Serial Peripheral Interface report

Made by
Job Tijhuis, 4275756
Wietse Bouwmeester, 4300807

26 November 2014

Abstract

This is the documentation about the SPI interface used for communicating with peripherals such as memory cards and sensors.

Contents

1 Introduction

One of the most used protocols to interface with peripherals is the Serial Peripheral Interface or SPI for short. For example, most external memory modules, which will be needed because the final chip is not large enough to fit a reasonable memory bank, use SPI for interfacing. Thus a SPI interface is key for a working final product.

2 Specification

The following specifications were determined:

- The interface should work according to the SPI de facto-standard.
- The SPI interface should have a baud-rate as high as possible.
- It needs to be possible to write to and read from the shift register located in the interface.
- The interface needs to have a enable line to indicate to start sending the shift register contents to the other receiving interface.
- The slave clock needs to be low while no data is being sent.

3 Design

In the specifications it is given that communication is to go according to the SPI de facto-standard. SPI does not define what the communication will be, but it does define the way the data is transferred. The SPI standard defines two communication lines: one from the master to the slave and one from the slave to the master. These lines are connected to shift registers where the outgoing line is connected to the most significant bit and the incoming to the least significant bit. The third line is a clock signal which tells the slave when to sample the line and when to shift its shift register. Since the shift register is such an integral part of SPI it was decided to separate the design in two parts: the shift register and a control system based on a Finite State Machine which controls when the register shifts and when the slave clock will be on. Because it is also important that the transmission stops after all eight bits have been transferred there is also a counter which counts the slave clock, this information is then used by the control.

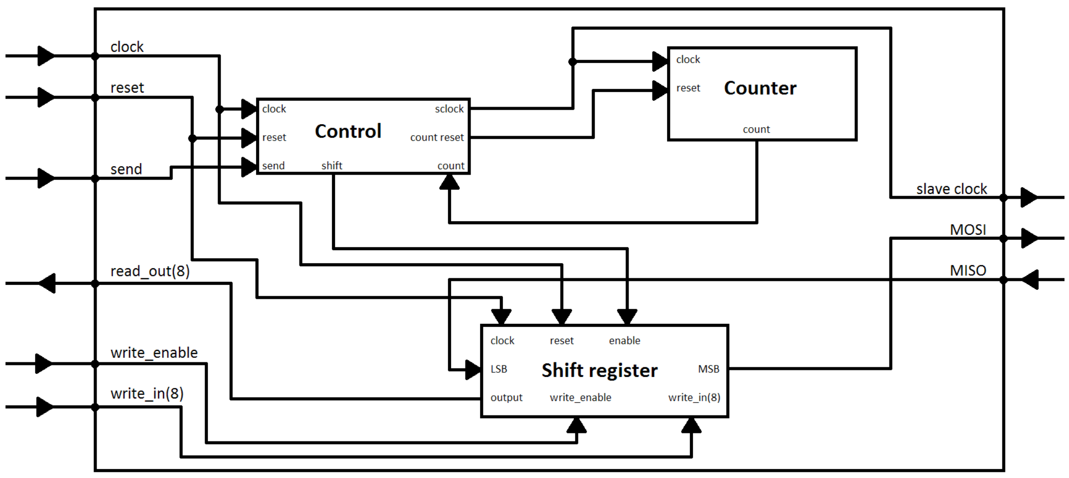

Figure 1: Diagram of SPI circuit

3.1 Control

Control is a FSM and has three states: the reset state in which it will be once it is reset, the idle state in which it is when not shifting and to which it will return once all eight bits have been shifted and the shifting state in which it will actually shift the bits out/in. The VHDL implementation is the implementation of the finite state diagram in figure **??**. The output in the different states is as in figure **??**.

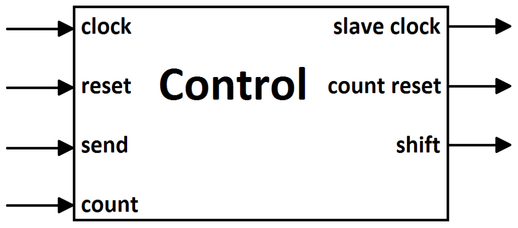

Figure 2: Diagram of Control

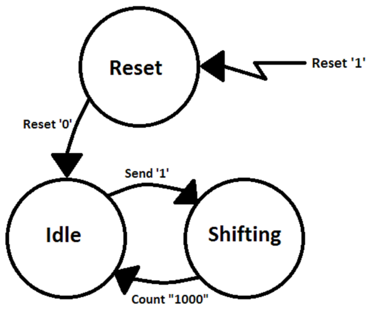

Figure 3: State diagram of Control FSM

State	Slave clock	Count reset	Shift
Reset	0	1	0
Idle	0	1	0
Shifting	not clk	0	1

Figure 4: Control FSM states

3.2 Shift register

The way the shift register works is as followed: it will reset and do not do anything else if the reset is high, it will synchronously read in 8 new bits if the write in enable is high and it will shift on the clock if enable is high and there is no new bits being loaded in. The input and outputs are as in figure **??**. The VHDL description can be found in **??**.

