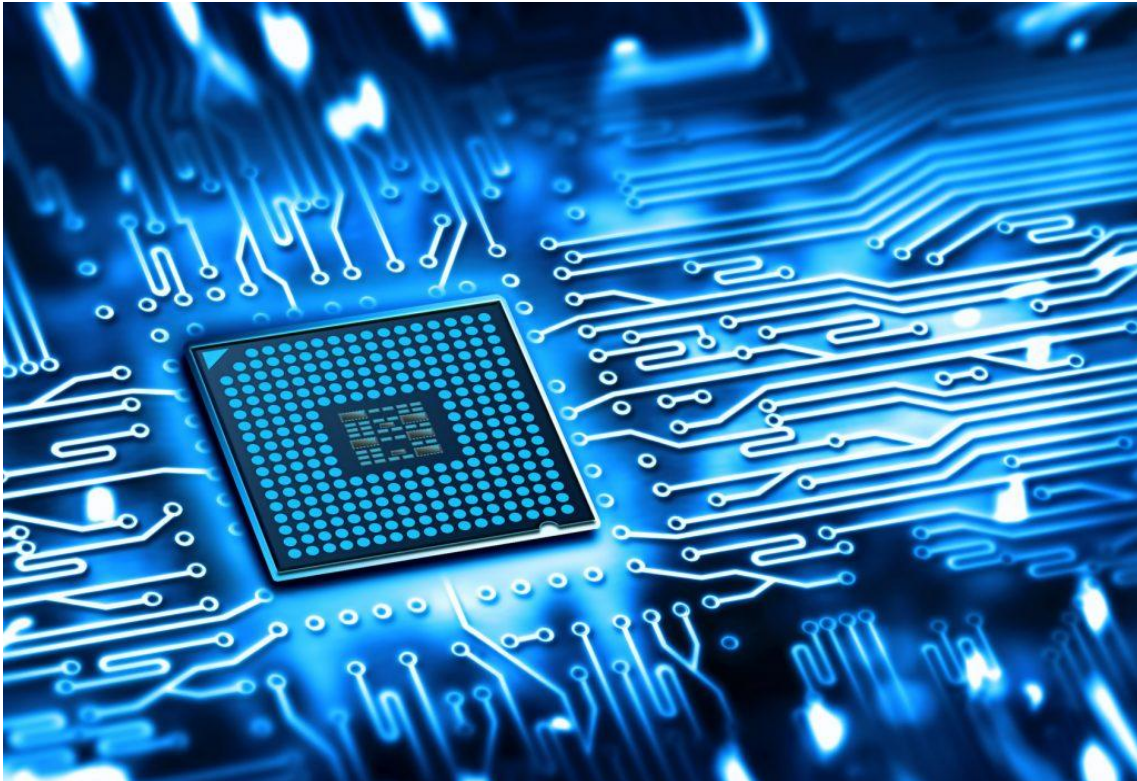


## **SPI Slave with single port RAM**

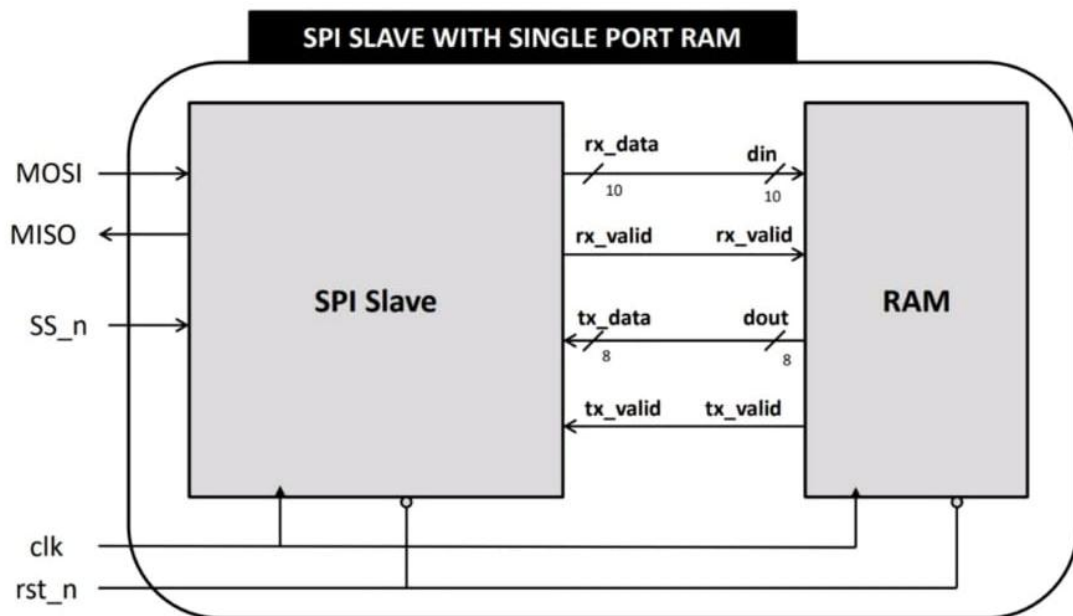


Prepared by  
Ali Abdel El Aleem Ali

## **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 slave