SPI-based communication

Daniel Paredes, Cristian Gil

1 SPI-based Communication protocol

The standard Serial Peripheral Interface (SPI) which uses the MASTER-SLAVE principle has 4 lines of data transmission (spi_clk, MISO, MOSI, SS). However, even when our designed protocol also uses the Master-Slave principle it only has 2 lines of data transmission since we only have one slave, and the communication only goes from the master to the slave. In other words, our communication line, as shown in Figure 1, has a generated clock (SPI_clk) and the MOSI (Master Output, Slave Input)

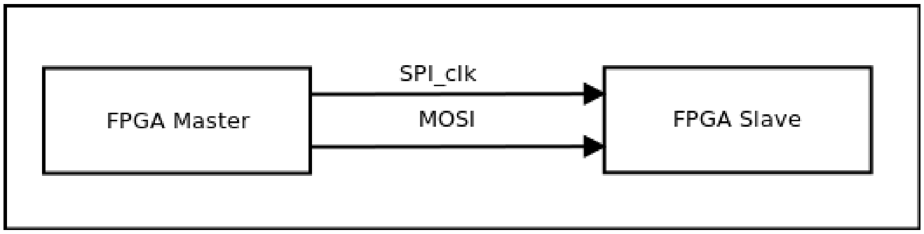

Figure 1: Communication between FPGA master and the slave

1.1 Details of our SPI-based protocol

Our protocol is based on five blocks:.

- A **Clock generator** generates the SPI clock
- **Rising edge detectors** detect the rising edges of the SPI clock. We need it because the SPI clock is not ideal (instant change between '0' and '1'); in other words, it takes some time to change from '0' to '1' and vice versa as can be observed in Figure 2..
- **Delays** that work as synchronizers since the SPI clock is not ideal we need to avoid reading data from any signal during value transitions.
- The **SPI transmission component** is in charge of sending the data through TX port

- The **SPI reception component** is in charge of receiving the data through RX port

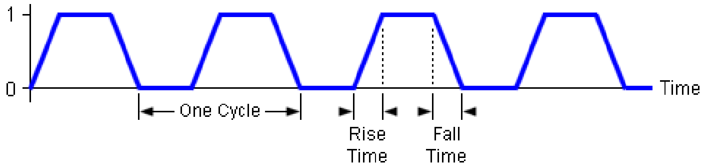

Figure 2: Non ideal clock signal

2 Communication Components

All the components in our design have a internal clock signal as well as an asynchronous reset signal.

2.1 Clock Generator

We used a state machine with only two states (ONE, ZERO) and a counter to generated the clock signal that will be used as `SPI clock`. In each state we reset the counter to zero and it starts counting up to some fixed maximum count, in our case is maximum count fixed to ten. Which means that our generated clock will run twenty times slower which is more than enough to ensure that there will not be missing bits.. The entity of this block is as follows:

```
entity clk_generator is
    port (  clk: in std_logic;
            reset: in std_logic;
            en: in std_logic;
            clk_out: out std_logic
         );
end entity clk_generator;
```

The output of a testbench can be observed in Figure 3.

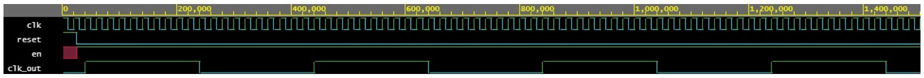

Figure 3: Testbench results - clock generator

2.2 Delay or synchronizer

This block is basically a flip flops type D. The entity definition is presented as well as the testbench result (Figure 4).

```
entity  delay_sync is
    port (  clk: in std_logic;
            reset: in std_logic;
            d: in std_logic;
            q: out std_logic
         );
end entity delay_sync;
```

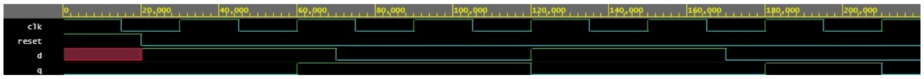

Figure 4: Testbench results - Delay or synchronizer

2.3 Rising Edge Detector

This component has the following entity definition:

```
entity rising_edge_detector is
    port (  clk: in std_logic;
            reset: in std_logic;
            clk_in: in std_logic;
            edge: out std_logic
         );
end entity rising_edge_detector;
```

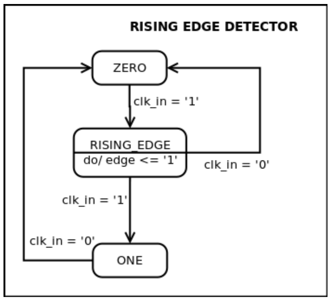

Figure 5: Rising Edge Detector

This block is based on a state machine with three states (ONE, ZERO, RISING_EDGE). The state machine is presented in Figure 5. The initial state is ZERO, once the clock changes to 1 the detector goes to RISING_EDGE state.

In the RISING_EDGE state `edge` is set to `1`, which means that a rising edge has been detected; afterwards, whether the clock signal is `0` or `1` the next state is either `ZERO` or `ONE`. If the next state is `ONE`, the component will be in that state until the clock signal changes to `0`.

The testbench result is presented in Figure 6.

