



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

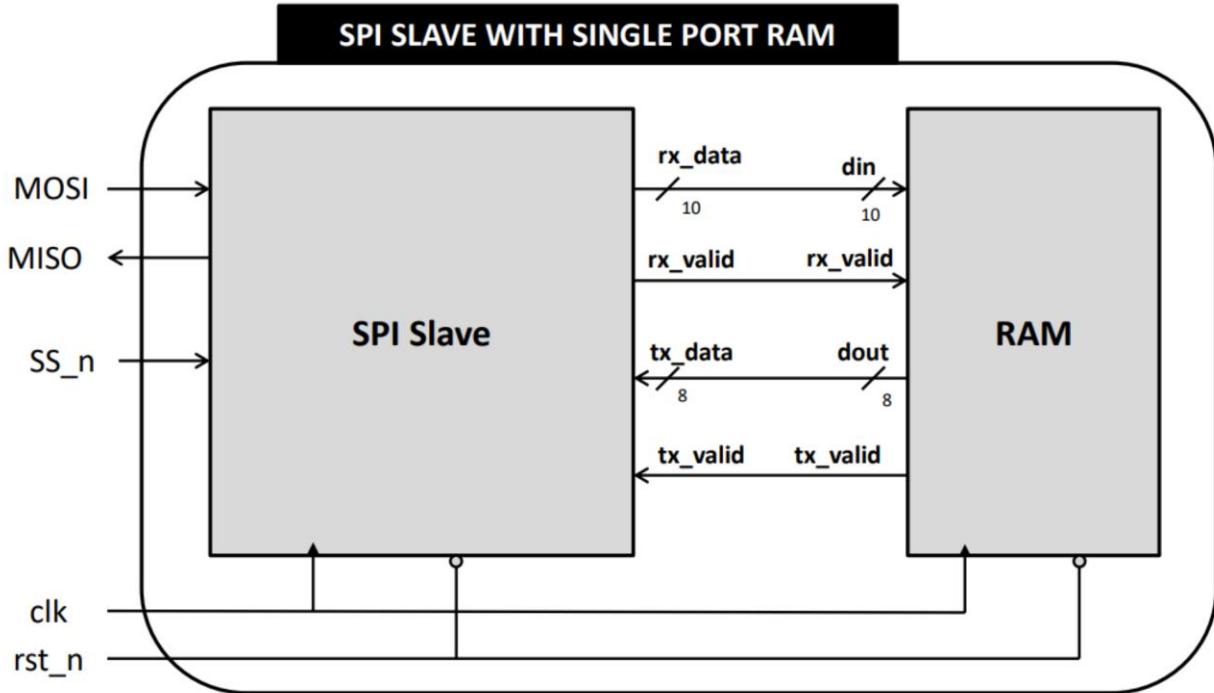
.....	0
Project Description	4
Overview of the SPI Communication Protocol	4
SPI slave interface	4
Single Port Async RAM	6
Verilog code	7
RTL code for counter module	7
RTL code for RAM	8
RTL code for SPI slave	9
Gray encoding:	9
Synthesis schematic	9
Implementation schematic	9
Utilities report.....	10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

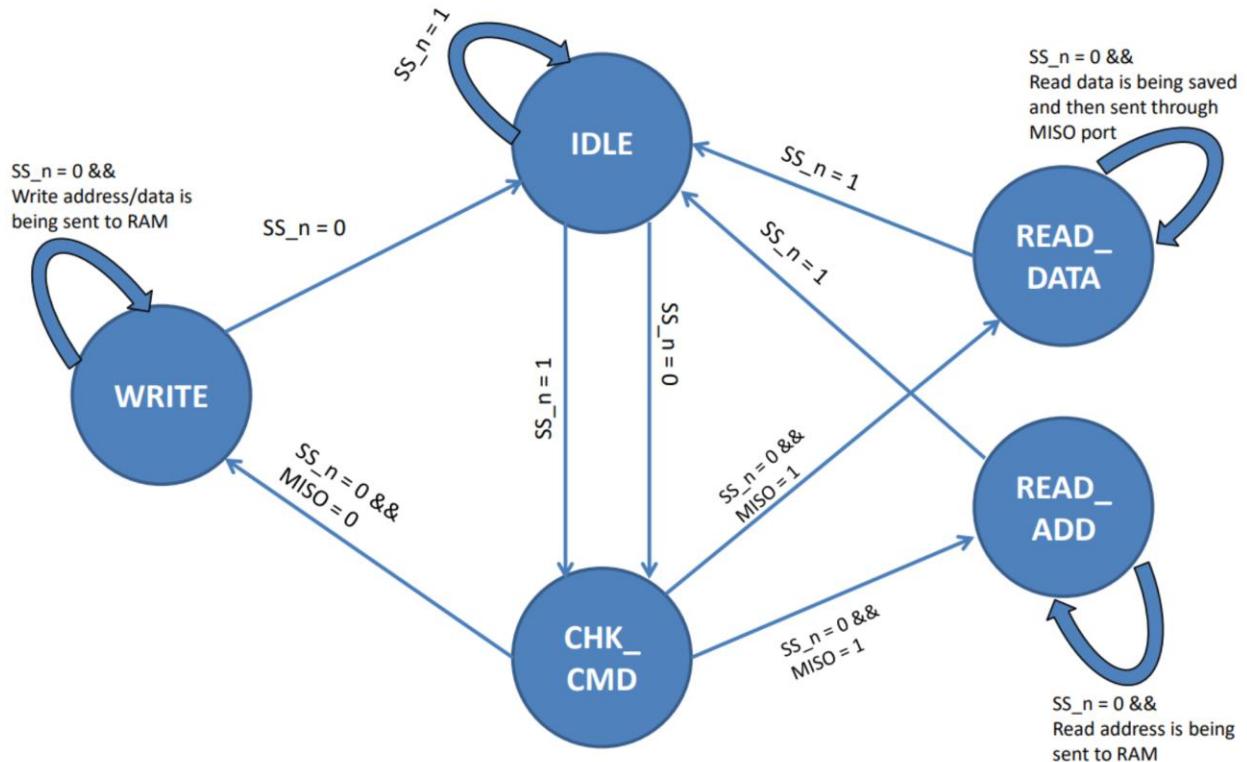
Overview of the SPI Communication Protocol



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.



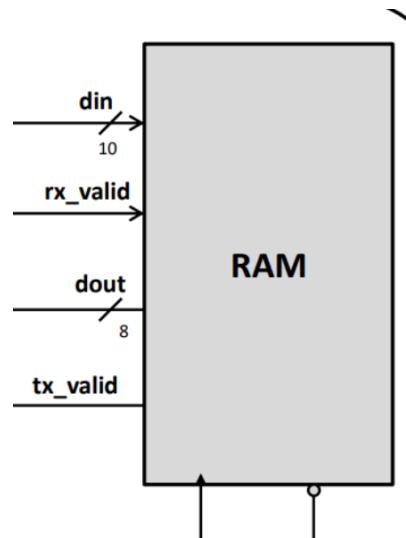
The SPI slave uses five states to manage communication and coordinate access to the RAM:

- **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

```
● ● ●
1 module down_counter #(parameter MOD=8, parameter COUNTER_BITS=3)(clk,rstn,enable,count);
2
3 input clk,rstn,enable;
4 output reg [COUNTER_BITS-1:0] count;
5
6 always @(posedge clk) begin
7     if(~rstn) begin
8         count<=MOD-1;
9     end
10    else if(enable) begin
11        if(count==0)begin
12            count<=MOD-1;
13        end
14        else begin
15            count<=count-1;
16        end
17    end
18 end
19 endmodule
```

