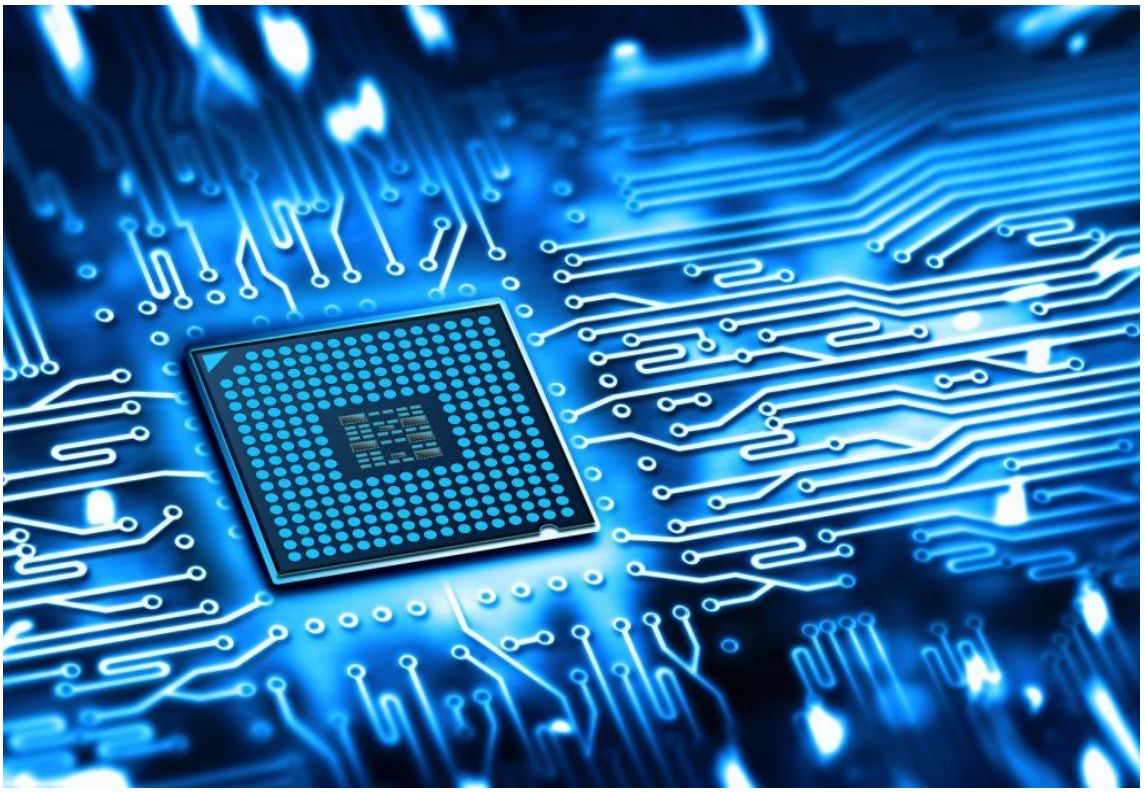
SPI Slave with single port RAM

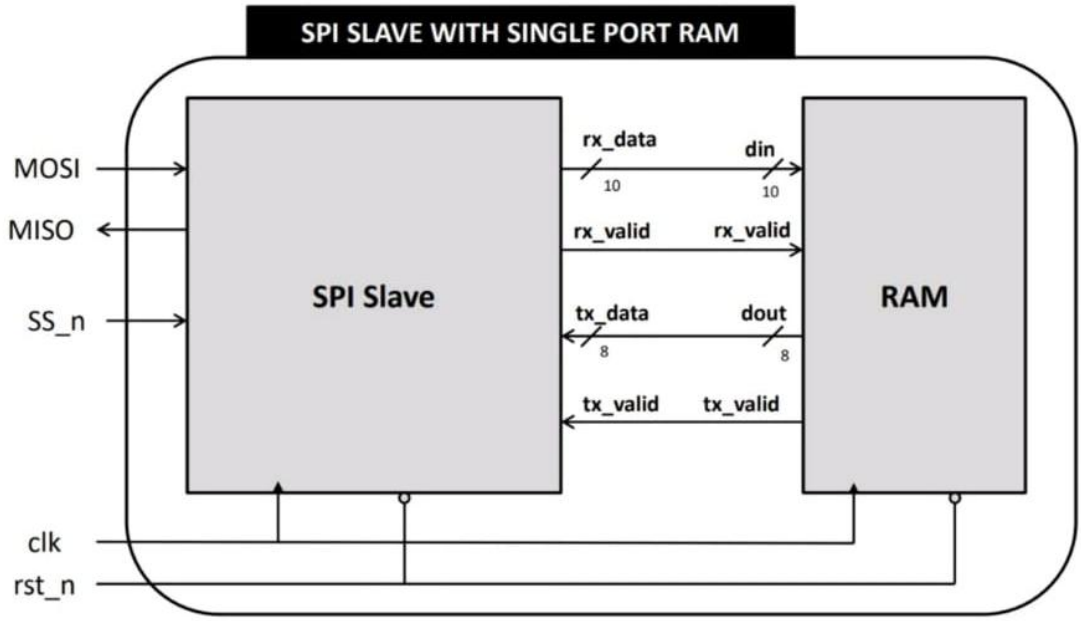


Prepared by
Ali Abdel El Aleem Ali

Abstract:

The Serial Peripheral Interface (SPI) is a high-speed, full-duplex, synchronous communication protocol widely used for data exchange between microcontrollers and peripheral devices. This project presents the design and implementation of an SPI communication system consisting of both Master and Slave modules. The SPI Master is responsible for generating the clock signal (SCLK) and controlling the data transfer sequence, while the SPI Slave responds to the Master's commands and exchanges data through the MOSI (Master Out Slave In) and MISO (Master In Slave Out) lines. The proposed design supports the standard SPI modes of operation, ensuring flexibility for various applications. Hardware Description Language (HDL) is used to model and verify the system functionality.