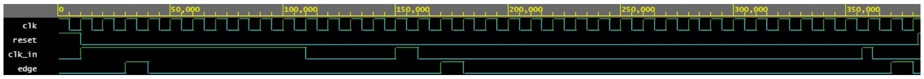

Figure 6: Testbench results - Rising edge detector

2.4 SPI TX component

The entity definition of this component is as follows:

```
entity spi_tx is
    port (  clk: in std_logic;
            reset: in std_logic;
            edge: in std_logic;
            data: in std_logic_vector (7 downto 0);
            empty_buf: out std_logic;
            tx: out std_logic
         );
end entity spi_tx;
```

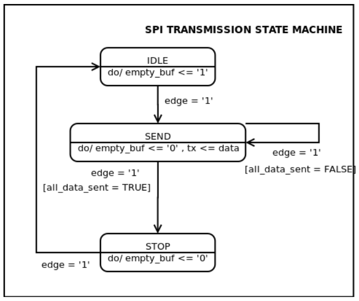

Figure 7: SPI transmission component state machine

The behavior of this component is shown in Figure 7. The SPI TX component will be in state IDLE until an `edge` is detected. Note when the component is in IDLE, it is waiting for data to send; thus, transmission buffer is empty (`empty_buf <= '1'`). Once an edge is detected the component goes to SEND state. It's in this state where the serial transmission takes place. In every detected edge a bit is sent through TX port, also the flag that states the transmission is taking place is sent to `0` (`empty_buf <= '0'`). In our case the least significant bit is sent first. Once all the data is sent (`all_data_sent = TRUE`), the component goes to STOP state. In this state the component makes sure that the last sent bit had enough time to be received without problems. Finally, in the next `edge` the component goes to IDLE state. The results of the testbench can be observed in Figure 8.

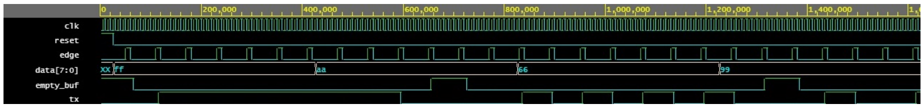

Figure 8: Testbench results - SPI transmission component

2.5 SPI RX component

The entity definition of this component is as follows:

```
entity spi_rx is
    port (  clk: in std_logic;
            reset: in std_logic;
            edge: in std_logic;
            rx: in std_logic;
            data: out std_logic_vector (7 downto 0);
            full_buf: out std_logic
         );
end entity spi_rx;
```

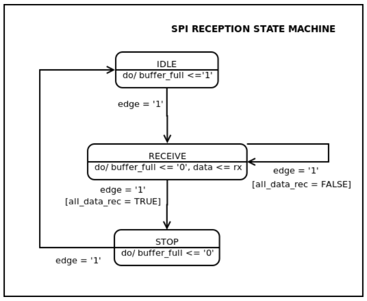

Figure 9: SPI reception component state machine

This component behavior is shown in Figure 9. The SPI RX component will be in state IDLE until an `edge` is detected. Similar to transmission block, the component reception buffer is set to 1 (`buffer_full <= '1'`) as long as it is IDLE state. When an edge is detected the component goes to SEND state starting the data frame reception. In every detected edge a bit is received through the RX port, also the flag that states the reception is taking place is set to `0` (`buffer_full <= '0'`). Note that the least significant bit is received first.

The reception is done by using a shifting register. Once all the data is received (`all_data_rec = TRUE`), the component goes to STOP state. In this state the component makes sure that all the bits have been received and the new data is ready to be read. Finally, in the next `edge` the component goes to IDLE state again.

The results of the testbench can be observed in Figure 10.

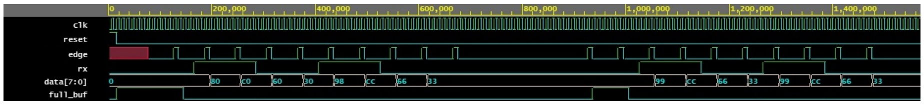

Figure 10: Testbench results - SPI reception component

3 SPI Transmission block

This block is in charge of transmiting the data from `DATA` using our SPI protocol. This block should be implemented in the master device. The internal block diagram is presented in Figure 11. Our transmission block has: 2 inputs: Enable SPI clock signal `EN` and data to transmit `DATA`; and 3 outputs: A buffer `empty_buffer` that works as feedback to other blocks that want to send data through our SPI channel, in other words, tells the other blocks whether this block is ready to send data (`empty_buf = '1'`) or not (`empty_buf = '0'`). The SPI generated clock `SPI_clk` used to synchronize the communication between master and slave, and the transmission pin `TX`.

As it can be observed in Figure 11, our design includes 4 blocks. A clock generator that will generated the `SPI_clk` that runs 20 times slower than the FPGA internal clock); a delay or synchronizer that is used to avoid reading data during ramps in the `SPI_clk`; a rising edge detector, used to detect rising edges in our `SPI_clk`; and the proper SPI transmission component which is in charge of transmiting the data.

The entity definition is as follows:

```vhdl
entity spi_block_tx is
    port (  clk: in std_logic;
            reset: in std_logic;
            data: in std_logic_vector (7 downto 0);
            en: in std_logic;
            empty_buf: out std_logic;
            clk_spi: out std_logic;
            tx: out std_logic
        );
end entity spi_block_tx;
```