RTL code for RAM

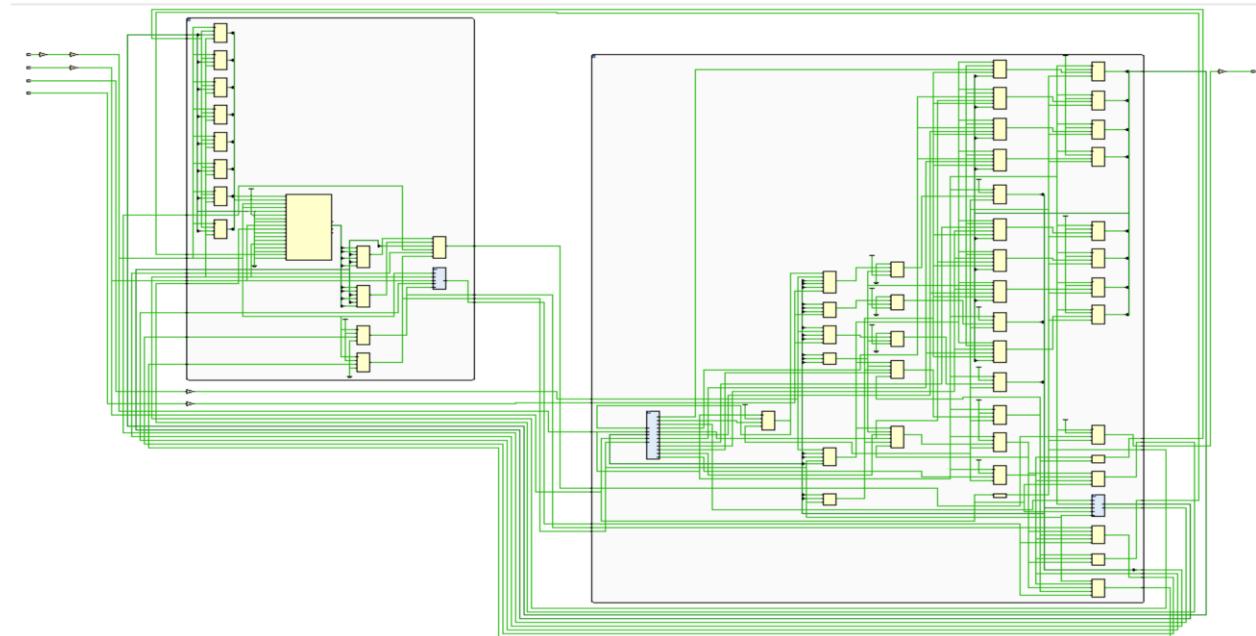
```
1 module SPR#(parameter MEM_DEPTH=256,parameter ADDR_SIZE=8)(din,clk,rstn,rx_valid,tx_valid,dout);
2
3   input [9:0]din;
4   input clk,rstn,rx_valid;
5   output reg tx_valid;
6   output reg [7:0]dout;
7
8   reg [7:0] MEM[255:0];
9
10  reg [7:0]addr; //internal address signal
11  wire [2:0] tx_counter;
12  reg sig;
13
14  down_counter #(8,3) data_out_counter(clk,rstn,sig,tx_counter);
15
16  always @(posedge clk) begin
17    if(!rstn) begin
18      dout<=8'b0;
19      addr<=8'b0;
20      tx_valid<=0;
21      sig<=0;
22    end
23    else begin
24      if(rx_valid) begin
25        case({din[9:8]})
26          2'b00: begin
27            addr<=din[7:0];
28            tx_valid<=0;
29
30            end
31            2'b01: begin
32              MEM[addr]<=din[7:0];
33              tx_valid<=0;
34
35            end
36            2'b10: begin
37              addr<=din[7:0];
38              tx_valid<=0;
39
40            end
41            2'b11: begin
42              dout<=MEM[addr];
43              tx_valid<=1;
44              sig<=1;
45            end
46            endcase
47        end
48        else if(tx_counter==0) begin
49          tx_valid<=0;
50          sig<=0;
51        end
52      end
53    end
54  endmodule
55
```

RTL code for SPI slave

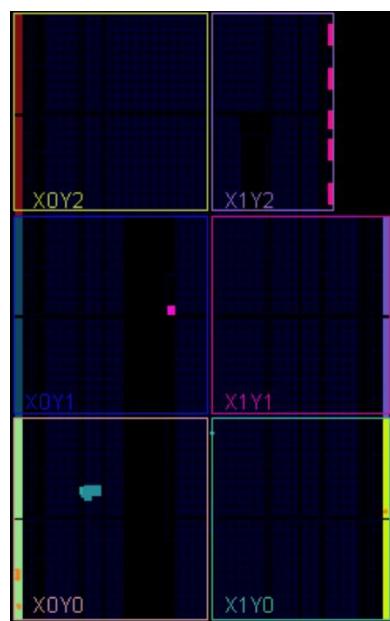
Here we want to work on the highest clock frequency, therefore, we will choose the encoding that performs on the maximum clock frequency.

Gray encoding:

Synthesis schematic



Implementation schematic



Utilities report

Tcl Console Messages Log Reports Design Runs Power DRC Methodology Timing Utilization x

Hierarchy Summary Slice Logic Slice LUTs (<1%) LUT as Logic (<1%) Slice Registers (<1%) Register as Latch (<1%) utilization_1

Name	Slice LUTs (303600)	Slice Registers (607200)	Slice (75900)	LUT as Logic (303600)	LUT Flip Flop Pairs (303600)	Block RAM Tile (1030)	Bonded IOB (600)	BUFGCTRL (32)
N SPI_WITH_SPR	43	39	16	43	26	0.5	5	1
RAM (SPR)	7	13	8	7	2	0.5	0	0
SPI (SPI_slave_op)	36	26	12	36	21	0	0	0

Timing report

Tcl Console Messages Log Reports Design Runs Power DRC Methodology Timing Utilization x

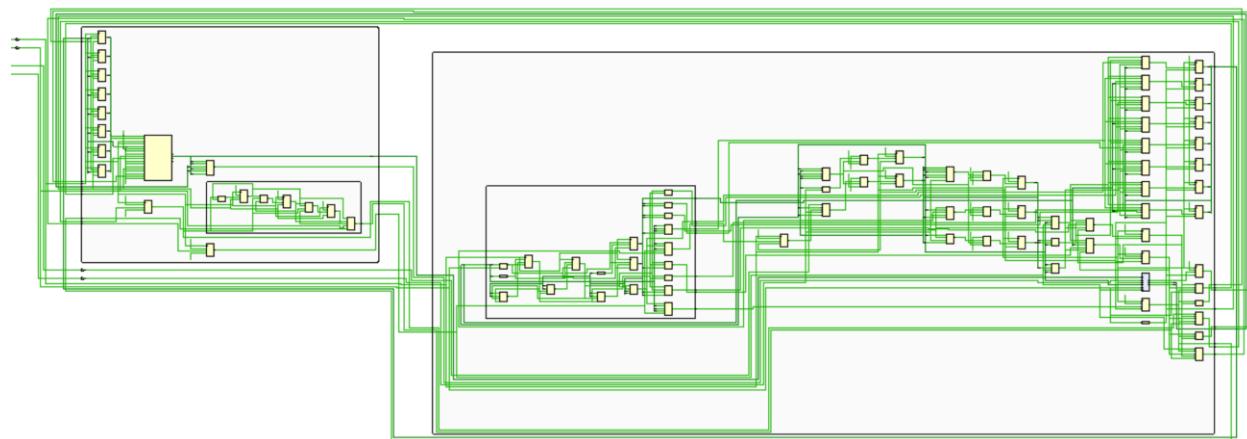
Design Timing Summary General Information Timer Settings Design Timing Summary Clock Summary (1) Check Timing (16) Intra-Clock Paths Inter-Clock Paths Other Path Groups Timing Summary - impl_1 (saved) Timing Summary - timing_1

Setup			Hold			Pulse Width		
Worst Negative Slack (WNS):	6.528 ns		Worst Hold Slack (WHS):	0.079 ns		Worst Pulse Width Slack (WPWS):	4.600 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:	77		Total Number of Endpoints:	77		Total Number of Endpoints:	39	

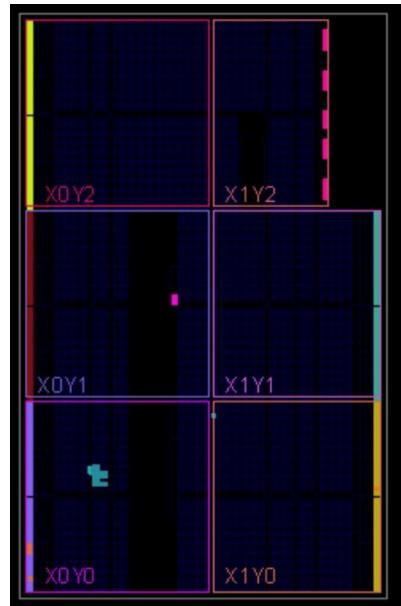
All user specified timing constraints are met.