Overview:



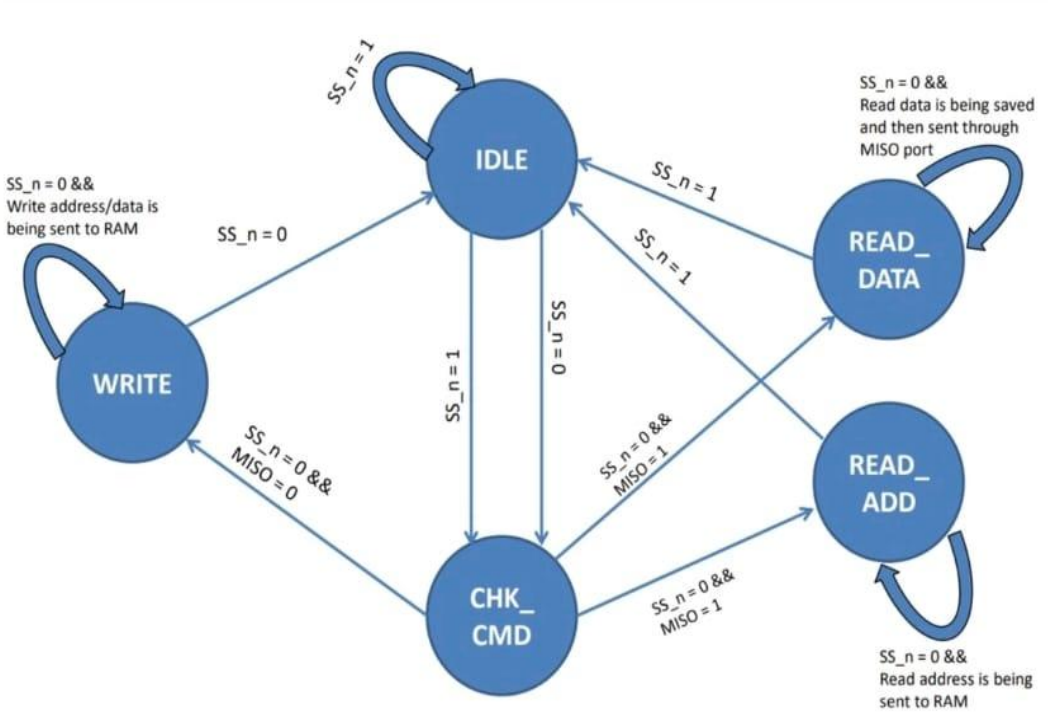
Serial Peripheral Interface (SPI) is a widely used synchronous communication Protocol that enables a master device to exchange data with one or more slave devices.
It operates over four main lines:
SCLK (clock), MOSI (master output salve input), MISO (master input salve output), and SS_n (slave select).
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, tx_valid.

States:

The SPI slave uses five states to marriage communication and



coordinate access to the RAM:

- **IDLE** : the default state, no active communication is taking place while I this state.

- **CHK_CMD**: entered once SS_n is asserted and communication begins. In this state the incoming MOSI bit is examined to determine the whether the operation is a read or write, and the FSM transitions accordingly.

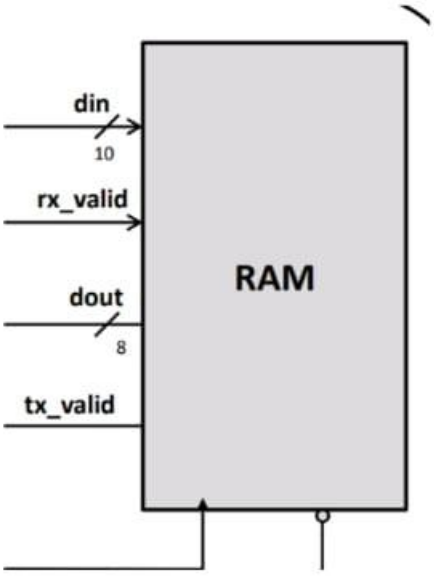
- **WRITE:** this state handles the write operation to RAM. Although the write process consists of two logical steps _ receiving the write address followed by the writing data_there is no need for dedicated RAM states like WRITE_ADDR and WRITE_DATA. After receiving the address, the FSM returns briefly to IDLE state and the re-turns CHK_CMD to interrupt the next byte, which is the write data. Because the address

and data are sent by the master in quick succession with no delay, the SPI slave can write the data to RAM immediately after receiving both, without needing intermediate RAM specific states.

- **READ_ADDR:** in this state the address of the location to be received from the master, after the address is captured, the FSM transitions to the next state. A delay is required here to allow 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.

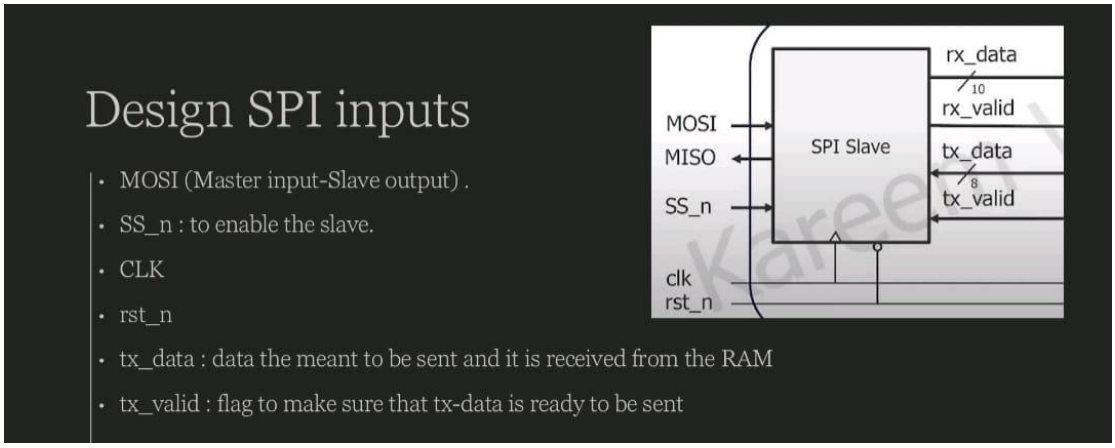
Single port Async RAM



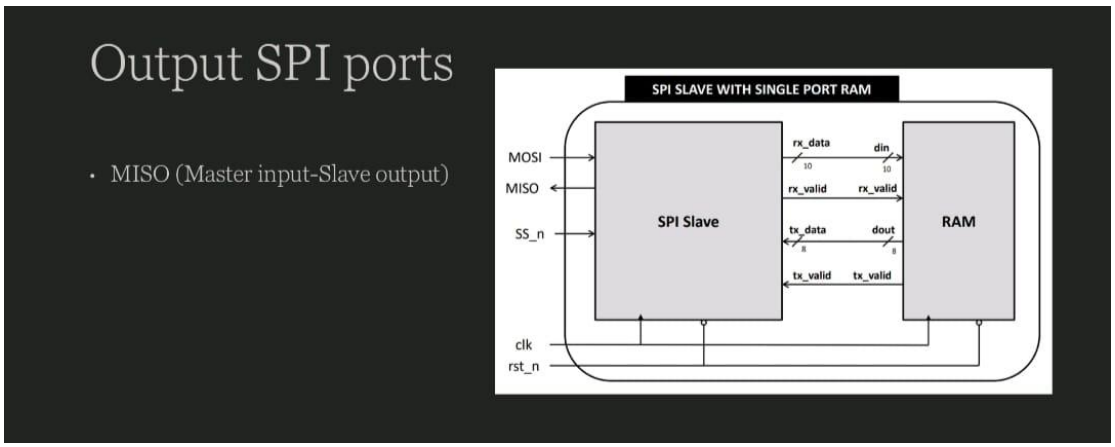
According to the most significant two bits of the din the RAM decides whether read_addr/read_data or write_addr/write_data. The rx_valid signal tell the RAM to accept the din data for the address while tx_valid tells the SPI to accept the dout data to the MISO.