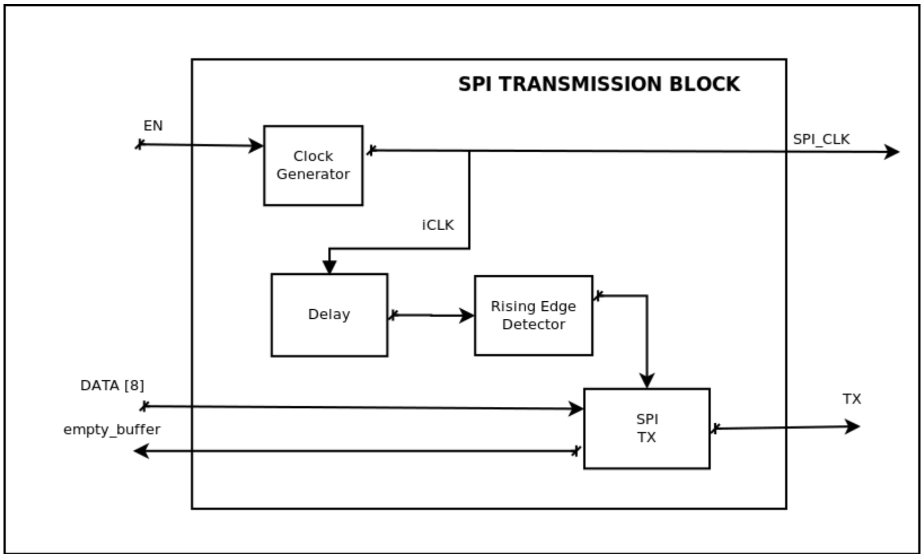

Figure 11: SPI TX - block diagram

A pseudocode of the main functionality of the SPI transmission block is presented

```
-- SPI transmission block pseudocode
while (EN = '1')
    generate(SPI_clk)
    while ( isSendingData )
        empty_buffer <= '0'
        if ( rising_edge_detector (SPI_clk) )
            TX <= send_BIT

    empty_buffer <= '1'
```

The testbench result can be observed in Figure 12.

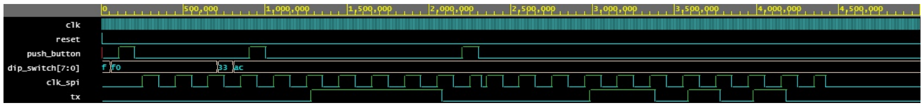

Figure 12: Testbench results - SPI Transmission block

4 SPI Reception block

This block is in charge of receiving the data frame from RX. This block should be implemented in the slave device. The internal block diagram is presented in Figure 13. This block has 2 inputs: the SPI generated clock `SPI_clk` form master, and the reception frame `RX`; also has two outputs: the received data `data` and a buffer `buffer_full` that tell other blocks that there is data ready to be read.

The entity definition is as follows:

```
entity spi_block_rx is
    port (  clk: in std_logic;
            reset: in std_logic;
            clk_spi: in std_logic;
            rx: in std_logic;
            data: out std_logic_vector (7 downto 0);
            full_buf: out std_logic
        );
end entity spi_block_rx;
```

As it can be observed in Figure 13, our design includes three blocks. A delay or synchronizer that is used to avoid reading data during ramps in the `SPI_clk`; rising edge detector, used to detect rising edges of our `SPI_clk`; and the SPI reception component which is in charge of receiving the data frame.

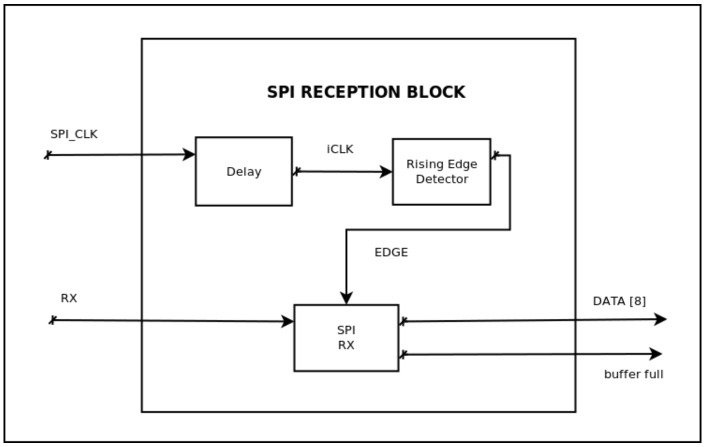

Figure 13: SPI RX - block diagram

A pseudocode of the main functionality of the SPI receiving block is as follows:

```
-- SPI reception block Pseudocode
while ( rising_edge_detected )
    buffer_full <= '0'
    data <= fill_with(rx_BIT)

buffer_full <= '1'
inMemory(data)
```

The testbench results of the SPI reception block can be observed in Figure 14.

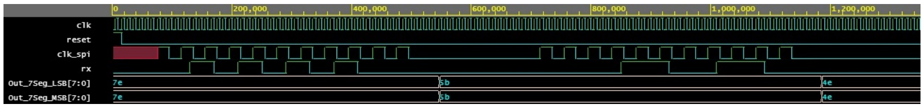

Figure 14: Testbench results - SPI Reception block

5 Test Case

We tested our SPI-based protocol using to FPGAs. The goal was to send a telegram from one FPGA to another using serial communication. Both FPGAs should be able to send a telegram, in other words, each FPGA was master and slave at the same time; thus, we used two lines of communication (Figure 15), from the FPGA number 1 to FPGA number 2 (comm_line_1to2) and vice versa (comm_line_2to1). It is important to mention that the lines of data transmission shown in Figure 1 were considered as **one line of communication**, for instance the `SPI_clk` and `RX-TX` lines are considered as one cummunication line.

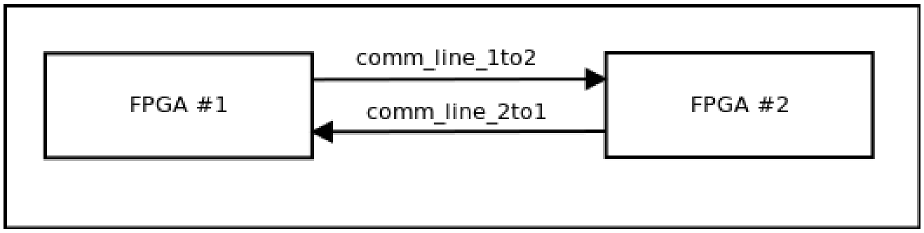

Figure 15: Bidirectional SPI-based communication

The telegram was 8-bit long and this information should be displayed in two 7-segment screen located in the other FPGA.