input), MISO (master input slave output), and SS\_n (slave select).

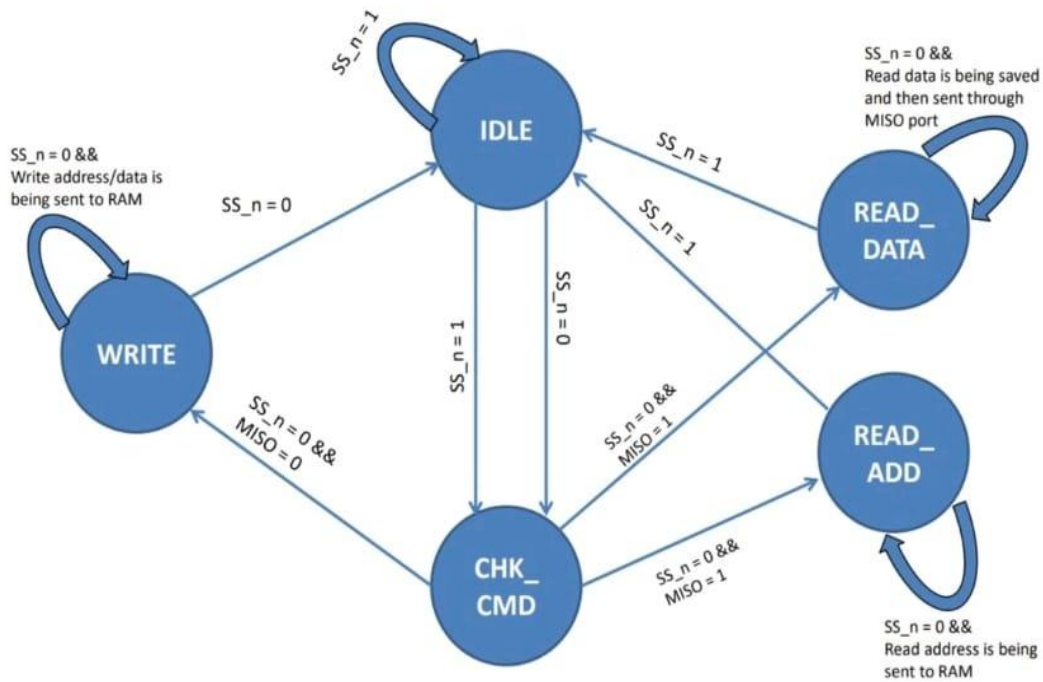
The master initiates communication and sends Serial data to either write to or read from peripherals such as memory modules.

