SPI.V:

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 2024/10/19 12:59:21

// Design Name:

// Module Name: SPI

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////



module SPI_driver(

    input  wire clk,

    output reg [2:0] cur_state,


    input wire  SPI_MISO,

    output reg  SPI_MOSI,

    output reg  SPI_CLK,

    output reg  SPI_EN,


    output reg busy,

    input wire command_read,

    input wire rx_read,

    input wire tx_read,

    input wire [1:0] Spi_rw,

    output reg [7:0] Spi_rx_reg,

    input wire [7:0] Spi_tx_reg

    );


localparam IDLE  = 3'b000;
localparam SPITX = 3'b001;

localparam SPIRX = 3'b010;

localparam SPIED = 3'b100;



reg [1:0] command_FIFO[15:0];

reg [7:0] tx_FIFO[15:0];

reg [7:0] rx_FIFO[15:0];

reg [3:0] command_addrw;

reg [3:0] command_addrr;

reg [3:0] tx_addrw;

reg [3:0] tx_addrr;

reg [3:0] rx_addrw;

reg [3:0] rx_addrr;

wire command_empty;

wire tx_empty;

wire rx_empty;


initial begin

   cur_state = 7'd0;

   SPI_MOSI = 1'b0;

   SPI_CLK = 1'b0;

   SPI_EN = 1'b0;

   busy = 1'b0;

   command_addrw = 4'b0;

   command_addrr = 4'd0;

   tx_addrw = 4'd0;

   tx_addrr = 4'd0;

   rx_addrw = 4'd0;

   rx_addrr = 4'd0;

end


assign tx_empty = &(~(tx_addrr^tx_addrw));

assign rx_empty = &(~(rx_addrr^rx_addrw));

assign command_empty = &(~(command_addrr^command_addrw));


always @(posedge clk) begin

   if (command_read == 1'b1) begin

      command_FIFO[command_addrw] <= Spi_rw;

      command_addrw <= command_addrw + 1;

   end

   if (tx_read == 1'b1) begin
```

```
        tx_FIFO[tx_addrw] <= Spi_tx_reg;
        tx_addrw <= tx_addrw + 1;
    end
    if (rx_read == 1'b1) begin
        if(rx_empty != 1'b1) begin
            Spi_rx_reg <= rx_FIFO[rx_addrr];
            rx_addrr <= rx_addrr + 1;
        end else begin
            Spi_rx_reg <= 8'b11111111;
        end
    end
end


reg[2:0] bit_counter;
reg[2:0] clk_counter;
reg[7:0] rx_temp_reg;

always @(posedge clk) begin
        case(cur_state)
            IDLE : begin
                busy <= 1'b0;
                if(command_empty == 1'b0)begin
                    if(command_FIFO[command_addrr] == 2'b01)begin
                        command_addrr  <= command_addrr + 1;
                        cur_state      <= SPITX;
                        clk_counter    <= 3'b000;
                        bit_counter    <= 3'b111;
                        busy <= 1'b1;
                    end
                    //add error detection if the first command out of IDLE is rx, this is incorrectly set by the controller
                    // also add error detection if when entering TX, check for TX_FIFO empty, if not, controller is incorrectly set
                end
            end
            SPITX: begin
                case(clk_counter)
                    3'b000 : begin
                        SPI_EN   <= 1'b1;
                        SPI_CLK  <= 1'b0;
                        SPI_MOSI <= tx_FIFO[tx_addrr][bit_counter];
                        busy <= 1'b1;
                        clk_counter <= clk_counter + 1;
                    end
                    3'b100 : begin
                        SPI_CLK <= 1'b1;
                        clk_counter <= clk_counter + 1;
                    end
                    3'b111 : begin
                        if(bit_counter != 3'b000) begin
                            bit_counter <= bit_counter -1;
                            clk_counter <= 3'b000;
                        end else begin
                            bit_counter <= 3'b111;
                            clk_counter <= 3'b000;
                            tx_addrr <= tx_addrr + 1;
                            if(command_empty == 1'b1)cur_state   <= SPIED;
                            else begin
                                if (command_FIFO[command_addrr] == 2'b01) cur_state <= SPITX;
                                else cur_state <= SPIRX;
                                command_addrr <= command_addrr + 1;
                                //add error detection if 2'b10 or 2'b01 is not the data read from the FIFO
                            end
                        end
                    end
                    default : begin
                        clk_counter <= clk_counter + 1;
                    end
                endcase
            end
            SPIRX: begin
                case(clk_counter)
                    3'b000 : begin
                        SPI_EN  <= 1'b1;
                        SPI_CLK <= 1'b0;
                        clk_counter <= clk_counter + 1;
                    end
                    3'b100 : begin
                        SPI_CLK  <= 1'b1;
                        rx_temp_reg[bit_counter] <= SPI_MISO;
                        clk_counter <= clk_counter +1;
                    end
                    3'b111 : begin
                        if(bit_counter != 3'b000) begin
                            bit_counter <= bit_counter - 1;
```