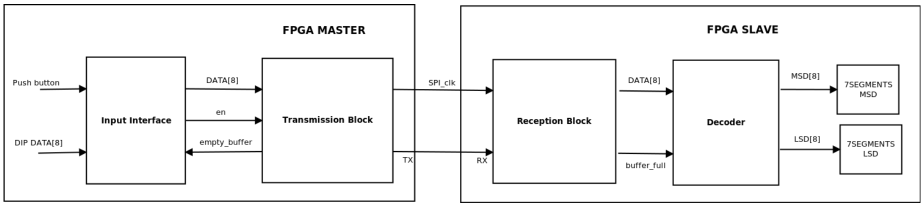

Figure 16: One-way communication - Block Diagram

In Figure 16 can be observed the only communication between master and slave. Note the inputs were a push button and a dip switch. **Input interface** block was in charge of manage the telegram, when the button was pressed the telegram given by the dip switch was sent through the transmission port (`TX`) if the transmission buffer were empty (`empty_buffer == 1`).

And after reception the **decoder block** used the data collected by **reception block** to decode the telegram in two data arrays (LSD - Least significant digit and MSD - Most significant digit), and both numbers were displayed in two 7-Segment displays.

We already discussed how the **reception block** and **transmission block** work. Now we will focus on the **Input interface** and **decoder**. The input interface block send the data once the user release the push button if the transmission buffer is free, if the transmission buffer is not free no data is sent. The decoder block read the data from reception block once reception buffer is full. Then we use a look up table to get the LSD and MSD in 7-Segment format.

VHDL code

Daniel Paredes, Cristan Gil

Bidirectional communication

```
-- Bidirectional block

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity bidirect_comm is
    port (
            clk: in std_logic;
            reset: in std_logic;

            -- Transmission ports
            push_button: in std_logic;
            dip_switch: in std_logic_vector(7 downto 0);
            clk_spi_out: out std_logic;
            tx: out std_logic;

            -- Reception ports
            clk_spi_in: in std_logic;
            rx: in std_logic;
            Out_7Seg_LSB: out std_logic_vector(7 downto 0);
            Out_7Seg_MSB: out std_logic_vector(7 downto 0)
        );
end entity bidirect_comm;

architecture arch of bidirect_comm is

    component Transmission_block is
        port(
            clk:    in  std_logic;
            reset:  in  std_logic;
            push_button:in  std_logic;
            dip_switch: in  std_logic_vector(7 downto 0);
            clk_spi: out std_logic;
            tx: out std_logic
        );
    end component Transmission_block;

    component Reception_block is
        port (
            clk: in std_logic;
            reset: in std_logic;
            clk_spi: in std_logic;
            rx: in std_logic;
            Out_7Seg_LSB: out std_logic_vector(7 downto 0);
            Out_7Seg_MSB: out std_logic_vector(7 downto 0)
        );
    end component Reception_block;

    signal iclk_spi: std_logic;
    signal iPIN: std_logic;

begin
    TX1: Transmission_block port map(
            clk  => clk,
            reset => reset,
            push_button => push_button,
            dip_switch => dip_switch,
            clk_spi => clk_spi_out,
            tx => tx );

    RX1: Reception_block port map(
            clk => clk,
            reset => reset,
            clk_spi => clk_spi_in,
            rx => rx,
            Out_7Seg_LSB => Out_7Seg_LSB,
            Out_7Seg_MSB => Out_7Seg_MSB );

end architecture arch;
```

Transmission_block

```
-- Transmission block

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity Transmission_block is
    port(
            clk:    in  std_logic;
            reset:  in  std_logic;
            push_button:in  std_logic;
            dip_switch: in  std_logic_vector(7 downto 0);
            clk_spi: out std_logic;
            tx: out std_logic
        );
end entity Transmission_block;

architecture arch of Transmission_block is

    component spi_block_tx is
        port (
            clk: in std_logic;
            reset: in std_logic;
            data: in std_logic_vector (7 downto 0);
            en: in std_logic;
            empty_buf: out std_logic;
            clk_spi: out std_logic;
            tx: out std_logic
        );
    end component spi_block_tx;

    component Interf_Inputs  is
        port (
            clk:    in  std_logic;
            reset:  in  std_logic;
            dip_switch: in std_logic_vector(7 downto 0);
            push_button:    in std_logic;
            empty_buf:  in  std_logic;
            en: out std_logic;
            data_out:   out std_logic_vector(7 downto 0)
        );

    end component Interf_Inputs;
    -- Internal signals

    signal iempty_buf: std_logic;
    signal ien: std_logic;
    signal idata: std_logic_vector(7 downto 0);

begin

    INPUTS_READ: Interf_Inputs port map(
            clk => clk,
            reset => reset,
            dip_switch => dip_switch,
            push_button => push_button,
            empty_buf => iempty_buf,
            en => ien,
            data_out => idata );

    SPI_TRANSMISSION: spi_block_tx port map(
            clk => clk,
            reset =>  reset,
            data => idata,
            en => ien,
            empty_buf => iempty_buf,
            clk_spi => clk_spi,
            tx =>  tx );

end architecture arch;
```