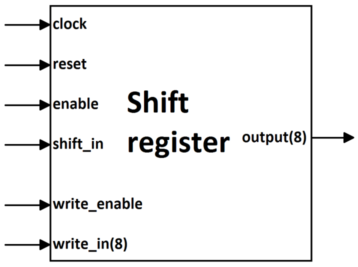

Figure 5: Diagram of shift register

3.3 Counter

Counter is a simple counter that goes up by one for every rising edge of the clock. It is a 4-bit counter which has a clock and reset as inputs and an output count as output. The VHDL description can be found in **??**

4 Results

The results are a SPI that works and correctly shifts out all bits and stops afterwards. The specifications that you can load in values to the shift register and read out of the shift register are also fulfilled. The circuit has been simulated on both simple logic level as wel on switch level, the results show that the switch level simulation does not differ from the logic simulation. These simulations can be seen in figure **??** and **??**. The actual layout on the chip can be seen in figure **??**.

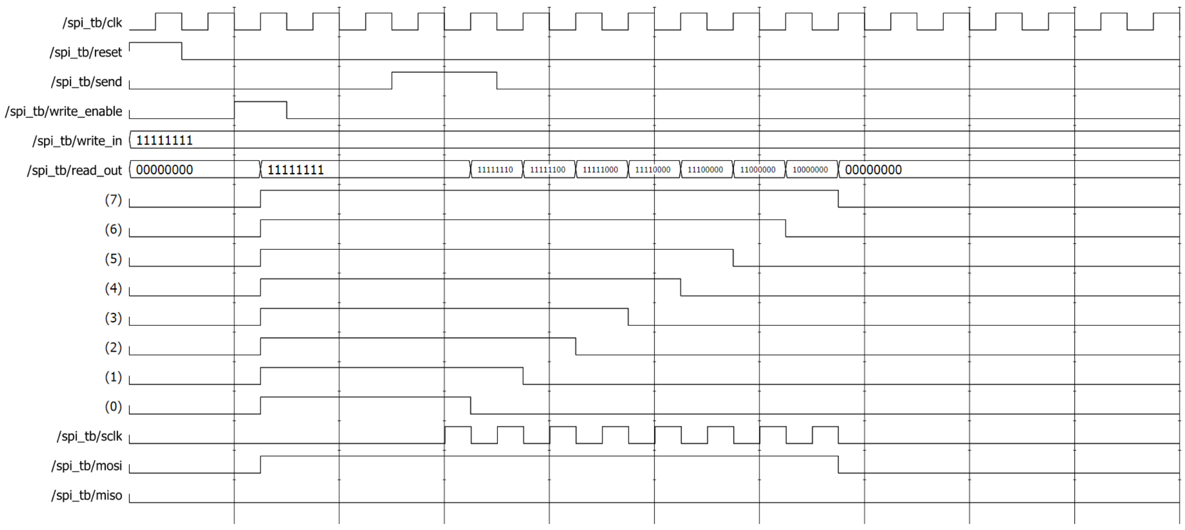

Figure 6: Modelsim simulation of SPI

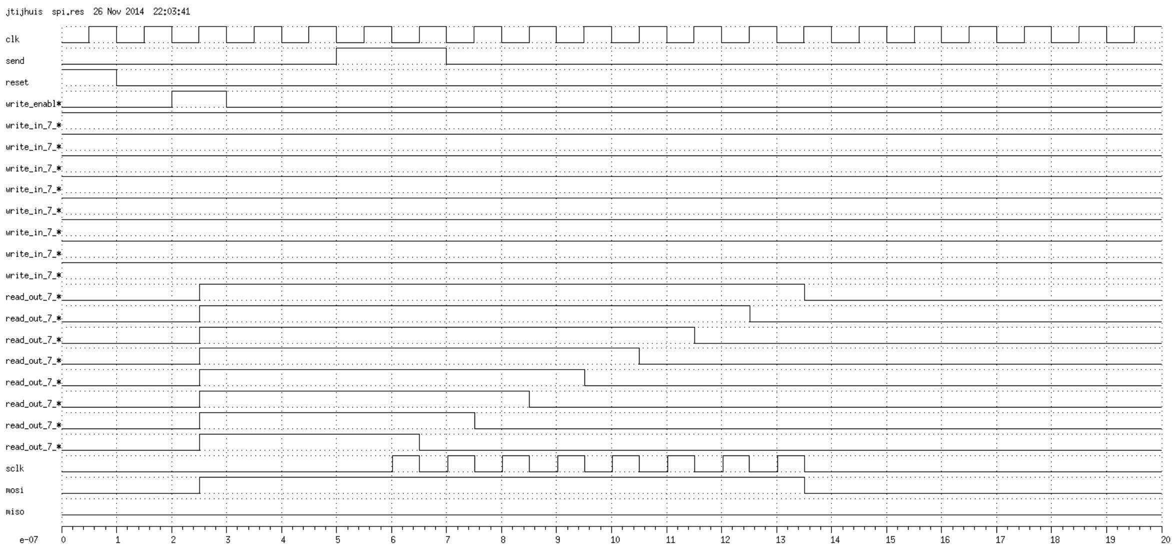

Figure 7: Switch level simulation of SPI

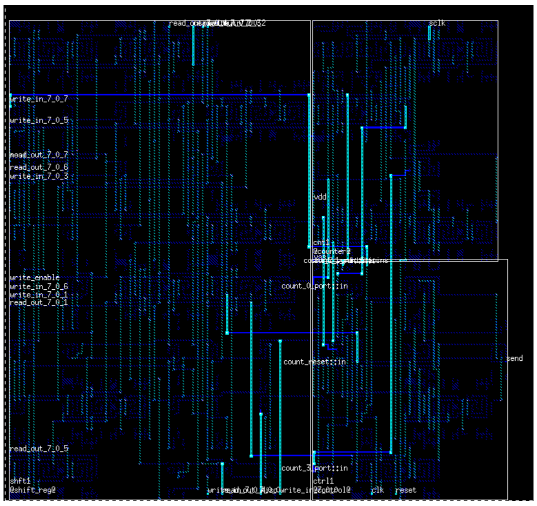

Figure 8: Layout of SPI on the chip

5 Conclusions

The SPI interface works according to the SPI de facto-standard. It also shifts transfers bits with the global system clock frequency, so the baud rate is as high as possible. It is possible to write data to and read data from the interface. The interface has an enable line which will indicate when the interface starts transferring data and the slave clock is low when no data is being transferred. Therefore can be concluded that the SPI interface meets the specifications. To improve the interface, slave select lines could be added. An other feature that could be added is a selectable slave clock speed since some peripherals might not support slave clock speeds up to a few MHz. The SLS and modelsim simulations of the interface are the same thus can be concluded that a working SPI interface has been designed.

References

[1] Serial Peripheral Interface Bus, http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus, accessed in November and October. 2014.

6 VHDL code

6.1 Control

```
library IEEE;
use IEEE.std_logic_1164.all;

entity control is
port(   clk:                    in std_logic;
                reset:                  in std_logic;
                send:                   in std_logic;
                count:                  in std_logic_vector(3 downto 0);
                shift:                  out std_logic;
                sclk:                   out std_logic;