One_hot encoding:

Synthesis schematic



Implementation schematic



Utilities report

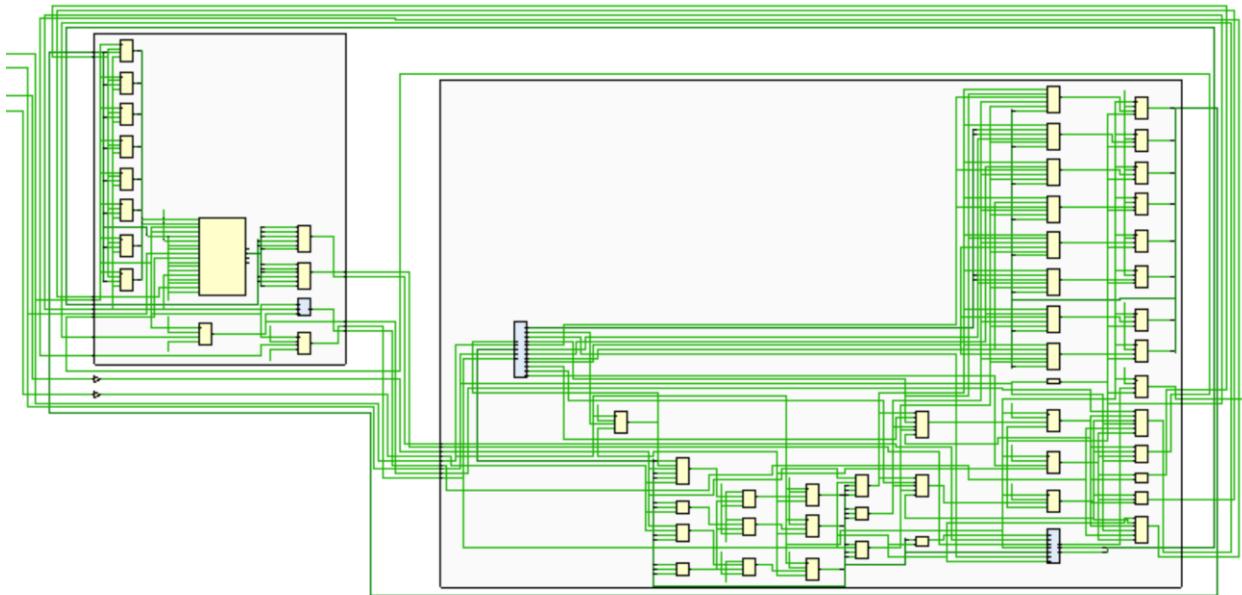
Utilization										
	Name	1	Slice LUTs (303600)	Slice Registers (607200)	Slice (75900)	LUT as Logic (303600)	LUT Flip Flop Pairs (303600)	Block RAM Tile (1030)	Bonded IOB (600)	BUFGCTRL (32)
	SPI_WITH_SPR	44	43	15	44	24	0.5	5	1	
	RAM (SPR)	5	13	5	5	2	0.5	0	0	
	SPI (SPI_slave_op)	40	30	15	40	21	0	0	0	

Timing report

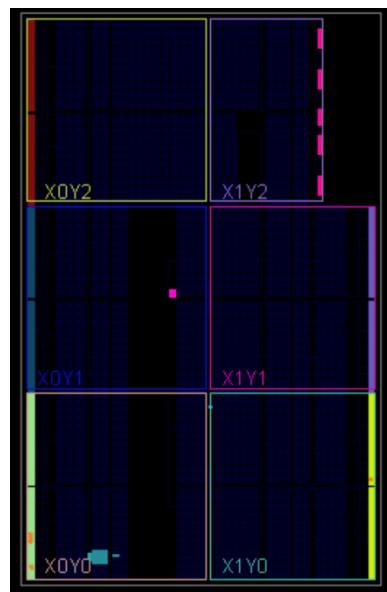
Design Timing Summary												
General Information			Setup			Hold						
Timer Settings		Worst Negative Slack (WNS): 6.554 ns		Worst Hold Slack (WHS): 0.061 ns		Worst Pulse Width Slack (WPWS): 4.600 ns						
Design Timing Summary		Total Negative Slack (TNS): 0.000 ns		Total Hold Slack (THS): 0.000 ns		Total Pulse Width Negative Slack (TPWS): 0.000 ns						
Clock Summary (1)		Number of Failing Endpoints: 0		Number of Failing Endpoints: 0		Number of Failing Endpoints: 0						
> Check Timing (34)		Total Number of Endpoints: 77		Total Number of Endpoints: 77		Total Number of Endpoints: 41						
> Intra-Clock Paths		All user specified timing constraints are met.										
Inter-Clock Paths												
Other Path Groups												

Johnson encoding:

Synthesis schematic



Implementation schematic



Utilities report

Tcl Console | Messages | Log | Reports | Design Runs | Power | DRC | Methodology | Timing | Utilization | ? - □ □

Hierarchy | Summary | Slice Logic | Slice LUTs (<1%) | LUT as Logic (<1%) | Slice Registers (<1%) | Register as Latch (<1%)

utilization_1

Name	Slice LUTs (303600)	Slice Registers (607200)	Slice (75900)	LUT as Logic (303600)	LUT Flip Flop Pairs (303600)	Block RAM Tile (1030)	Bonded IOB (600)	BUFGCTRL (32)
N SPI_WITH_SPR	45	39	13	45	24	0.5	5	1
RAM (SPR)	5	13	7	5	2	0.5	0	0
SPI (SPI_slave_op)	40	26	12	40	20	0	0	0

Timing report:

Tcl Console | Messages | Log | Reports | Design Runs | Power | DRC | Methodology | Timing | Utilization | ? - □ □

Design Timing Summary | General Information | Timer Settings | Design Timing Summary | Clock Summary (1) | Check Timing (16) | Intra-Clock Paths | Inter-Clock Paths | Other Path Groups

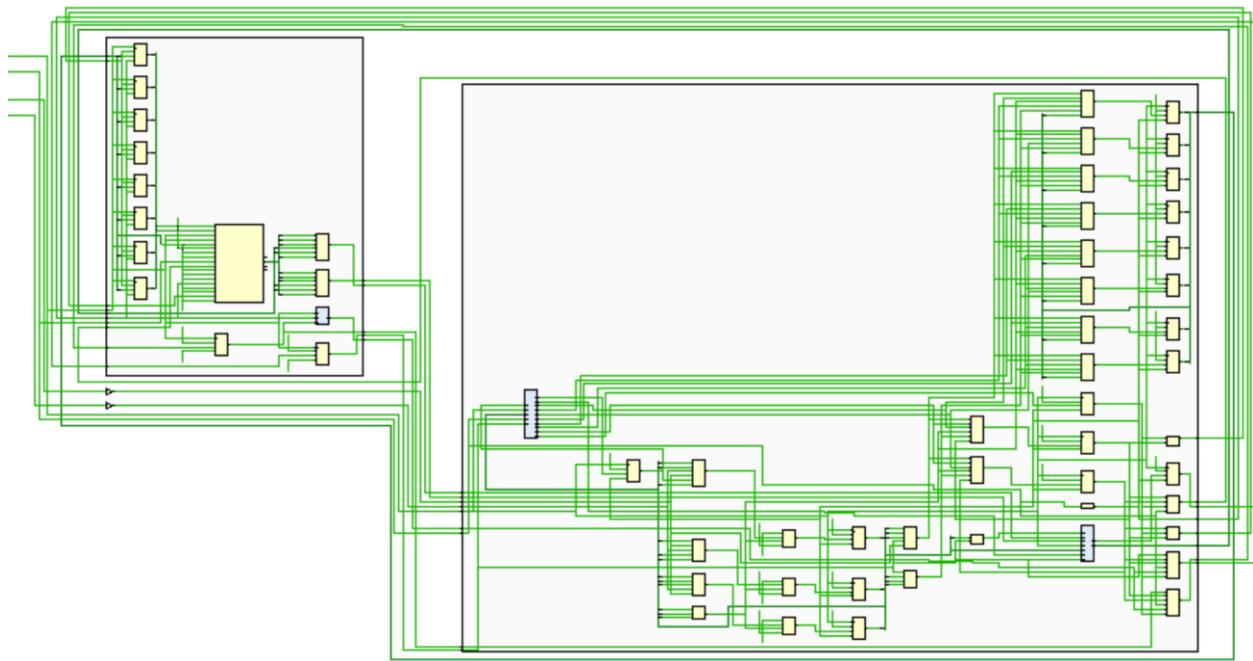
All user specified timing constraints are met.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 6.535 ns	Worst Hold Slack (WHS): 0.088 ns	Worst Pulse Width Slack (WPWS): 4.600 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: 77	Total Number of Endpoints: 77	Total Number of Endpoints: 39

