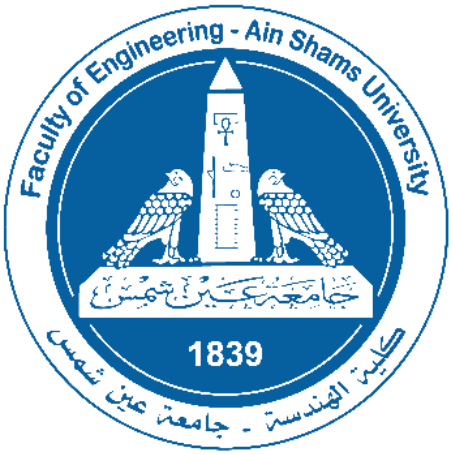


Faculty Of Engineering Ain Shams University

Electronics and Communications Department

Project (2)

SPI Slave with Single Port RAM

Prepared by

Omar Mohamed Hussein Mostafa

Table of Contents

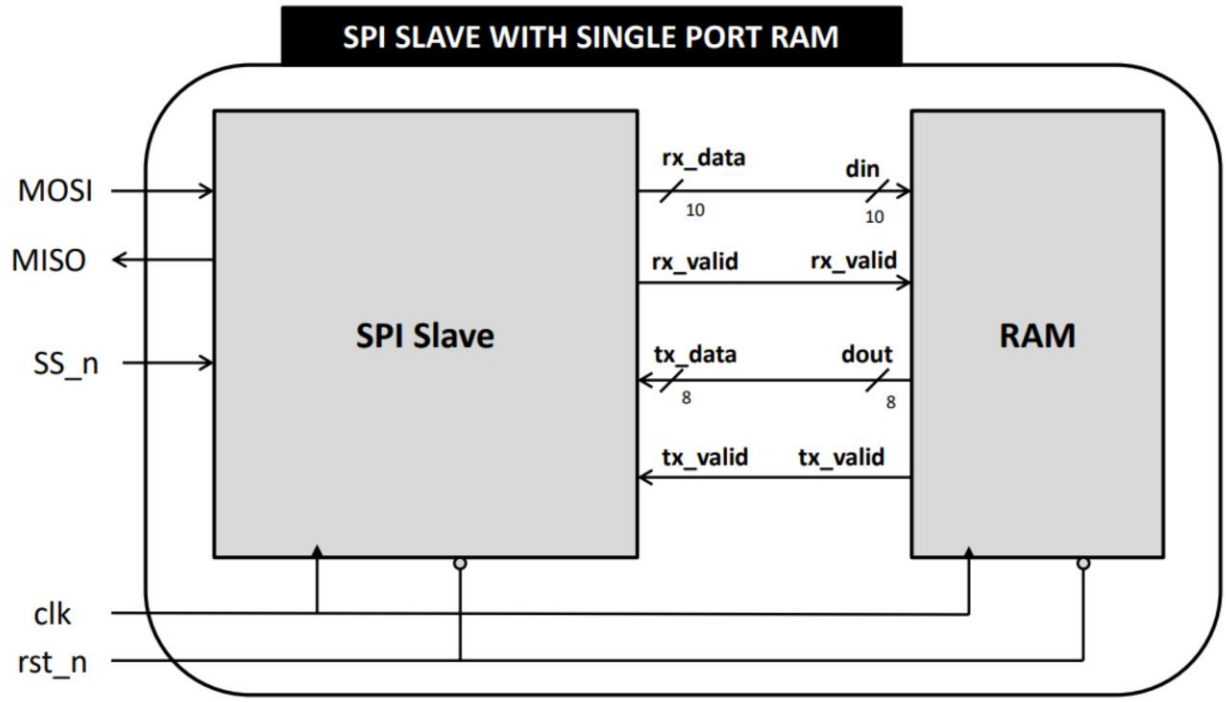
.....0
Project Description 4
Overview of the SPI Communication Protocol 4
SPI slave interface 4
Single Port Async RAM 6
Verilog code 7
RTL code for counter module 7
RTL code for RAM 8
RTL code for SPI slave 9
Gray encoding: 9
Synthesis schematic 9
Implementation schematic 9
Utilities report.....10
Timing report 10
One_hot encoding:.....10
Synthesis schematic 10
Implementation schematic 11
Utilities report.....11
Timing report 11
Johnson encoding: 12
Synthesis schematic 12
Implementation schematic 12
Utilities report.....13
Timing report: 13
Binary encoding:.....13

Synthesis schematic 13
Implementation schematic 14
Utilities report 14
Timing report 14
RTL code for the wrapped SPI with the RAM 17
Test bench code 18
Questasim Waveform 22
Reset and SS_n check 22
Write address check 22
Write data check 23
Read address check 24
Read data check 24
Repeated code check for the same output 25
Lint check 26
RTL analysis using vivado 26
RTL schematic 26
Synthesis analysis using vivado 28
Technology Schematic 28
Utilities report 29
Timing report 29
Messages 29
Debug setting 30
Implementation using vivado 31
Schematic 31
Utilities report 31
Timing report 32

Messages.....32
Bitstream generation.....32
Encoding in vivado 32
Debug cores 33

Project Description

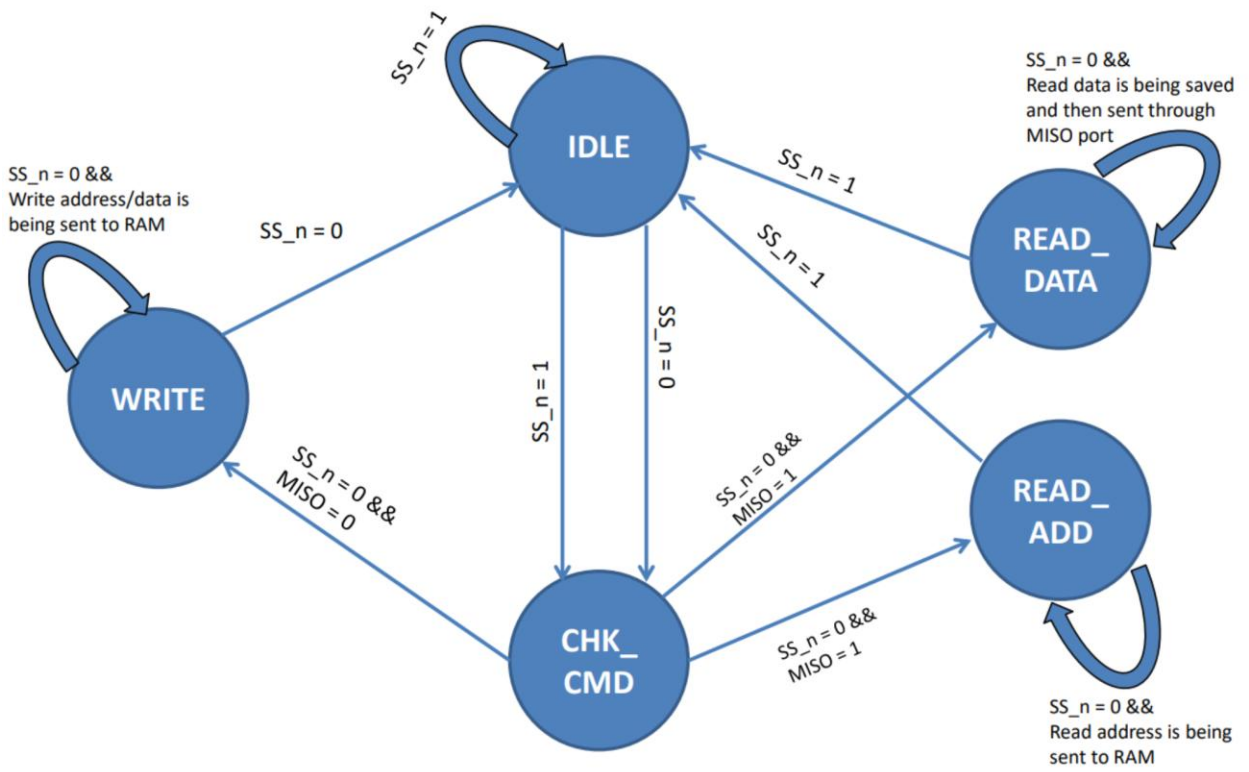
Overview of the SPI Communication Protocol



Serial Peripheral Interface (SPI**)** is a widely used synchronous communication protocol that enables a master device to exchange data with one or more slave devices. It operates over four main lines: SCLK (clock), MOSI (Master Out Slave In), MISO (Master In Slave Out), and SS_n (Slave Select). The master initiates communication and sends serial data to either write to or read from peripherals such as memory modules.

SPI slave interface

The SPI slave receives data from the master and, based on the received commands, generates control signals to the RAM to perform either a read or write operation through the signals rx_data,rx_valid,tx_data and tx_valid.



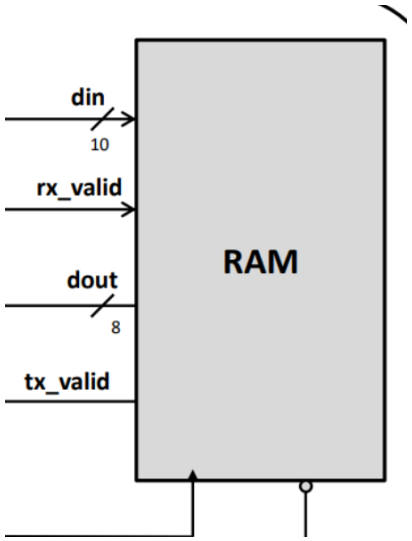
The SPI slave uses five states to manage communication and coordinate access to the RAM:

- **IDLE:** The default/reset state. No active communication is taking place while in this state.
- **CHK_CMD:** Entered once SS_n is asserted and communication begins. In this state the incoming MOSI bit is examined to determine whether the operation is a read or a write, and the FSM transitions accordingly.
- **WRITE:** This state handles the write operation to RAM. Although the write process consists of two logical steps—receiving the write address followed by the writing data—there is no need for dedicated RAM states like WRITE_ADDR and WRITE_DATA. After receiving the address, the FSM returns briefly to IDLE and then re-enters CHK_CMD to interpret the next byte, which is the write data. Because the address and data are sent by the master in quick

succession with no delay or separate transaction, the SPI slave can write the data to RAM immediately after receiving both, without needing intermediate RAM-specific states.

