

SPI SLAVE INTERFACE

Digital design project

Done by

Saeed Maher Saeed & Sameh Mohammed Sameh

Table of Contents

Introduction	3
RAM design:.....	4
The RTL code design:	4
The testbench code:	5
The simulation snippets:.....	7
○ waveform:	7
○ Memory values:	8
The lint tool:	9
The synthesis tool:.....	9
○ Check errors:	9
○ The whole schematic:	10
○ The zoomed in schematic “zoomed on the selected part on the picture”:	10
SPI Slave design:.....	11
The RTL code design:	11
The testbench code:	14
The simulation snippets:.....	17
The lint tool:	18
The synthesis tool:.....	18
○ Check errors:	18
○ timing reports:	19
○ Schematic:	20
Top module:.....	21
The RTL code design:	21
The testbench code:	21
The simulation snippets:.....	24
○ waveform:	24
○ Memory values:	26
The lint tool:	27

The synthesis tool “Vavido”:	27
➤ For one hot code encoding:	27
1. Elaboration:	28
1.1 MSG of no errors:	28
1.2 Schematic:	28
2. Synthesis:	28
2.1 MSG of no errors:	28
2.2 Schematic:	29
2.3 The critical path:	30
2.4 Timing report:	31
2.5 Utilization report:	31
3. Implementation:	31
3.1 MSG of no errors:	31
3.2 Timing report:	32
3.3 Utilization report:	32
3.4 Device:	32