Interface Inputs

```
library ieee;
use ieee.std_logic_1164.all;

-- When SPI TX block is ready, a signal informs it to take the updated output
entity Interf_Inputs is
    port (
            clk:    in  std_logic;
            reset:  in  std_logic;
            dip_switch: in std_logic_vector(7 downto 0);
            empty_buf:  in  std_logic;
            push_button:    in std_logic;
            en: out std_logic;
            data_out:   out std_logic_vector(7 downto 0)
    );
end entity Interf_Inputs;

architecture Data_Sending of Interf_Inputs is
    type STATES is (IDLE, PRESS_BUTTON, CHECK_BUFF, WAIT_BUFFER, SEND);
    -- Current state (reg) and the next one (next) --> Then "next" is updated earler
    signal state_next, state_reg: STATES;
    signal data_next, data_reg: std_logic_vector(7 downto 0);
    signal en_next, en_reg: std_logic ;

begin
    -- FSMD: state and data registers
    process (clk, reset)
    begin
        -- To ensure the system will work, the reset button must be pushed at first,
        -- after configuring the HW
        if (reset = '1') then
            state_reg <= IDLE;
            en_reg <= '0';
            data_reg <= "00000000";
        elsif (rising_edge(clk)) then
            state_reg <= state_next;
            en_reg <= en_next;
            data_reg <= data_next;
        end if;
    end process;

    -- control flow and data path
    process (state_reg,  data_reg, empty_buf, push_button,
            en_reg, dip_switch)
    begin
        state_next <= state_reg;
        en_next <= en_reg;
        data_next <= data_reg;

        case state_reg is
            when IDLE =>
                en_next <= '0';
                -- Button has been pressed
                if (push_button = '1') then
                        state_next <= PRESS_BUTTON;
                end if;

            when PRESS_BUTTON =>
                en_next <= '0';
                -- wait until the button is released
                if (push_button = '0')  then
                    state_next <= CHECK_BUFF;
                end if;

            -- the flag that says whether the transmission is empty or not
            -- is delayed so we have to wait
            -- Note that the data in the dip_switch have been already move to the
            -- transmission buffer
            when CHECK_BUFF =>
```

```
                if (empty_buf = '1') then
                        state_next <= WAIT_BUFFER;
                        data_next <= dip_switch;
                        en_next <= '1';
                else
                        en_next <= '0';
                        state_next <= IDLE;
                end if;

            when WAIT_BUFFER =>
                en_next <= '1';
                if  (empty_buf = '0') then
                    state_next <= SEND;
                end if;
            when SEND =>
                en_next <= '1';
                data_next <= data_reg;
                if (empty_buf = '1') then
                        state_next <= IDLE;
                        en_next <= '0';
                end if;
            end case;
    end process;

    data_out <= data_reg;
    en <= en_reg;

end architecture Data_Sending;
```

SPI Block tx

```
-- spi block tx

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity spi_block_tx is
    port (  clk: in std_logic;
            reset: in std_logic;
            data: in std_logic_vector (7 downto 0);
            en: in std_logic;
            empty_buf: out std_logic;
            clk_spi: out std_logic;
            tx: out std_logic
        );
end entity spi_block_tx;

architecture arch of spi_block_tx is

component clk_generator
    port (
            clk : in std_logic;
        reset : in std_logic;
        en : in std_logic;
        clk_out : out std_logic
    );
end component;

component spi_tx
    port (
            clk: in std_logic;
            reset: in std_logic;
            edge: in std_logic;
            data: in std_logic_vector (7 downto 0);
            empty_buf: out std_logic;
            tx: out std_logic
         );
end component;

component  delay_sync
    port (
            clk: in std_logic;
            reset: in std_logic;
            d: in std_logic;
            q: out std_logic
         );
end component;

component rising_edge_detector
    port (
            clk: in std_logic;
            reset: in std_logic;
            clk_in: in std_logic;
            edge: out std_logic
         );
end component;

-- Internal signals
signal iclk_spi: std_logic;
signal iclk_delayed: std_logic;
signal irising_edge: std_logic;

begin

    SPI_CLK_GEN: clk_generator port map (
        clk => clk,
        reset => reset,
        en => en,
        clk_out => iclk_spi);

    DELAY_1: delay_sync port map(
        clk => clk,
        reset => reset,
        d => iclk_spi,
        q => iclk_delayed);

    RISING_EDGE_TX: rising_edge_detector port map(
        clk => clk,
        reset => reset,
        clk_in => iclk_delayed,
        edge => irising_edge);
    SPI_TX_Blk: spi_tx port map(
        clk => clk,
        reset => reset,
        edge => irising_edge,
        data => data,
        empty_buf => empty_buf,
        tx => tx);

    clk_spi <= iclk_delayed;
end architecture arch;
```

Clock Generator

```
-- clk generator for the SPI communication
-- clk by default 0

library ieee;
use ieee.std_logic_1164.all;
--use ieee.numeric_std.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity clk_generator is
    port (  clk: in std_logic;
            reset: in std_logic;
            en: in std_logic;
            clk_out: out std_logic
         );
end entity clk_generator;

architecture arch of clk_generator is
    type STATES is (ONE, ZERO);
    signal state_next, state_reg: STATES;
    signal clk_next, clk_reg: std_logic;
    signal count_next, count_reg: std_logic_vector (3 downto 0);
    signal MAX_COUNT: std_logic_vector(3 downto 0) := "1010"; --10
    --constant MAX_COUNT: integer := 10;
begin

    -- FSMD: state and data registers
    process(clk, reset)
    begin
        if (reset = '1') then
            state_reg <= ONE;
            clk_reg <= '0';
            count_reg <= (others => '0');
        elsif ( rising_edge(clk)) then
            state_reg <= state_next;
            clk_reg <= clk_next;
            count_reg <= count_next;
        end if;
    end process;

    -- Control logic and data path
    process(state_reg, count_reg, en)
    begin
        state_next <= state_reg;
        count_next <= count_reg;
        if (en = '1') then
            case state_reg is
                when ONE =>
                    -- Count 10 cycles
                    clk_next <= '1';
                    if ( count_reg = (MAX_COUNT-1)  ) then
                        count_next <= (others => '0');
                        state_next <= ZERO;
                    else
                        count_next <= (count_reg + 1);
                        state_next <= ONE;
                    end if;
                when ZERO =>
                    clk_next <= '0';
                    -- Count 10 cycles
                    if ( count_reg = (MAX_COUNT-1)   ) then
                        count_next <= (others => '0');
                        state_next <= ONE;
                    else
                        count_next <= (count_reg + 1);
                        state_next <= ZERO;
                    end if;
           end case;
        else
            clk_next <= '0';
            count_next <= (others => '0');
            state_next <= ONE;
        end if;
    end process;
   clk_out <= clk_reg;
end architecture arch;
```