```
              clk_counter <= 3'b000;
            end else begin
              bit_counter <= 3'b111;
              clk_counter <= 3'b000;
              rx_FIFO[rx_addrw] <= rx_temp_reg;
              rx_addrw <= rx_addrw + 1;
              if(command_empty == 1'b1) cur_state <= SPIED;
              else begin
                if(command_FIFO[command_addrr] == 2'b01) cur_state <= SPITX;
                else cur_state <= SPIRX;
                command_addrr <= command_addrr + 1;
              end
            end
          end
          default : begin
            clk_counter <= clk_counter + 1;
          end
        endcase
      end
      SPIED : begin
        case(clk_counter)
          3'b000 : begin
            SPI_EN  <=1'b1;
            SPI_CLK <=1'b0;
            clk_counter <= clk_counter + 1;
          end
          3'b100 : begin
            SPI_EN   <= 1'b0;
            SPI_CLK  <= 1'b0;
            SPI_MOSI <= 1'b0;
            clk_counter <= 3'b000;
            cur_state <= IDLE;
          end
          default : begin
            clk_counter <= clk_counter + 1;
          end
        endcase
      end
    endcase
end


endmodule
```

SPI_controller.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2024/09/22 14:34:42
// Design Name:
// Module Name: TS_controller
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module SPI_controller(
    input clk,

    input wire [31:0] PC_rx,
    input wire [31:0] PC_addr,
    input wire [31:0] PC_val,
    output reg [31:0] PC_tx,

    output reg command_read,

    output reg rx_read,

    output reg tx_read,

    output reg [7:0] tx_byte,

    output reg [1:0] rw,

    output reg [9:0] cur_state,

    input wire [7:0] rx_byte,

    input wire busy

    );


    reg busy_reg = 0;

    reg [7:0] tx_byte_reg;

    reg [7:0] rx_byte_reg;

    reg [2:0] read_counter;

    reg [31:0] PC_rx_reg1, PC_rx_reg2;


    localparam idle_     = 10'b0000000001;

    localparam start_wr  = 10'b0000000010;

    localparam tx_wr1    = 10'b0000000100;

    localparam tx_wr2    = 10'b0000001000;

    localparam end_wr    = 10'b0000010000;

    localparam start_rt  = 10'b0000100000;

    localparam tx_rt     = 10'b0001000000;

    localparam rx_rt     = 10'b0010000000;

    localparam wait_rt   = 10'b0100000000;

    localparam end_rt    = 10'b1000000000;


    initial begin
```

```
    cur_state <= idle_;
    PC_rx_reg1 <= 0;
    PC_rx_reg2 <= 0;
    tx_byte_reg <= 0;
    rx_byte_reg <= 0;
    busy_reg <= 1'b0;
    read_counter <= 0;
end

integer i;

always @(posedge clk) begin
    for (i=0; i<8; i=i+1) begin
        tx_byte[i] <= tx_byte_reg[i];
        rx_byte_reg[i] <= rx_byte[i];
    end
end


always @(posedge clk) begin
    case (cur_state)
        idle_ : begin
            command_read <= 1'b0;
            tx_read <= 1'b0;
            rx_read <= 1'b0;
            PC_rx_reg1 <= PC_rx;
            PC_rx_reg2 <= PC_rx_reg1;
            if (PC_rx_reg2[0] == 1'b0 && PC_rx_reg1[0] == 1'b1) begin
                cur_state <= start_wr;
            end
            if (PC_rx_reg2[1] == 1'b0 && PC_rx_reg1[1] == 1'b1) begin
                cur_state <= start_rt;
            end
        end
        //Write single byte
        start_wr: begin
            tx_byte_reg <= {1'b1, PC_addr[6:0]};
            cur_state <= tx_wr1;
        end
        tx_wr1: begin
            tx_byte_reg <= PC_val[7:0];
            command_read <= 1'b1;
            rw <= 2'b01;
            tx_read <= 1'b1;
            cur_state <= tx_wr2;
        end
        tx_wr2: begin
            command_read <= 1'b1;
            rw <= 2'b01;
            tx_read <= 1'b1;
            cur_state <= end_wr;
        end
        end_wr : begin
            tx_byte_reg <= {8{1'b0}};
            command_read <= 1'b0;
            tx_read <= 1'b0;
            cur_state <= idle_;
        end
```

```
//Read two byte
start_rt: begin
   tx_byte_reg <= {1'b0, PC_addr[6:0]};
   cur_state <= tx_rt;
end
tx_rt: begin
   command_read <= 1'b1;
   rw <= 2'b01;
   tx_read <= 1'b1;
   cur_state <= rx_rt;
end
rx_rt : begin
   command_read <= 1'b1;
   tx_read <= 1'b0;
   rw <= 2'b10;
   cur_state <= wait_rt;
end
wait_rt : begin
   tx_byte_reg <= {8{1'b0}};
   command_read <= 1'b0;
   rw <= 2'b00;
   busy_reg <= busy;
   if (busy_reg == 1'b1 && busy == 1'b0) begin
      rx_read <= 1'b1;
      cur_state <= end_rt;
   end
end
end_rt: begin
   rx_read <= 1'b0;
```

```
                read_counter <= read_counter + 1;
                if (read_counter == 2) begin
                    PC_tx[7:0] <= rx_byte_reg;
                    read_counter <= 0;
                    cur_state <= idle_;
                end
            end
            default : begin
                tx_byte_reg <= {8{1'b0}};
                cur_state <= idle_;
            end
        endcase
    end


endmodule
```

Python

```
# -*- coding: utf-8 -*-

#%%
# import various libraries necessary to run your Python code
import time   # time related library
import sys,os    # system related library
ok_sdk_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\Python\\x64"
ok_dll_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\lib\\x64"

sys.path.append(ok_sdk_loc)   # add the path of the OK library
```

```
os.add_dll_directory(ok_dll_loc)


import ok    # OpalKelly library
#%%
def reset_image_sensor():
   dev.SetWireInValue(0x01, 1)
   dev.UpdateWireIns()
   time.sleep(1)
   dev.SetWireInValue(0x01, 0)
   dev.UpdateWireIns()
   time.sleep(0.1)
#%%
def write_to_device(reg_addr, value):
   dev.SetWireInValue(0x00, 0)
   dev.UpdateWireIns()
   dev.SetWireInValue(0x02, reg_addr)
   dev.SetWireInValue(0x03, value)
   dev.UpdateWireIns()  # Update the WireIns
   time.sleep(0.2)
   dev.SetWireInValue(0x00, 1) # Write trigger
   dev.UpdateWireIns()  # Update the WireIns
   time.sleep(0.2)
   dev.SetWireInValue(0x00, 0)
   dev.UpdateWireIns()  # Update the WireIns


#%%
def read_from_device(reg_addr):
   dev.SetWireInValue(0x00, 0)
   dev.UpdateWireIns()  # Update the WireIns
```

```
    time.sleep(0.2)
    dev.SetWireInValue(0x02, reg_addr)
    dev.SetWireInValue(0x00, 2)  # Read trigger
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(0.2)
    dev.UpdateWireOuts()
    read = dev.GetWireOutValue(0x20)
#    if slave_addr == 0x3C:
#        m_L = read // 2**8
#        m_H = read - (m_L * 2**8)
#        read =  m_H * 2**8 + m_L
#    if read >= 2**15:
#        read = read - 2**16 # deal with 2's complement
    dev.SetWireInValue(0x00, 0)
    dev.UpdateWireIns()
    return read
#%%
# Define FrontPanel device variable, open USB communication and
# load the bit file in the FPGA
dev = ok.okCFrontPanel()  # define a device for FrontPanel communication
SerialStatus=dev.OpenBySerial("")      # open USB communication with the OK board
# We will NOT load the bit file because it will be loaded using JTAG interface from Vivado

# Check if FrontPanel is initialized correctly and if the bit file is loaded.
# Otherwise terminate the program
print("----------------------------------------------------")
if SerialStatus == 0:
    print ("FrontPanel host interface was successfully initialized.")
else:
    print ("FrontPanel host interface not detected. The error code number is:" +
str(int(SerialStatus)))
    print("Exiting the program.")
    sys.exit ()


#%% Reg and value constants
start1_h = 3
start1_l = 4
#%%
# Define the two variables that will send data to the FPGA
# We will use WireIn instructions to send data to the FPGA
time.sleep(1)
reset_image_sensor()
write_to_device(3, 8)
write_to_device(4, 160)
write_to_device(57, 3)
write_to_device(58, 44)
write_to_device(59, 240)
write_to_device(60, 10)
write_to_device(69, 9)
write_to_device(80, 2)
write_to_device(83, 187)
write_to_device(97, 240)
write_to_device(98, 10)
write_to_device(100, 112)
write_to_device(101, 98)
write_to_device(102, 34)
write_to_device(103, 64)
write_to_device(106, 94)
write_to_device(107, 110)

write_to_device(108, 91)

write_to_device(109, 82)

write_to_device(110, 80)

write_to_device(117, 91)

print(read_from_device(3))

print(read_from_device(4))

print(read_from_device(57))

print(read_from_device(58))

print(read_from_device(59))

print(read_from_device(60))

print(read_from_device(69))

print(read_from_device(80))

print(read_from_device(83))

print(read_from_device(97))

print(read_from_device(98))

print(read_from_device(100))

print(read_from_device(101))

print(read_from_device(102))

print(read_from_device(103))

print(read_from_device(106))

print(read_from_device(107))

print(read_from_device(108))

print(read_from_device(109))

print(read_from_device(110))

print(read_from_device(117))


dev.Close
```