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
    end
end
```

```
    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

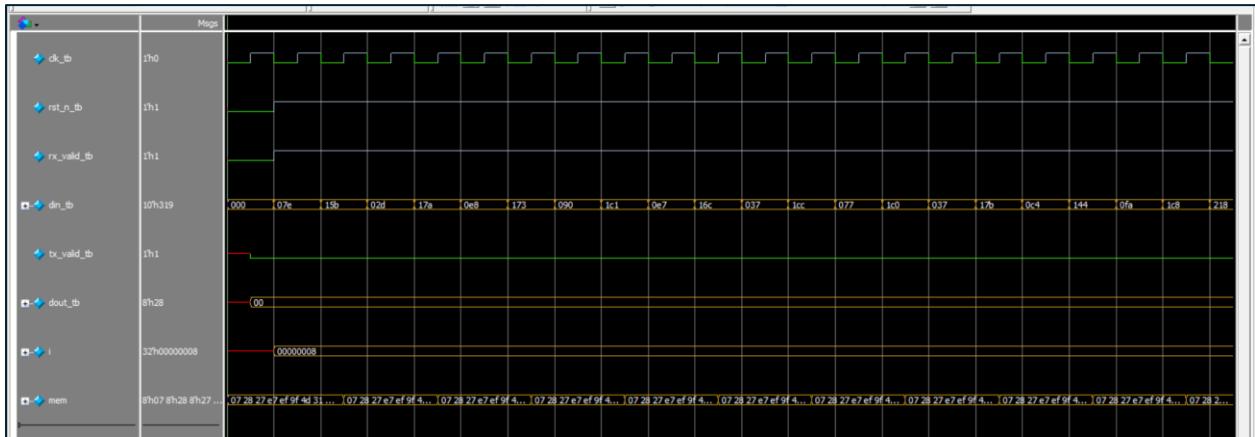
$stop;
end

endmodule
```

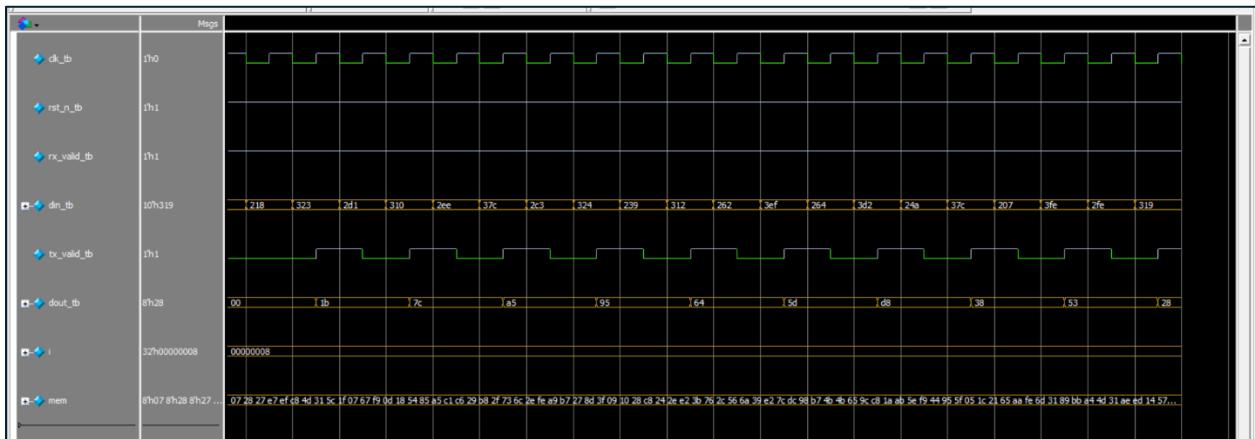
The simulation snippets:

- **waveform:**

- waveform for writing process:



- waveform for reading process:



- **Memory values:**

- memory initial values before running the simulation:

Address	Data
000000ff	07 28 27 e7 ef 9f 4d 31 5c 1f 07 67 f9 0d 18 54 85 a5
000000ed	c1 c6 29 b8 2f 0 18 2e fe a9 b7 27 8d 3f 09 10 28 c8
000000db	24 2e e2 3b 76 2c 56 6a 39 e2 7c dc 98 b7 4b 4b 65 9c
000000c9	c8 1a ab 5e f9 31 95 5f 05 1c 21 65 aa fe 6d 31 89 bb
000000b7	a4 4d 31 ae ed 14 57 68 ba db ff 9f 4e f0 fa 62 01 ee
000000a5	55 1a ca cd b7 e0 eb a2 d1 86 51 32 4d c3 90 c0 0e b3
00000093	51 d8 22 5b 85 e6 2f 61 70 2b ea a4 46 20 08 45 20 ff
00000081	0b 13 2c ba f6 89 cb 8e 2a 20 db 1b a6 f1 d2 fa 0f b5
0000006f	83 de 67 60 2d c4 f7 3e 80 60 0a d8 01 5d 46 0b ef e2
0000005d	2b f0 1a 87 e5 12 12 73 5b 72 76 ac 41 32 0b 2c 2b 08
0000004b	c5 38 47 9c 86 75 aa 53 b4 bd 57 d3 c6 b1 5a 6e 91 3c
00000039	64 7d 71 76 23 3a 34 e7 a3 5b 46 11 7b d4 39 4f a6 2d
00000027	fb 8d 21 14 cd c0 14 f1 f8 08 52 93 19 78 90 1b ca 98
00000015	b9 54 83 d4 7c 5d 0f de 92 9c f9 b6 da 31 53 23 3d 9c
00000003	85 8c 09 86
ffffffffff	

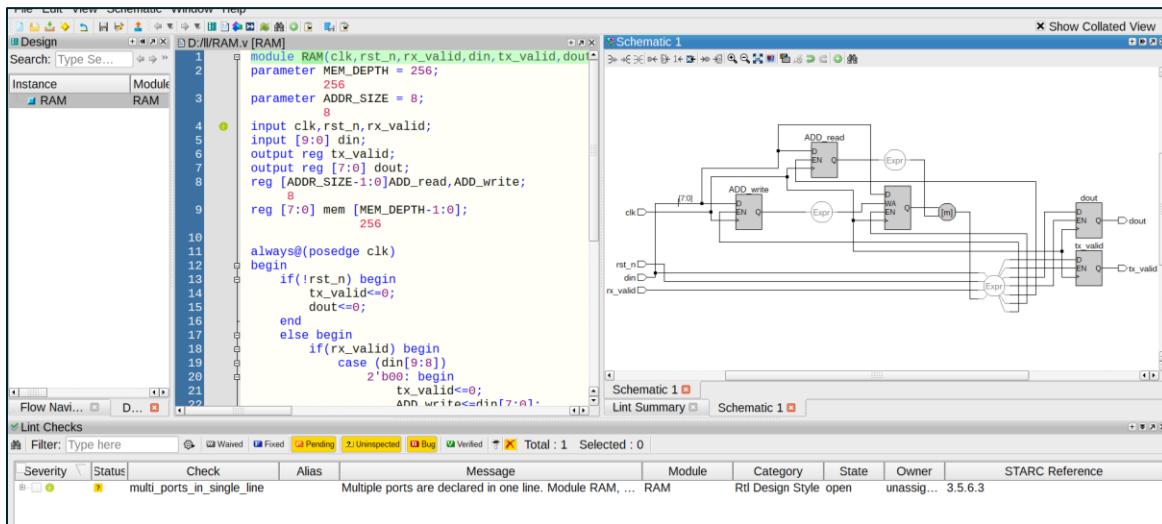
- memory values after running the simulation:

Address	Data
000000ff	07 28 27 e7 ef 9f 4d 31 5c 1f 07 67 f9 0d 18 54 85 a5
000000ed	c1 c6 29 b8 2f 0 18 2e fe a9 b7 27 8d 3f 09 10 28 c8