- **READ_ADD:** In this state the address of the location to be read is received from the master. After the address is captured, the FSM transitions to the next state. A delay is required here to allow the single-port RAM to output the data at that address.
- **READ_DATA:** The data fetched from RAM is shifted out to the master over MISO in this state.

Single Port Async RAM



According to the most significant two bits of the din the RAM decides whether to read_add/read_data or write_add/write_data. The rx_valid signals tell the ram to accept the din data for the address while the tx_valid tells the SPI to accept the dout data to the MISO.

Verilog code

RTL code for counter module

```
module down_counter#(parameter MOD=8,parameter COUNTER_BITS=3)(clk,rstn,enable,count);

input clk,rstn,enable;
output reg [COUNTER_BITS-1:0] count;

always @(posedge clk ) begin
    if(~rstn) begin
        count<=MOD-1;
    end
    else if(enable) begin
        if(count==0)begin
            count<=MOD-1;
        end
        else begin
           count<=count-1;
        end
    end
end
endmodule
```

RTL code for RAM

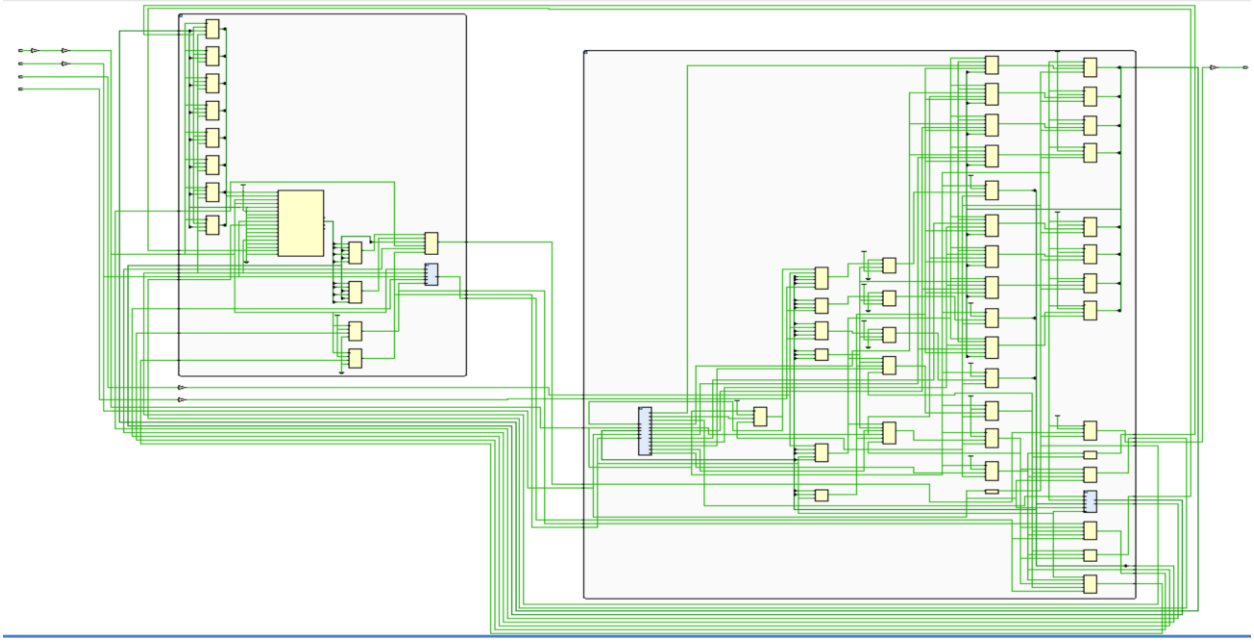
```
module SPR#(parameter MEM_DEPTH=256,parameter ADDR_SIZE=8)(din,clk,rstn,rx_valid,tx_valid,dout);

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

reg [7:0] MEM[255:0];

reg [7:0]addr; //internal address signal
wire [2:0] tx_counter;
reg sig;

down_counter #(8,3)data_out_counter(clk,rstn,sig,tx_counter);

always @(posedge clk) begin
    if(!rstn) begin
        dout<=8'b0;
        addr<=8'b0;
        tx_valid<=0;
        sig<=0;
    end
    else begin
        if(rx_valid) begin
            case({din[9:8]})
                2'b00: begin
                    addr<=din[7:0];
                    tx_valid<=0;

                end
                2'b01: begin
                    MEM[addr]<=din[7:0];
                    tx_valid<=0;

                end
                2'b10: begin
                    addr<=din[7:0];
                    tx_valid<=0;

                end
                2'b11: begin
                    dout<=MEM[addr];
                    tx_valid<=1;
                    sig<=1;
                end
            endcase
        end
        else if(tx_counter==0) begin
            tx_valid<=0;
            sig<=0;
        end
    end
end
endmodule

```

RTL code for SPI slave

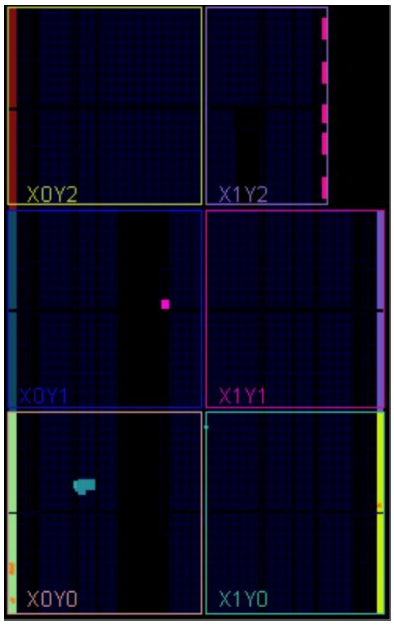
Here we want to work on the highest clock frequency, therefore, we will choose the encoding that performs on the maximum clock frequency.

Gray encoding:

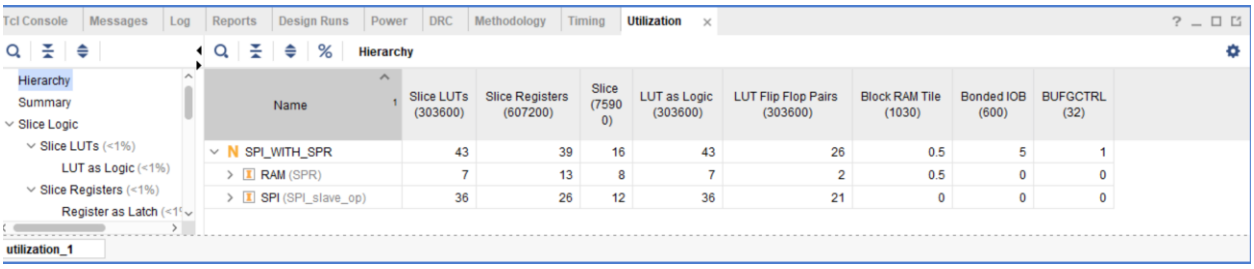
Synthesis schematic



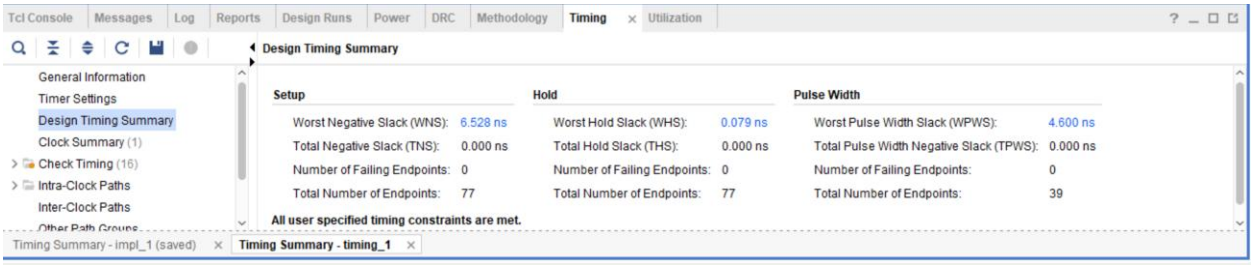
Implementation schematic



Utilities report

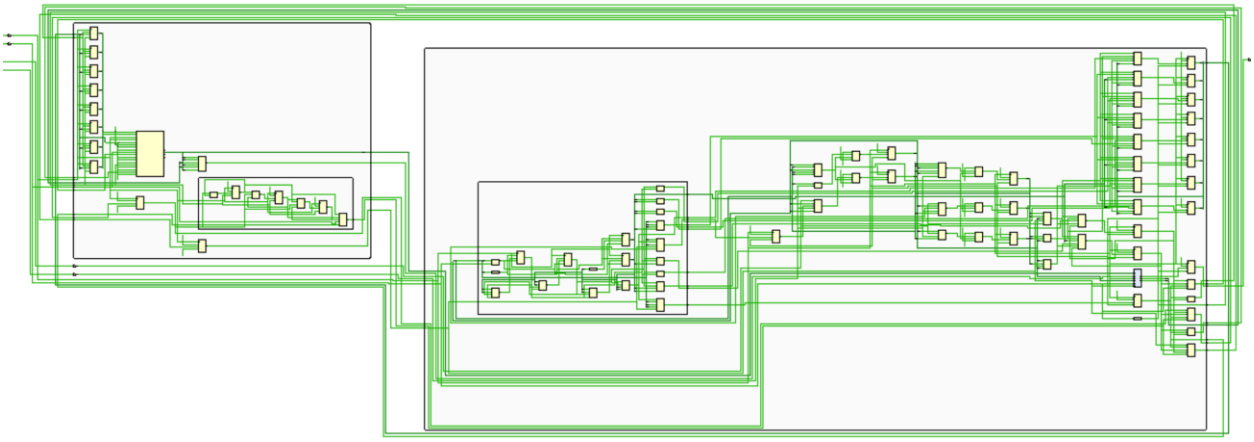


Timing report

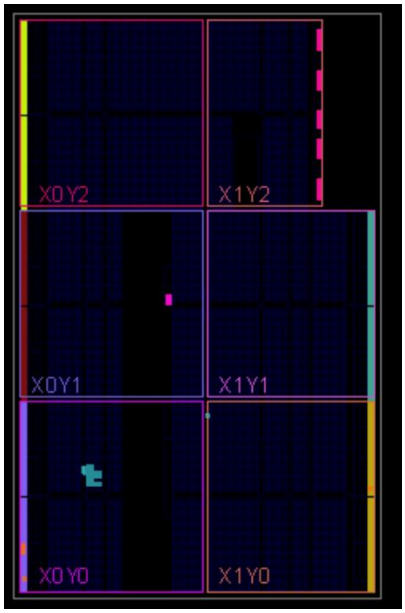


One_hot encoding:

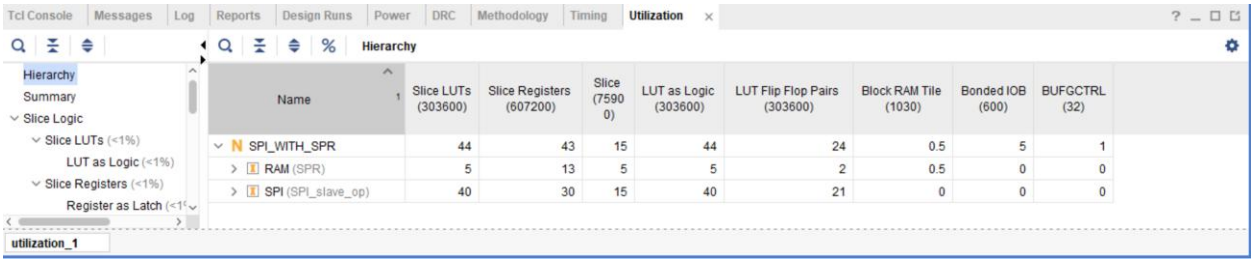
Synthesis schematic



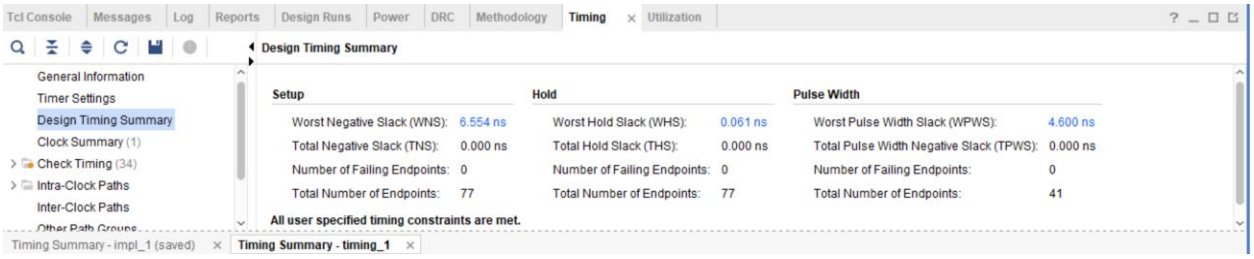
Implementation schematic



Utilities report

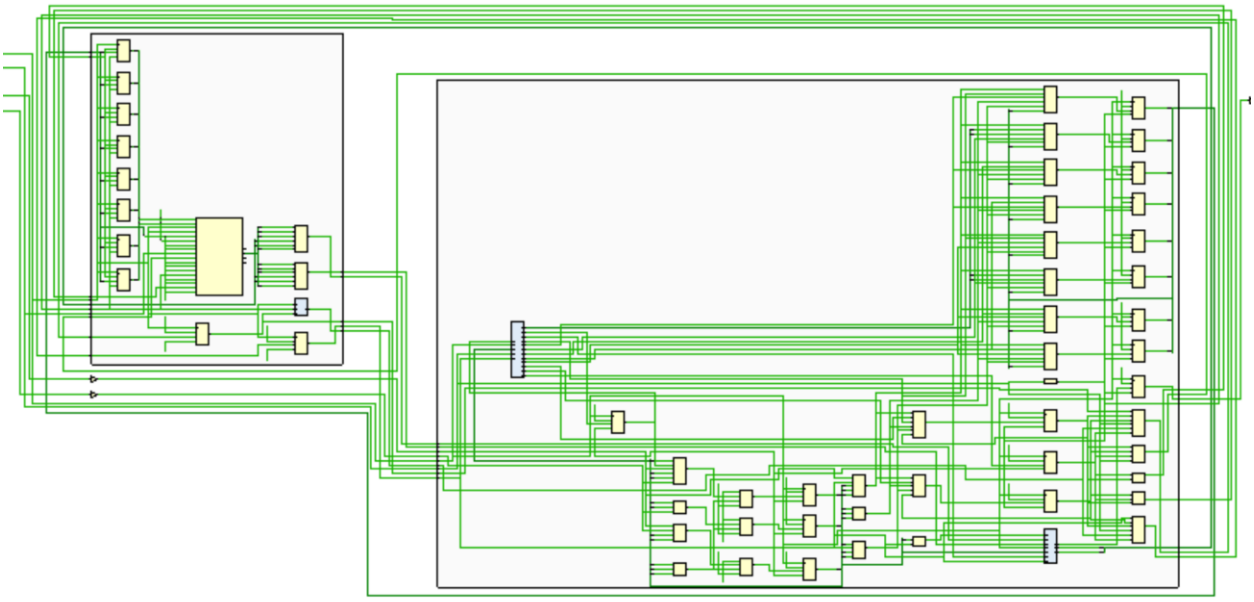


Timing report

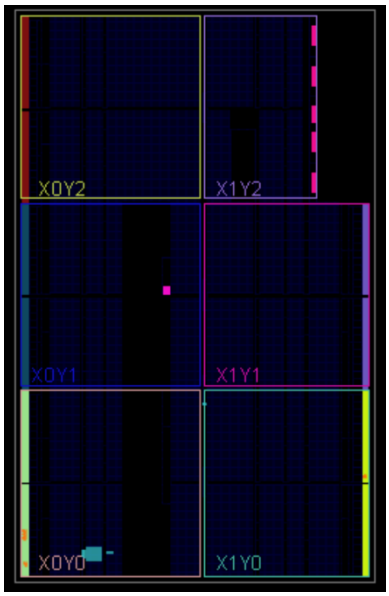


Johnson encoding:

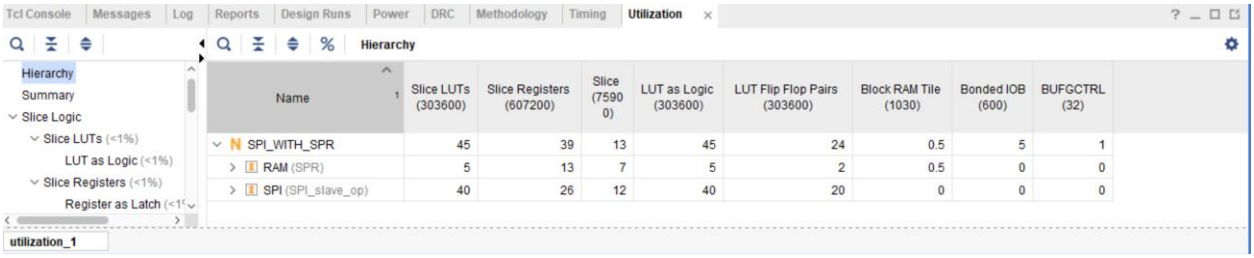
Synthesis schematic



Implementation schematic



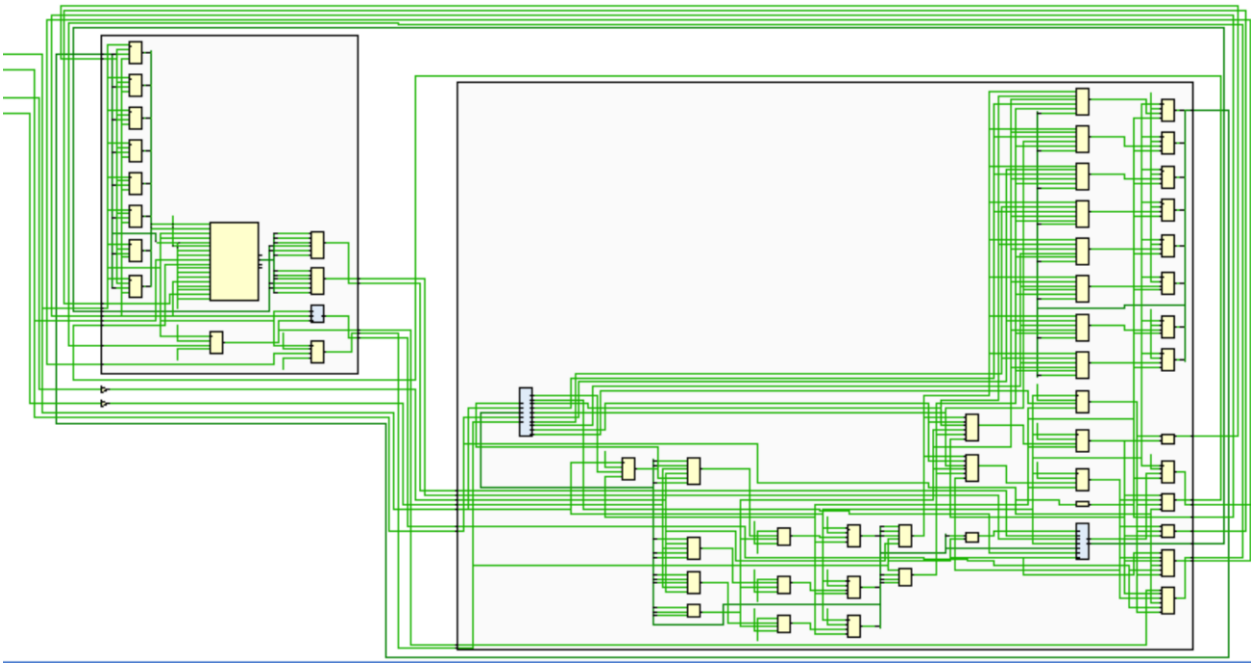
Utilities report



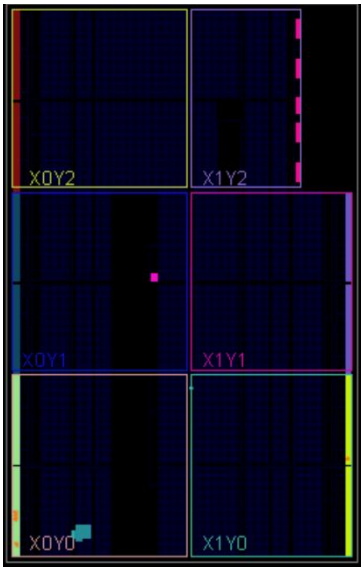
Timing report:

Binary encoding:

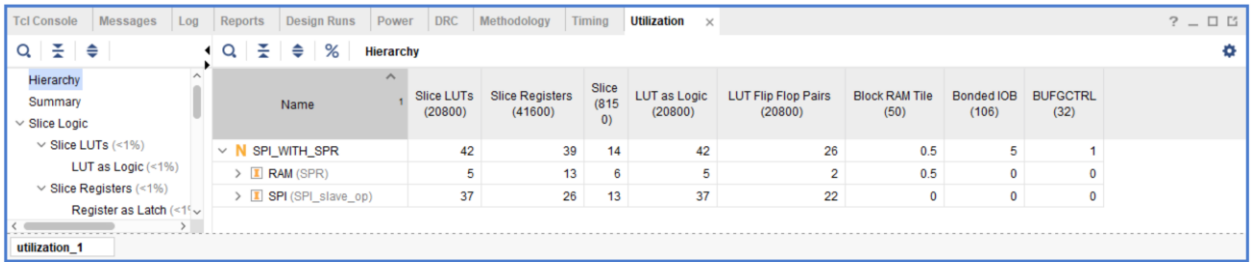
Synthesis schematic



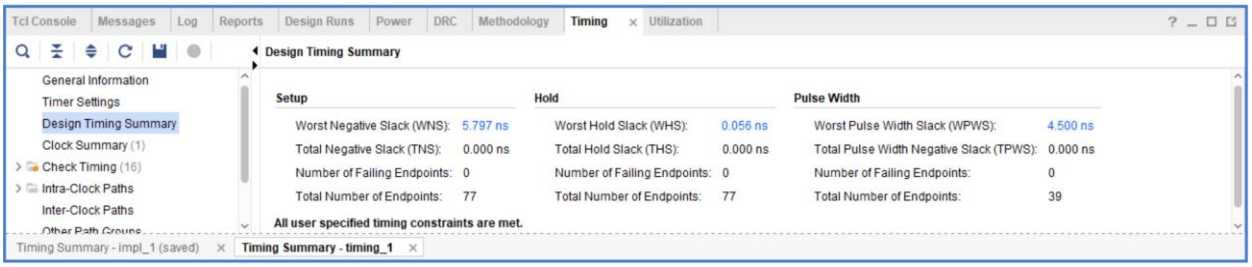
Implementation schematic



Utilities report



Timing report



So we will stick with the **one_hot** encoding as it gives us the best timing slack which will perform as the maximum clock frequency.

Therefore, our RTL code for the SPI slave would be:

```
module SPI_slave_op(MOSI,MISO,clk,rstn,rx_data,tx_data,rx_valid,tx_valid,SS_n);

//our state encoding;
parameter IDLE=3'b000;
parameter CHK_CMD=3'b001;
parameter WRITE =3'b010;
parameter READ_ADD=3'b011;
parameter READ_DATA=3'b100;


//SPI slave ports
input MOSI,clk,rstn,tx_valid,SS_n;
input [7:0]tx_data;
output reg MISO,rx_valid;
output reg [9:0]rx_data;

(*fsm_encoding="one_hot"*)
reg [2:0]ns,cs;//next and current state

//internal signals
wire [2:0]count;
wire [3:0]count_10;
reg Read_add_data_inx;

//counters
down_counter #(10,4) down_counter_10(clk,rstn && cs!=CHK_CMD ,cs!=IDLE,count_10);//counter counts from 9 to 0 ,10 states ,10 bits
down_counter #(8,3)tx_to_MISO(clk,rstn&& (cs!=CHK_CMD),(cs==READ_DATA && tx_valid),count);

//next state logic
always @(*) begin
    case (cs)
        IDLE:begin
            if(SS_n) ns=IDLE;
            else begin
                ns=CHK_CMD;
            end
        end
        CHK_CMD:begin
            if(SS_n) ns=IDLE;
            else begin
                if(MOSI) begin
                    if(!Read_add_data_inx) begin//read addr
                        ns=READ_ADD;
                    end
                    else if (Read_add_data_inx) begin
                        ns=READ_DATA;
                    end
                end
                else begin
                    ns=WRITE;
                end
            end
        end
        WRITE:begin
            if(SS_n) ns=IDLE;
            else begin
                ns=WRITE;
            end
        end

```

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

//memory logic
always @(posedge clk) begin
    if (!rstn) begin
        cs<=3'b000;
    end
    else begin
        cs<=ns;
    end
end

//output logic
always @(posedge clk) begin
    if(!rstn) begin
        rx_data<=8'b0;
        rx_valid<=0;
        MISO<=0;
        Read_add_data_inx<=0;
    end
    else begin
        case(cs)
            WRITE: begin
                rx_data[count_10]<=MOSI;
                MISO<=0;
                if (count_10==4'b0) begin
                   rx_valid<=1;
                end
                else begin
                        rx_valid<=0;
                    end
            end
            READ_ADD: begin
                rx_data[count_10]<=MOSI;
                MISO<=0;
                if (count_10==4'b0) begin
                   rx_valid<=1;
                    Read_add_data_inx<=1;
                end
                else begin
                        rx_valid<=0;
                    end
            end
            READ_DATA: begin
                if(!tx_valid) begin
                    rx_data[count_10]<=MOSI;
                    MISO<=0;
                    if (count_10==4'b0) begin
                       rx_valid<=1;
                       Read_add_data_inx<=0;
                    end
                    else begin
                        rx_valid<=0;
                    end
                end
                else begin
                    MISO<=tx_data[count]; // getting the data out of the SPI through the MISO
                    rx_valid<=0;
                    rx_data<='b0;
                end
            end
```

```
            CHK_CMD :begin
                rx_valid<=0;
                rx_data<=0;
                MISO<=0;
            end
            IDLE: begin
                rx_valid<=0; // reset all values to start communication again
                rx_data<=0;
                MISO<=0;
            end
            default :begin
                rx_valid<=0;
                rx_data<=0;
                MISO<=0;
            end
        endcase
    end
end
endmodule
```

RTL code for the wrapped SPI with the RAM

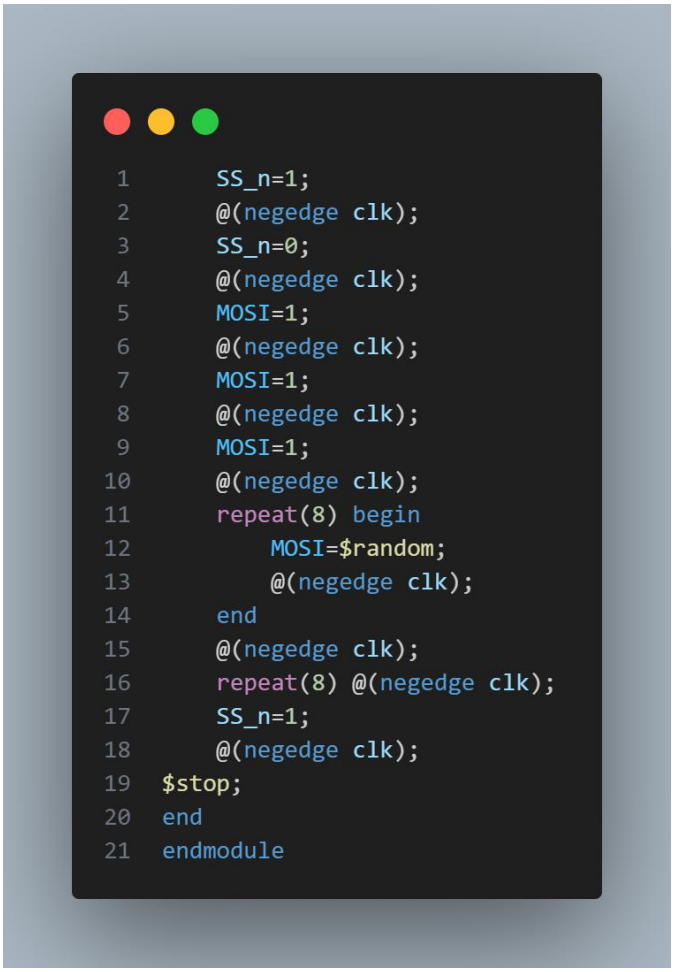
```
module SPI_WITH_SPR#(parameter MEM_DEPTH=256,parameter ADDR_SIZE=8)(clk,rstn,MISO,MOSI,SS_n);

input clk,rstn,SS_n,MOSI;
output MISO;

wire tx_valid,rx_valid;
wire [7:0] tx_data;
wire[9:0] rx_data;

SPI_slave_op SPI(MOSI,MISO,clk,rstn,rx_data,tx_data,rx_valid,tx_valid,SS_n);
SPR #(MEM_DEPTH,ADDR_SIZE)RAM(rx_data,clk,rstn,rx_valid,tx_valid,tx_data);

endmodule
```

Test bench code

```
module SPI_WITH_SPR_tb();

reg clk,rstn,SS_n,MOSI;
wire MISO;

SPI_WITH_SPR #(256,8)DUT(clk,rstn,MISO,MOSI,SS_n);