### 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 manage communication and



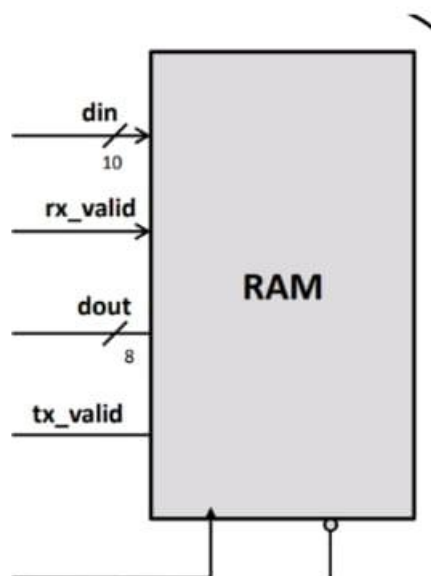
coordinate access to the RAM:

- **IDLE**: the default 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 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 then returns to 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



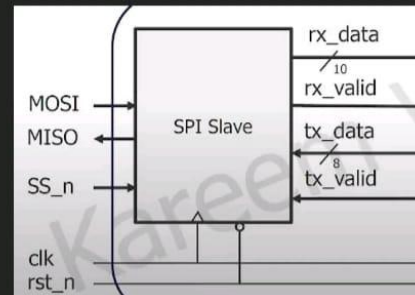
According to the most significant two bits of the din the RAM decides whether read\_addr/read\_data or write\_addr/write\_data. The rx\_valid signal tell the RAM to accept the din data for the address while tx\_valid tells the SPI to accept the dout data to the MISO.

### Design Procedure:

## 1. SPI slave inputs

### Design SPI inputs

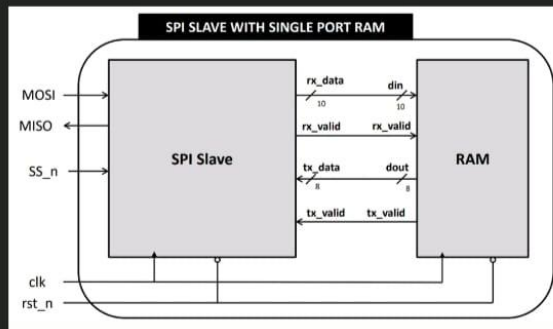
- MOSI (Master input-Slave output) .
- SS\_n : to enable the slave.
- CLK
- rst\_n
- tx\_data : data the meant to be sent and it is received from the RAM
- tx\_valid : flag to make sure that tx-data is ready to be sent



## 2. SPI slave outputs

### Output SPI ports

- MISO (Master input-Slave output)



## 3. RAM parameters

### RAM parameters

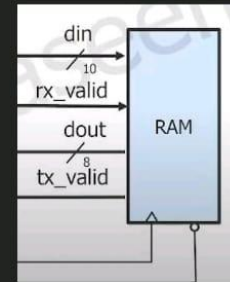
#### ■ Parameters

- MEM\_DEPTH, Default: 256
- ADDR\_SIZE, Default: 8

## 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 `din` 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

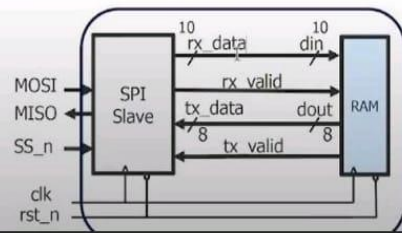
port	Din[9:8]	command	description
din	00	write	Hold <code>din[7:0]</code> as write address
	01		Write <code>din[7:0]</code> inside the write address that got before
	10	read	Hold <code>din[7:0]</code> as read address
	11		Get the data inside the read address that got before and put it in <code>dout</code>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



## 7. Write address

### RAM Write Command – Write Address

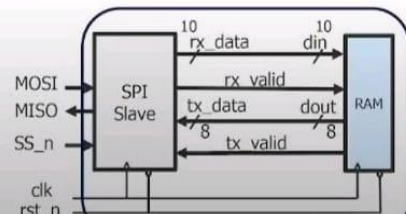
1. Master will start the write command by sending the write address value,  $rx\_data[9:8] = din[9:8] = 2'b00$
2.  $SS\_n = 0$  to tell the SPI Slave that the master will begin communication
3. SPI Slave check the first received bit on MOSI port '0' which is a control bit to let the slave determine which operation will take place "write in this case". SPI Slave then expects to receive 10 more bits, the first 2 bits are "00" on two clock cycles and then the  $wr\_address$  will be sent on 8 more clock cycles
4. Now the data is converted from serial "MOSI" to parallel after writing the  $rx\_data[9:0]$  bus
5.  $rx\_valid$  will be HIGH to inform the RAM that it should expect data on  $din$  bus
6.  $din$  takes the value of  $rx\_data$
7. RAM checks on  $din[9:8]$  and find that they hold "00"
8. RAM stores  $din[7:0]$  in the internal write address bus
9.  $SS\_n = 1$  to end communication from Master side