000000db	24 2e e2 3b 76 2c 56 6a 39 e2 7c dc 98 b7 4b 4b 65 9c
000000c9	c8 1a ab 5e f9 31 95 5f 05 1c 21 65 aa fe 6d 31 89 bb
000000b7	a4 4d 31 ae ed 14 57 68 ba db ff 9f 4e f0 fa 62 01 ee
000000a5	55 1a ca cd b7 e0 eb a2 d1 86 51 32 4d c3 90 c0 0e b3
00000093	51 d8 22 5b 85 e6 2f 61 70 2b ea a4 46 20 08 45 20 ff
00000081	0b 13 2c ba f6 89 cb 8e 2a 20 c0 1b a6 f1 d2 fa 0f b5
0000006f	83 de 67 60 2d c4 f7 3e 80 60 0a d8 01 5d 46 0b ef e2
0000005d	2b f0 1a 87 e5 12 12 73 5b 72 76 ac 41 32 0b 2c 2b 08
0000004b	c5 38 47 9c 86 75 aa 53 b4 bd 57 d3 c6 b1 5a 6e 91 3c
00000039	64 7d 7b 76 23 3a 34 e7 a3 5b 46 11 7a d4 39 4f a6 2d
00000027	fb 8d 21 14 cd c0 14 f1 f8 08 52 93 19 78 90 1b ca 98
00000015	b9 54 83 d4 7c 5d 0f de 92 9c f9 b6 da 31 53 23 3d 9c
00000003	85 8c 09 86
ffffffffff	

From the snippets, we can find that the data wrote in their correct 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.



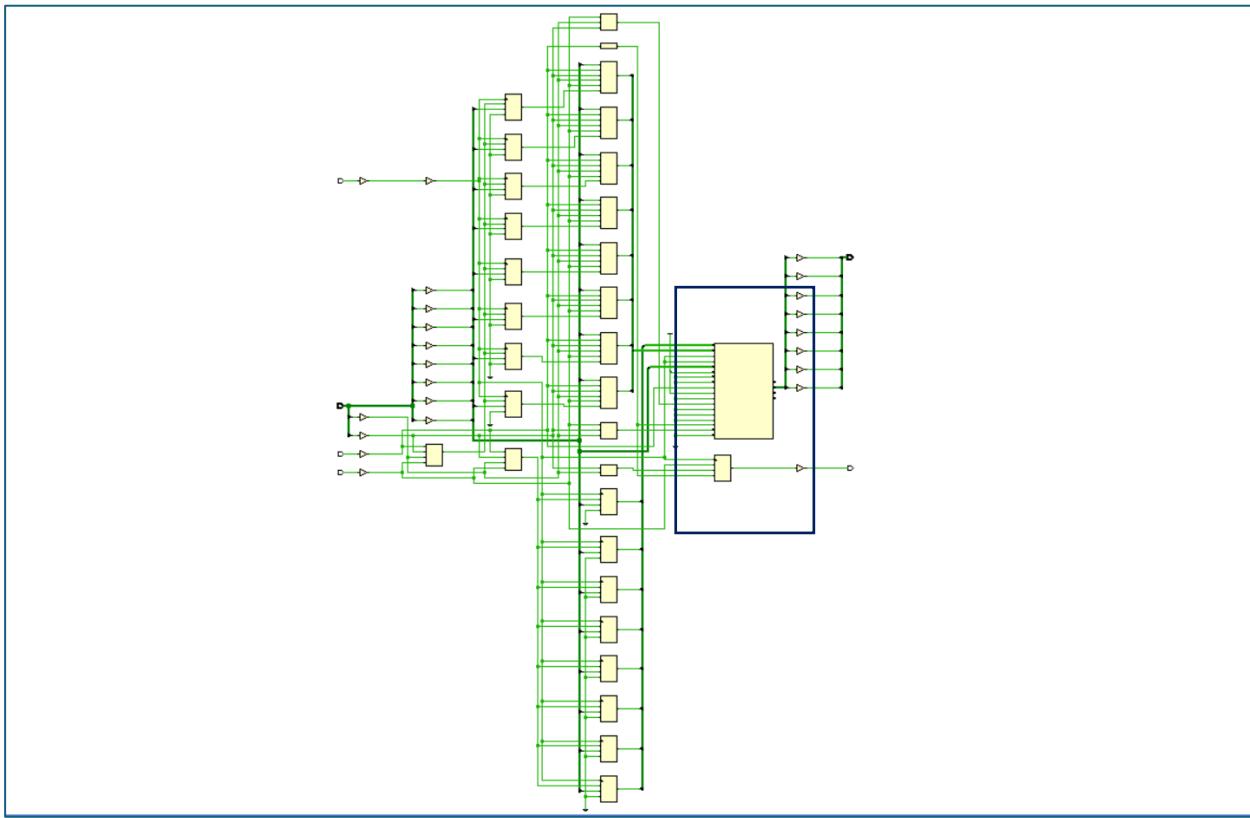
The synthesis tool:

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

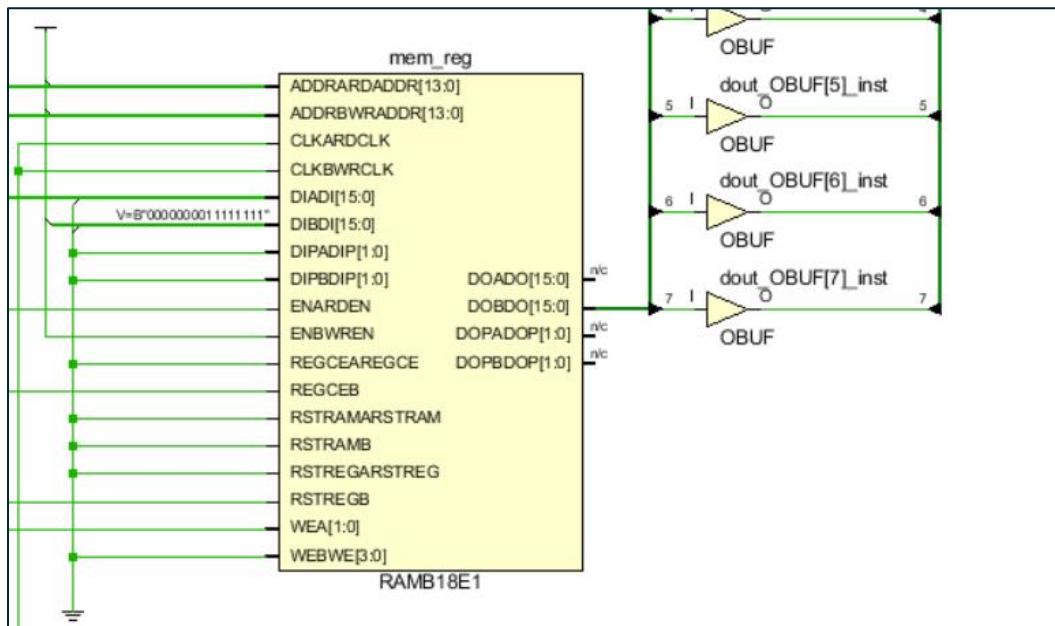
○ Check errors:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	14	0	20800	0.07
LUT as Logic	14	0	20800	0.07
LUT as Memory	0	0	9600	0.00
Slice Registers	17	0	41600	0.04
Register as Flip Flop	17	0	41600	0.04
Register as Latch	0	0	41600	0.00
F7 Muxes	0	0	16300	0.00
F8 Muxes	0	0	8150	0.00

- 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
        end
    endcase
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
        end
    endcase
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