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

initial begin
//reset
    rstn=0;
    MOSI=0;
    SS_n=1;
    repeat(20) begin
    @(negedge clk);
    if(MOSI!=0) begin
        $display("Error in reset");
        $stop;
    end
    end
//IDLE test
    rstn=1;
    repeat(10) begin
        @(negedge clk);
        if(MOSI!=0) begin
        $display("Error in IDLE");
        $stop;
    end
    end
//write add test
    SS_n=0;
    @(negedge clk);//now we are at the CHk_CMD
    MOSI=0;
    @(negedge clk);//now we are at the WRITE
    MOSI=0;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);//now we sent 2 zero for the ram to write address
    repeat(8) begin //random address
        MOSI=$random;
        @(negedge clk);
    end
    SS_n=1;//end communication ,now the ram should have taken this value for the address
    @(negedge clk);
    SS_n=0;//start communication
    @(negedge clk);//at CHK_CMD
     MOSI=0;//go to write state
    @(negedge clk);//at write
    MOSI=0;
    @(negedge clk);
    MOSI=1;//write data
    @(negedge clk);

```

```
repeat(8) begin //write this random data
    MOSI=$random;
    @(negedge clk);
end
SS_n=1;//end comminication
@(negedge clk);
SS_n=0;//start communication
@(negedge clk);//at check command
MOSI=1;
@(negedge clk);//at Read command
MOSI=1;
@(negedge clk);
MOSI=0;
@(negedge clk);//we sent 10 for the ram to read add of the following
    MOSI=0;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);// the address is 01111110 which is 7E in hexa which was the previous random add
SS_n=1;//end communication
@(negedge clk);
SS_n=0;
@(negedge clk);
MOSI=1;//read
@(negedge clk);
MOSI=1;
@(negedge clk);
MOSI=1;
@(negedge clk); //read data at address 7E which is the one we wrote in the write state
repeat(8) begin
    MOSI=$random;
    @(negedge clk); //dummy bits
end
@(negedge clk);
repeat(8) @(negedge clk);//data out
SS_n=1;//end communication
@(negedge clk);

```

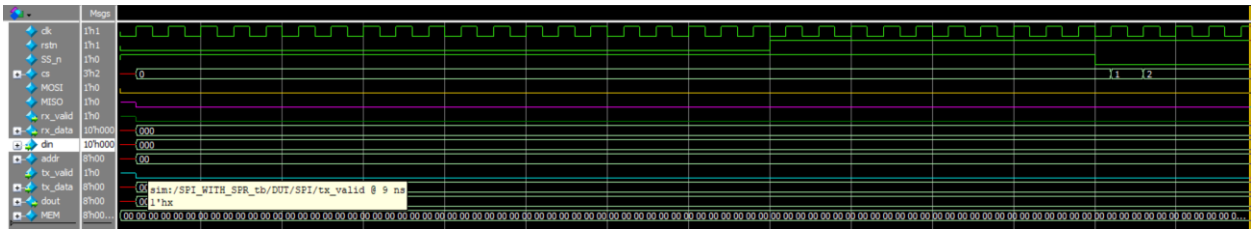
```
//////////////repeat the same process again to ensure every thing is right every time///////////////

    SS_n=0;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);
    repeat(8) begin
        MOSI=$random;
        @(negedge clk);
    end
    SS_n=1;
    @(negedge clk);
    SS_n=0;
    @(negedge clk);
     MOSI=0;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    repeat(8) begin
        MOSI=$random;
        @(negedge clk);
    end
    SS_n=1;
    @(negedge clk);
    SS_n=0;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=0;
    @(negedge clk);
        MOSI=0;
        @(negedge clk);
        MOSI=1;
        @(negedge clk);
        MOSI=1;
        @(negedge clk);
        MOSI=1;
        @(negedge clk);
        MOSI=1;
        @(negedge clk);
        MOSI=1;
        @(negedge clk);
        MOSI=1;
        @(negedge clk);
        MOSI=0;
        @(negedge clk);
```