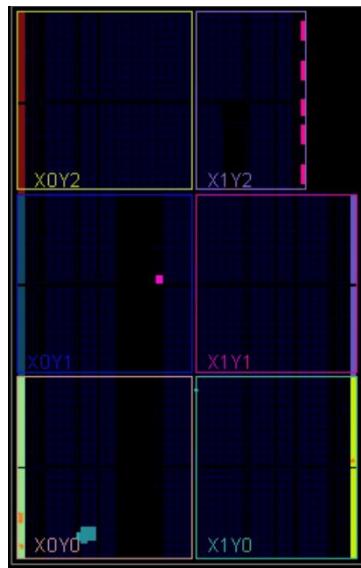
Timing Summary - impl_1 (saved) | Timing Summary - timing_1

Binary encoding:

Synthesis schematic



Implementation schematic



Utilities report

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
N SPI_WITH_SPR	42	39	14	42	26	0.5	5	1
> RAM (SPR)	5	13	6	5	2	0.5	0	0
> SPI (SPI_slave_op)	37	26	13	37	22	0	0	0

Timing report

Design Timing Summary		
General Information	Setup	Hold
Timer Settings	Worst Negative Slack (WNS): 5.797 ns	Worst Hold Slack (WHS): 0.056 ns
Design Timing Summary	Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns
Clock Summary (1)	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
> Check Timing (16)	Total Number of Endpoints: 77	Total Number of Endpoints: 77
> Intra-Clock Paths	All user specified timing constraints are met.	
Inter-Clock Paths		
Other Path Groups		

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:

```
1 module SPI_slave_op(MOSI,MISO,clk,rstn,rx_data,tx_data,rx_valid,tx_valid,ss_n);
2
3 //our state encoding;
4 parameter IDLE=3'b000;
5 parameter CHK_CMD=3'b001;
6 parameter WRITE =3'b010;
7 parameter READ_ADD=3'b011;
8 parameter READ_DATA=3'b100;
9
10
11 //SPI slave ports
12 input MOSI,clk,rstn,tx_valid,ss_n;
13 input [7:0]tx_data;
14 output reg MISO,rx_valid;
15 output reg [9:0]rx_data;
16
17 (*fsm_encoding="one_hot")
18 reg [2:0]ns; //next and current state
19
20 //internal signals
21 wire [2:0]count;
22 wire [3:0]count_10;
23 reg Read_add_data_inx;
24
25 //counters
26 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
27 down_counter #(8,3)tx_to_MISO(clk,rstn&& (cs!=CHK_CMD),(cs==READ_DATA && tx_valid),count);
28
29 //next state logic
30 always @(*) begin
31     case (cs)
32         IDLE:begin
33             if(ss_n) ns=IDLE;
34             else begin
35                 ns=CHK_CMD;
36             end
37         end
38         CHK_CMD:begin
39             if(ss_n) ns=IDLE;
40             else begin
41                 if(MOSI) begin
42                     if(!Read_add_data_inx) begin//read addr
43                         ns=READ_ADD;
44                     end
45                     else if (Read_add_data_inx) begin
46                         ns=READ_DATA;
47                     end
48                 end
49                 else begin
50                     ns=WRITE;
51                 end
52             end
53         end
54         WRITE:begin
55             if(ss_n) ns=IDLE;
56             else begin
57                 ns=WRITE;
58             end
59         end
60     end
61 
```

```

1      READ_ADD:begin
2          if(SS_n) ns=IDLE;
3          else begin
4              ns=READ_ADD;
5          end
6      end
7      READ_DATA: begin
8          if(SS_n) ns=IDLE;
9          else begin
10             ns=READ_DATA;
11         end
12     end
13   endcase
14 end
15
16 //memory logic
17 always @(posedge clk) begin
18     if (!rstn) begin
19         cs<=3'b000;
20     end
21     else begin
22         cs<=ns;
23     end
24 end
25
26 //output logic
27 always @(posedge clk) begin
28     if(!rstn) begin
29         rx_data<=8'b0;
30         rx_valid<=0;
31         MISO<=0;
32         Read_add_data_inx<=0;
33     end
34     else begin
35         case(cs)
36             WRITE: begin
37                 rx_data[count_10]<=MOSI;
38                 MISO<=0;
39                 if (count_10==4'b0) begin
40                     rx_valid<=1;
41                 end
42                 else begin
43                     rx_valid<=0;
44                 end
45             end
46             READ_ADD: begin
47                 rx_data[count_10]<=MOSI;
48                 MISO<=0;
49                 if (count_10==4'b0) begin
50                     rx_valid<=1;
51                     Read_add_data_inx<=1;
52                 end
53                 else begin
54                     rx_valid<=0;
55                 end
56             end
57             READ_DATA: begin
58                 if(!tx_valid) begin
59                     rx_data[count_10]<=MOSI;
60                     MISO<=0;
61                     if (count_10==4'b0) begin
62                         rx_valid<=1;
63                         Read_add_data_inx<=0;
64                     end
65                     else begin
66                         rx_valid<=0;
67                     end
68                 end
69                 else begin
70                     MISO<=tx_data[count]; // getting the data out of the SPI through the MISO
71                     rx_valid<=0;
72                     rx_data<='b0;
73                 end
74             end

```

```
1      CHK_CMD :begin
2          rx_valid<=0;
3          rx_data<=0;
4          MISO<=0;
5      end
6      IDLE: begin
7          rx_valid<=0; // reset all values to start communication again
8          rx_data<=0;
9          MISO<=0;
10     end
11     default :begin
12         rx_valid<=0;
13         rx_data<=0;
14         MISO<=0;
15     end
16     endcase
17 end
18 end
19 endmodule
```

RTL code for the wrapped SPI with the RAM

```
1 module SPI_WITH_SPR#(parameter MEM_DEPTH=256,parameter ADDR_SIZE=8)(clk,rstn,MISO,MOSI,SS_n);
2
3 input clk,rstn,SS_n,MOSI;
4 output MISO;
5
6 wire tx_valid,rx_valid;
7 wire [7:0] tx_data;
8 wire[9:0] rx_data;
9
10 SPI_slave_op SPI(MOSI,MISO,clk,rstn,rx_data,tx_data,rx_valid,tx_valid,SS_n);
11 SPR #(MEM_DEPTH,ADDR_SIZE)RAM(rx_data,clk,rstn,rx_valid,tx_valid,tx_data);
12
13 endmodule
```

Test bench code

```
● ● ●
1 module SPI_WITH_SPR_tb();
2
3 reg clk,rstn,SS_n,MOSI;
4 wire MISO;
5
6 SPI_WITH_SPR #(256,8)DUT(clk,rstn,MISO,MOSI,SS_n);
7
8 initial begin
9   clk=0;
10  forever begin
11    #10 clk=~clk;
12  end
13 end
14
15 initial begin
16 //reset
17   rstn=0;
18   MOSI=0;
19   SS_n=1;
20   repeat(20) begin
21     @(negedge clk);
22     if(MOSI!=0) begin
23       $display("Error in reset");
24       $stop;
25     end
26   end
27 //IDLE test
28   rstn=1;
29   repeat(10) begin
30     @(negedge clk);
31     if(MOSI!=0) begin
32       $display("Error in IDLE");
33       $stop;
34     end
35   end
36 //write add test
37   SS_n=0;
38   @(negedge clk); //now we are at the CHk_CMD
39   MOSI=0;
40   @(negedge clk); //now we are at the WRITE
41   MOSI=0;
42   @(negedge clk);
43   MOSI=0;
44   @(negedge clk); //now we sent 2 zero for the ram to write address
45   repeat(8) begin //random address
46     MOSI=$random;
47     @(negedge clk);
48   end
49   SS_n=1; //end communication ,now the ram should have taken this value for the address
50   @(negedge clk);
51   SS_n=0; //start communication
52   @(negedge clk); //at CHk_CMD
53   MOSI=0; //go to write state
54   @(negedge clk); //at write
55   MOSI=0;
56   @(negedge clk);
57   MOSI=1; //write data
58   @(negedge clk);
59
```