Design Procedure:

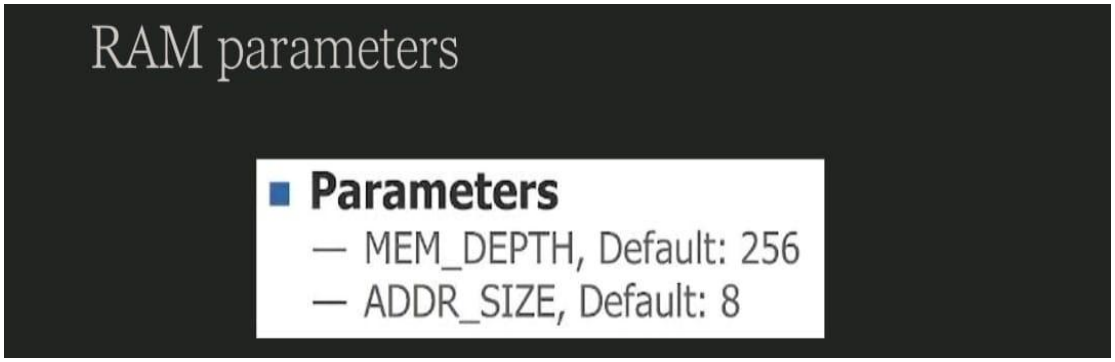
1. SPI slave inputs



2. SPI slave outputs



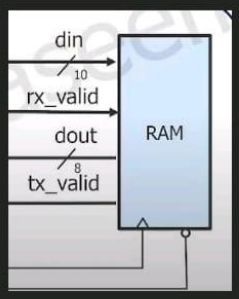
3. RAM parameters



4. RAM inputs

Design RAM inputs

- din : 10-bit bus that carry the data that the RAM will use
- rx_valid : flag used to make sure that fin is ready to be used
- CLK
- rst_n



5. RAM outputs

Output RAM ports

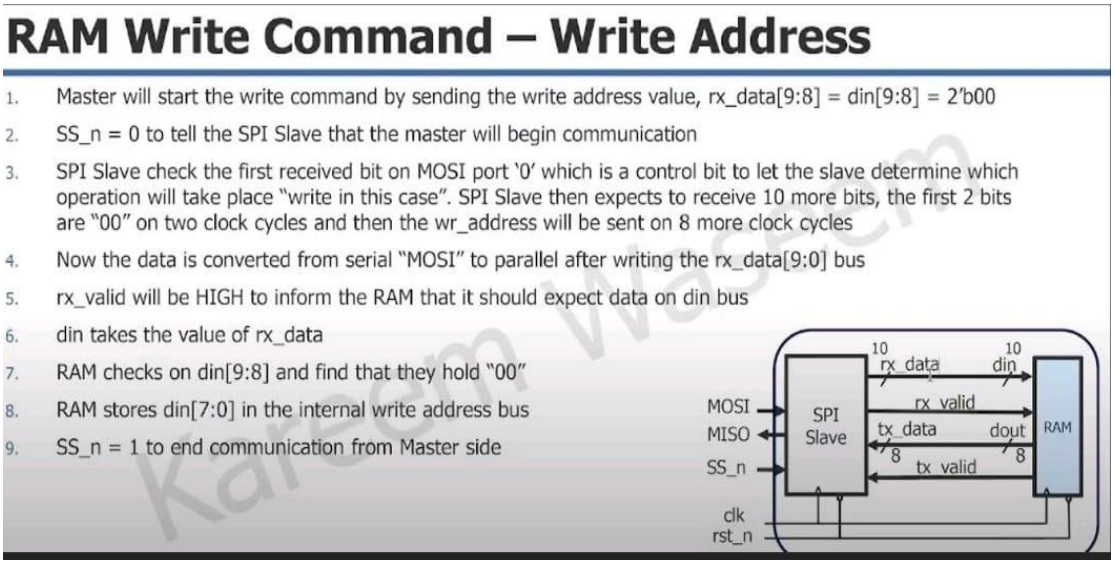
- dout : data sent to the SPI slave
- tx_valid : flag used to tell the SPI slave that dout is ready to be received

6. Din most 2bit

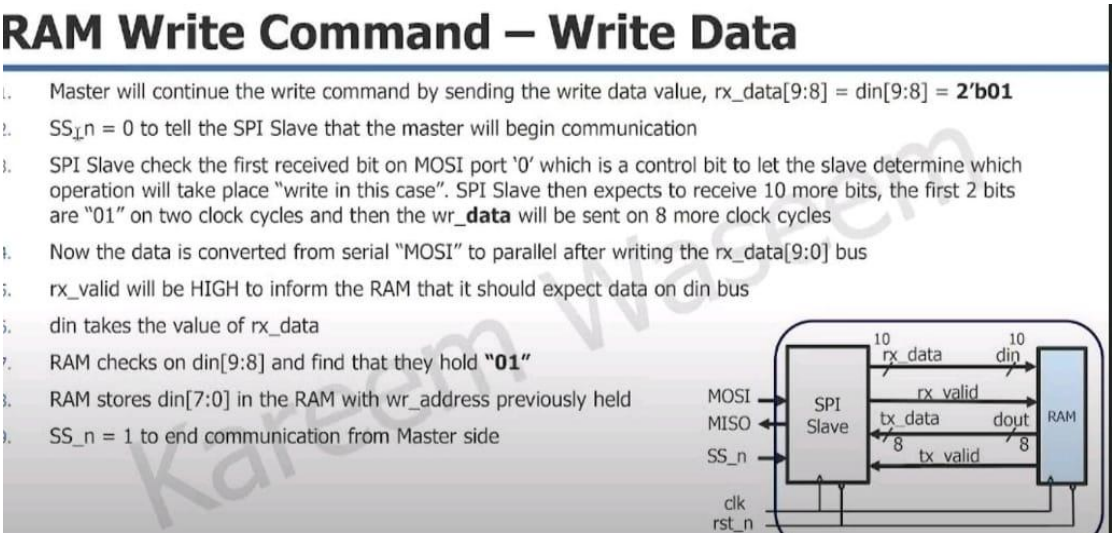
din most two significant bits descriptions

port	Din[9:8]	command	description
din	00	write	Hold din[7:0] as write address
	01		Write din[7:0] inside the write address that got before
	10	read	Hold din[7:0] as read address
	11		Get the data inside the read address that got before and put it in dout and make tx_valid high *note ignore din[7:0]