```
    SS_n=1;
    @(negedge clk);
    SS_n=0;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    MOSI=1;
    @(negedge clk);
    repeat(8) begin
        MOSI=$random;
        @(negedge clk);
    end
    @(negedge clk);
    repeat(8) @(negedge clk);
    SS_n=1;
    @(negedge clk);
$stop;
end
endmodule
```

Questasim Waveform

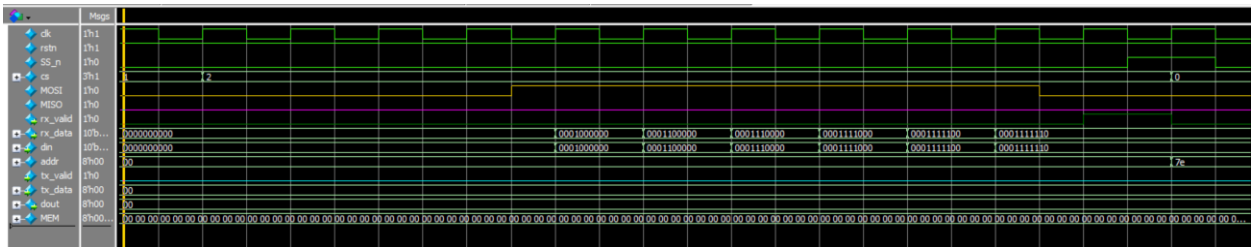
Reset and SS_n check



Here, as long as the RESET is low, all outputs are zeros.

Also, if the RESET is HIGH and SS_n is high the cs would be IDLE(zero).

Write address check



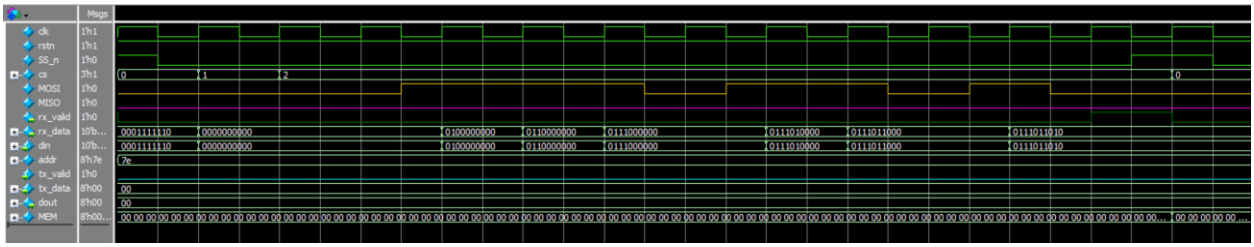
The SS_n became **0**, the communication starts.

The cs goes from IDLE to CHK_CMD then to Write according to the MOSI.

Then 10 bits during 10 clock cycles are sent with the most significant two bits are **00** to write address, after the ten cycles are finished the rx_valid became high to tell the ram to accept the din as an address.

Then the SS_n became **1** to end the communication.

Write data check



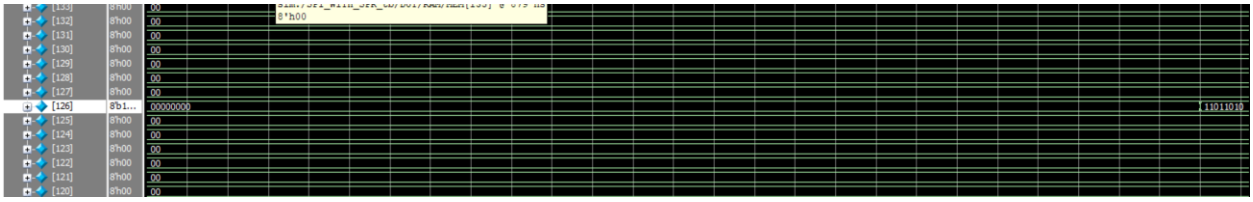
The SS_n became **0**, the communication starts.

The cs goes from IDLE to CHK_CMD then to Write according to the MOSI.

Then 10 bits during 10 clock cycles are sent with the most significant two bits are **01** to write data, after the ten cycles are finished the rx_valid became high to tell the ram to write the din in the previous address with was $\mathbf{126_D}$ $\boldsymbol{or}$ $\mathbf{7E_H}$.

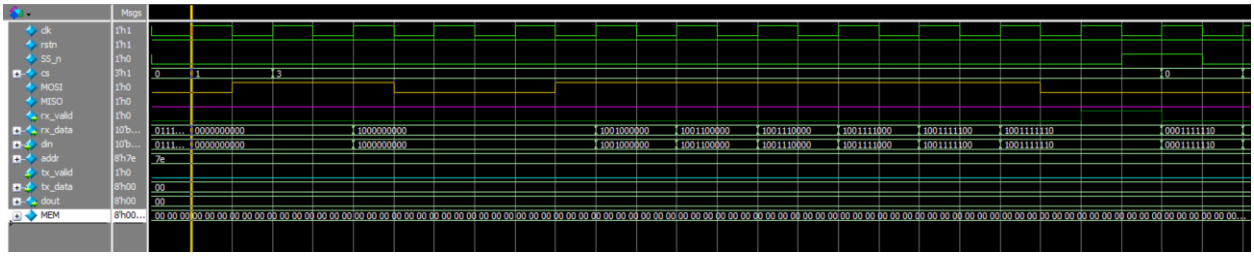
Then the SS_n became **1** to end the communication.

Let's go to MEM **[126]** to see if the data **11011010** is written or not.



Therefore the write commands are working well ✓.

Read address check



The SS_n became **0**, the communication starts.

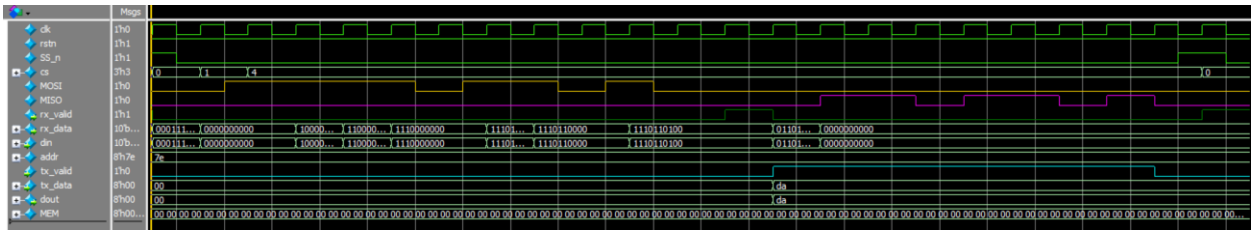
The cs goes from IDLE to CHK_CMD then to READ according to the MOSI.

We will of course read address first before we read the data.

Then 10 bits during 10 clock cycles are sent with the most significant two bits are **10** to read address, after the ten cycles are finished the rx_valid became high to tell the ram to accept the din as an address to be read later.

Then the SS_n became **1** to end the communication.

Read data check



The SS_n became **0**, the communication starts.

The cs goes from IDLE to CHK_CMD then to READ according to the MOSI.