## 8. Write data

### RAM Write Command – Write Data

1. Master will continue the write command by sending the write data value,  $rx\_data[9:8] = din[9:8] = 2'b01$
2.  $SS\_n = 0$  to tell the SPI Slave that the master will begin communication
3. SPI Slave check the first received bit on MOSI port '0' which is a control bit to let the slave determine which operation will take place "write in this case". SPI Slave then expects to receive 10 more bits, the first 2 bits are "01" on two clock cycles and then the  $wr\_data$  will be sent on 8 more clock cycles
4. Now the data is converted from serial "MOSI" to parallel after writing the  $rx\_data[9:0]$  bus
5.  $rx\_valid$  will be HIGH to inform the RAM that it should expect data on  $din$  bus
6.  $din$  takes the value of  $rx\_data$
7. RAM checks on  $din[9:8]$  and find that they hold "01"
8. RAM stores  $din[7:0]$  in the RAM with  $wr\_address$  previously held
9.  $SS\_n = 1$  to end communication from Master side

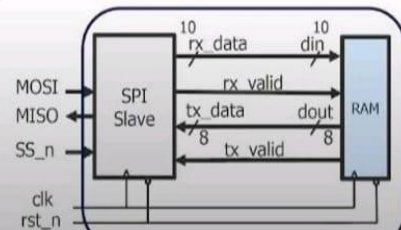




## 9. Read address

### RAM Read Command – Read Address

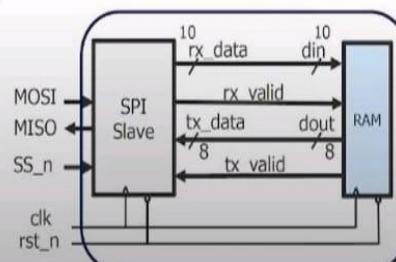
1. Master will start the write command by sending the read address value,  $rx\_data[9:8] = din[9:8] = 2'b10$
2.  $SS\_n = 0$  to tell the SPI Slave that the master will begin communication
3. SPI Slave check the first received bit on MOSI port '1' which is a control bit to let the slave determine which operation will take place "read in this case". SPI Slave then expects to receive 10 more bits, the first 2 bits are "10" on two clock cycles and then the  $rd\_address$  will be sent on 8 more clock cycles
4. Now the data is converted from serial "MOSI" to parallel after writing the  $rx\_data[9:0]$  bus
5.  $rx\_valid$  will be HIGH to inform the RAM that it should expect data on  $din$  bus
6.  $din$  takes the value of  $rx\_data$
7. RAM checks on  $din[9:8]$  and find that they hold "10"
8. RAM stores  $din[7:0]$  in the internal read address bus
9.  $SS\_n = 1$  to end communication from Master side



## 10. Read data

### RAM Read Command – Read Data

1. Master will start the write command by sending the read address value,  $rx\_data[9:8] = din[9:8] = 2'b11$
2.  $SS\_n = 0$  to tell the SPI Slave that the master will begin communication
3. SPI Slave check the first received bit on MOSI port '1' which is a control bit to let the slave determine which operation will take place "read in this case". SPI Slave then expects to receive 10 more bits, the first 2 bits are "11" on two clock cycles and then dummy data will be sent and ignored since the master is waiting for the data to be sent from slave side
4. Now the data is converted from serial "MOSI" to parallel after writing the  $rx\_data[9:0]$  bus
5.  $din$  takes the value of  $rx\_data$
6. RAM reads  $din[9:8]$  and find that they hold "11"
7. RAM will read from the memory with  $rd\_address$  previously held
8. RAM will assert  $tx\_valid$  to inform slave that data out is ready
9. Slave reads  $tx\_data$  and convert it into serial out data on MISO port
10.  $SS\_n = 1$ , Master ends communication after receiving data "8 clock cycles"