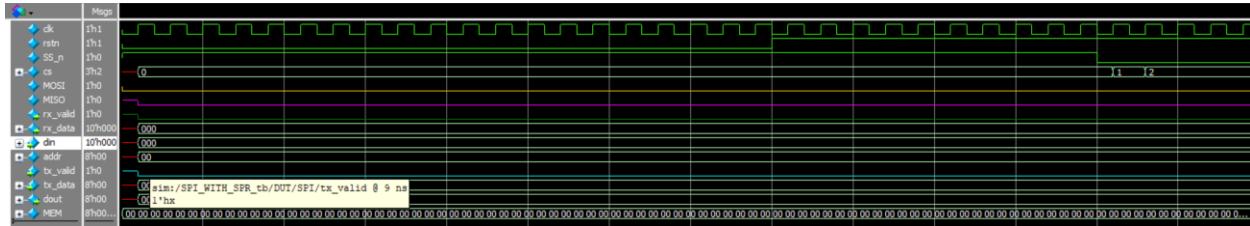
```
1 repeat(8) begin //write this random data
2     MOSI=$random;
3     @(negedge clk);
4 end
5 SS_n=1;//end communication
6 @(negedge clk);
7 SS_n=0;//start communication
8 @(negedge clk); //at check command
9 MOSI=1;
10 @(negedge clk); //at Read command
11 MOSI=1;
12 @(negedge clk);
13 MOSI=0;
14 @(negedge clk); //we sent 10 for the ram to read add of the following
15 MOSI=0;
16 @(negedge clk);
17 MOSI=1;
18 @(negedge clk);
19 MOSI=1;
20 @(negedge clk);
21 MOSI=1;
22 @(negedge clk);
23 MOSI=1;
24 @(negedge clk);
25 MOSI=1;
26 @(negedge clk);
27 MOSI=1;
28 @(negedge clk);
29 MOSI=0;
30 @(negedge clk); // the address is 01111110 which is 7E in hexa which was the previous random add
31 SS_n=1;//end communication
32 @(negedge clk);
33 SS_n=0;
34 @(negedge clk);
35 MOSI=1;//read
36 @(negedge clk);
37 MOSI=1;
38 @(negedge clk);
39 MOSI=1;
40 @(negedge clk); //read data at address 7E which is the one we wrote in the write state
41 repeat(8) begin
42     MOSTI=$random;
43     @(negedge clk); //dummy bits
44 end
45 @(negedge clk);
46 repeat(8) @(negedge clk); //data out
47 SS_n=1;//end communication
48 @(negedge clk);
49
```

```
1 //////////////repeat the same process again to ensure every thing is right every time///////////
2
3     SS_n=0;
4     @(negedge clk);
5     MOSI=0;
6     @(negedge clk);
7     MOSI=0;
8     @(negedge clk);
9     MOSI=0;
10    @(negedge clk);
11    repeat(8) begin
12        MOSI=$random;
13        @(negedge clk);
14    end
15    SS_n=1;
16    @(negedge clk);
17    SS_n=0;
18    @(negedge clk);
19    MOSI=0;
20    @(negedge clk);
21    MOSI=0;
22    @(negedge clk);
23    MOSI=1;
24    @(negedge clk);
25    repeat(8) begin
26        MOSI=$random;
27        @(negedge clk);
28    end
29    SS_n=1;
30    @(negedge clk);
31    SS_n=0;
32    @(negedge clk);
33    MOSI=1;
34    @(negedge clk);
35    MOSI=1;
36    @(negedge clk);
37    MOSI=0;
38    @(negedge clk);
39        MOSI=0;
40        @(negedge clk);
41        MOSI=1;
42        @(negedge clk);
43        MOSI=1;
44        @(negedge clk);
45        MOSI=1;
46        @(negedge clk);
47        MOSI=1;
48        @(negedge clk);
49        MOSI=1;
50        @(negedge clk);
51        MOSI=1;
52        @(negedge clk);
53        MOSI=0;
54        @(negedge clk);
```

```
1      SS_n=1;
2      @(negedge clk);
3      SS_n=0;
4      @(negedge clk);
5      MOSI=1;
6      @(negedge clk);
7      MOSI=1;
8      @(negedge clk);
9      MOSI=1;
10     @(negedge clk);
11     repeat(8) begin
12         MOSI=$random;
13         @(negedge clk);
14     end
15     @(negedge clk);
16     repeat(8) @(negedge clk);
17     SS_n=1;
18     @(negedge clk);
19 $stop;
20 end
21 endmodule
```

Questasim Waveform

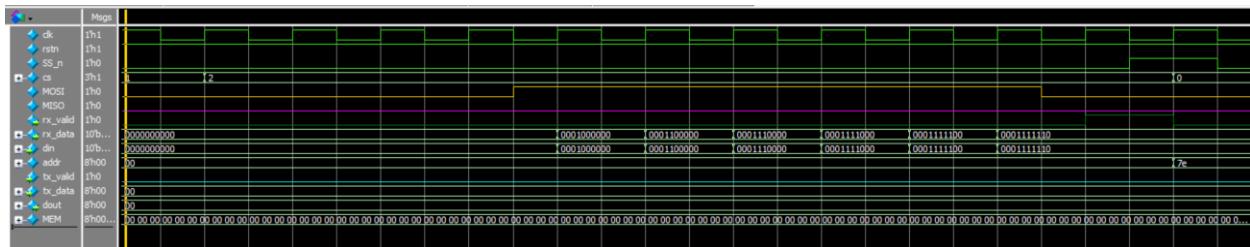
Reset and SS_n check



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



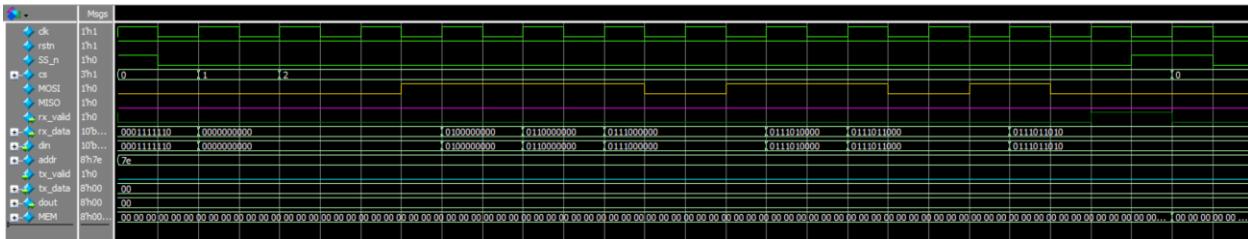
The SS_n became **0**, the communication starts.

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



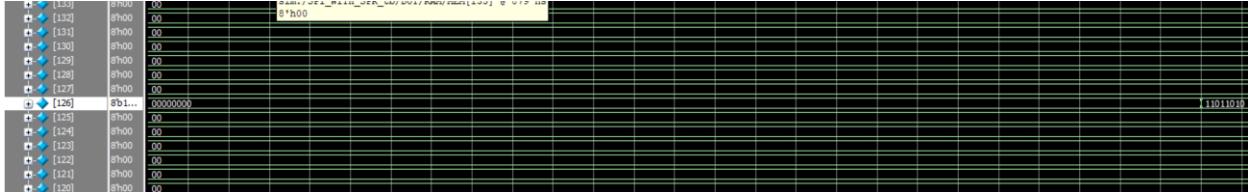
The SS_n became **0**, the communication starts.

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 **126_D or 7E_H**.

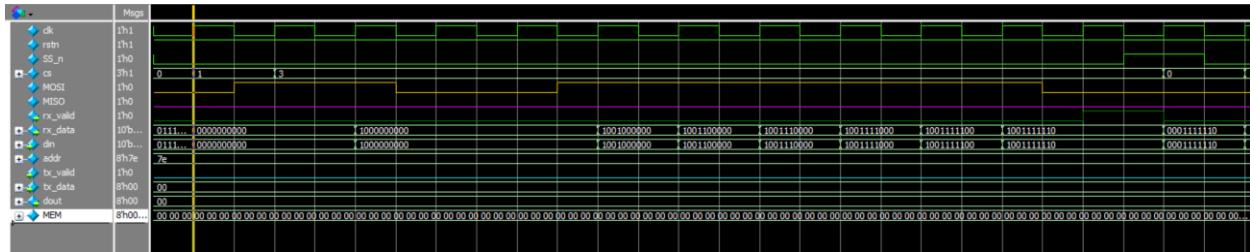
Then the SS_n became **1** to end the communication.

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.