After reading the address the previous state, now it's time to read the data in this address

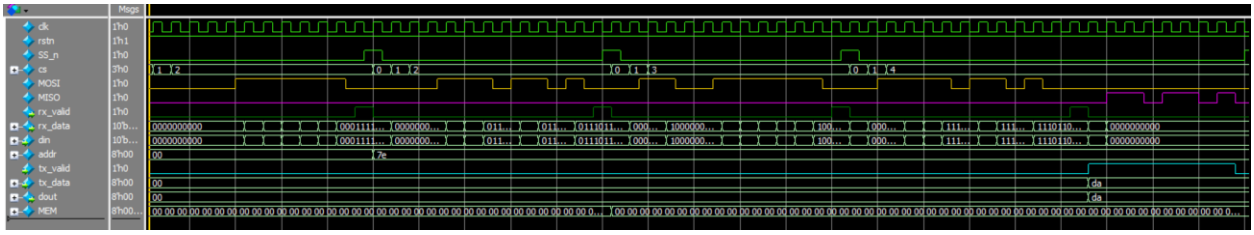
Then 10 bits during 10 clock cycles are sent with the most significant two bits are **11** to read address, therefore the RAM understood that the next bits are dummy bits, and it should read the data stored in the read address.

So after the dummy bits are sent the tx_valid raise up and the dout is loaded with the data, then the MISO starts to send the data out to the master.

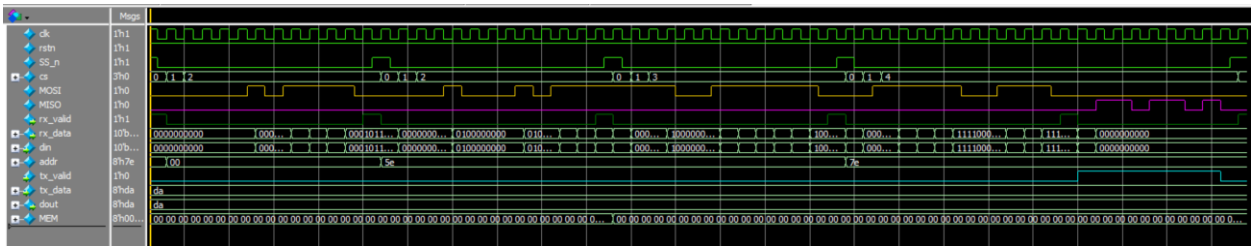
Then the SS_n became **1** to end the communication.

Repeated code check for the same output

First cycle:

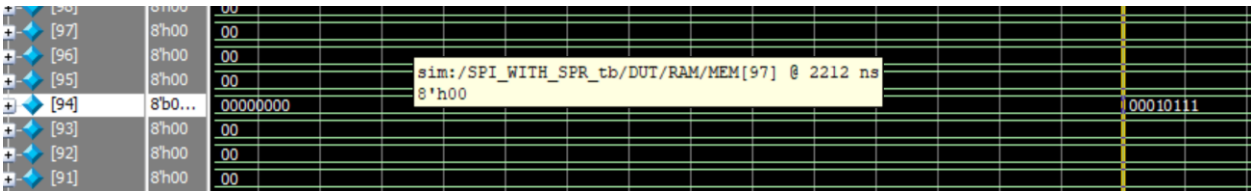


Second cycle:

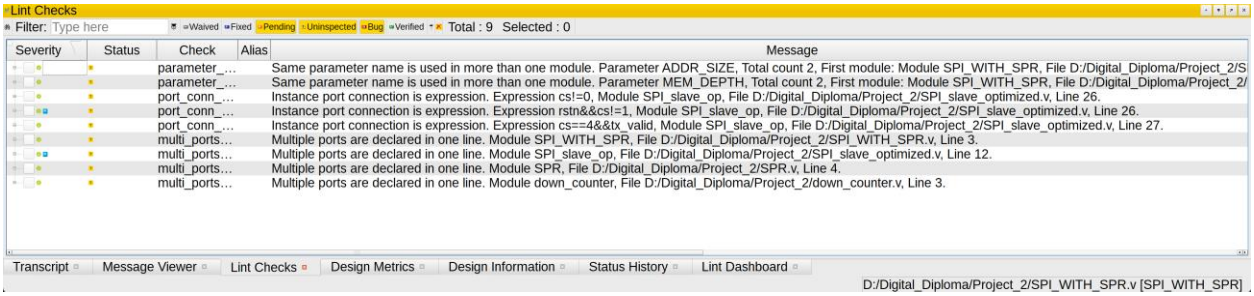


The reading cycles are the same as we inserted the same inputs, but for the write cycles we loaded another values, so the write address became $\mathbf{5E_H or\ 94_D}$ and the write data is **00010111**

MEM [94] check:

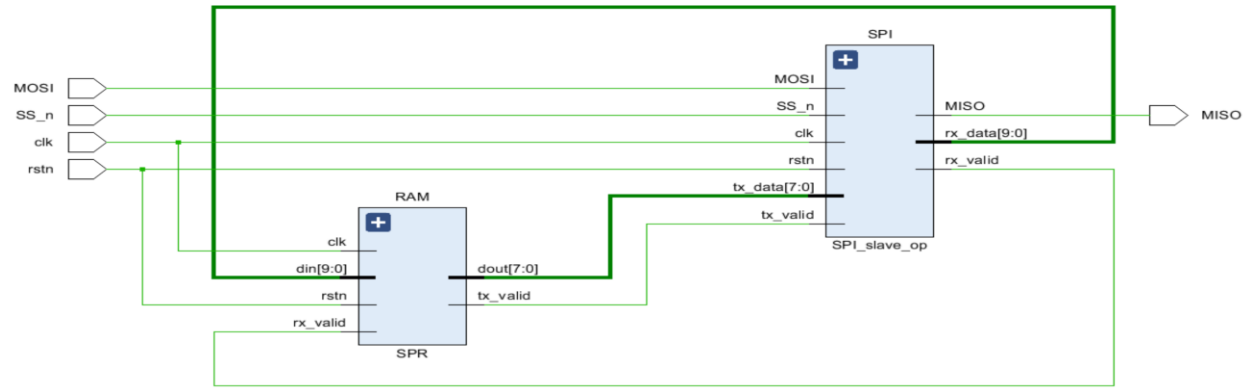


Lint check

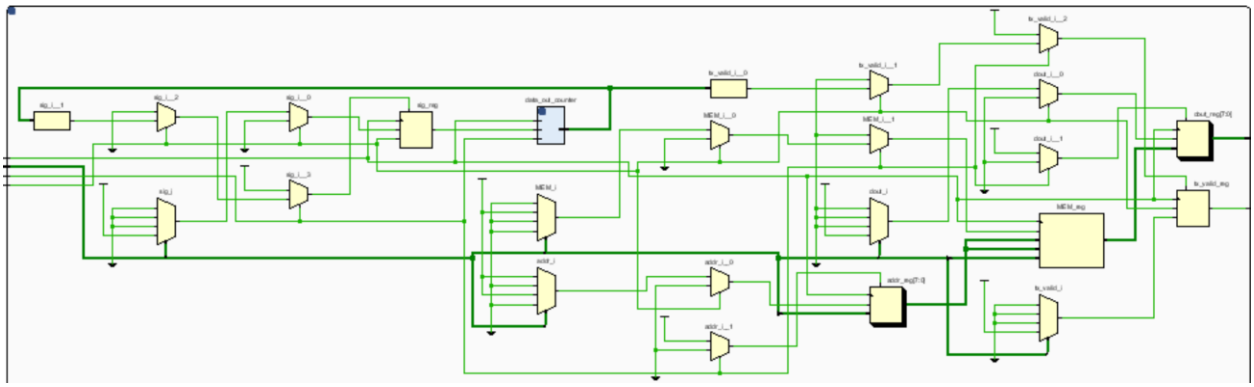


RTL analysis using vivado

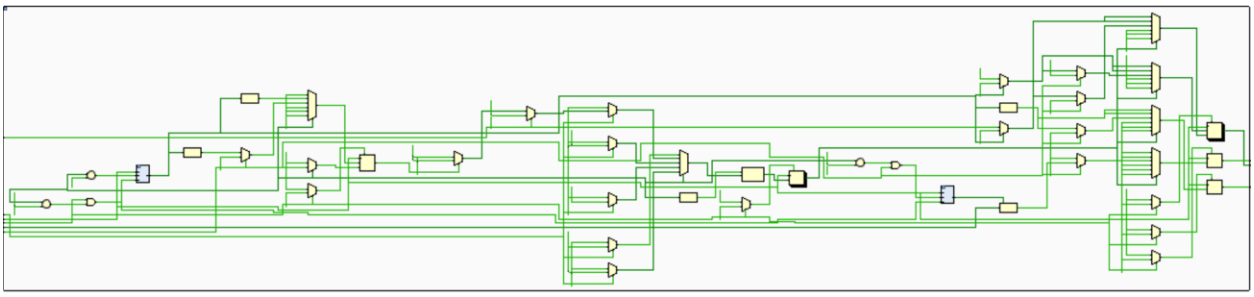
RTL schematic



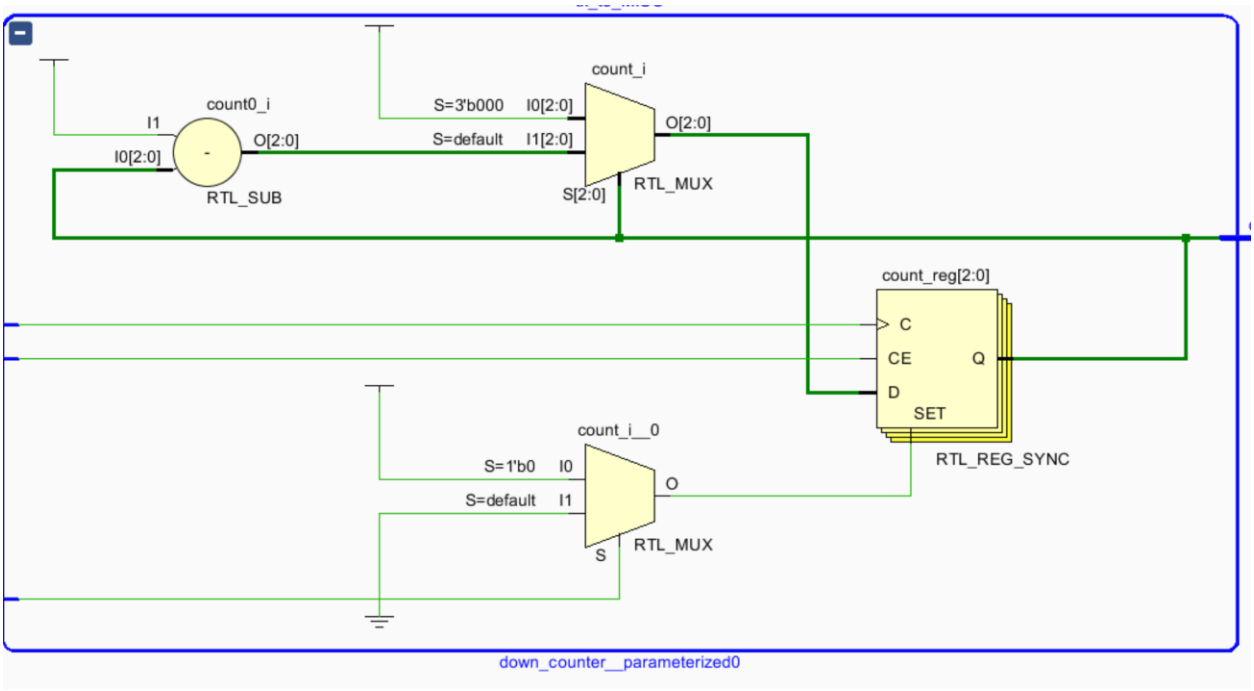
For a deeper look inside the RAM:



For a deeper look inside the SPI slave:

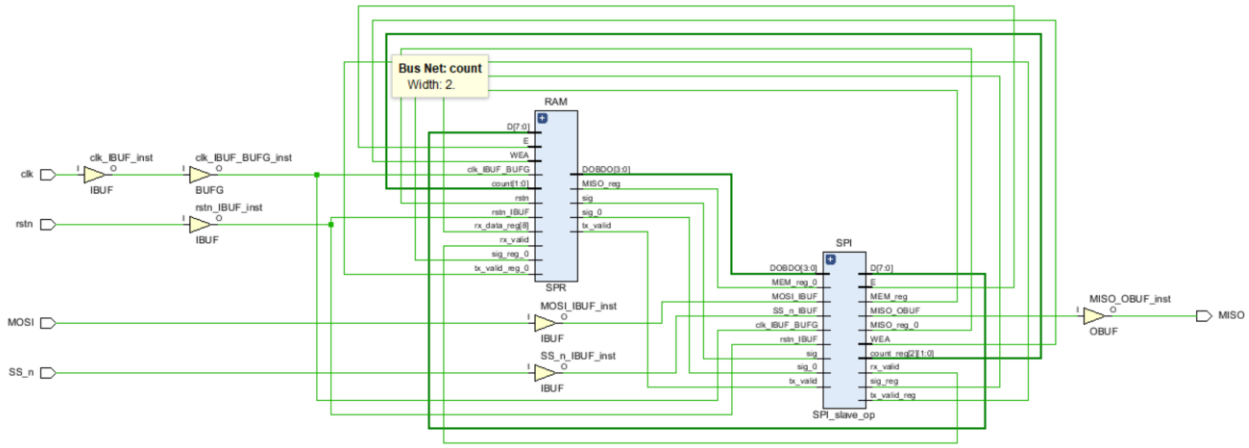


For a deeper look inside the counter:

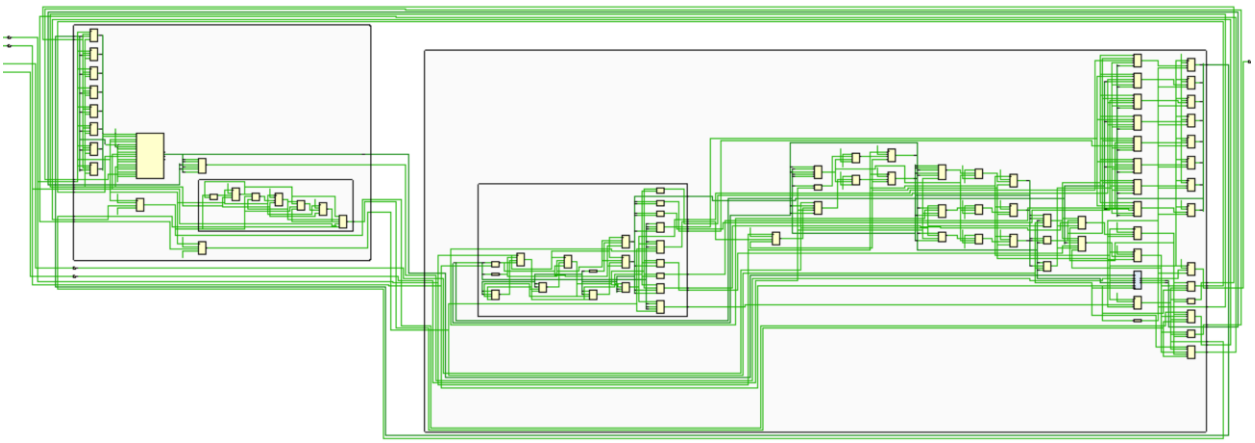


Synthesis analysis using vivado

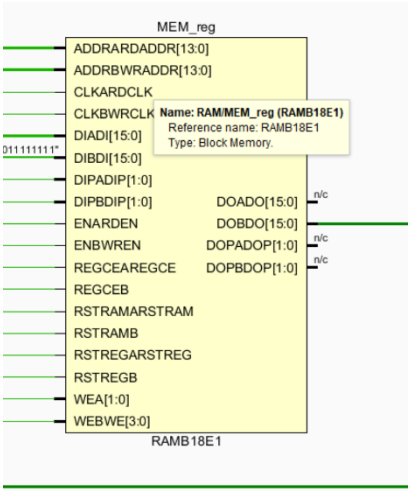
Technology Schematic



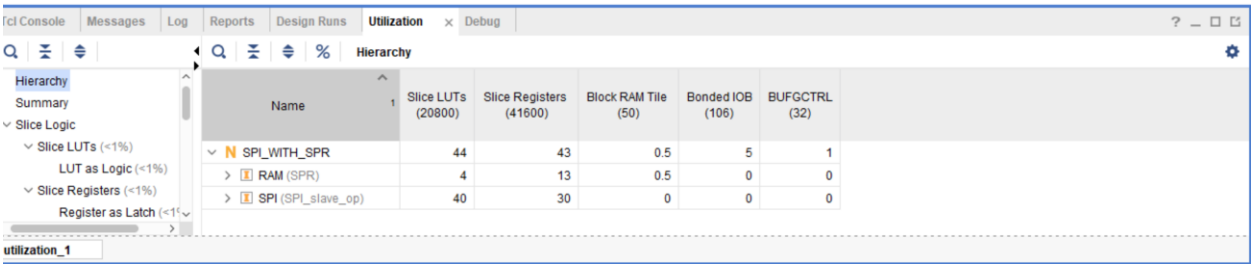
With more details inside:



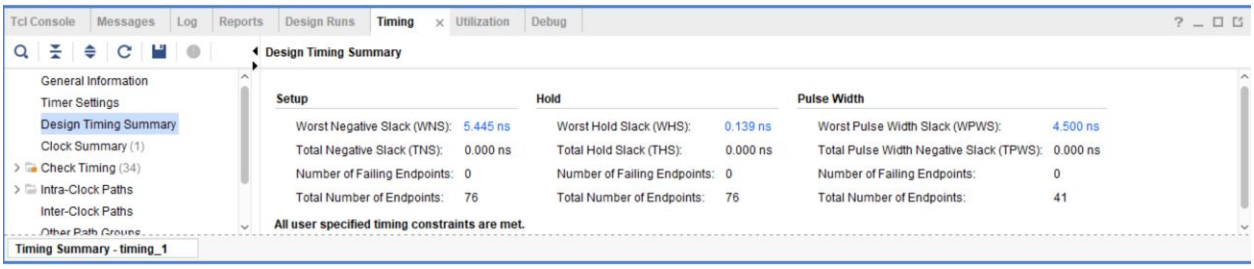
Zooming at the big yellow block:



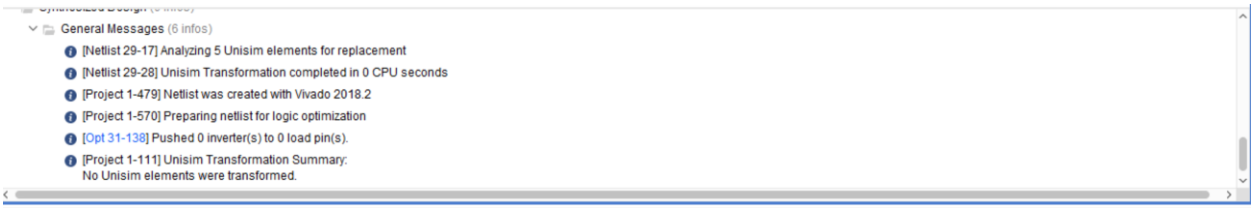
Utilities report



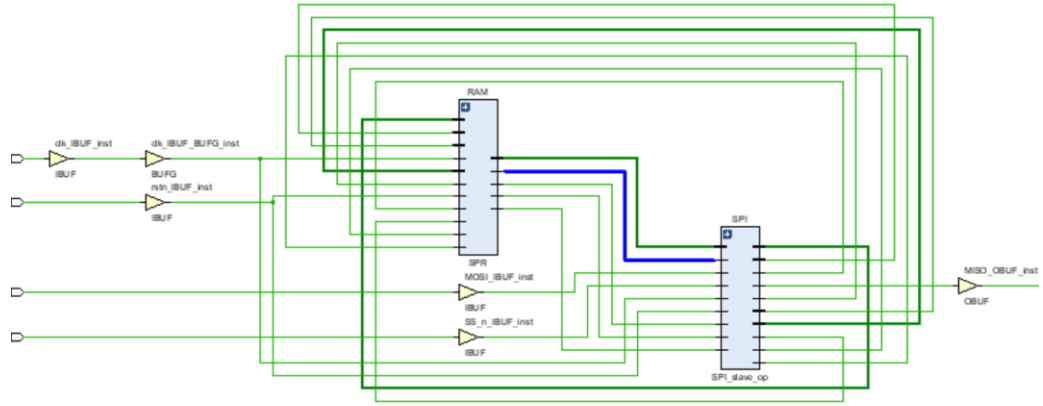
Timing report



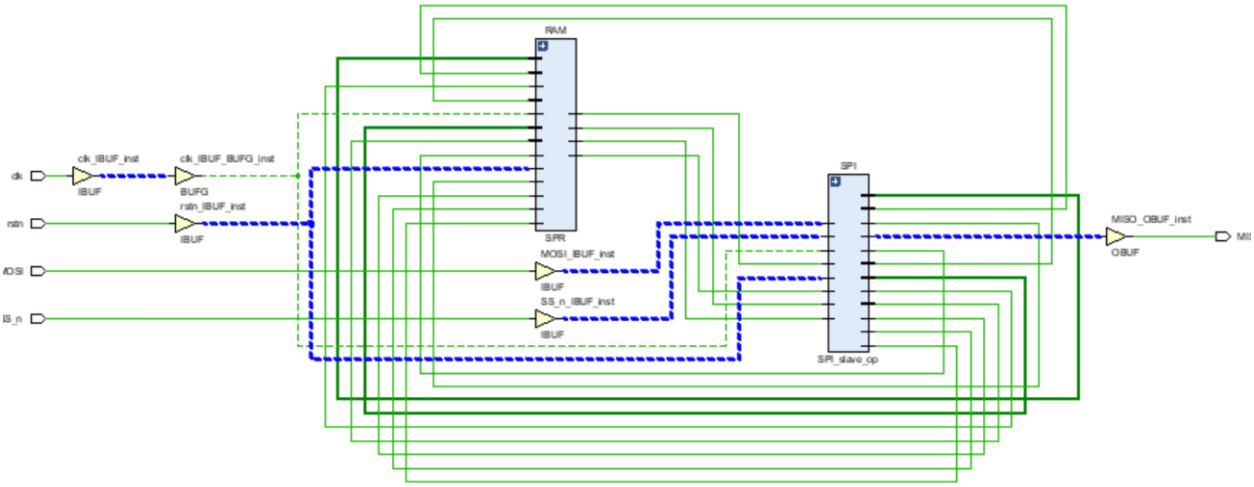
Messages



Worst path:

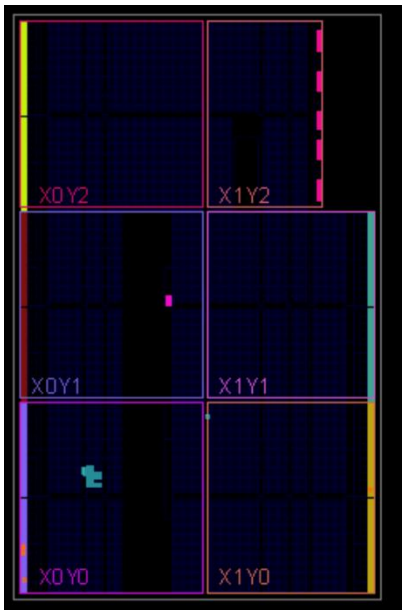


Debug setting

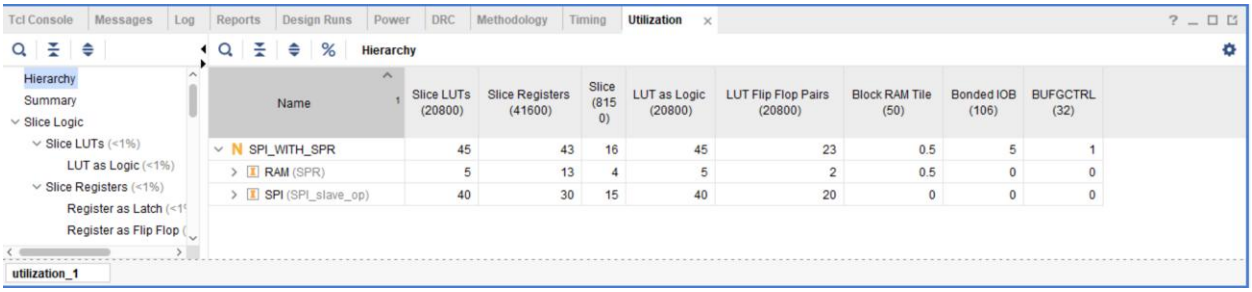


Implementation using vivado

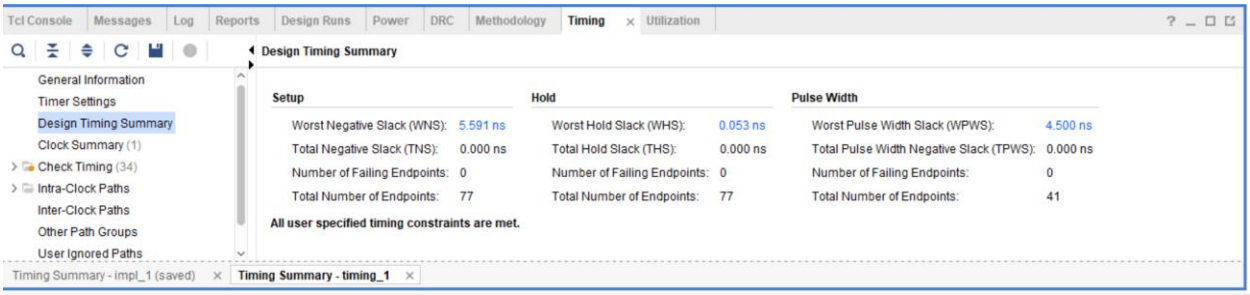
Schematic



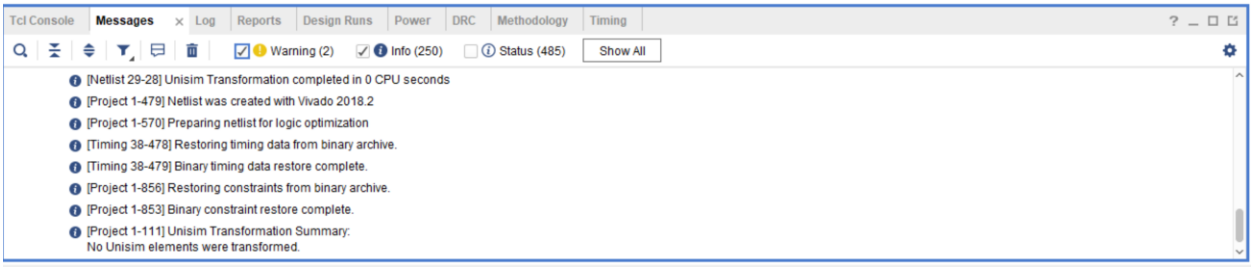
Utilities report



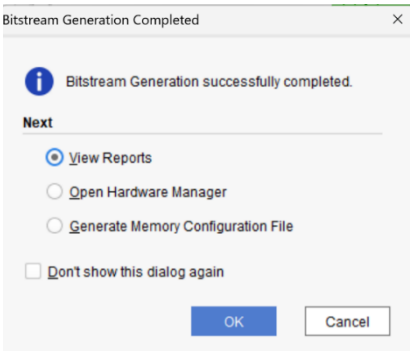
Timing report



Messages



Bitstream generation



Encoding in vivado

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

As we said before the one hot encoding was the best with the maximum frequency clock.

Debug cores