## RTL for RAM:

```
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; // SYNC reset

//declaring outputs

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

//declaring 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 <= 0;
                end
                2'b01: // write data
                begin
                    MEM[addr] <= din[7:0];
                    tx_valid <= 0;
                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
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
        end
    endcase
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  //inistantiation
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);
    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);
```

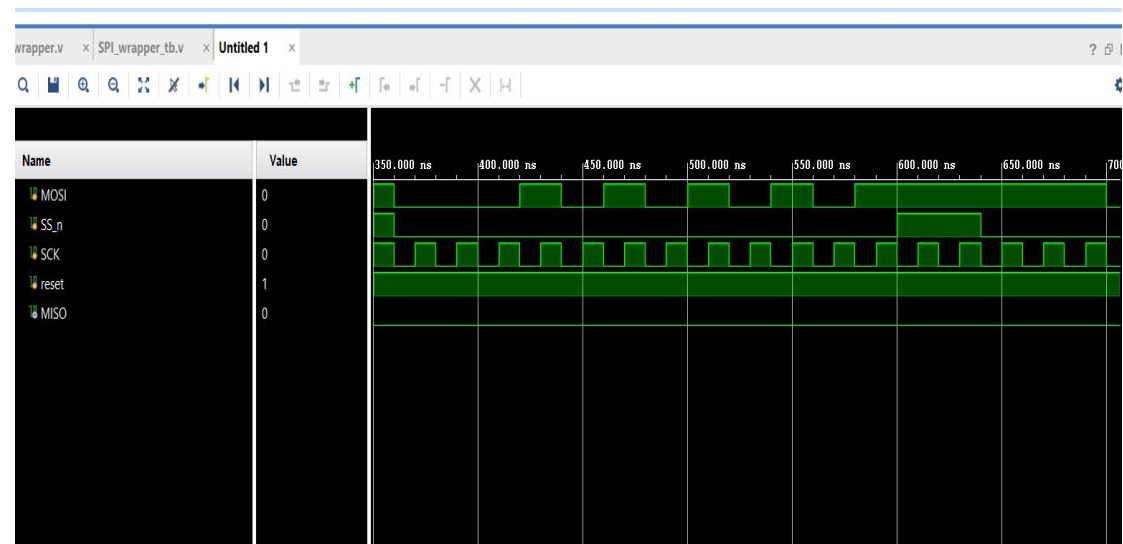
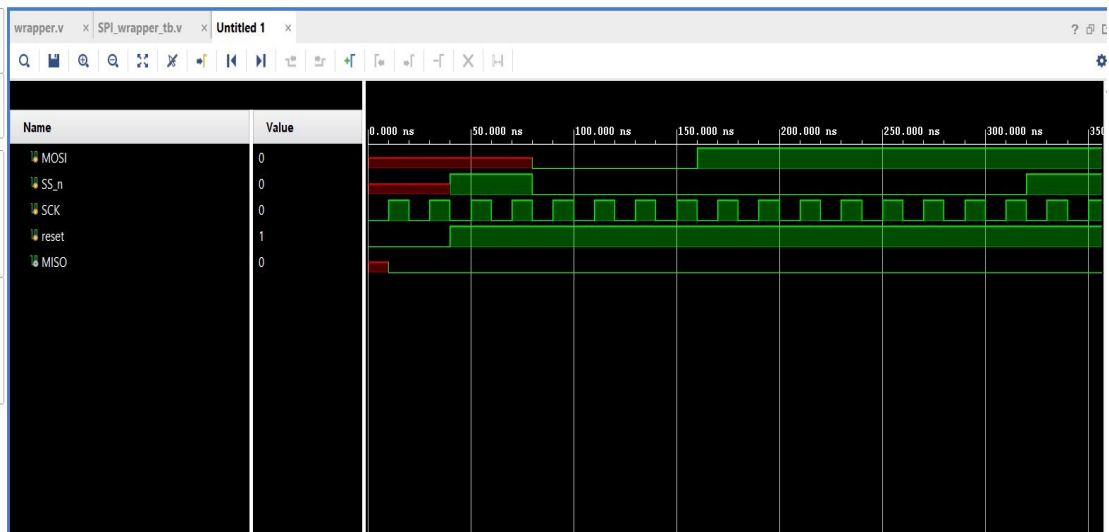
```

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);
    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

