Projet de Systems-on-Chip (CS-309)

Lab 1: Joystick Interface

ESER Can
MAROUF Imad Eddine

Supervisor : Pr. BEUCHAT René
École Polytechnique Fédéral de Lausanne, Switzerland

***Abstract*—In this report, we presents the different steps of the design of our own FPGA-based system which is able to control a pan tilt composed of two servo motors and a camera using joysticks. We describe how we designed our programmable interface and simulated it. We present also how we programmed our programmable interface in C in order to test it on an Intel Cyclone V FPGA. This report is in the context of "Projet de Systems-on-Chip CS-309" course supervised by Pr. BEUCHAT René.**

INTRODUCTION

As part of "Projet de Systems-on-Chip course, we had laboratories to familiarize ourselves with the design in VHDL of an embedded system Intel Cyclone V SE SoC FPGA and program it in C with NIOS II. SoC FPGAs come in a wide range of programmable logic densities with many system-level functions : a dual-core ARM Cortex-A9 Hard Processor System, embedded peripherals, multiport memory controllers, serial transceivers, and PCI Express ports. VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as FPGA. Nios II allows software engineer to program different registers defined in our programmable interface.

The objective of this first laboratory is to create a full FPGA-based system (containing a processor, memory, and a programmable interface). In order to do this, we had put into application our design methodology of some programmables interfaces with the FPGA Design Software Intel Quartus. More precisely, we designed and integrated our own programmable interface going from designing Pulse Width Modulation interface 1.0 with VHDL and C throughtout Lab 1.0 and 1.1, and in Lab 1.2 we interfaced them with module MCP3203 to use joysticks to control a pan tilt composed of two servo motors and a camera at the top. In this laboratory, camera control is neglected. We will focus only on the control of two motors with the generation of two PWM signals with different duty cycles which depends on signal from joystick and converted by the Analog to Digital converter. This system allows us to rotate the camera in 2 axis left-right, and up-down.

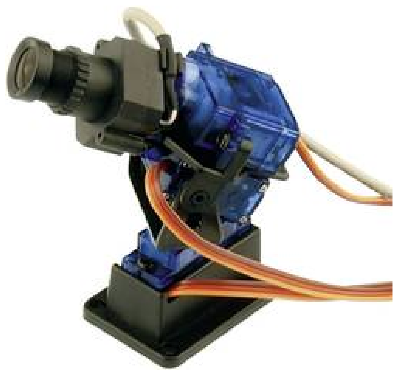
Figure 1: Pan tilt with two servos and camera.

I. Master component and Avalon bus

Our programmable interface contains a NIOS II processor. Nios II is an embedded-processor architecture designed specifically for the Intel family of FPGA. It has a configurable number of different devices : RAM/ROM memory, UART to manage a serial line, timer, PIO (Parallel Input Output). Apart from these devices implemented by Intel, the user has the possibility to add that he will have written himself which are seen by NIOS as simple memory locations accessed in read and write with the ability to handle interruptions. The internal architecture of the NIOS processor is designed to provide several advantages.

The NIOS processor can communicate with surrounding components via an Avalon bus which is an interface that indicates port connections between the master and slave components. Many bus designs exist in the industry but our FPGA of Intel use the bus called the Avalon bus. The operation of the Avalon bus is quite simple. The Avalon bus uses memory-mapped I/O to ensure the link between the master and the slave, so the same address bus is used to access memory and I/O devices. It means that the master can look for the slave at specific addresses in the master's address space. The diagram below allows us to understand our full system. It's composed of a NIOS II Processor which is the master, SRAM, an Avalon Bus and our two programmable interfaces : MCP3204 used to convert the analog signal coming from the joystick to digital signal. This interface contains also the Serial Peripheral Interface (SPI), Pulse Width Modulation Generator allows to generate two PWM signals depending on the ADC value in order to control both servos.

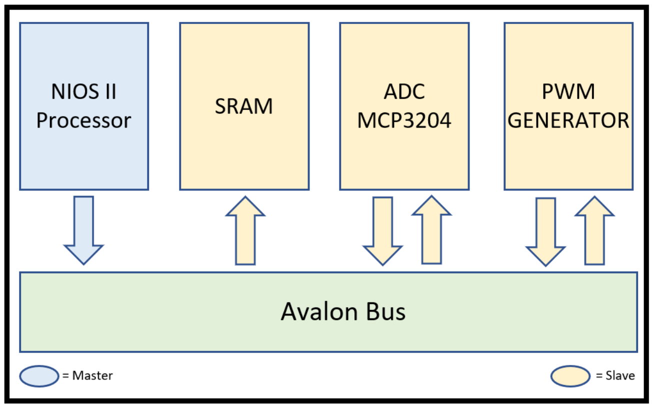

Figure 2: Diagram of our full system.

II. Analog to Digital Conversion

In order to use input device ”Playstation joysticks”, we have to convert the analog output of joysticks which are VRx and VRy pins to digital values. Therefore, in this laboratory, we used MCP3204 as show in Figure 3.

MCP3203 is 12-bit ADC and has 4 channels which is sufficient in our case, because we have 2 analog inputs.

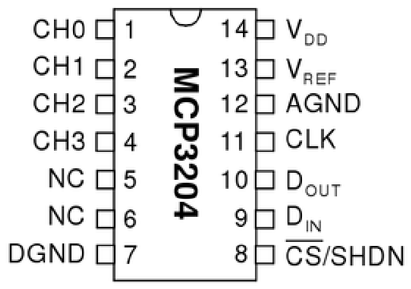

Figure 3: MCP3204

- *CH0 - CH3*: are the channels that we will use to convert analog values.
- *Din* : is used to load channel configuration, we will use single-ended configuration which mean:
  [Single/Diff bit is assigned to 1], [D2 bit: will not be used] and [D1, D0] will represent number of the channel used.
- *CS/SHDN* : is used to initiate communication with the device when pulled low and it will end the conversion.
- *CLK* : 1 MHz, we used clock divider to realize it.

Datasheet of MCP3203 is provided in Annexe

III. Design of the architecture

A. Global architecture

Now, we have to determine a communication protocol to transmit the digital values from MCP3204 to our FPGA which will be used to move the servos motors.

MCP3204 uses Serial Peripheral interface(SPI), this communication protocol is based on connection between Slave and Masters components, which are in our project ADC and FPGA respectively. In order to establish, the communication between them, we have to design a VHDL module that can supply commands to the ADC and can read the converted values back through the SPI bus. This task is divided into two modules.

The Figure 4 represent the top level of our model. We can see 2 different components. **MCP3204 Manager** is the interface between the Avalon Bus and the architecure. It instructs which channel to convert. **MCP3204 SPI** communicates with MCP3204 module to obtain the converted value, then send back the received data to the manager.

B. Clock Divider

In order for MCP3204 SPI to function efficiently, we need to create a clock of 1MHz frequency, we used a clock divider module provided in the project. Which generates a raising

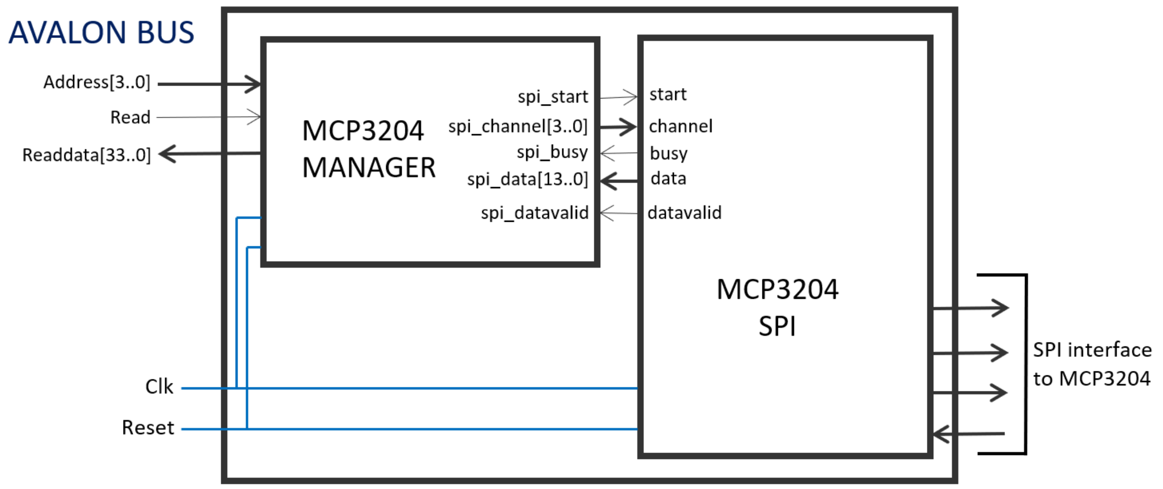

Figure 4: Top-Level block diagram of MCP3204 SPI

and a falling edge pulses at each 1MHz because the internal clock of FPGA is 50MHz, as show in code in annexe.

C. *Finite State Machines*

In this lab, we have already MCP3204 Manager coded. Therefore, we had only to design MCP3204 SPI. Since, the later has many states and operations we had to create a finite state machines which does the work in more efficient way as illustrated on the Figure 5.

From Figure 6, we see that in order for SPI module to function we need to provide input signal Start = '1', to start a conversion process, after that the SPI module will change the value of busy signal to be "1".

Throughout the conversion process, the FSM functions on the falling edge of CLK (1MHz) as illustrated on the Figure 5. After that, we move to the next state SGL where we select "single-ended" configuration, we assign MOSI = 1, move to the next state D2 which is not important because we are using 4 channels. Next state, is D1 which the most significant bit in channels selection therefore we use channel input signal MSB (MOSI = channel[1]), D0 as LSB (MOSI = channel[0]).

In order to avoid conflicts while writing, we had to create a wait state, where we wait for the next falling edge to start writing into MOSI signal bit by bit. MCP3204 is 12-bit ADC, and MOSI signal is one-bit output, then we had to create a counter which counts 13 cycles (including NULL bit), in each cycle we write into MOSI bit by bit starting from MSB to LSB. After that, we assign data signal to 12-bit from internal signal of 13-bits (removing NULL-bit) taking into consideration that counter functions on raising edge and Data_valid signal should be assigned to "1" throughtout the writing state and chip-select signal should be assigned to "1" after writing state is done. Afterwards, we move to "End" state where we assign busy signal and data_valid to "0", in order to start the process again.

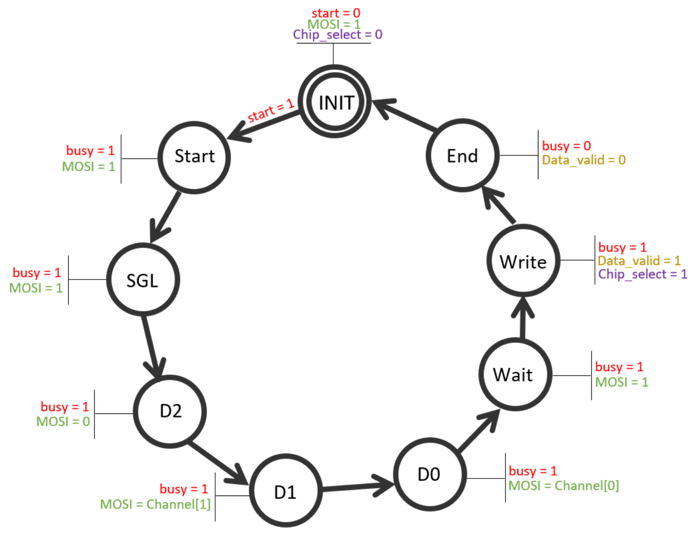

Figure 5: Final State Machine of MCP3204_SPI.

D. *Software*

After our programmable interface is designed, compiled and launched on our FPGA, we used NIOS II System Architect Design to program the NIOS II Processor with completing a C program.