```

```

        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;

```

```

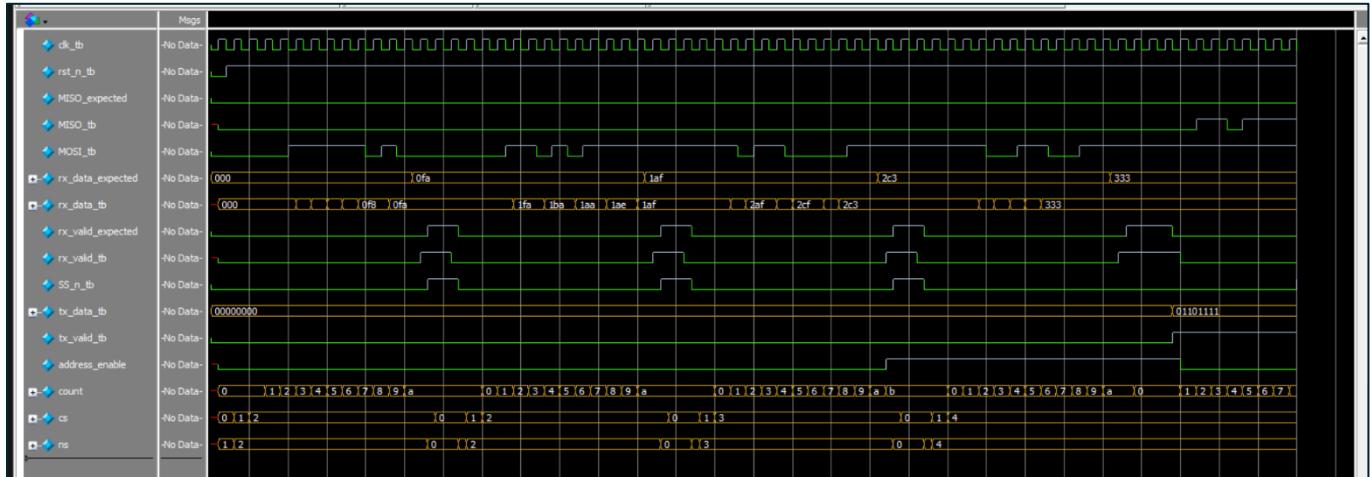
repeat(2) @(negedge clk_tb); // goes to READ_DATA state
    spi_send_10bits(10'b11_0011_0011);
    rx_data_expected=10'b11_0011_0011;
    @(negedge clk_tb);
    rx_valid_expected=1;
    @(negedge clk_tb);
    if (MISO_tb != MISO_expected || rx_data_expected != rx_data_tb ||
rx_valid_expected != rx_valid_tb) 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

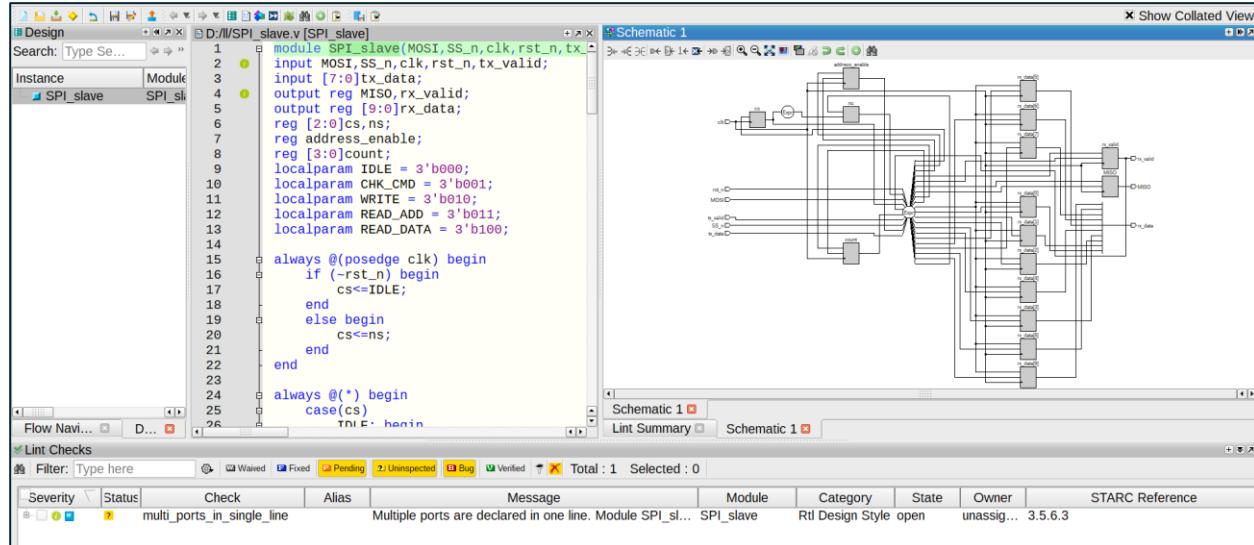
```

The simulation snippets:



The lint tool:

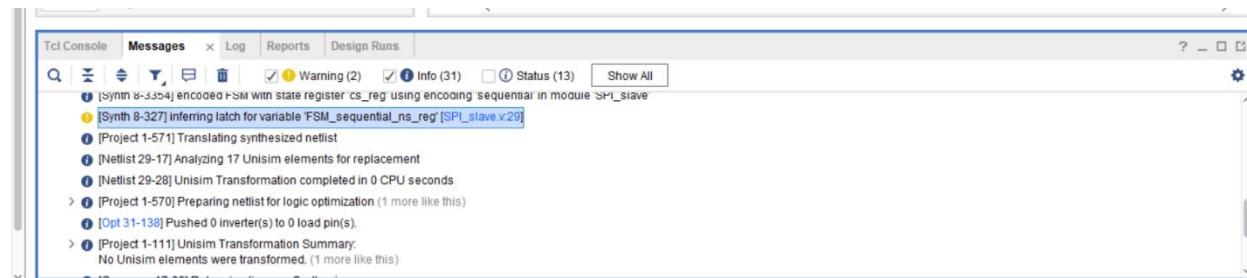
In this part, it checks that this module is synthesizable or not also if it shows the correct RTL schematic.