```

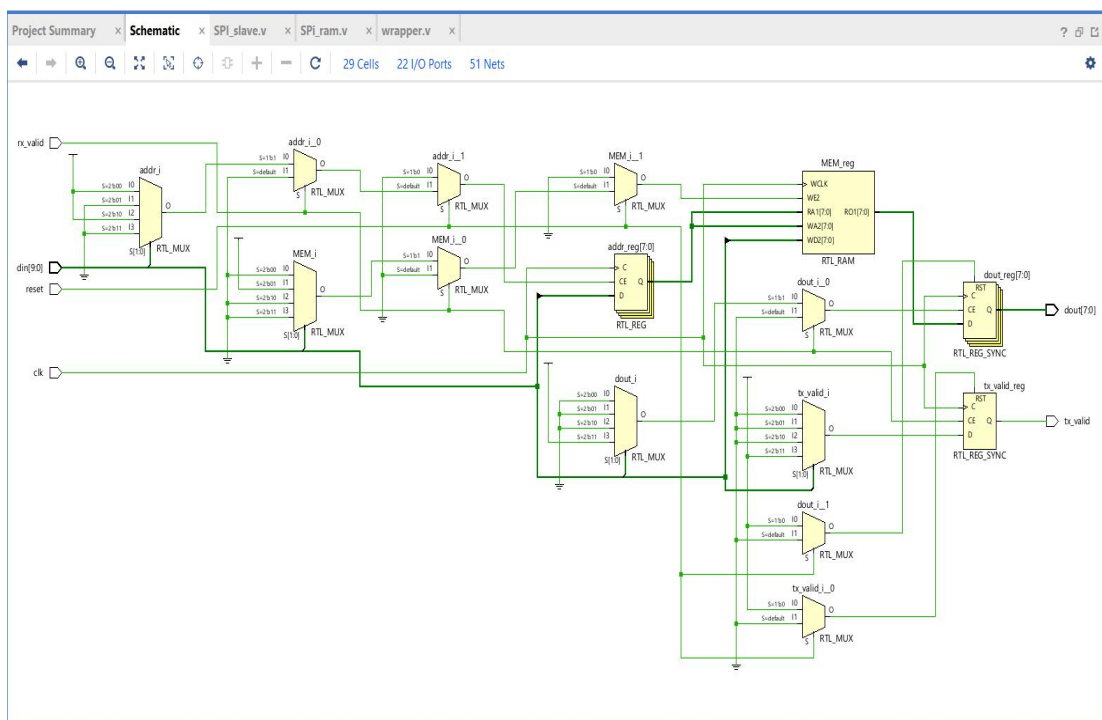


Waveform:

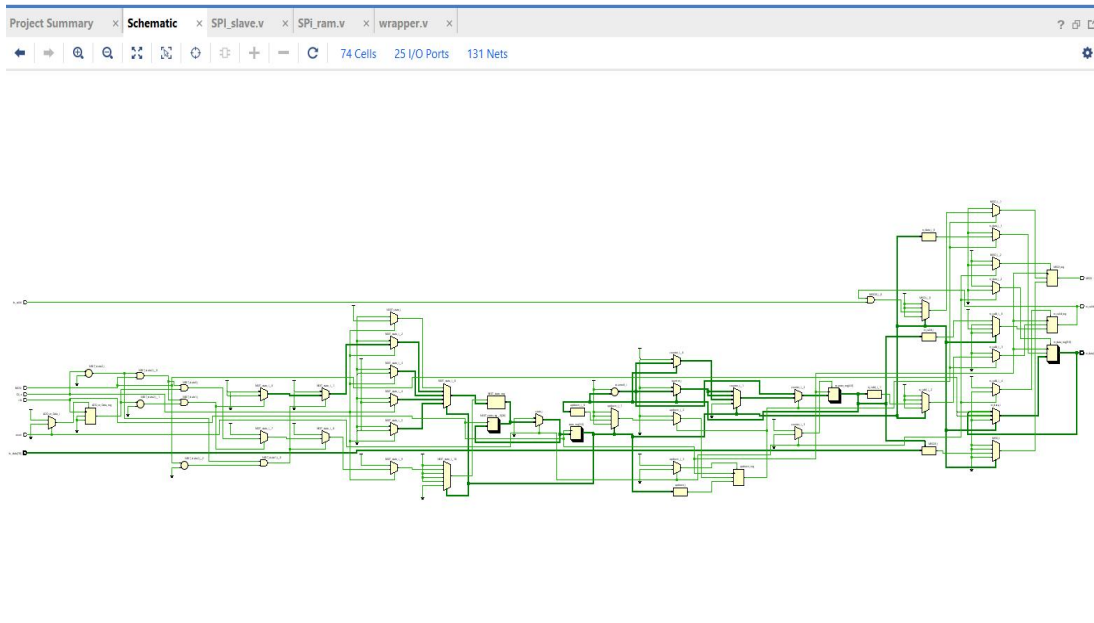




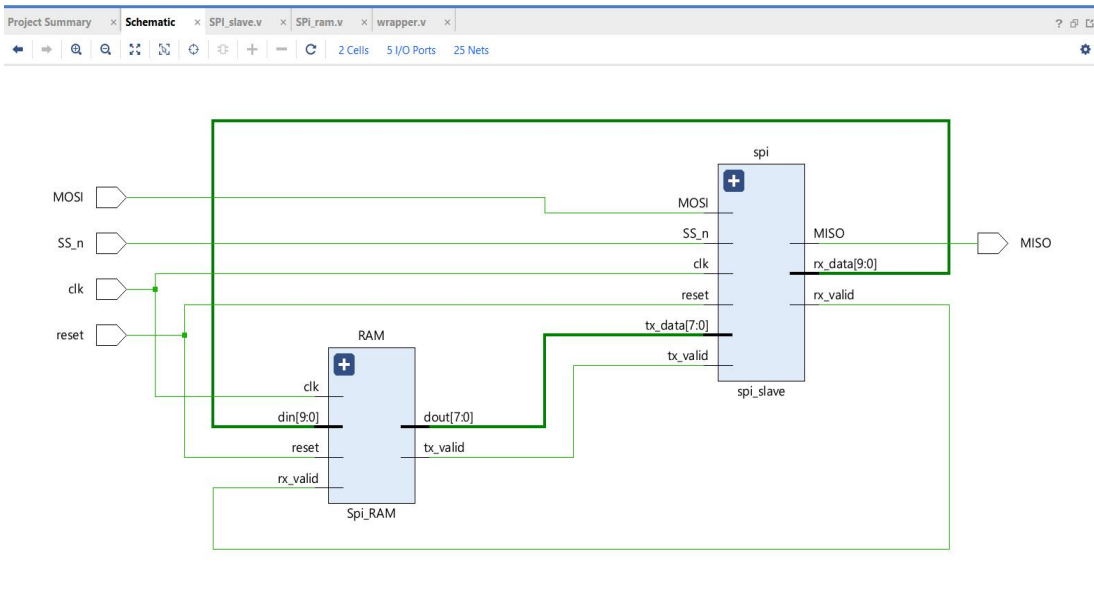
## RTL Schematic for RAM:



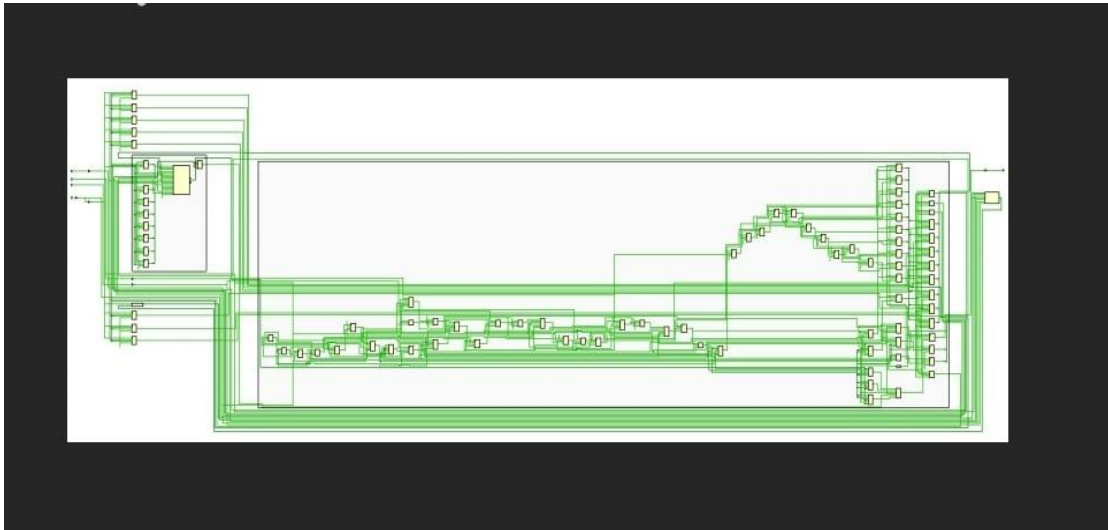
## RTL Schematic for SPI slave:



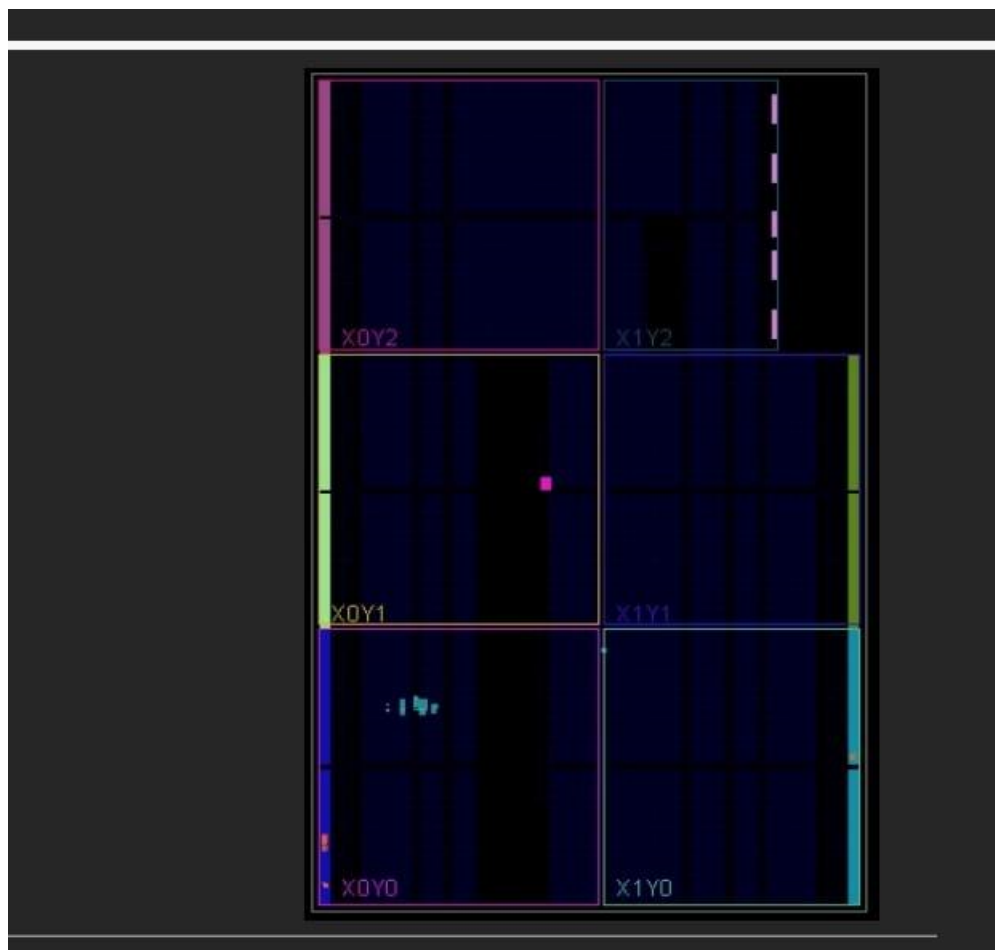
## RTL Schematic for slave with RAM:



## 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