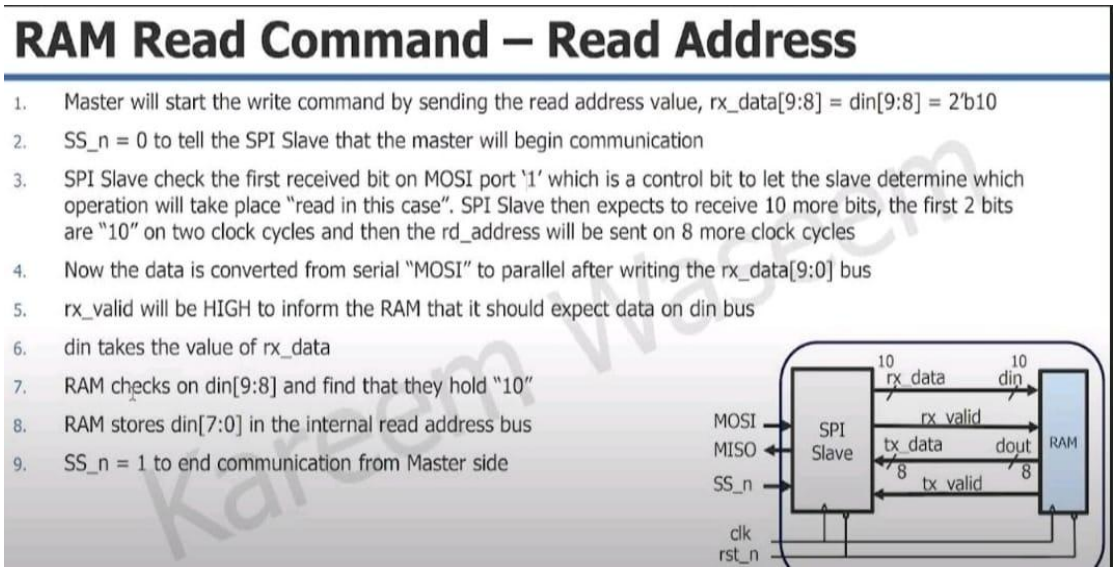
7. Write address



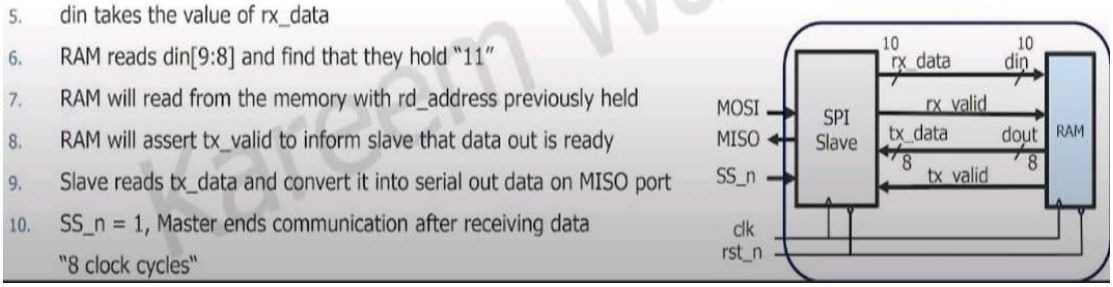
8. Write data



9. Read address



10. Read data



RTL for RAM:

```
AM.v
module Spi_RAM

( din , rx_valid, dout, tx_valid, clk, reset);
parameter MEM_depth = 256;
parameter ADDR_size = 8;

//declaring inputs

input [9:0] din;  // carry the data that will be used by RAM
input rx_valid;   // flag used to make sure that din is ready to be used
input clk;
input reset;      // SYC reset


//declaring outputs

output reg [7:0] dout; // data sent to SPI salve
output reg tx_valid;   // Flag for indicate to slave that data stores to ram

//decalring memory
reg [ADDR_size-1:0] addr;   // internal address for RAM
reg [7:0] MEM [0:MEM_depth-1]; // memory declaration

//logic
always@(posedge clk)
      begin
          if(~reset)
             begin
                    dout <= 0;
                    tx_valid <= 0;
             end
          else
             begin
                    if(rx_valid)
                       begin
                             case(din[9:8])
                                 begin
                                   2'b00:  //write address
                                      begin
                                         addr <= din[7:0]
                                         tx_vlid <= o;
                                      end
                                   2'b01:   // write data
                                      begin
                                         MEM[addr] <= din[7:0];
                                         tx_valid <= o0
                                      end

                                   2'b10:   // read address
                                      begin
                                          addr <= din[7:0];
                                          tx_valid <= 0;


                                      end

                                   2'b11:   // read data
                                      begin
                                           dout <= MEM[addr];
                                           tx_valid <= 1;
                                      end
                                 end
                             endcase
                       end
             end
      end
endmodule
```