Delay

```
-- Delay based on  Flip flops type D
-- One clock delay
-- This block will delay the spi_clk
-- can be seen as synchronization block

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity  delay_sync is
    port (  clk: in std_logic;
            reset: in std_logic;
            d: in std_logic;
            q: out std_logic
         );
end entity delay_sync;

architecture arch of delay_sync is
```

```
    signal q1_next, q1_reg: std_logic;
    signal q2_next, q2_reg: std_logic;
begin
    -- FSMD: state and data registers
    process (clk, reset)
    begin
        if (reset  = '1') then
            q1_reg <= '0';
            q2_reg <= '0';
        elsif ( rising_edge(clk)) then
            q1_reg <= q1_next;
            q2_reg <= q2_next;
        end if;
    end process;

-- Control logic and data path
    process (d, q1_reg)
    begin
        q1_next <= d;
        q2_next <= q1_reg;
    end process;

    q <= q2_reg;
end architecture arch;
```

Rising Edge detector

```
-- rising edge detector
-- It will be used to detect the rising edge of the SPI clk

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rising_edge_detector is
    port (  clk: in std_logic;
            reset: in std_logic;
            clk_in: in std_logic;
            edge: out std_logic
         );
end entity rising_edge_detector;

architecture arch of rising_edge_detector is
    type STATES is (ONE, ZERO, R_EDGE);
    signal state_next, state_reg: STATES;
    signal q_next, q_reg: std_logic; -- used to detect edges
begin

    -- FSMD: state and data registers
    process (clk, reset)
    begin
        if (reset = '1') then
            state_reg <= ZERO;
            q_reg <= '0';
        elsif (rising_edge(clk)) then
```

```
            state_reg <= state_next;
            q_reg <= q_next;
        end if;
    end process;

    -- control flow and data path
    process (state_reg, clk_in)
    begin
        state_next <= state_reg;
        q_next <= '0';
        case state_reg is
            when ZERO =>
                if ( clk_in ='1') then
                    state_next <= R_EDGE;
                end if;
            when ONE =>
                if ( clk_in = '0') then
                    state_next <= ZERO;
                end if;
            -- There was a change from 0 to 1
            when R_EDGE =>
                q_next <= '1';
                if ( clk_in = '0') then
                    state_next <= ZERO;
                else
                    state_next <= ONE;
                end if;
        end case;
    end process;

    edge <= q_reg;
end architecture arch;
```

SPI tx

```
-- SPI transmitter
-- LSB is sent first

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity spi_tx is
    port (  clk: in std_logic;
            reset: in std_logic;
            edge: in std_logic;
            data: in std_logic_vector (7 downto 0);
            empty_buf: out std_logic;
            tx: out std_logic
         );
end entity spi_tx;

architecture arch of spi_tx is
    type STATES is (IDLE, SEND, STOP);
    signal state_next, state_reg: STATES;
    signal buf_next, buf_reg: std_logic_vector (7 downto 0);
    signal count_next, count_reg: unsigned (3 downto 0);
    signal empty_next, empty_reg: std_logic;
    signal tx_next, tx_reg: std_logic;
    constant DBIT: integer := 8;
begin

    -- FSMD: state and data registers
    process ( clk, reset)
    begin
        if (reset = '1') then
            state_reg <= IDLE;
            buf_reg <= (others => '0');
            count_reg <= (others => '0');
            empty_reg <= '1';
            tx_reg <= '0';
    elsif ( rising_edge(clk) ) then
            state_reg <= state_next;
            buf_reg <= buf_next;
            count_reg <= count_next;
            empty_reg <= empty_next;
            tx_reg <= tx_next;
        end if;
    end process;

    -- Control logic and data path
    process (edge, state_reg, count_reg, buf_reg,
         tx_reg, data)
    begin
        state_next <= state_reg;
        count_next <= count_reg;
        buf_next <= buf_reg;
        empty_next <= '1';
        tx_next <= '0';

        case state_reg is
            -- wait for a detected rising edge clock signal
            when IDLE =>
                if (edge ='1') then
                    state_next <= SEND;
                    buf_next <= data;
                end if;
            -- send the 8 bits frame
            when SEND =>
                empty_next <= '0';
                if ( edge ='1') then
                    if ( count_reg = DBIT  ) then
                        count_next <= (others => '0');
                        state_next <= STOP;
            --            empty_next <= '1';
                    else
                        count_next <= (count_reg + 1);
                        -- LSB is sent first
                        -- so we need to rotate the buffer
                        buf_next <= '0' & buf_reg(7 downto 1);
                        tx_next <= buf_reg(0);
                    end if;
                else
                    tx_next <= tx_reg;
                end if;

            -- end of transmission
            when STOP =>
                empty_next <= '0';
                if ( edge = '1') then
                    empty_next <= '1';
                    state_next <= IDLE;
                end if;
        end case;
    end process;

    empty_buf <= empty_reg;
    tx <= tx_reg;
end architecture arch;
```