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:

The screenshot shows the Xilinx Vivado IDE interface with two main windows open:

- Schematic > SPI_slave.v:** Shows the module structure with 105 nets and 92 leaf cells.
- synth_1_synth_synthesis_report_0 - synth_1:** Displays synthesis logs. Key messages include:
 - INFO: [Synth 8-5544] ROM "ns" won't be mapped to Block RAM because address size (1) smaller than threshold (5)
 - INFO: [Synth 8-3354] encoded FSM with state register 'cs_reg' using encoding 'sequential' in module 'SPI_slave'
 - WARNING: [Synth 8-327] inferring latch for variable 'FSM_sequential_ns_reg' [D:/11/project_2/srcs/sources_1/imports/11]
- Source File Properties:** Shows SPI_slave.v is enabled.
- Tcl Console:** Shows the completion of RTL Optimization Phase 2.
- Timing > Design Timing Summary:** Shows timing constraints for sequential encoding. Key values:

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 6.497 ns	Worst Hold Slack (WHS): 0.200 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: 22	Total Number of Endpoints: 22	Total Number of Endpoints: 21

○ gray encoding:

The screenshot shows the Xilinx Vivado IDE interface with two main windows open:

- Schematic > SPI_slave.v:** Shows the module structure with 105 nets and 92 leaf cells.
- synth_1_synth_synthesis_report_0 - synth_1:** Displays synthesis logs. Key messages include:
 - INFO: [Synth 8-5544] ROM "ns" won't be mapped to Block RAM because address size (1) smaller than threshold (5)
 - INFO: [Synth 8-3354] encoded FSM with state register 'cs_reg' using encoding 'gray' in module 'SPI_slave'
 - WARNING: [Synth 8-327] inferring latch for variable 'FSM_gray_ns_reg' [D:/11/project_2/srcs/sources_1/imports/11/SPI_si
- Source File Properties:** Shows SPI_slave.v is enabled.
- Tcl Console:** Shows the completion of RTL Optimization Phase 2.
- Timing > Design Timing Summary:** Shows timing constraints for gray encoding. Key values:

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 6.328 ns	Worst Hold Slack (WHS): 0.209 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: 22	Total Number of Endpoints: 22	Total Number of Endpoints: 21

- o one hot encoding:

The screenshot shows the Xilinx Vivado IDE interface with two main windows open:

- Schematic View:** Shows the SPI_slave.v module structure with various components and connections.
- Synthesis Report:** Titled "synth_1_synth_synthesis_report_0 - synth_1", it displays state encoding details. A table shows the mapping from State to New Encoding and Previous Encoding:

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

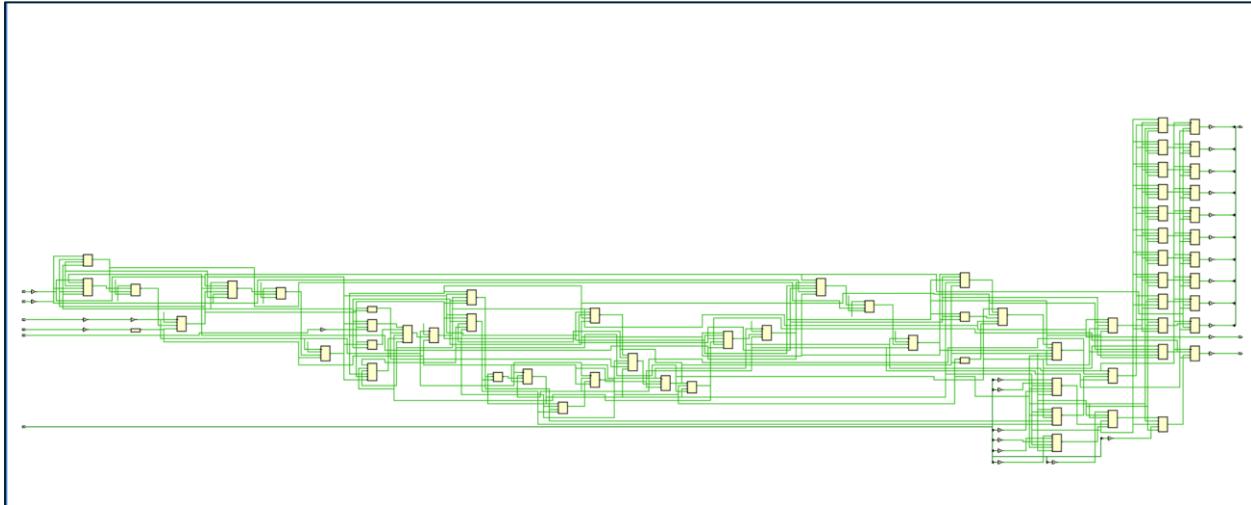
- Timing Report:** Titled "Design Timing Summary", it provides timing constraints and analysis. Key values include:

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 6.770 ns	Worst Hold Slack (WHS): 0.209 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 Falling Endpoints: 0	Number of Falling Endpoints: 0	Number of Falling Endpoints: 0
Total Number of Endpoints: 22	Total Number of Endpoints: 22	Total Number of Endpoints: 23

 A message at the bottom states: "All user specified timing constraints are met."

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

- o **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
        begin
            for (i = 9; i >= 0; i = i - 1) begin
                MOSI_tb = data_in[i];
                @(negedge clk_tb);
            end
        end
    end
end
```

```

endtask

initial begin
    // initializing 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);

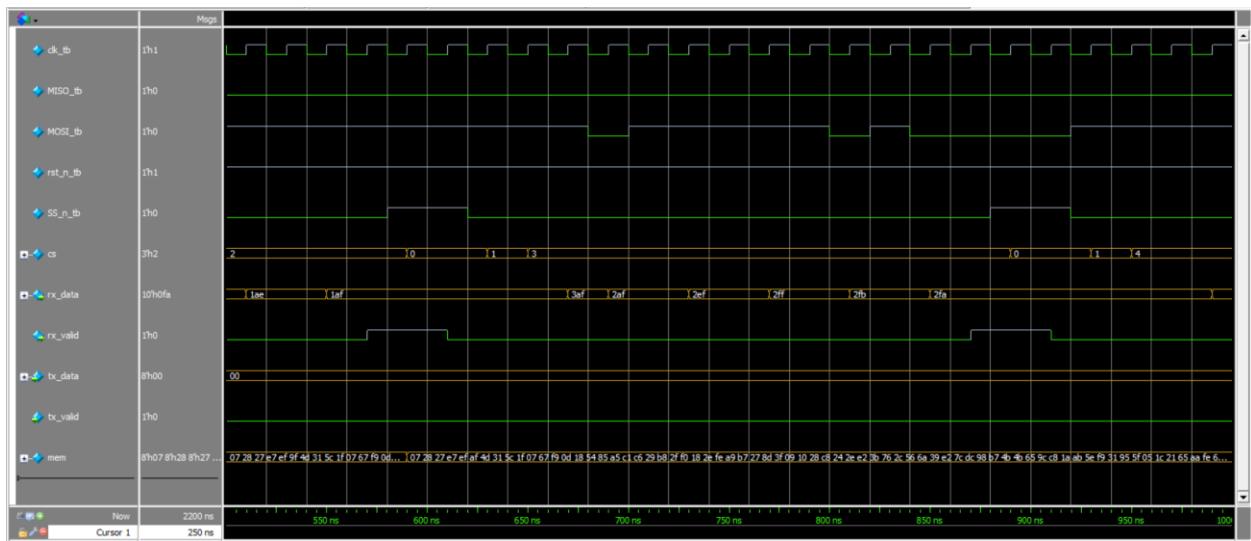
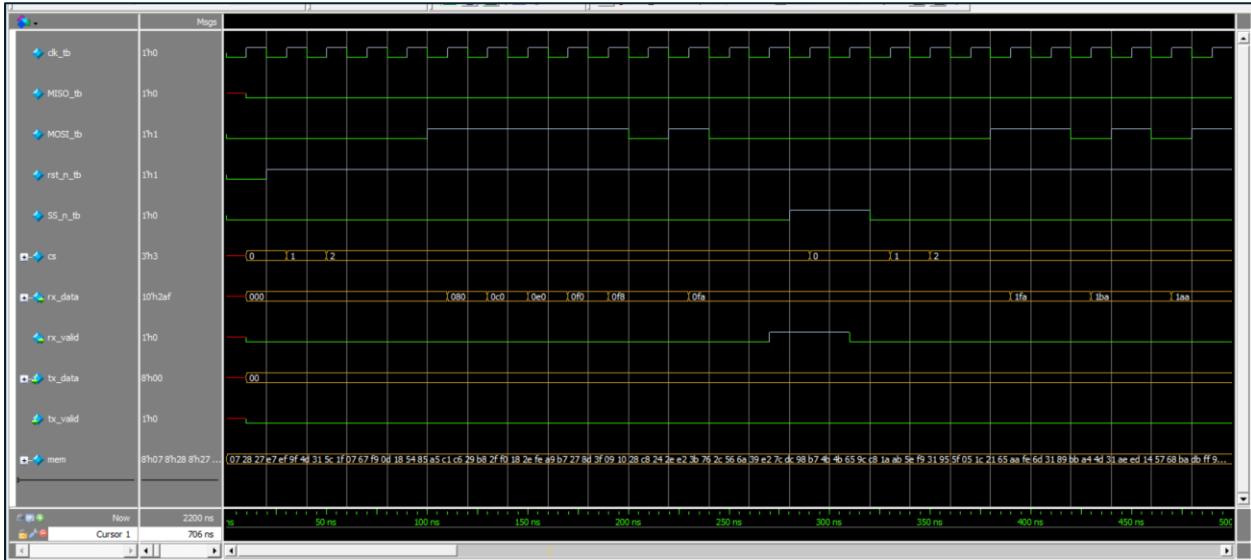
$stop;
end