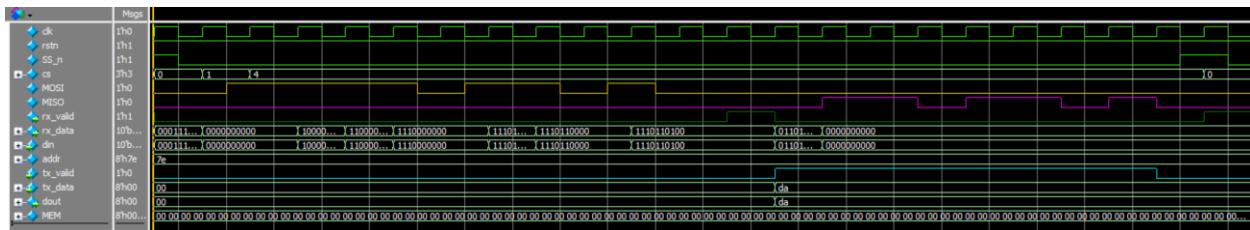
The cs goes from IDLE to CHK_CMD then to READ according to the MOSI.

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:

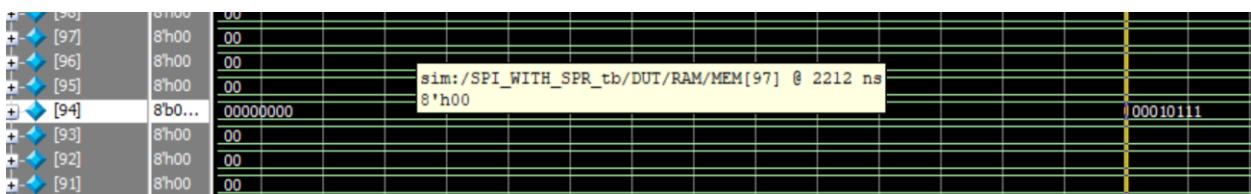


Second cycle:



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 **5E_H** or **94_D** and the write data is **00010111**

MEM [94] check:

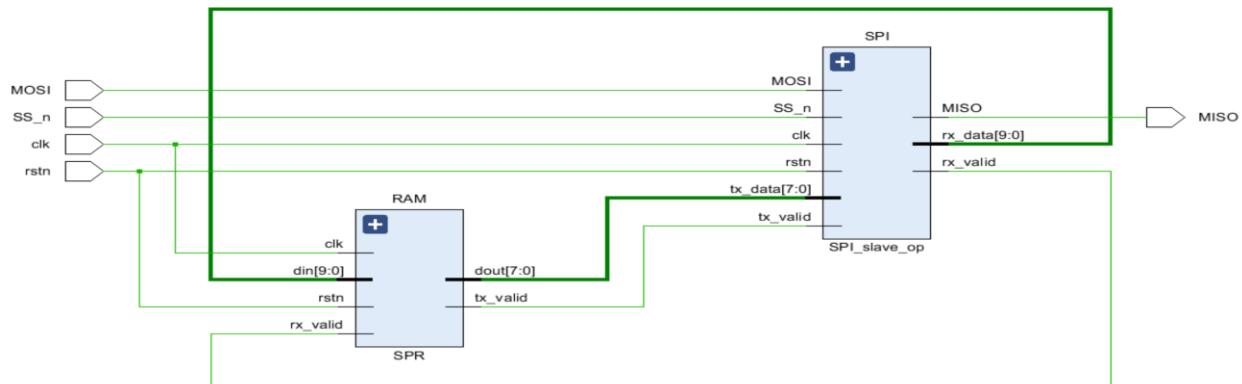


Lint check

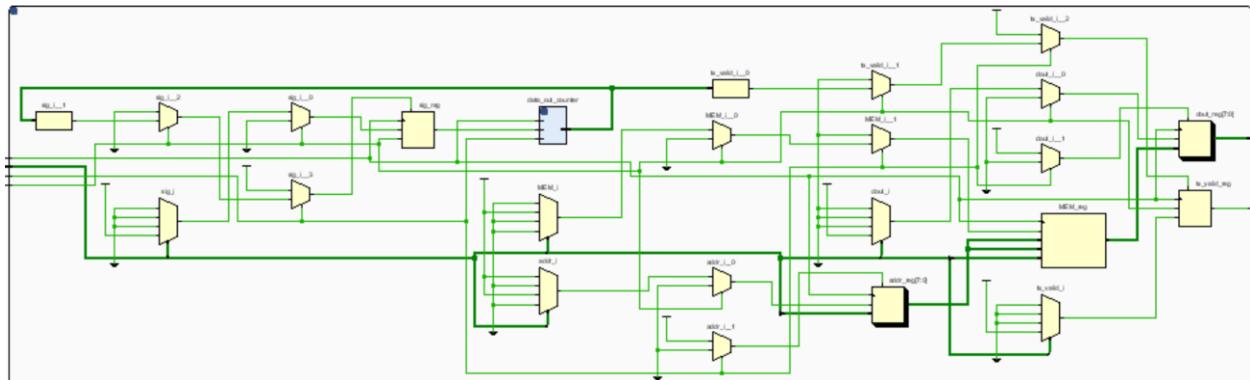
Lint Checks			
* Filter: Type here	Waived	Fixed	Pending
parameter...	Same parameter name is used in more than one module. Parameter ADDR_SIZE, Total count 2, First module: Module SPI_WITH_SPR, File D/Digital_Diploma/Project_2/SPI_SLAVE_optimized.v, Line 26.	Parameter MEM_DEPTH, Total count 2, First module: Module SPI_SLAVE_OP, File D/Digital_Diploma/Project_2/SPI_SLAVE_optimized.v, Line 26.	Bug
port_conn...	Instance port connection is expression. Expression cs!=0, Module SPI_SLAVE_OP, File D/Digital_Diploma/Project_2/SPI_SLAVE_optimized.v, Line 26.	Instance port connection is expression. Expression rstin&&cs!=1, Module SPI_SLAVE_OP, File D/Digital_Diploma/Project_2/SPI_SLAVE_optimized.v, Line 26.	Verified
multi_ports...	Multiple ports are declared in one line. Module SPI_SLAVE_OP, File D/Digital_Diploma/Project_2/SPI_SLAVE_optimized.v, Line 3.	Multiple ports are declared in one line. Module SPI_SLAVE_OP, File D/Digital_Diploma/Project_2/SPI_SLAVE_optimized.v, Line 27.	
multi_ports...	Multiple ports are declared in one line. Module SPI_SLAVE_OP, File D/Digital_Diploma/Project_2/SPI_SLAVE_optimized.v, Line 12.	Multiple ports are declared in one line. Module SPI_SLAVE_OP, File D/Digital_Diploma/Project_2/SPI_SLAVE_optimized.v, Line 4.	
multi_ports...	Multiple ports are declared in one line. Module down_counter, File D/Digital_Diploma/Project_2/down_counter.v, Line 3.		