RTL for SPI slave

slave.v

```
module spi_slave ( MOSI, MISO, SS_n, clk, reset, rx_data, rx_valid, tx_data, tx_valid);

//declaring inputs

input clk;
input reset;
input SS_n; //enable for slave
input tx_valid;
input [7:0] tx_data;
input MOSI;
/*****************************/
//declaring output
output reg rx_valid;
output reg [9:0] rx_data;
output reg MISO;
/*****************************/
//FSM
parameter IDLE = 0, CHK_CMD = 1, WRITE = 2, READ_DATA = 3, READ_ADD = 4; //states

reg [2:0] state,NEXT_state; // Logic for FSM
reg  ADD_or_Data;
reg [4:0] counter;
reg updown;


// state register

always@(posedge clk)
     begin
        if(~reset)
          begin
            state <= IDLE;
            NEXT_state <= IDLE;
          end
        else
          begin
            state <= NEXT_state;
          end
     end
/*************************************/
//next state logic

always@(*)
     begin
        case(state)
          IDLE:
           begin
              if(SS_n ==1)
                NEXT_state <= IDLE;
              else
                NEXT_state <= CHK_CMD;
           end
          CHK_CMD:
           begin
              if(SS_n == 0)
                NEXT_state <= IDLE;
              else if(SS_n == 0 && MOSI == 0)
                NEXT_state <= WRITE;
              else if(SS_n == 0 && MOSI == 1 && ADD_or_Data == 0 )
                NEXT_state <= READ_ADD;
              else if(SS_n == 0 && MOSI == 1 && ADD_or_Data == 1 )
                NEXT_state <= READ_DATA;
           end
          WRITE:
           begin
              if(SS_n == 1)
                NEXT_state <= IDLE;
              else
                NEXT_state <= WRITE;
           end
          READ_DATA:
           begin
              if(SS_n == 1)
                NEXT_state <= IDLE;
              else
                NEXT_state <= READ_DATA;
           end
          READ_ADD:
           begin
              if(SS_n == 1)
                NEXT_state <= IDLE;
              else
                NEXT_state <= READ_ADD;
           end
        endcase
     end
```

```
/***********************************/
// output logic
always@(posedge clk)
  begin
    if(~reset) begin
        rx_valid <= 0;
        counter <= 0;
        updown <= 0;
        rx_data <= 0;
        ADD_or_Data <= 0;
        MISO <= 0;
    end
    else begin
        if(~updown) begin
            counter <= counter+1;
        end
        else begin
            counter <= counter+1;
        case(state)
        WRITE:begin
            updown <= 0;
            MISO <= 0;
            if(counter == 9) begin
                counter <= 0;
                rx_valid <=1;
            end
            else
            rx_valid <= 0;
            rx_data <= {rx_data[8:0],MOSI};
        READ_ADD:begin
            updown <= 0;
            MISO <= 0;
            if(counter == 9) begin
                counter <= 0;
                rx_valid <= 0;
            end
            else
            rx_valid <= 0;
            rx_data <= {rx_data[8:0],MOSI};
            end
        READ_DATA:begin
            if(counter == 9 && (~updown)) begin
                rx_valid <= 0;
                rx_data <= {rx_data[8:0],MOSI};
            end
            if(counter == 9)begin
                counter <= 8;
                rx_valid <= 1;
                updown <= 1;
            end
            if(rx_valid && tx_valid)
                MISO <= tx_data[counter];
            end
             default: begin
                 rx_valid <= 0;
                 rx_data <= 0;
                 counter <= 0;
                 MISO <= 0;
                 updown <= 0;
             end
        endcase
        end
        end
  end


endmodule
```

RTL for Wrapper (slave with RAM):

SPI_SLAVE_RAM.v

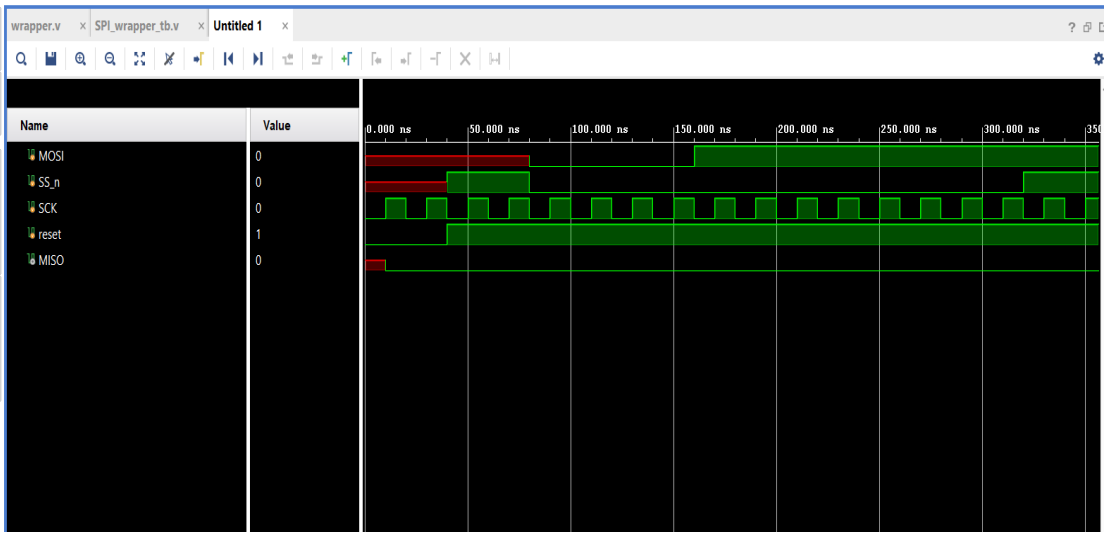
```
1  module SPI_final( MOSI, MISO, SS_n, clk, reset);
2  input MOSI,SS_n,clk,reset_n;
3  output MISO;
4  wire [9:0] rx_data;
5  wire rx_valid,tx_valid;
6  wire [7:0] tx_data;
7
8
9  //inistanitiation
10
11 spi_slave spi(MOSI, MISO, SS_n, clk, reset, rx_data, rx_valid, tx_data, tx_valid);
12 Spi_RAM RAM(rx_data, rx_valid, tx_data, tx_valid, clk, reset);
13
14 endmodule
```

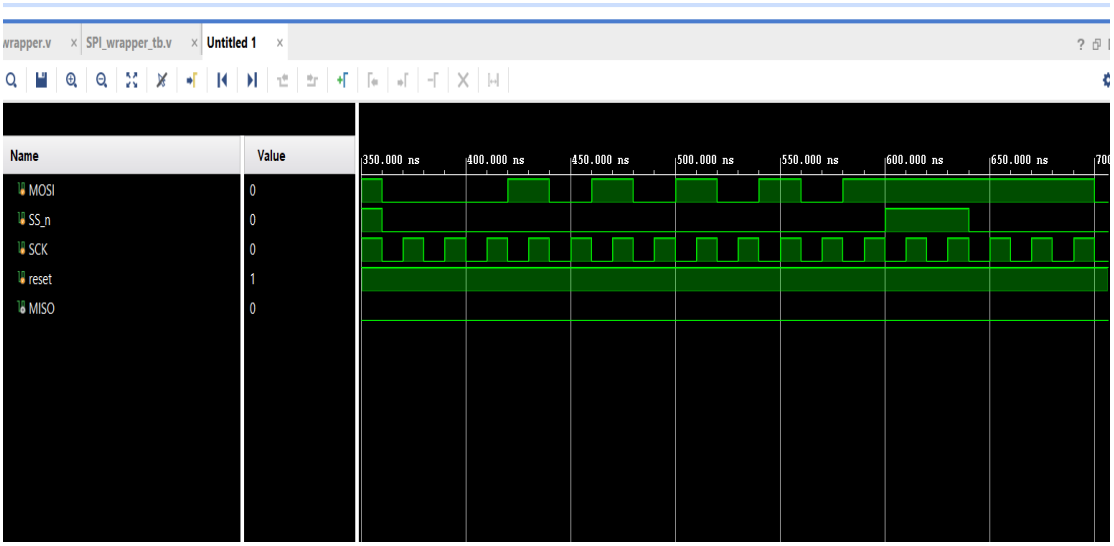
Testbench:

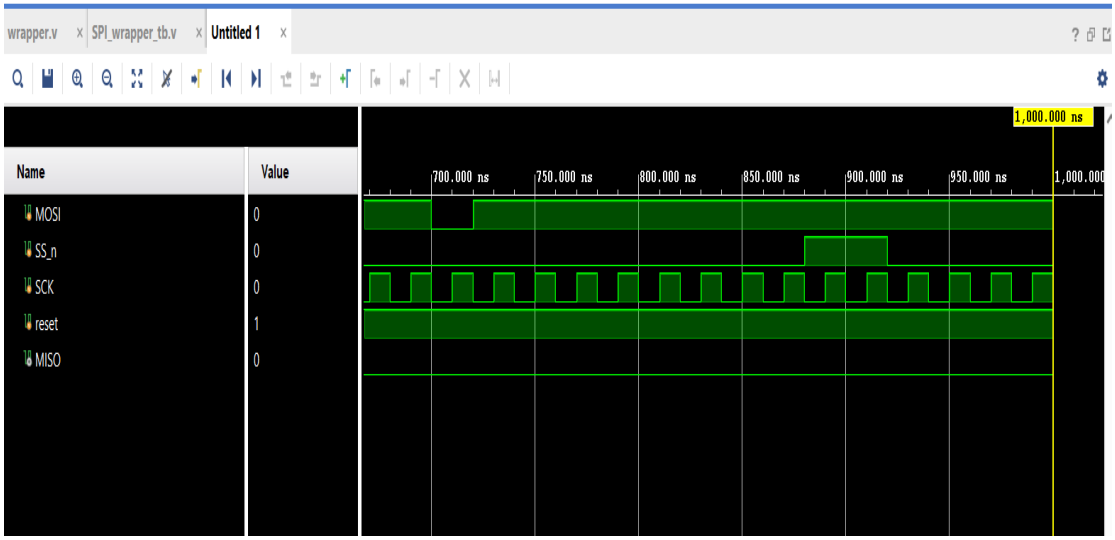
```
module SPI_wrapper_tb();
reg MOSI,SS_n,SCK,reset;
wire MISO;
SPI_final DUT( MOSI, MISO, SS_n, SCK,reset);


initial begin
    clk = 0;
     forever
       begin
        #10 clk = ~clk;
       end
end
initial begin
        reset=0;
        repeat(2)
        @(negedge SCK);
        reset=1;
        SS_n=1;
        repeat(2)
        @(negedge SCK);
        SS_n=0;
        MOSI=0;
        repeat(2)
        @(negedge SCK);
        MOSI=0;
        @(negedge SCK);
        MOSI=0;
        @(negedge SCK);
      MOSI=1;
      @(negedge SCK);
      MOSI=1;
      @(negedge SCK);
      MOSI=1;
      @(negedge SCK);
      MOSI=1;
      @(negedge SCK);
      MOSI=1;
      @(negedge SCK);
      MOSI=1;
      @(negedge SCK);
      MOSI=1;
      @(negedge SCK);
      SS_n=1;
   repeat(2)
      @(negedge SCK);
      SS_n=0;
      MOSI=0;
      repeat(2)
      @(negedge SCK);
      MOSI=0;
      @(negedge SCK);
      MOSI=1;
      @(negedge SCK);
      MOSI=0;
      @(negedge SCK);
      MOSI=1;
      @(negedge SCK);
       MOSI=0;
       @(negedge SCK);
       MOSI=1;
       @(negedge SCK);
       MOSI=0;
       @(negedge SCK);
       MOSI=1;
       @(negedge SCK);
       MOSI=0;
       @(negedge SCK);
       MOSI=1;
       @(negedge SCK);
       SS_n=1;
    repeat(2)
       @(negedge SCK);
       SS_n=0;
       MOSI=1;
       repeat(2)
       @(negedge SCK);
       MOSI=1;
       @(negedge SCK);
       MOSI=0;
       @(negedge SCK);
       MOSI=1;
       @(negedge SCK);
       MOSI=1;
       @(negedge SCK);
```

```
        MOSI=1;
        @(negedge SCK);
        MOSI=1;
        @(negedge SCK);
        MOSI=1;
        @(negedge SCK);
        MOSI=1;
        @(negedge SCK);
        MOSI=1;
        @(negedge SCK);
        SS_n=1;
    repeat(2)
        @(negedge SCK);
        SS_n=0;
        MOSI=1;
        repeat(2)
        @(negedge SCK);
        MOSI=1;
        @(negedge SCK);
        MOSI=1;
        @(negedge SCK);
        MOSI=0;
        @(negedge SCK);
        MOSI=0;
        @(negedge SCK);
        MOSI=1;
        @(negedge SCK);
         @(negedge SCK);
         MOSI=1;
         @(negedge SCK);
         MOSI=0;
         @(negedge SCK);
         MOSI=1;
         @(negedge SCK);
         MOSI=1;
         repeat(9)
         @(negedge SCK);
         SS_n=1;
          repeat(3)
         @(negedge SCK);
    $stop;
    end
endmodule
```