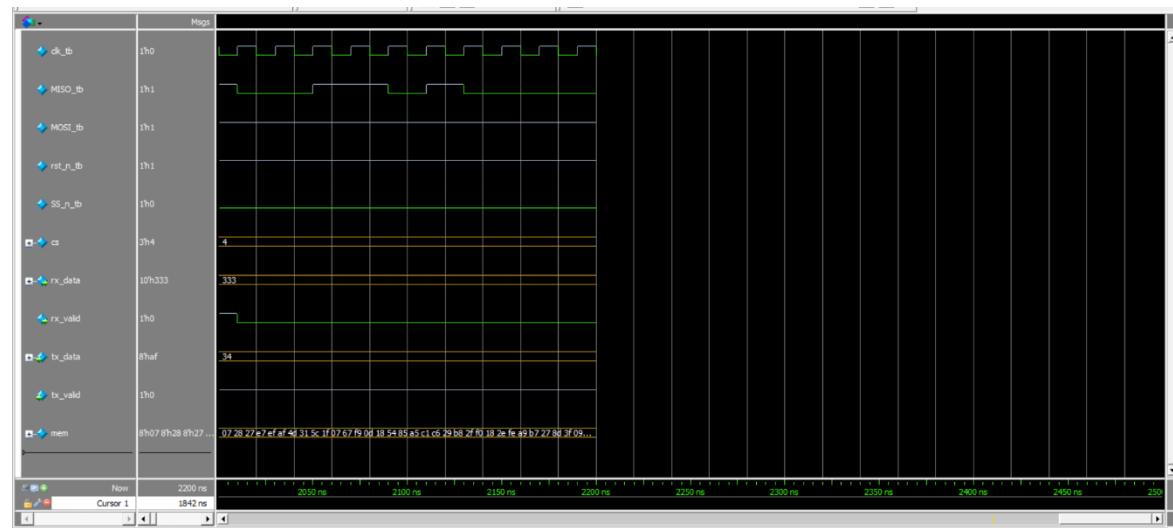
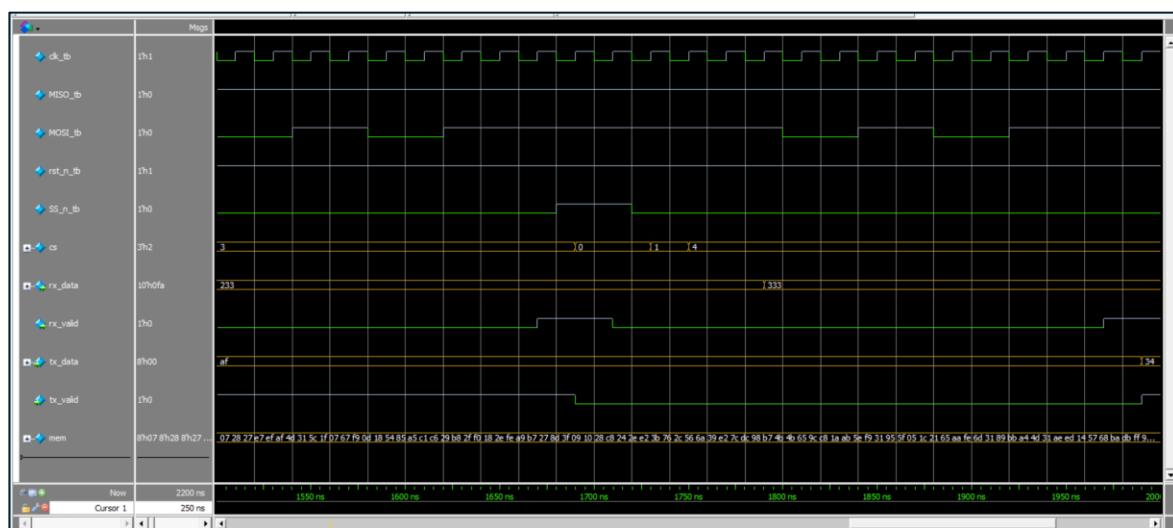
endmodule

```

The simulation snippets:

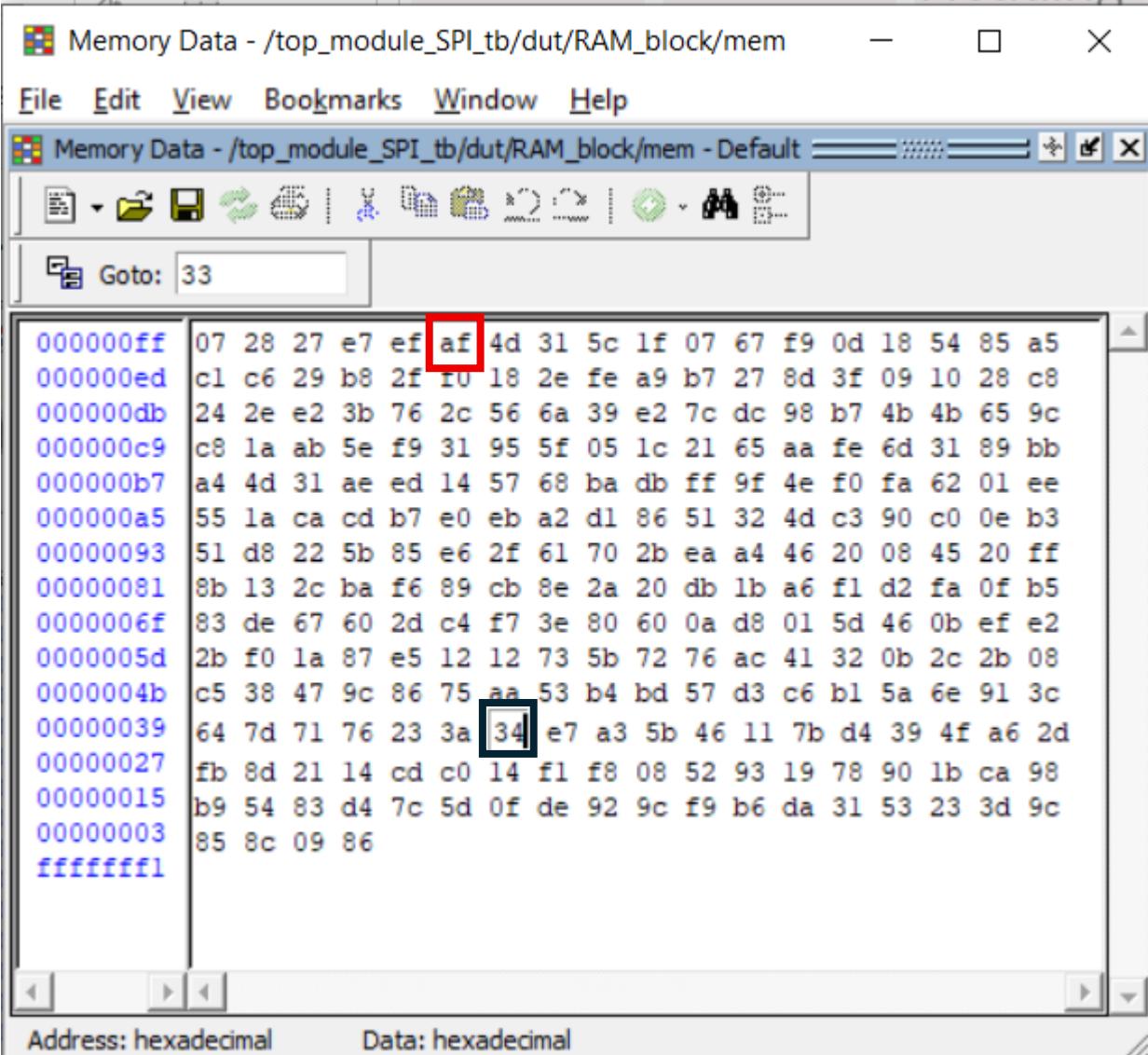
- **waveform:**





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.

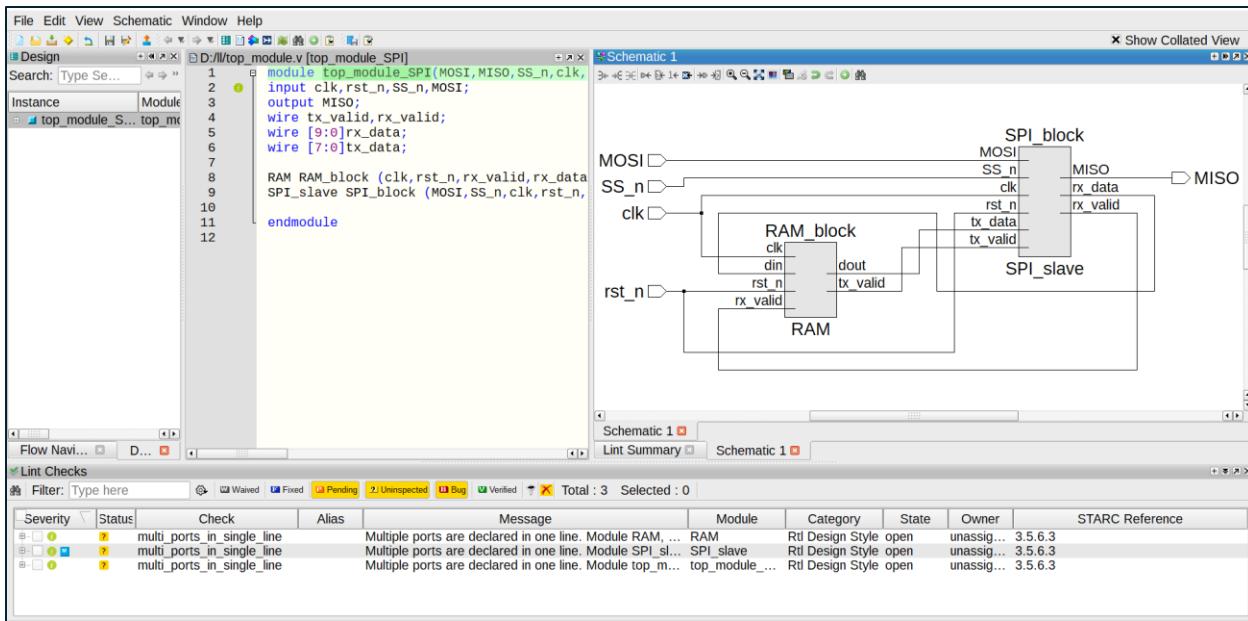
- **Memory values:**



The screenshot shows a memory dump window titled "Memory Data - /top_module_SPI_tb/dut/RAM_block/mem". The window has a menu bar with File, Edit, View, Bookmarks, Window, and Help. Below the menu is a toolbar with various icons. A search bar at the top says "Goto: 33". The main area displays memory contents in two columns. The first column is addresses and the second is data. Addresses shown include 000000ff, 000000ed, 000000db, 000000c9, 000000b7, 000000a5, 00000093, 00000081, 0000006f, 0000005d, 0000004b, 00000039, 00000027, 00000015, 00000003, and ffffff1. The data column contains hex values like 07, 28, e7, ef, af, 4d, 31, 5c, 1f, 07, 67, f9, 0d, 18, 54, 85, a5, c1, c6, 29, b8, 2f, f0, 18, 2e, fe, a9, b7, 27, 8d, 3f, 09, 10, 28, c8, 24, 2e, e2, 3b, 76, 2c, 56, 6a, 39, e2, 7c, dc, 98, b7, 4b, 4b, 65, 9c, c8, 1a, ab, 5e, f9, 31, 95, 5f, 05, 1c, 