RTL analysis using vivado

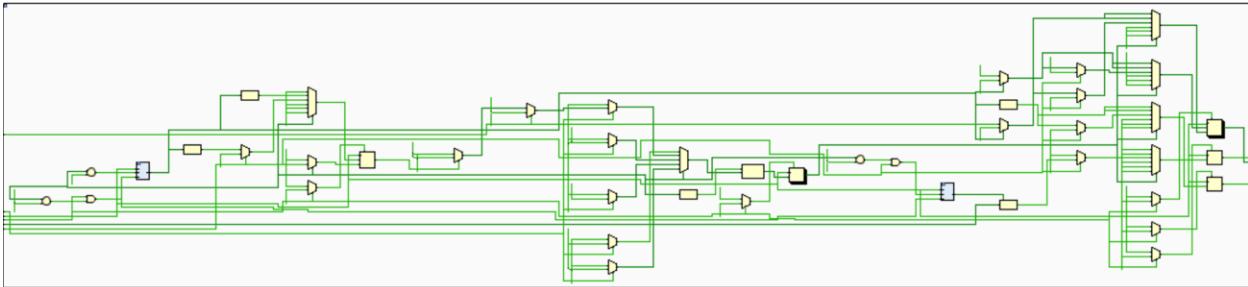
RTL schematic



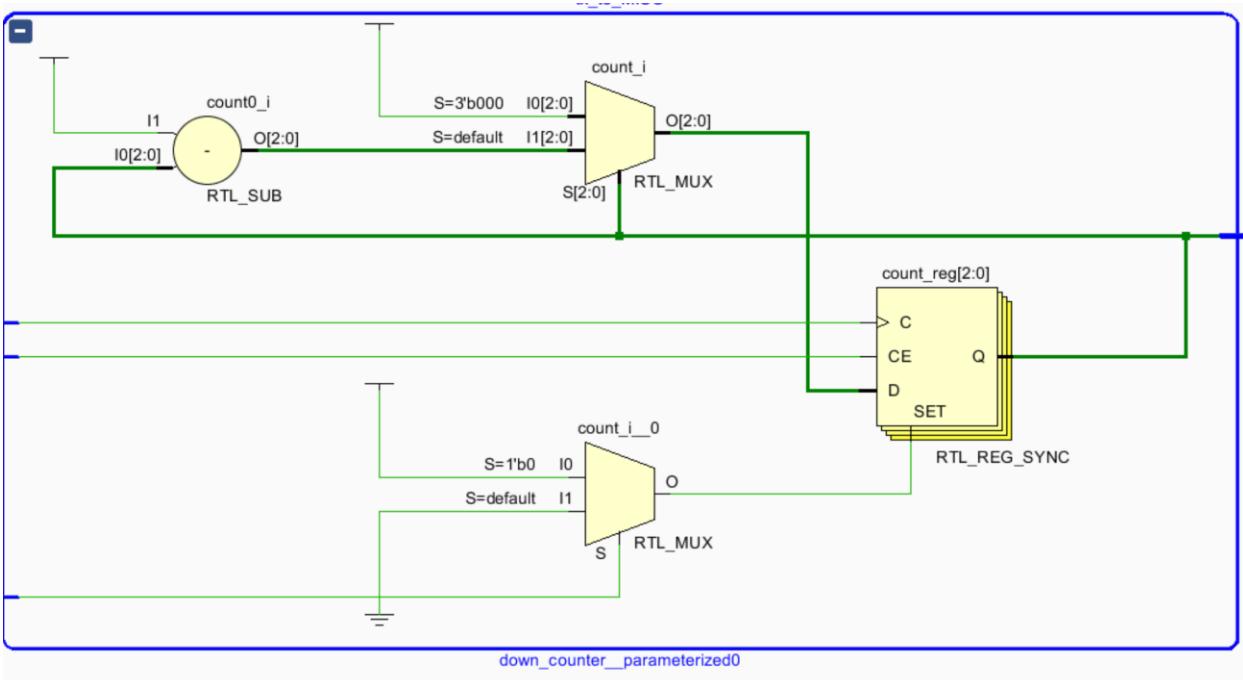
For a deeper look inside the RAM:



For a deeper look inside the SPI slave:

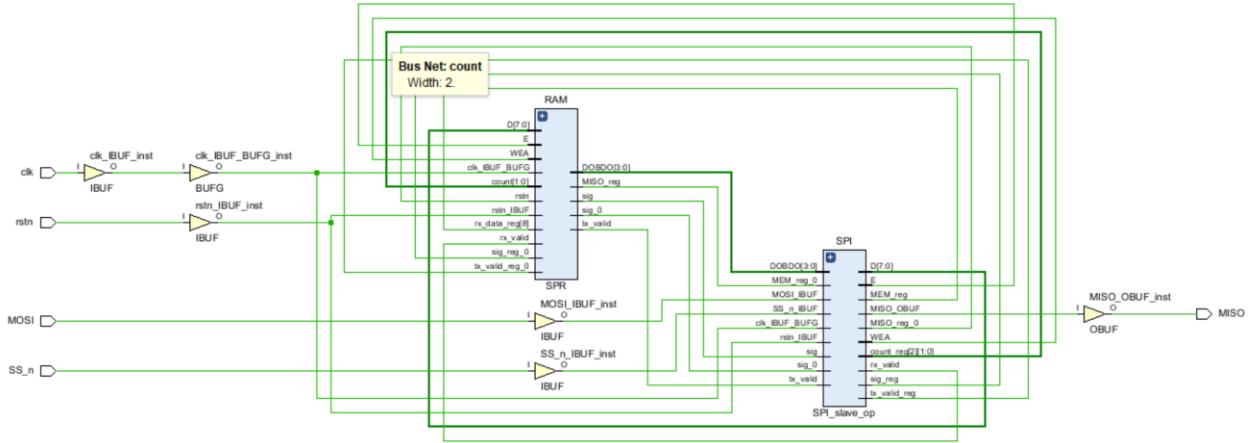


For a deeper look inside the counter:

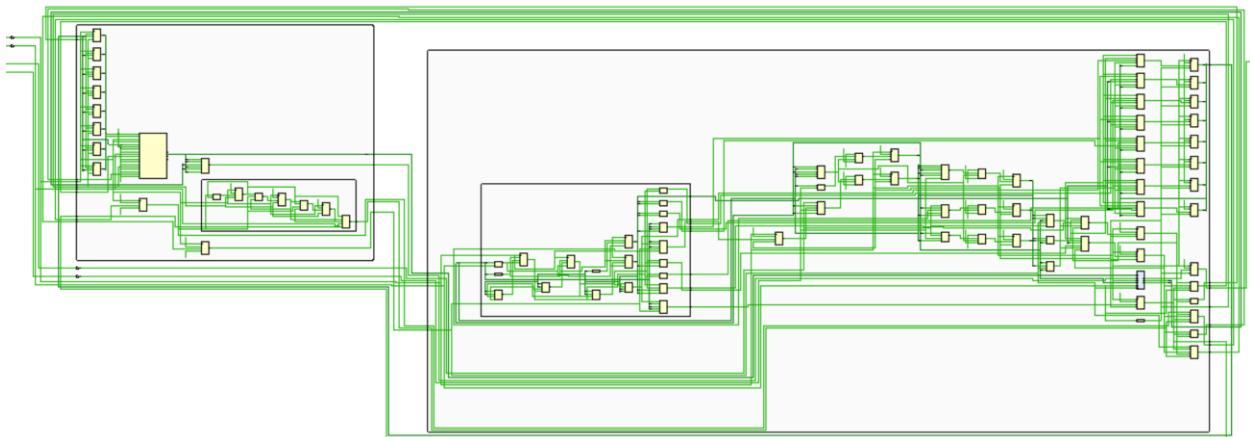


Synthesis analysis using vivado

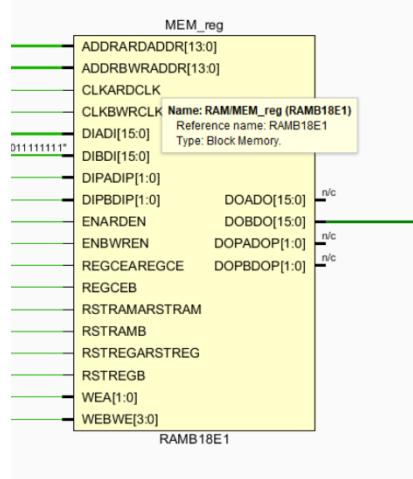
Technology Schematic



With more details inside:



Zooming at the big yellow block:



Utilities report

Utilization						
	Name	Slice LUTs (20800)	Slice Registers (41600)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
Hierarchy						
Summary						
Slice Logic						
Slice LUTs (<1%)	N SPI_WITH_SPR	44	43	0.5	5	1
LUT as Logic (<1%)	RAM (SPR)	4	13	0.5	0	0
Slice Registers (<1%)	SPI (SPI_slave_op)	40	30	0	0	0
Register as Latch (<1%)						

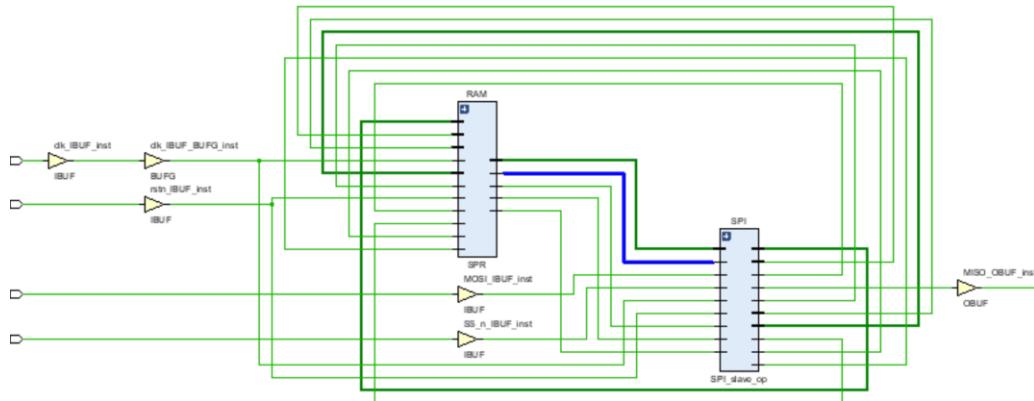
Timing report

Timing							
Design Timing Summary							
General Information		Setup		Hold			
Timer Settings		Worst Negative Slack (WNS):	5.445 ns	Worst Hold Slack (WHS):	0.139 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Design Timing Summary		Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWNS):	0.000 ns
Clock Summary (1)		Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Check Timing (34)		Total Number of Endpoints:	76	Total Number of Endpoints:	76	Total Number of Endpoints:	41
Intra-Clock Paths		All user specified timing constraints are met.					
Inter-Clock Paths							
Other Path Groups							