                c_reset:                out std_logic
        );
end entity control;

architecture behavioural of control is
        type control_state is (reset_state,idle,shifting);
        signal state : control_state;
        signal clk_switch : std_logic;
begin
        process(clk,reset)
                begin
                if(reset = '1') then
                        state <= reset_state;
                else
                        if rising_edge(clk)  then
                                case state is
                                        when reset_state =>
                                                if(reset = '0') then
                                                        state <= idle;
                                                else
                                                        state <= reset_state;
                                                end if;
                                        when idle =>
                                                if(send = '1') then
                                                        state <= shifting;
                                                else
                                                        state <= idle;
                                                end if;
                                        when shifting =>
                                                if(count="1000") then
                                                        state <= idle;
                                                else
                                                        state <= shifting;
                                                end if;
                                end case;
                        end if;
                end if;
        end process;

        process(state)
        begin
                if(state=reset_state) then
                        c_reset <= '1';
                        clk_switch <= '0';
                elsif(state=idle) then
```

```
                        c_reset <= '1';
                        clk_switch <= '0';
                elsif(state=shifting) then
                        c_reset <= '0';
                        clk_switch <= '1';
                else
                        c_reset <= '1';
                        clk_switch <= '0';
                end if;
        end process;

        sclk <=         (not(clk and clk_switch)) after 1 ns
                                when (state=shifting) else '0';

        shift <= clk_switch;
end behavioural;
```