21, 65, aa, fe, 6d, 31, 89, bb, a4, 4d, 31, ae, ed, 14, 57, 68, ba, db, ff, 9f, 4e, f0, fa, 62, 01, ee, 55, 1a, ca, cd, b7, e0, eb, a2, d1, 86, 51, 32, 4d, c3, 90, c0, 0e, b3, 51, d8, 22, 5b, 85, e6, 2f, 61, 70, 2b, ea, a4, 46, 20, 08, 45, 20, ff, 00000081, 8b, 13, 2c, ba, f6, 89, cb, 8e, 2a, 20, db, 1b, a6, f1, d2, fa, 0f, b5, 0000006f, 83, de, 67, 60, 2d, c4, f7, 3e, 80, 60, 0a, d8, 01, 5d, 46, 0b, ef, e2, 0000005d, 2b, f0, 1a, 87, e5, 12, 12, 73, 5b, 72, 76, ac, 41, 32, 0b, 2c, 2b, 08, 0000004b, c5, 38, 47, 9c, 86, 75, aa, 53, b4, bd, 57, d3, c6, b1, 5a, 6e, 91, 3c, 00000039, 64, 7d, 71, 76, 23, 3a, 34, e7, a3, 5b, 46, 11, 7b, d4, 39, 4f, a6, 2d, 00000027, fb, 8d, 21, 14, cd, c0, 14, f1, f8, 08, 52, 93, 19, 78, 90, 1b, ca, 98, 00000015, b9, 54, 83, d4, 7c, 5d, 0f, de, 92, 9c, f9, b6, da, 31, 53, 23, 3d, 9c, 00000003, 85, 8c, 09, 86, ffffff1

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:

The screenshot shows a synthesis report window titled "synth_1_synth_synthesis_report_0 - synth_1". The report displays the state encoding for an FSM:

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