```
create_debug_core u_ila_0 ila
set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
set_property ALL_PROBE_SAME_MU_CNT 1 [get_debug_cores u_ila_0]
set_property C_ADV_TRIGGER false [get_debug_cores u_ila_0]
set_property C_DATA_DEPTH 1024 [get_debug_cores u_ila_0]
set_property C_EN_STRG_QUAL false [get_debug_cores u_ila_0]
set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ila_0]
set_property C_TRIGIN_EN false [get_debug_cores u_ila_0]
set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
set_property port_width 1 [get_debug_ports u_ila_0/clk]
connect_debug_port u_ila_0/clk [get_nets [list clk_IBUF_BUFG]]
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe0]
set_property port_width 1 [get_debug_ports u_ila_0/probe0]
connect_debug_port u_ila_0/probe0 [get_nets [list clk_IBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe1]
set_property port_width 1 [get_debug_ports u_ila_0/probe1]
connect_debug_port u_ila_0/probe1 [get_nets [list MISO_OBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe2]
set_property port_width 1 [get_debug_ports u_ila_0/probe2]
connect_debug_port u_ila_0/probe2 [get_nets [list MOSI_IBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe3]
set_property port_width 1 [get_debug_ports u_ila_0/probe3]
connect_debug_port u_ila_0/probe3 [get_nets [list rstn_IBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe4]
set_property port_width 1 [get_debug_ports u_ila_0/probe4]
connect_debug_port u_ila_0/probe4 [get_nets [list SS_n_IBUF]]
set_property C_CLK_INPUT_FREQ_HZ 300000000 [get_debug_cores dbg_hub]
set_property C_ENABLE_CLK_DIVIDER false [get_debug_cores dbg_hub]
set_property C_USER_SCAN_CHAIN 1 [get_debug_cores dbg_hub]
connect_debug_port dbg_hub/clk [get_nets clk_IBUF_BUFG]
```