Reception_block

```
-- Reception block

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity Reception_block is
    port (  clk: in std_logic;
            reset: in std_logic;
            clk_spi: in std_logic;
            rx: in std_logic;
            Out_7Seg_LSB: out std_logic_vector(7 downto 0);
            Out_7Seg_MSB: out std_logic_vector(7 downto 0)
        );
end entity Reception_block;

architecture arch of Reception_block is
    component spi_block_rx is
        port (
            clk: in std_logic;
            reset: in std_logic;
            clk_spi: in std_logic;
            rx: in std_logic;
            data: out std_logic_vector (7 downto 0);
            full_buf: out std_logic
        );
    end component spi_block_rx;

    component decoder_block is
        port (
            clk:    in  std_logic;
            reset:  in  std_logic;
            full_buf:  in  std_logic;
            data_in:    in  std_logic_vector(7 downto 0);
            Out_7Seg_LSB: out std_logic_vector(7 downto 0);
            Out_7Seg_MSB: out std_logic_vector(7 downto 0)
        );
    end component decoder_block;

-- Internal signals
    signal idata : std_logic_vector(7 downto 0);
    signal ifull_buf: std_logic;

begin
    SPI_RECEPTION: spi_block_rx port map(
            clk => clk,
            reset  => reset,
            clk_spi => clk_spi,
            rx =>  rx,
            data => idata,
            full_buf => ifull_buf);

    DECODER:  decoder_block port map(
            clk => clk,
            reset =>  reset,
            full_buf =>  ifull_buf,
            data_in => idata,
            Out_7Seg_LSB => Out_7Seg_LSB,
            Out_7Seg_MSB => Out_7Seg_MSB  );

end architecture arch;
```

SPI BLOCK RX

```
-- spi block rx

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity spi_block_rx is
    port (  clk: in std_logic;
            reset: in std_logic;
            clk_spi: in std_logic;
            rx: in std_logic;
            data: out std_logic_vector (7 downto 0);
            full_buf: out std_logic
        );
end entity spi_block_rx;

architecture arch of spi_block_rx is

component spi_rx
    port (  clk: in std_logic;
            reset: in std_logic;
            edge: in std_logic;
            rx: in std_logic;
            data: out std_logic_vector (7 downto 0);
            full_buf: out std_logic
         );
end component;
```

```
component  delay_sync
    port (
            clk: in std_logic;
            reset: in std_logic;
            d: in std_logic;
            q: out std_logic
         );
end component;

component rising_edge_detector
    port (
            clk: in std_logic;
            reset: in std_logic;
            clk_in: in std_logic;
            edge: out std_logic
         );
end component;

-- Internal signals
signal iclk_delayed: std_logic;
signal irising_edge: std_logic;

begin

    SYNCHRONIZER: delay_sync port map(
        clk => clk,
        reset => reset,
        d => clk_spi,
        q => iclk_delayed);

    RISING_EDGE_TX: rising_edge_detector port map(
        clk => clk,
        reset => reset,
    clk_in => iclk_delayed,
    edge => irising_edge);

    SPI_RX_Blk: spi_rx port map(
        clk => clk,
    reset => reset,
    edge => irising_edge,
    rx => rx,
    data => data,
        full_buf => full_buf);

end architecture arch;
```

SPI RX

```
-- SPI receiver

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity spi_rx is
    port (  clk: in std_logic;
            reset: in std_logic;
            edge: in std_logic;
            rx: in std_logic;
            data: out std_logic_vector (7 downto 0);
            full_buf: out std_logic
         );
end entity spi_rx;

architecture arch of spi_rx is
    type STATES is (IDLE, RECEIVE, STOP);
    signal state_next, state_reg: STATES;
    signal buf_next, buf_reg: std_logic_vector (7 downto 0);
    signal count_next, count_reg: unsigned (3 downto 0);
    signal full_next, full_reg: std_logic;
    constant DBIT: integer := 8;
begin

    -- FSMD: state and data registers
    process (clk, reset)
    begin
        if (reset = '1') then
            state_reg <= IDLE;
            buf_reg <= (others => '0');
            count_reg <= (others => '0');
            full_reg <= '0';
    elsif ( rising_edge(clk) ) then
            state_reg <= state_next;
            buf_reg <= buf_next;
            count_reg <= count_next;
            full_reg <= full_next;
        end if;
    end process;

    -- Control logic and data path
    process (edge, state_reg, count_reg, buf_reg,
         full_reg)
    begin
        state_next <= state_reg;
        count_next <= count_reg;
        buf_next <= buf_reg; --(others => '0');
        full_next <= full_reg;

        case state_reg is
            -- wait until an rising edge of SPI clk is detected
            when IDLE =>
                if (edge ='1') then
                    state_next <= RECEIVE;
                end if;
                -- if there is not edge, it means the data
                -- can be read
                full_next <= '1';
            -- receive the 8 bit data frame
            when RECEIVE =>
                full_next <= '0';
                if ( edge ='1') then
                    if ( count_reg = DBIT  ) then
                        count_next <= (others => '0');
                        state_next <= STOP;
                        -- full_next <= '1';
                    else
                        count_next <= (count_reg + 1);
                        -- LSB is received first
                        -- so we need to rotate the buffer
                        buf_next <= rx & buf_reg(7 downto 1);
                    end if;
                else
                    buf_next <= buf_reg;
                end if;
            -- all the date have been received
            when STOP =>
                full_next <= '0';
                if ( edge = '1') then
                    full_next <= '1';
                    state_next <= IDLE;
                end if;
        end case;
    end process;

    full_buf <= full_reg;
    data  <= buf_reg;
end architecture arch;
```