INFO: [Synth 8-6430] The Block RAM mem_reg may get memory collision error if read and write address collide. Use attrib^

INFO: [Synth 8-3354] encoded FSM with state register 'cs_reg' using encoding 'one-hot' in module 'SPI_slave'

WARNING: [Synth 8-327] inferring latch for variable 'FSM_onehot_ns_reg' [D:/last/project_3.srcc/sources_1/imports/last/

Finished RTL Optimization Phase 2 : Time (s): cpu = 00:00:25 ; elapsed = 00:00:27 . Memory (MB): peak = 752.852 ; gain

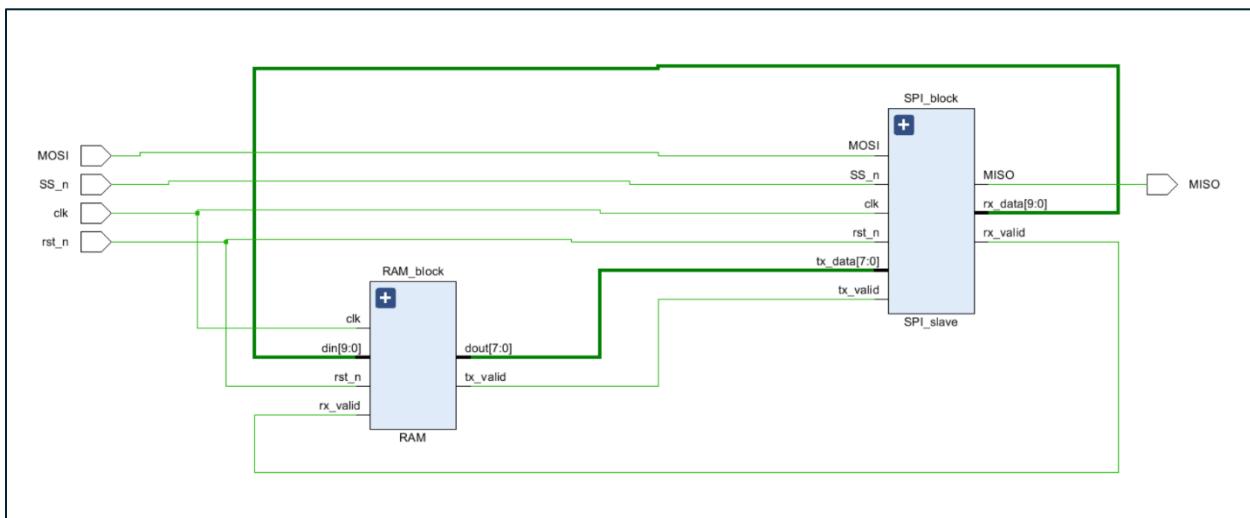
Report RTL Partitions:

1. Elaboration:

1.1 MSG of no errors:



1.2 Schematic:

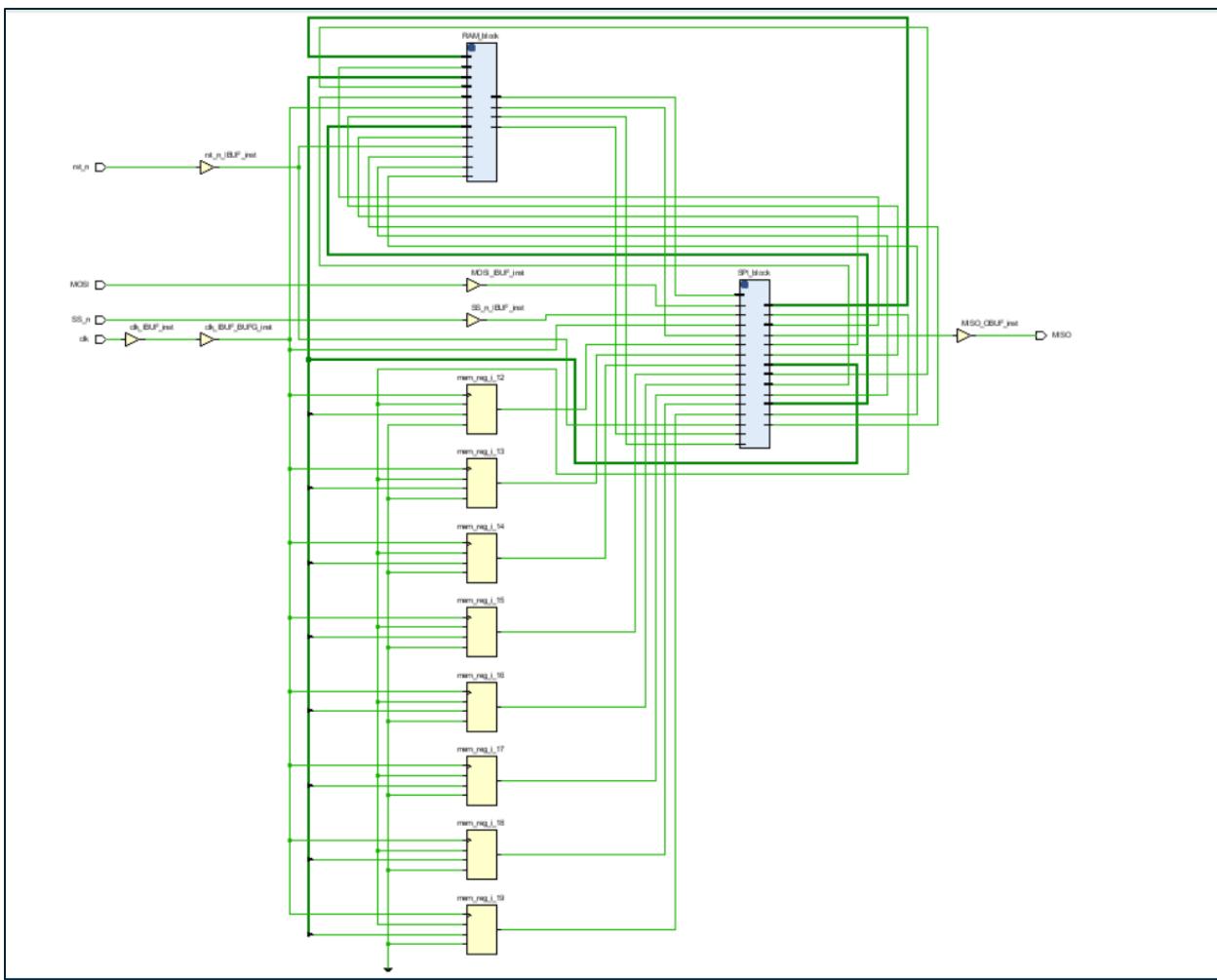


2. Synthesis:

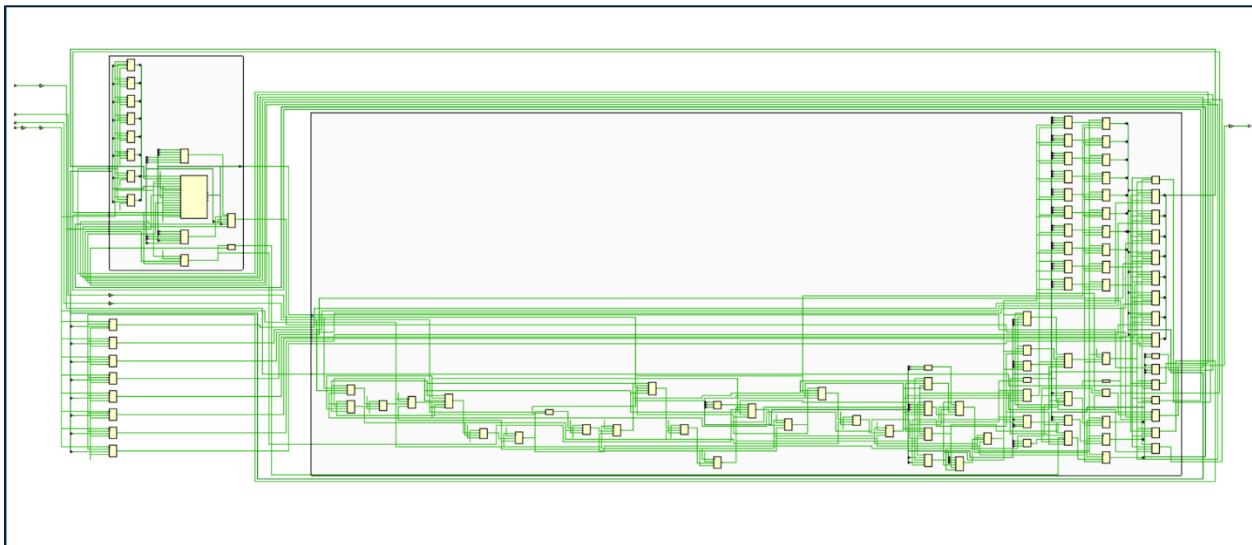
2.1 MSG of no errors:



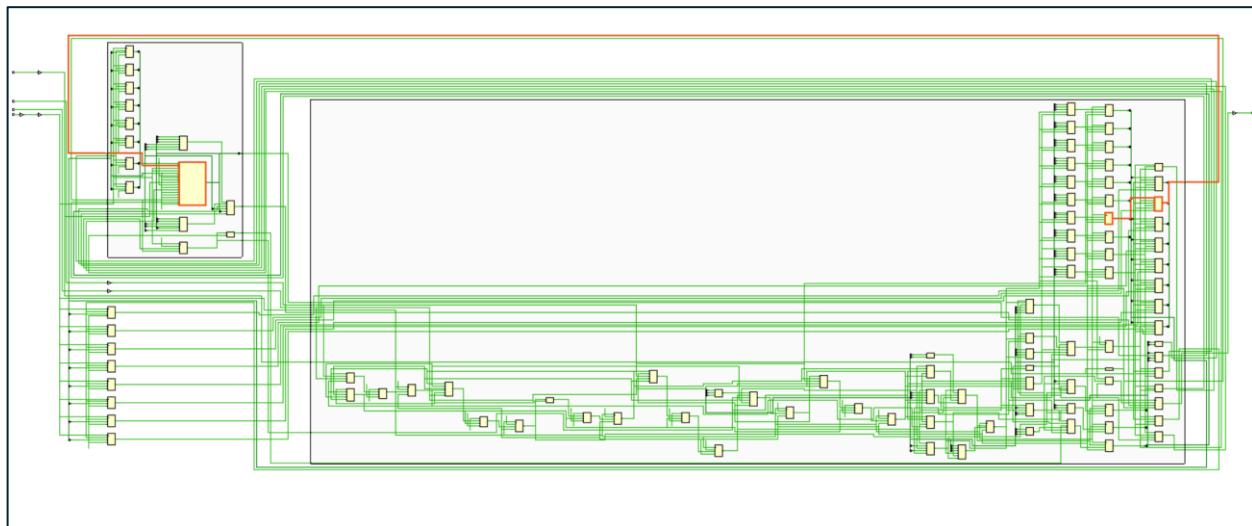
2.2 Schematic:



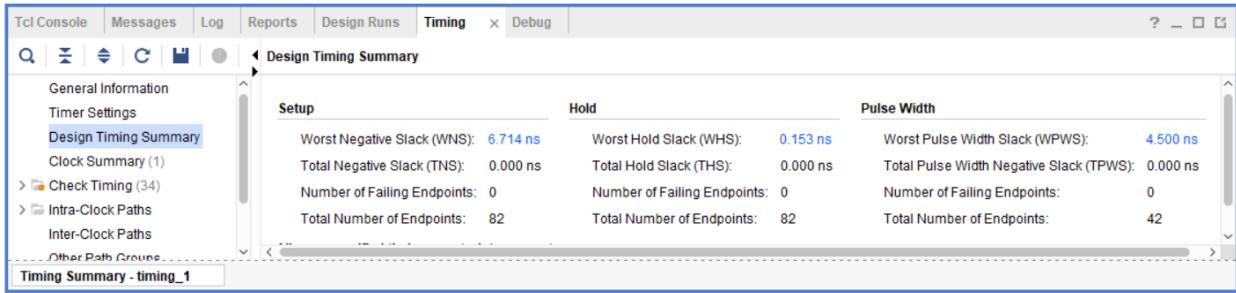
- Showing the RAM and SPI schematics from inside



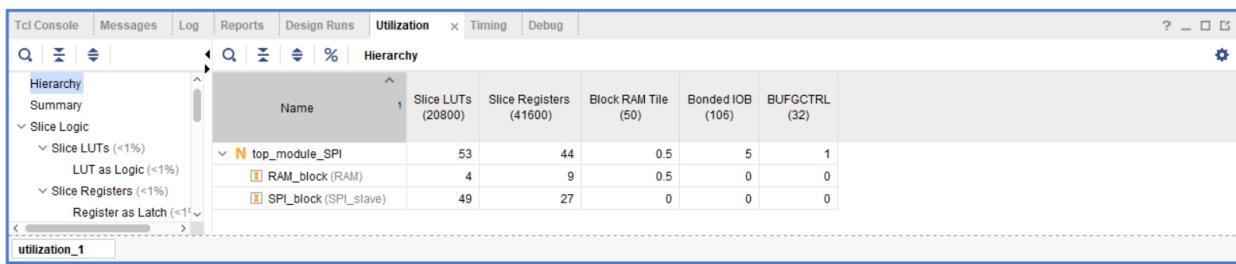
2.3 The critical path:



2.4 Timing report:

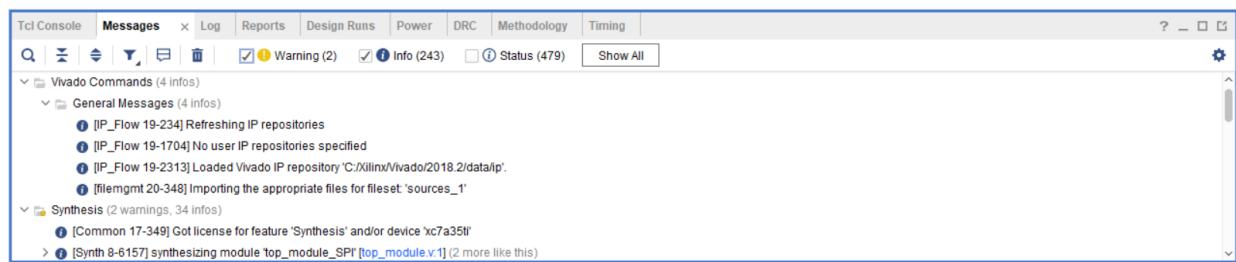


2.5 Utilization report:

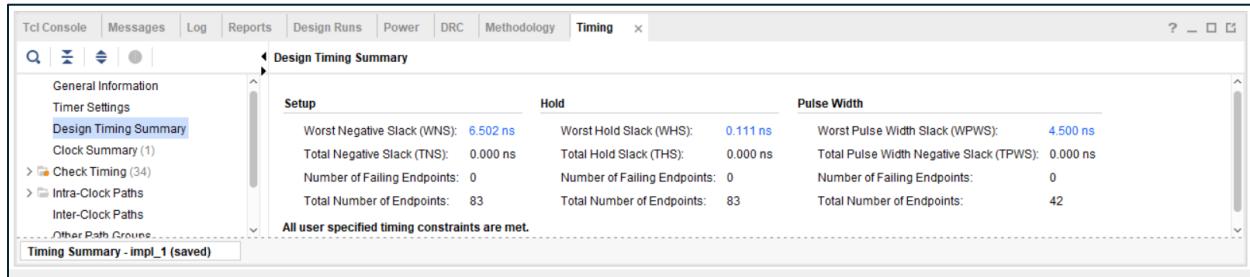


3. Implementation:

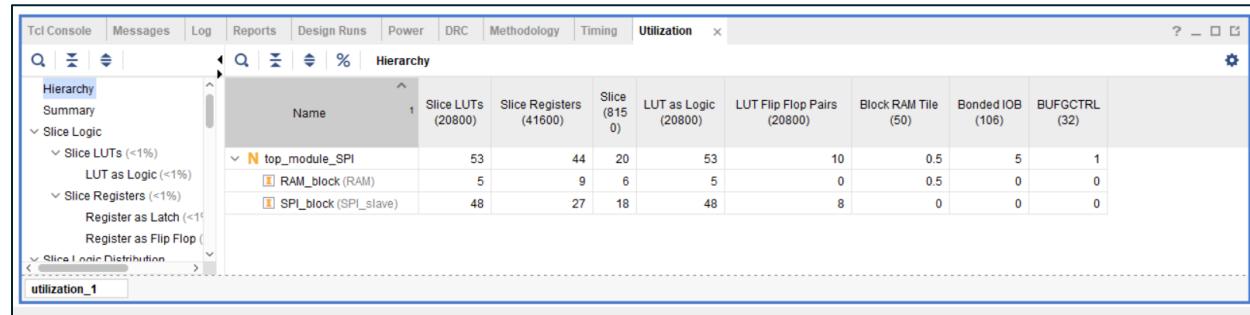
3.1 MSG of no errors:



3.2 Timing report:



3.3 Utilization report:



3.4 Device:

