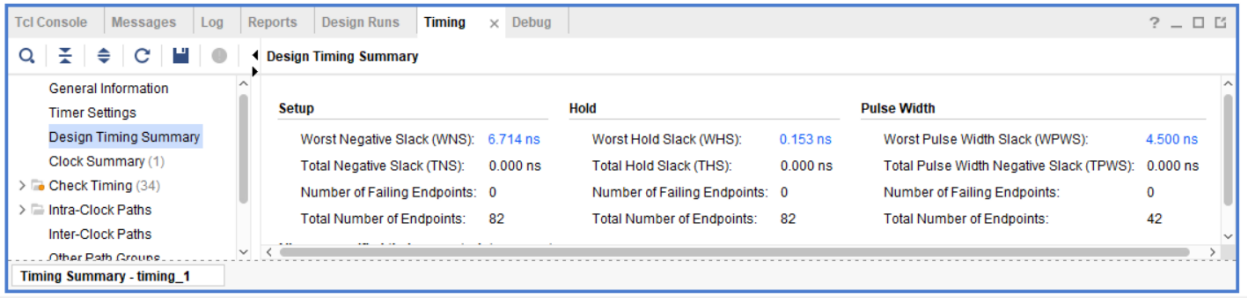
- Showing the RAM and SPI schematics from inside



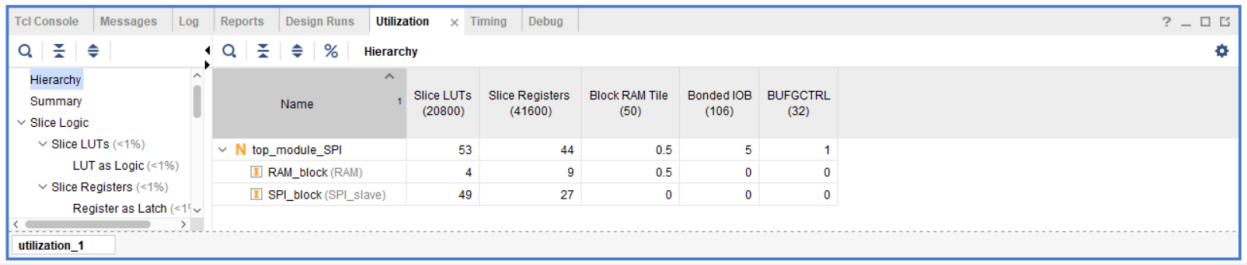
2.3 The critical path:



2.4 Timing report:

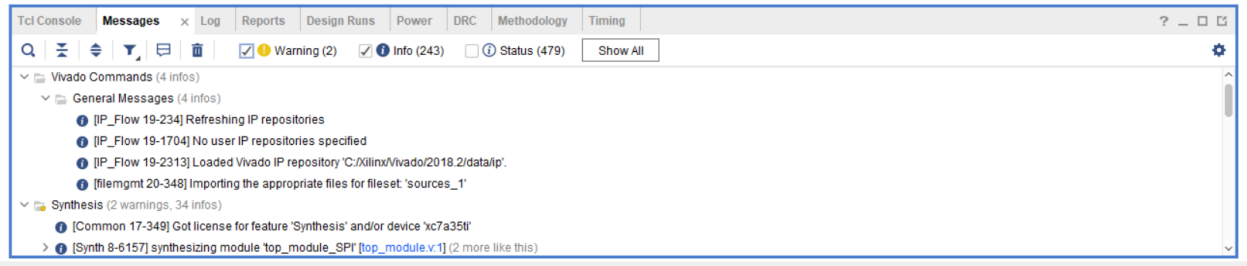


2.5 Utilization report:

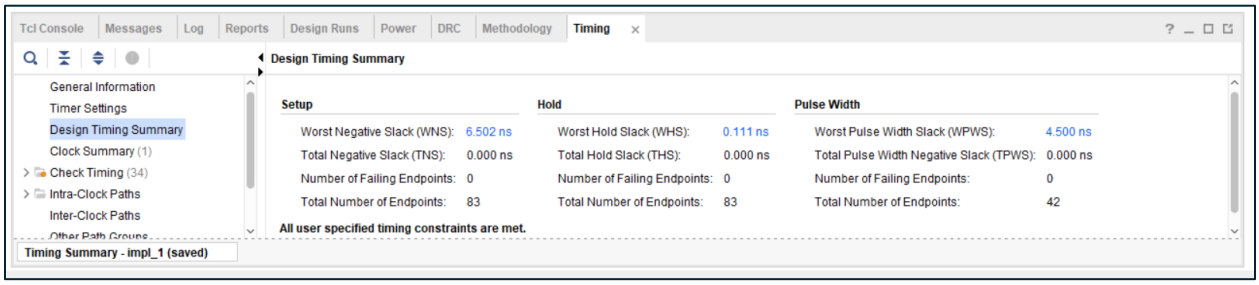


3. Implementation:

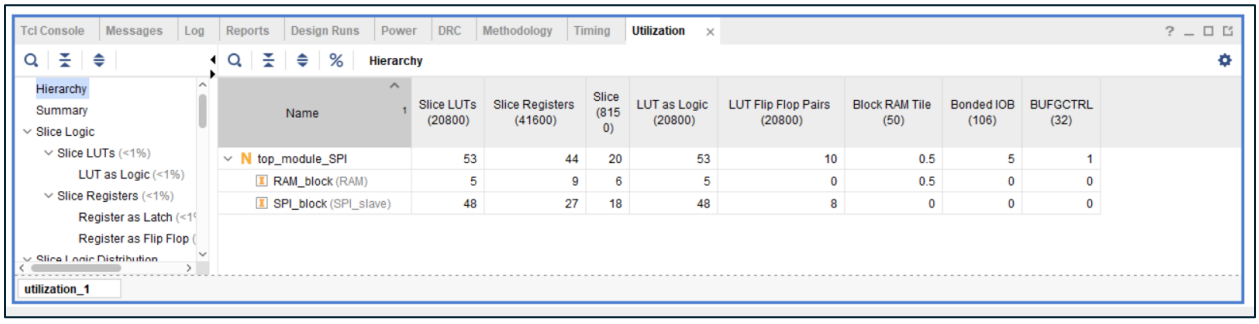
3.1 MSG of no errors:



3.2 Timing report:



3.3 Utilization report:



3.4 Device: