Университет ИТМО
Кафедра Вычислительной техники

Системы ввода/вывода и периферийные устройства

Лабораторная работа 1

Разработка контроллера ввода/вывода

Интерфейс SPI

Работу выполнили студенты группы P3415
Фомин Евгений
Халанский Дмитрий

2016

Содержание

1 Задание

Для интерфейса SPI разработать модели контроллера ввода-вывода на уровне регистровых передач с использованием языка Verilog и в виде поведенческой модели с использование языка SystemC. Для каждой из моделей разработать тестовое окружение и провести тестирование обеих моделей.

2 Краткая характеристика интерфейса

SPI(Serial Peripheral Interface) — последовательный синхронный стандарт передачи данных в дуплексном режиме. В SPI используются четыре цифровых сигнала:

1. MOSI (Master Out Slave In) — линия передачи данных от ведущего устройства к ведомому
2. MISO (Master In Slave Out) — вход ведущего, выход ведомого
3. SCLK — тактовый сигнал для ведомых устройств
4. SS — выбор ведомого устройства

Передача осуществляется пакетами длиной, как правило, 8 бит. Ведущее устрйоство инициирует цикл связи установкой низкого уровня на выходе `SS` для выбора устройства, с которым необходимо установить соединение. При низком уровне сигнала `SS` ведомое устройство по тактовому сигналу `SCLK` считывает значение на входе `MOSI`, выставляя значение к отправке на выход `MISO`. Побитовый обмен происходит с

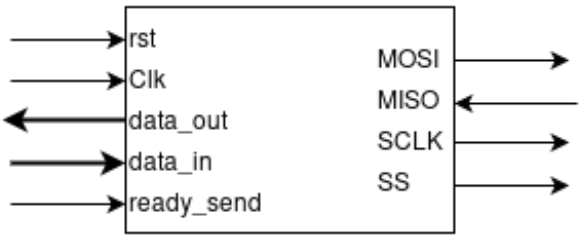
синхросигналом SCLK, пока не будет передан последний бит пакета. Затем на линии SS устанавливается высокий уровень.

3 Разработанный контроллер

3.1 Описание функций и особенностей разработанного контроллера

Разработанный контроллер SPI поддерживает работу только в режиме *Master* с произвольным интерфейсом доступа к процессорному ядру. Контроллер работает в режиме *SPI mode 0*: сигнал синхронизации начинается с низкого уровня и выборка данных производится по переднему фронту. Частота синхросигнала, по которому работает контроллер и внешнее устройство, меньше частоты тактового сигнала процессора в 4 раза.

3.2 Описание интерфейсов контроллера



На данный момент разработанный контроллер работает в режиме *Master* с одним ведомым устройством, поэтому внешнее устройство взаимодействует с ним по четырехпроводному интерфейсу с одной линией SS. Интерфейс для работы с процессорным ядром в дальнейшем будет доработан, поскольку его текущее назначение — только тестирование и отладка контроллера. Ввиду максимальных упрощений он предоставляет 2 входа и 1 выход: data_in, ready_send и data_out соответсвенно. Первый из них представляет собой шину, на которую процессорное ядро выставляет пакет к отправке, второй — сигнал к началу обмена. Полученные данные с внешнего устройства будут на выходе data_out.

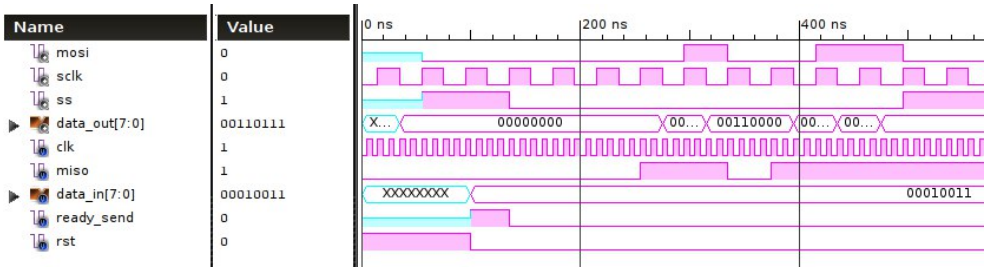
4 Тестирование контроллера

4.1 Методика

Так как SPI является стандартом передачи данных в дуплексном режиме, было необходимо провеить правильность отправки и получения пакета данных в один цикл обмена. Гипотетическим процессорным ядром был выствлен байт данных и сигнал готовности к обмену. Эти данные можно обнаружить на временной диаграмме на выходе MOSI. Данные на вход контроллера подавались тестовым окружением при низком

уровне SS в соответствии с синхросигналом — ожидалось их обнаружить в регистре `data_out` после окончания цикла обмена.

4.2 Временные диаграммы



5 Исходные тексты

5.1 Verilog

Контроллер

```
‘timescale 1ns / 1ps

module spi(
    input clk,
    input miso,
    input[7:0] data_in,
    input ready_send,
    input rst,
    output reg mosi,
    output reg sclk = 0,
    output reg ss,
    output reg[7:0] data_out
    );

localparam IDLE = 0;
localparam EXGE = 1;

reg[7:0] state;
reg[3:0] counter;
reg[7:0] data_in_reg;
reg      clk_tmp = 0;

always @(posedge clk) begin
   clk_tmp <= !clk_tmp;
   if (clk_tmp) begin
      sclk <= !sclk;
   end
end

always @(posedge sclk) begin
   if (rst) begin
      state <= IDLE;
   end
   case(state)
```

```
      IDLE: begin
         mosi <= 1'b0;
         ss <= 1;
         if(ready_send) begin
            counter <= 8;
            ss <= 0;
            state <= EXGE;
            data_in_reg <= data_in;
         end
      end
      EXGE: begin
         if (counter == 0) begin
            mosi <= 1'b0;
            ss <= 1;
            state <= IDLE;
         end else begin
            mosi <= data_in_reg[counter - 1]; // send
            counter <= counter - 1;
         end
      end
   endcase
end

always @(negedge sclk) begin
    if (rst) begin
        data_out <= 0;
    end else if (state == EXGE) begin
      data_out[counter[2:0]] <= miso; // receive
   end
end

endmodule
```

Тестовое окружение

```
`timescale 1ns / 1ps

module test_spi;

        // Inputs
        reg clk;
        reg miso;
        reg[7:0] data_in;
        reg ready_send;
        reg rst;

        // Outputs
        wire mosi;
        wire sclk;
        wire ss;
        wire[7:0] data_out;

        // Instantiate the Unit Under Test (UUT)
        spi uut (
    .clk(clk),
    .miso(miso),
    .data_in(data_in),
    .ready_send(ready_send),
    .rst(rst),
    .mosi(mosi),
    .sclk(sclk),
    .ss(ss),
    .data_out(data_out)
    );
```

```
        initial begin
                clk = 0;
                miso = 0;
                rst = 1;
                #100;

                rst = 0;
                data_in = 8'b00010011; // 13
                ready_send = 1;
                @(ss == 1);
                ready_send = 0;
                #1;
                // 37 8'b00110111
                @(posedge sclk);
                miso = 0;
                @(posedge sclk);
                miso = 0;
                @(posedge sclk);
                miso = 1;
                @(posedge sclk);
                miso = 1;
                @(posedge sclk);
                miso = 0;
                @(posedge sclk);
                miso = 1;
                @(posedge sclk);
                miso = 1;
                @(posedge sclk);
                miso = 1;

                #500
                $finish;
        end

        always
        #5 clk = ! clk;

endmodule
```

5.2 SystemC

Контроллер

```
#include "systemc.h"

/* A module to halve the frequency of a clock. */
SC_MODULE(halved_clk)
{
  sc_in<bool> clk;
  sc_out<bool> hclk;

  void invert() {
    hclk = !hclk;
  }

  SC_CTOR(halved_clk) {
    SC_METHOD(invert);
    sensitive << clk.pos();
  }

};

/* A module to reduce the frequency of a clock to a quarter of the original. */
```

```
SC_MODULE(quarted_clk)
{
  sc_in<bool> clk;
  sc_out<bool> qclk;

  halved_clk c1, c2;
  sc_signal<bool> tmp;

 SC_CTOR(quarted_clk): c1("C1"), c2("C2") {
    c1.clk(clk);
    c1.hclk(tmp);

    c2.clk(tmp);
    c2.hclk(qclk);
  }

};

/* A module for clock that only optionally ticks. */
SC_MODULE(opt_clk)
{
  sc_in<bool> clk, n_en;
  sc_out<bool> oclk;

  void tick() {
    oclk.write(clk.read() && !n_en.read());
  }

  SC_CTOR(opt_clk) {
    SC_METHOD(tick);
    sensitive << clk;
  }
};

/* The part of SPI that's responsible for receiving. */
SC_MODULE(spi_rx)
{
sc_out<sc_uint<8> > data_out;
sc_in<sc_uint<3> > ctr;
sc_in<bool> clk, in, rst;

void rx() {
    if (rst) {
        data_out.write(0);
    } else {
        sc_uint<8> data_out_tmp = data_out.read();
        data_out_tmp[ctr.read()] = in.read();
        data_out.write(data_out_tmp);
    }
}

SC_CTOR(spi_rx) {
    SC_METHOD(rx);
    sensitive << clk.pos();
}
};

/* The part of SPI that's responsible for transmission. */
SC_MODULE(spi_tx)
{
sc_in<sc_uint<8> > data_in;
sc_in<sc_uint<3> > ctr;
sc_in<bool> clk;
sc_out<bool> out;
```

```
void tx() {
    out = data_in.read()[ctr.read()];
}

SC_CTOR(spi_tx) {
    SC_METHOD(tx);
    sensitive << clk.pos();
}
};

/* Main loop thread for SPI master. */
SC_MODULE(spi_master_loop)
{
sc_in<bool> clk, rst, enable;
sc_in<sc_uint<8> > input;
sc_out<bool> ss, do_tx, do_rx;
sc_out<sc_uint<3> > ctr;
sc_out<sc_uint<8> > cache;

void state() {
    bool done = false;
    bool enabled = false;
    int ctr_temp = 0;
    ss.write(true);

    while (true) {
        wait();
        do_tx = false;
        do_rx = false;
        if (clk) {
            if (rst) {
                ss.write(true);
                cache = 0;
                enabled = false;
                done = false;
                ctr_temp = 0;
            } else if (!done && enabled) {
                do_tx = true;
                if (ctr_temp == 7) {
                    done = true;
                } else {
                    ++ctr_temp;
                    ctr.write(ctr_temp);
                }
            }
        } else {
            if (done) {
                ss.write(true);
                done = false;
                enabled = false;
            } else if (enabled) {
                do_rx = true;
            } else if (enable && !enabled) {
                ss.write(false);
                ctr_temp = -1;
                cache = input;
                enabled = true;
            }
        }
    }
}

SC_CTOR(spi_master_loop) {
    SC_THREAD(state);
    sensitive << clk;
```

```
}
};

/* SPI master. */
SC_MODULE(spi_master)
{
sc_out<sc_uint<8> > data_in;
sc_out<sc_uint<8> > data_out;
sc_in<bool> clk, rst, enable, miso;
sc_out<bool> sclk, ss, mosi;

spi_rx rx;
spi_tx tx;
quarted_clk qclk;
opt_clk oclk;
spi_master_loop lp;

sc_signal<sc_uint<3> > ctr;
sc_signal<sc_uint<8> > cache;
sc_signal<bool> do_tx, do_rx, qclk_perm;

SC_CTOR(spi_master):
    rx("RX"), tx("TX"), qclk("QCLK"), lp("LP"),
    do_tx("do_tx"), do_rx("do_rx"), oclk("OCLK"), qclk_perm("QCLK_") {
        qclk.clk(clk);
        qclk.qclk(qclk_perm);

        oclk.clk(qclk_perm);
        oclk.n_en(ss);
        oclk.oclk(sclk);

        lp.clk(qclk_perm);
        lp.rst(rst);
        lp.enable(enable);
        lp.input(data_in);
        lp.ss(ss);
        lp.do_tx(do_tx);
        lp.do_rx(do_rx);
        lp.ctr(ctr);
        lp.cache(cache);

        rx.data_out(data_out);
        rx.ctr(ctr);
        rx.clk(do_rx);
        rx.in(miso);
        rx.rst(rst);

        tx.data_in(cache);
        tx.ctr(ctr);
        tx.clk(do_tx);
        tx.out(mosi);
    }
};
```

Тестовое окружение

```
#include "systemc.h"

SC_MODULE(test_slave) {
sc_out<bool> miso, rst, enable;
sc_in<bool> mosi, clk, ss;
sc_out<sc_uint<8> > data_in;

void send() {
```

```
    wait();
    rst.write(true);
    wait();
    rst.write(false);
    data_in.write(0x37);
    enable.write(true);
    wait();
    while (ss.read() != true) {
        wait();
    }
    wait();
    miso.write(1);
    wait();
    miso.write(1);
    wait();
    miso.write(0);
    wait();
    miso.write(0);
    wait();
    miso.write(1);
    wait();
    miso.write(0);
    wait();
    miso.write(0);
    wait();
    miso.write(0);
    wait();
    for (int i = 0; i < 10; ++i) {
        wait();
    }
    sc_stop();
}

SC_CTOR(test_slave) {
    SC_THREAD(send);
    sensitive << clk.pos();
}

};
```