6.2 Counter

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

entity counter is
        port (  clk             : in    std_logic;
                reset           : in    std_logic;
                count_out       : out   std_logic_vector (3 downto 0)
        );
end entity counter;

architecture behavioural of counter is

        signal  count   : unsigned (3 downto 0);

begin
        -- Dit process genereert het register
        process (clk, reset)
        begin
        if (reset = '1') then
                count   <= (others => '0');     -- zet op 0 bij reset
        else
                if (rising_edge (clk)) then
                        count   <= count + 1;
                end if;
        end if;
        end process;
        -- Dit process berekent de nieuwe count-waarde
--      process (count)
--      begin
--                      new_count       <= count + 1;
--      end process;
        count_out       <= std_logic_vector (count);

end behavioural;
```

6.3 Shift register

```
library IEEE;
use IEEE.std_logic_1164.all;

entity shift_reg is
port(   clk:                    in std_logic;
                reset:                  in std_logic;
                enable:                 in std_logic;
                shift_in:               in std_logic;
                reg_set:                in std_logic;
                reg_write:              in std_logic_vector (7 downto 0);
                output:                 out std_logic_vector (7 downto 0)
        );
end shift_reg;

architecture behavioral of shift_reg is

        signal reg_shift,shifted_reg: std_logic_vector (7 downto 0);

begin

        shifted_reg(7 downto 1) <= reg_shift(6 downto 0);
        shifted_reg(0) <= shift_in;

        process(clk, reset)
        begin
        if(reset = '1') then
                reg_shift <= (others => '0');
        else
                if(rising_edge(clk)) then
                        if(reg_set = '1') then
                                reg_shift <= reg_write;
                        else
                                if(enable = '1') then
                                        reg_shift <= shifted_reg;
                                else
                                        reg_shift <= reg_shift;
                                end if;
                        end if;
                end if;
        end if;
        end process;

        output <= reg_shift;

end behavioral;
```

6.4 SPI

```
library IEEE;
use IEEE.std_logic_1164.all;

entity spi is
        port (  clk             : in    std_logic;
                send            : in    std_logic;
                reset           : in    std_logic;
                write_enable    : in    std_logic;
                write_in        : in    std_logic_vector (7 downto 0);
                read_out        : out   std_logic_vector (7 downto 0);
                sclk            : out   std_logic;
                mosi            : out   std_logic;
                miso            : in    std_logic
        );
end entity spi;

architecture structural of spi is

component counter is
        port (  clk             : in    std_logic;
                reset           : in    std_logic;
                count_out       : out   std_logic_vector (3 downto 0)
        );
end component counter;

component shift_reg is
port(   clk:                    in std_logic;
        reset:                  in std_logic;
        enable:                 in std_logic;
        shift_in:               in std_logic;
        reg_set:                in std_logic;
        reg_write:              in std_logic_vector (7 downto 0);
        output:                 out std_logic_vector (7 downto 0)
        );
end component shift_reg;
```

```
component control is
port(   clk:                 in std_logic;
            reset:           in std_logic;
            send:            in std_logic;
            count:           in std_logic_vector(3 downto 0);
            shift:           out std_logic;
            sclk:            out std_logic;
            c_reset:         out std_logic
        );
end component control;

signal count : std_logic_vector(3 downto 0);
signal output : std_logic_vector(7 downto 0);
signal count_reset,switch_clk : std_logic;
signal shift_reset,shift : std_logic;

begin
        sclk <= switch_clk;
        read_out <= output;
        mosi <= output(7);
cnt1:  counter port map (switch_clk,count_reset,count);
shft1: shift_reg port map (clk,reset,shift,miso,write_enable,write_in,output);
ctrl1: control port map (clk,reset,send,count,shift,switch_clk,count_reset);

end structural;
```

6.5 Shift register

```
library IEEE;
use IEEE.std_logic_1164.all;

entity spi_tb is
end spi_tb;

architecture structural of spi_tb is

component spi is
        port (  clk            : in    std_logic;
                send           : in    std_logic;
                reset          : in    std_logic;
                write_enable   : in    std_logic;
                write_in       : in    std_logic_vector (7 downto 0);
                read_out       : out   std_logic_vector (7 downto 0);
                sclk           : out   std_logic;
                mosi           : out   std_logic;
                miso           : in    std_logic
        );
end component spi;

signal clk: std_logic := '0';
signal reset: std_logic;
signal send: std_logic;
signal write_enable : std_logic;
signal write_in: std_logic_vector(7 downto 0);
signal read_out: std_logic_vector(7 downto 0);
signal sclk: std_logic;
signal mosi: std_logic;
signal miso: std_logic;
begin
spi1: spi port map (clk, send, reset, write_enable,write_in,read_out,sclk,
                                             mosi,miso);

clk <= not clk after 50 ns;

reset <= '1' after 0 ns,
         '0' after 100 ns;

write_enable <= '0' after 0 ns,
                            '1' after 200 ns,
                            '0' after 300 ns;

write_in <= "11111111";

miso <= '0';

send <= '0' after 0 ns,
                '1' after 500 ns,
                '0' after 700 ns,
                '1' after 2600 ns,
                '0' after 2700 ns;

end structural;
```