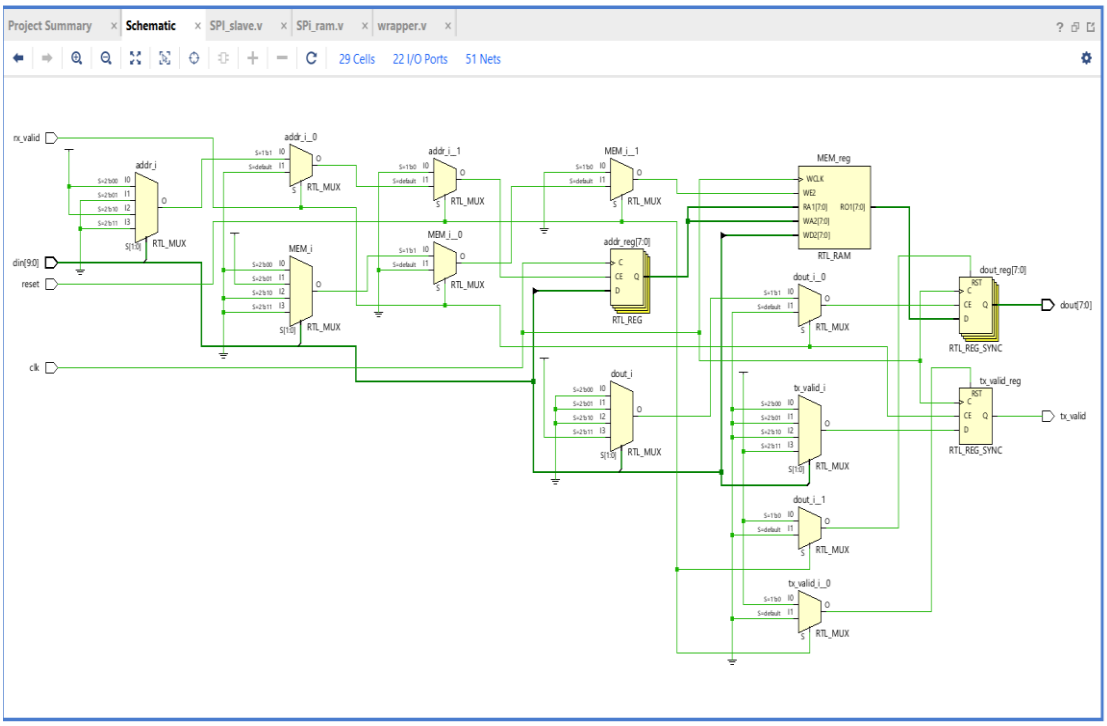
Waveform:


wrapper.v
SPI_wrapper_tb.v
Untitled 1
Name
Value
MOSI
SS_n
SCK
reset
MISO
0
0
0
1
0
0.000 ns
50.000 ns
100.000 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns

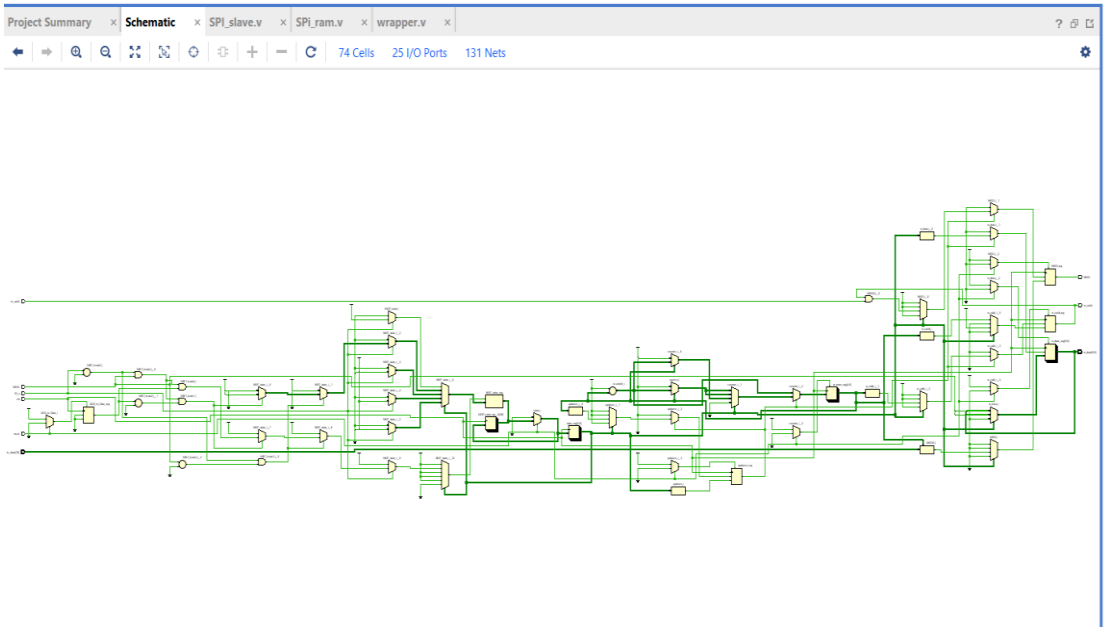

SPI_wrapper_tb.v
Untitled 1
Name
Value
MOSI
SS_n
SCK
reset
MISO
0
0
0
1
0
350.000 ns
400.000 ns
450.000 ns
500.000 ns
550.000 ns
600.000 ns
650.000 ns


wrapper.v
SPI_wrapper_tb.v
Untitled 1
Name
Value
MOSI
SS_n
SCK
reset
MISO
0
0
0
1
0
700.000 ns
750.000 ns
800.000 ns
850.000 ns
900.000 ns
950.000 ns
1,000.000 ns

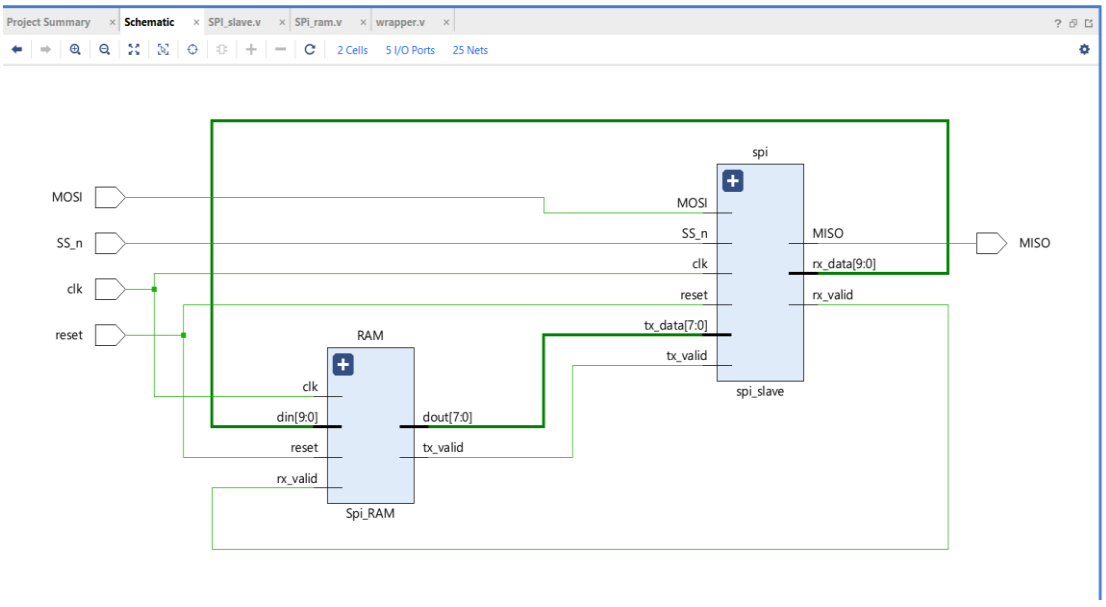
RTL Schematic for RAM:



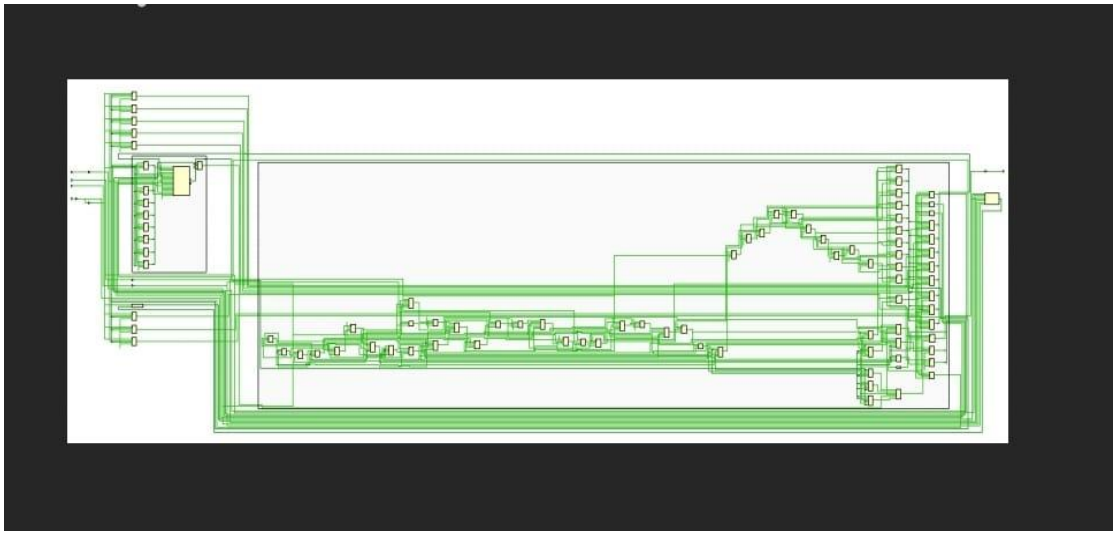
RTL Schematic for SPI slave:



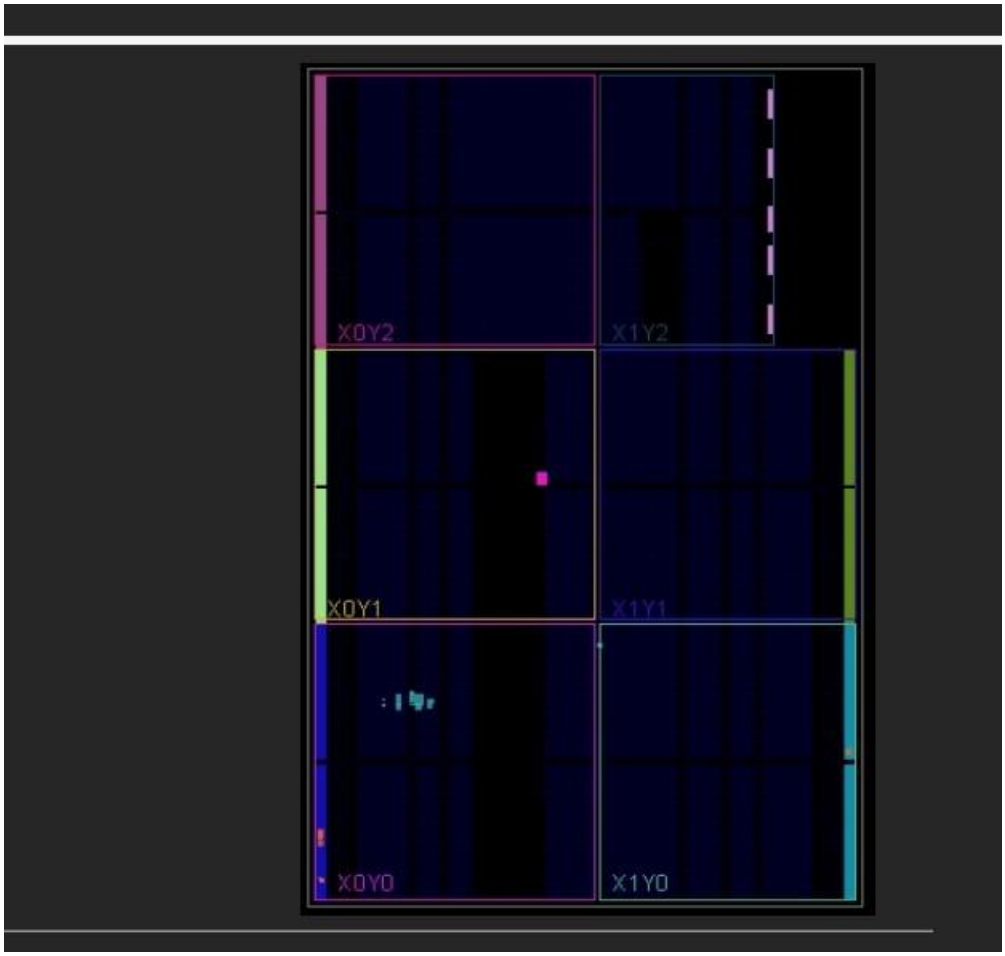
RTL Schematic for slave with RAM:



Synthesis schematic:



Implementation:



Grey encoding for our FSM:

- **Timing Report**

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.714 ns	Worst Hold Slack (WHS):	0.142 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:	110	Total Number of Endpoints:	110	Total Number of Endpoints:	50

All user specified timing constraints are met.

- **Utilization Report**

Name	Slice LUTs (20800)	Slice Registers (41600)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
N spi_wrapper	38	50	0.5	5	1
ram (ram)	1	8	0.5	0	0
spi_slave (spi_slave)	37	34	0	0	0