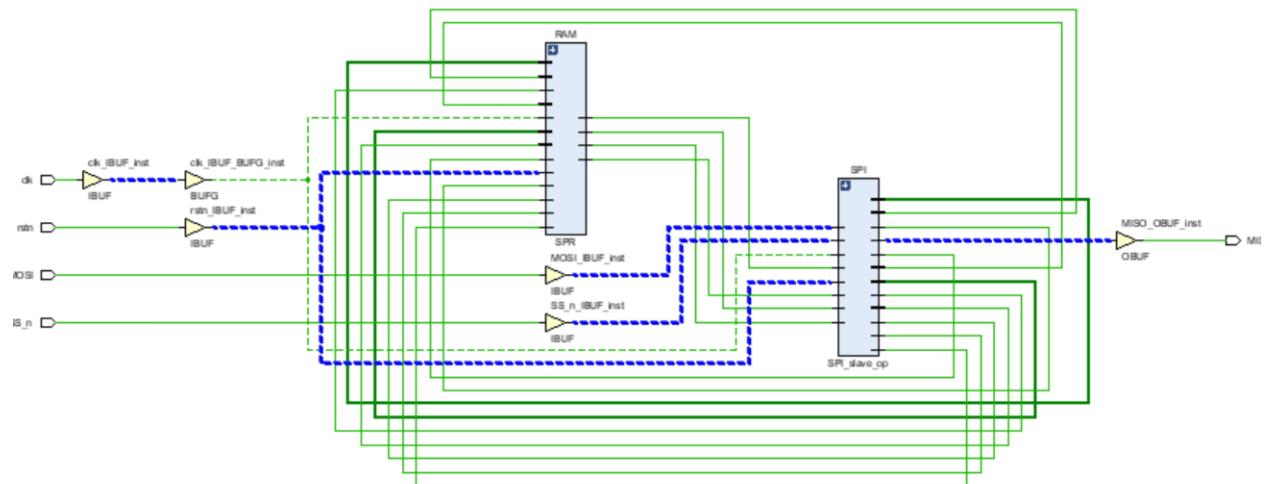
Messages

General Messages (6 infos)	
Info	[Netlist 29-17] Analyzing 5 Unisim elements for replacement
Info	[Netlist 29-28] Unisim Transformation completed in 0 CPU seconds
Info	[Project 1-479] Netlist was created with Vivado 2018.2
Info	[Project 1-570] Preparing netlist for logic optimization
Info	[Opt 31-138] Pushed 0 inverter(s) to 0 load pin(s).
Info	[Project 1-111] Unisim Transformation Summary: No Unisim elements were transformed.

Worst path:

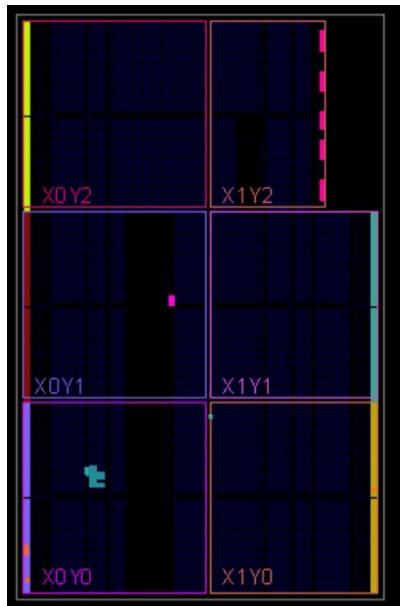


Debug setting



Implementation using vivado

Schematic



Utilities report

The screenshot shows the Vivado Utilities report window. The left sidebar displays a hierarchical tree with categories like Hierarchy, Summary, and Slice Logic. The main area is a table showing resource utilization for three components: SPI_WITH_SPR, RAM (SPR), and SPI (SPI_slave_op).

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
N SPI_WITH_SPR	45	43	16	45	23	0.5	5	1
RAM (SPR)	5	13	4	5	2	0.5	0	0
SPI (SPI_slave_op)	40	30	15	40	20	0	0	0

Timing report

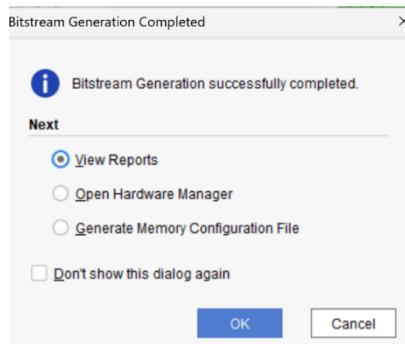
The screenshot shows the Vivado interface with the 'Timing' tab selected. The main window displays the 'Design Timing Summary' with three columns: Setup, Hold, and Pulse Width. Key values include Worst Negative Slack (WNS) at 5.591 ns, Worst Hold Slack (WHS) at 0.053 ns, and Worst Pulse Width Slack (WPWS) at 4.500 ns. A note at the bottom states 'All user specified timing constraints are met.'

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 5.591 ns	Worst Hold Slack (WHS): 0.053 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: 77	Total Number of Endpoints: 77	Total Number of Endpoints: 41

Messages

The screenshot shows the Vivado 'Messages' window with the 'Messages' tab selected. It displays a list of build logs, mostly informational messages (info) and warnings (warning). Some key entries include: '[Netlist 29-28] Unisim Transformation completed in 0 CPU seconds', '[Project 1-479] Netlist was created with Vivado 2018.2', and '[Project 1-570] Preparing netlist for logic optimization'. There are also several warning messages related to timing and binary archive restoration.

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

```
1  create_debug_core u_ila_0 ila
2  set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
3  set_property ALL_PROBE_SAME_MU_CNT 1 [get_debug_cores u_ila_0]
4  set_property C_ADV_TRIGGER false [get_debug_cores u_ila_0]
5  set_property C_DATA_DEPTH 1024 [get_debug_cores u_ila_0]
6  set_property C_EN_STRG_QUAL false [get_debug_cores u_ila_0]
7  set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ila_0]
8  set_property C_TRIGIN_EN false [get_debug_cores u_ila_0]
9  set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
10 set_property port_width 1 [get_debug_ports u_ila_0/clk]
11 connect_debug_port u_ila_0/clk [get_nets [list clk_IBUF_BUFG]]
12 set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe0]
13 set_property port_width 1 [get_debug_ports u_ila_0/probe0]
14 connect_debug_port u_ila_0/probe0 [get_nets [list clk_IBUF]]
15 create_debug_port u_ila_0 probe
16 set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe1]
17 set_property port_width 1 [get_debug_ports u_ila_0/probe1]
18 connect_debug_port u_ila_0/probe1 [get_nets [list MISO_OBUF]]
19 create_debug_port u_ila_0 probe
20 set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe2]
21 set_property port_width 1 [get_debug_ports u_ila_0/probe2]
22 connect_debug_port u_ila_0/probe2 [get_nets [list MOSI_IBUF]]
23 create_debug_port u_ila_0 probe
24 set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe3]
25 set_property port_width 1 [get_debug_ports u_ila_0/probe3]
26 connect_debug_port u_ila_0/probe3 [get_nets [list rstn_IBUF]]
27 create_debug_port u_ila_0 probe
28 set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe4]
29 set_property port_width 1 [get_debug_ports u_ila_0/probe4]
30 connect_debug_port u_ila_0/probe4 [get_nets [list SS_n_IBUF]]
31 set_property C_CLK_INPUT_FREQ_HZ 300000000 [get_debug_cores dbg_hub]
32 set_property C_ENABLE_CLK_DIVIDER false [get_debug_cores dbg_hub]
33 set_property C_USER_SCAN_CHAIN 1 [get_debug_cores dbg_hub]
34 connect_debug_port dbg_hub/clk [get_nets clk_IBUF_BUFG]
35
36
```