Decoder

```
-- Block of Interf_7Seg, Control_7Seg and Bin_To_BCD
library ieee;
use ieee.std_logic_1164.all;

entity decoder_block is
    port (
            clk:    in  std_logic;
            reset:  in  std_logic;
            full_buf : in  std_logic;
            data_in: in  std_logic_vector(7 downto 0);
            Out_7Seg_LSB: out std_logic_vector(7 downto 0);
            Out_7Seg_MSB: out std_logic_vector(7 downto 0)
    );
end entity decoder_block;

architecture arch of decoder_block is
    component Interf_7Seg is
            port (
                clk: in  std_logic;
                reset: in std_logic;
                buffer_full:in std_logic;
                data_in: in  std_logic_vector(7 downto 0);
                data_out: out std_logic_vector(7 downto 0)
            );
    end component Interf_7Seg;
    component decoder7seg is
        port (
                encoded_input:  in  std_logic_vector(7 downto 0);
                Out_7Seg_LSB: out std_logic_vector(7 downto 0);
                Out_7Seg_MSB: out std_logic_vector(7 downto 0)
            );
    end component decoder7seg;

    signal iEncodedData: std_logic_vector(7 downto 0);

begin
        g7: Interf_7Seg port map (clk, reset, full_buf, data_in, iEncodedData);
        g8: decoder7seg port map ( iEncodedData, Out_7Seg_LSB, Out_7Seg_MSB);

end architecture arch;
```

Interface 7 Segments

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- DATA is taken from the SPI RX block
entity Interf_7Seg is
    port (
        clk:        in  std_logic;
        reset:      in  std_logic;
        buffer_full:in  std_logic;
        data_in:    in  std_logic_vector(7 downto 0);
        data_out:   out std_logic_vector(7 downto 0)
    );
end entity Interf_7Seg ;

architecture arch of Interf_7Seg  is
    type STATES is (READING, UPDATE);
    signal state_next, state_reg: STATES;
    signal data_next, data_reg: std_logic_vector(7 downto 0);

begin
    -- FSMD: state and data registers
    process (clk, reset)
    begin
        if (reset = '1') then
            state_reg <= READING;
            data_reg <= "00000000";
        elsif (rising_edge(clk)) then
            state_reg <= state_next;
            data_reg <= data_next;
        end if;
    end process;

    -- control flow and data path
    process (state_reg, buffer_full, data_in, data_reg)
    begin
```

```
        state_next <= state_reg;
        data_next <= "00000000";

        case state_reg is
            -- wait until a new data arrives
            when READING =>
                data_next <= data_reg;
                if (buffer_full = '1') then
                        state_next <= UPDATE;
                end if;
            -- update data
            when UPDATE =>
                data_next <= data_in;
                state_next <= READING;
        end case;
    end process;

    data_out <= data_reg;
end architecture arch;
```

Decoder 7 Segments

```
-- It separates the input of 8 bits in a 2 independent BCD numbers
-- each for 7 segment display
library ieee;
use ieee.std_logic_1164.all;

entity decoder7seg is
  port (
    encoded_input:    in  std_logic_vector(7 downto 0);
    Out_7Seg_LSB: out std_logic_vector(7 downto 0);
    Out_7Seg_MSB: out std_logic_vector(7 downto 0)
  );
end entity decoder7seg;

architecture arch of decoder7seg is
    component bcd_7seg
        port (
            data_in:    in  std_logic_vector(3 downto 0);
            data_out:   out std_logic_vector(7 downto 0)
        );
    end component bcd_7seg;

    signal in_LSB, in_MSB: std_logic_vector(3 downto 0);
    signal out_LSB, out_MSB: std_logic_vector(7 downto 0);

    begin
      in_LSB <= encoded_input(3 downto 0);
      in_MSB <= encoded_input(7 downto 4);

      g5: bcd_7seg port map (in_LSB, out_LSB);
      g6: bcd_7seg port map (in_MSB, out_MSB);

      Out_7Seg_LSB <= out_LSB;
      Out_7Seg_MSB <= out_MSB;
```

```
end architecture arch;
```

bcd to 7 segments

```
-- Entity that transforms a 4-array into a BCD number to 7 segment display
library ieee;
use ieee.std_logic_1164.all;

entity bcd_7seg is
    port (
        data_in:    in  std_logic_vector(3 downto 0);
        data_out:   out std_logic_vector(7 downto 0)
  );
end entity bcd_7seg;

architecture arch of bcd_7seg is
begin
    process (data_in) is
    begin
        case data_in is
            when "0000" => data_out <= "01111110";  -- 0x7E (Number 0)
            when "0001" => data_out <= "00110000";  -- 0x30 (Number 1)
            when "0010" => data_out <= "01101101";  -- 0x6D (Number 2)
            when "0011" => data_out <= "01111001";  -- 0x79 (Number 3)
            when "0100" => data_out <= "00110011";  -- 0x33 (Number 4)
            when "0101" => data_out <= "01011011";  -- 0x5B (Number 5)
            when "0110" => data_out <= "01011111";  -- 0x5F (Number 6)
            when "0111" => data_out <= "01110000";  -- 0x70 (Number 7)
            when "1000" => data_out <= "01111111";  -- 0x7F (Number 8)
            when "1001" => data_out <= "01111011";  -- 0x7B (Number 9)
            when "1010" => data_out <= "01110111";  -- 0x77 (Number A)
            when "1011" => data_out <= "00011111";  -- 0x1F (Number B)
            when "1100" => data_out <= "01001110";  -- 0x4E (Number C)
            when "1101" => data_out <= "00111101";  -- 0x3D (Number D)
            when "1110" => data_out <= "01001111";  -- 0x4F (Number E)
            when others => data_out <= "01000111";  -- 0x47 (Number F)
        end case;
    end process;

end architecture arch;
```