To do this, we used a IO Header file regs.h. This header file, defines the core's register map. So it allows us to use

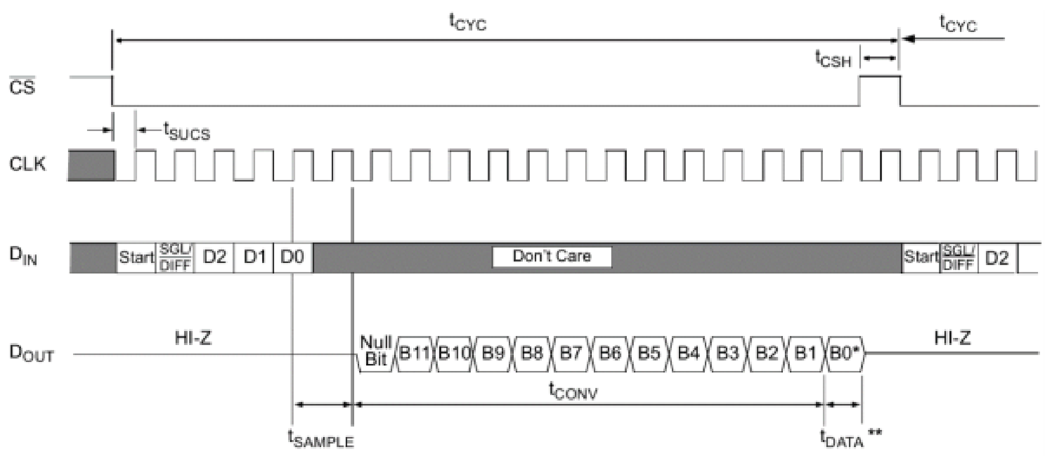

Figure 6: MCP3204 SPI Communication Protocol

macros to access low-level hardware instead of hard-coding various numbers to make the code much easier to read and to debug when necessary. We should also add system.h to access to base addresses of global architecture as defined by Qsys and io.h in order to read and write to system peripherals which allow us to use simple instruction in our code.

First, we had to complete mcp3204_read() function, we used IORD_32DIRECT(BASE, OFFSET): Make a 32-bit read access at the location with address BASE+OFFSET, after asserting that number of channels is less than 4.

Secondly, we had to convert digital values obtained from ADC to the required type of output ranging from from 0 to 4095. For vertical axe, we see that it is inverted, so we used absolute value function to get the desired range:

```
uint32_t joysticks_read_left_vertical(
    joysticks_dev *dev) {
    uint32_t output = mcp3204_read(&(dev->mcp3204)
    , JOYSTICK_LEFT_VRY_MCP3204_CHANNEL);
    return abs(output - JOYSTICKS_MAX_VALUE);
}
```

The table bellow shows the unit's register map. All the unit's registers are read-only, as it is impossible to write to the joysticks, so we omit any write related signals from the Avalon bus to simplify the design. Finally, the result of each conversion is stored in one of these 4 internal registers and incoming requests on the MCSP3204 Manager are served directly from the contents of these registers.

OFFSET	ADDRESS	Name	Access
0	0x00	CHANNEL_0	RO
1	0x04	CHANNEL_1	RO
2	0x08	CHANNEL_2	RO
3	0xC	CHANNEL_3	RO

Table I: MCP3204 MANAGER Register map

Thirdly, we had to complete interpolation function in "app.c" which is used to map the range values obtained from ADC readings of the joysticks to the range of values used by PMW generator.

```
uint32_t interpolate(uint32_t input,
                     uint32_t input_lower_bound,
                     uint32_t input_upper_bound,
                     uint32_t output_lower_bound,
                     uint32_t output_upper_bound)
    {

    double p = (float) (output_upper_bound -
    output_lower_bound) / (input_upper_bound -
    input_lower_bound);

    return output_lower_bound + (uint32_t) (p * (
    input - input_lower_bound));
}
```

IV. Results

Once our programmable interface is designed, we simulated it on Modelsim to check its operation before programming on NIOS and running it on the FPGA.

As illustrated in Figure 7, we see that the full design functions correctly, where we want to read values from channel 0, and data signal is getting the correct converted values from ADC from MSB to LSB over a vector of 12-bits on each raising edge of SCLK signal (1MHz).

Additionally, in Figure 7, data_valid signal is assigned to "1" throughout the writing process, where we send the converted value to data signal bit by bit using the counter, then data_valid changes its value from "1" to "0" when we move to the next state "end", same for the busy signal is assigned "1" throughout the first half of the simulation. Therefore, the master module cannot send commands to SPI module unless the writing process is finished.

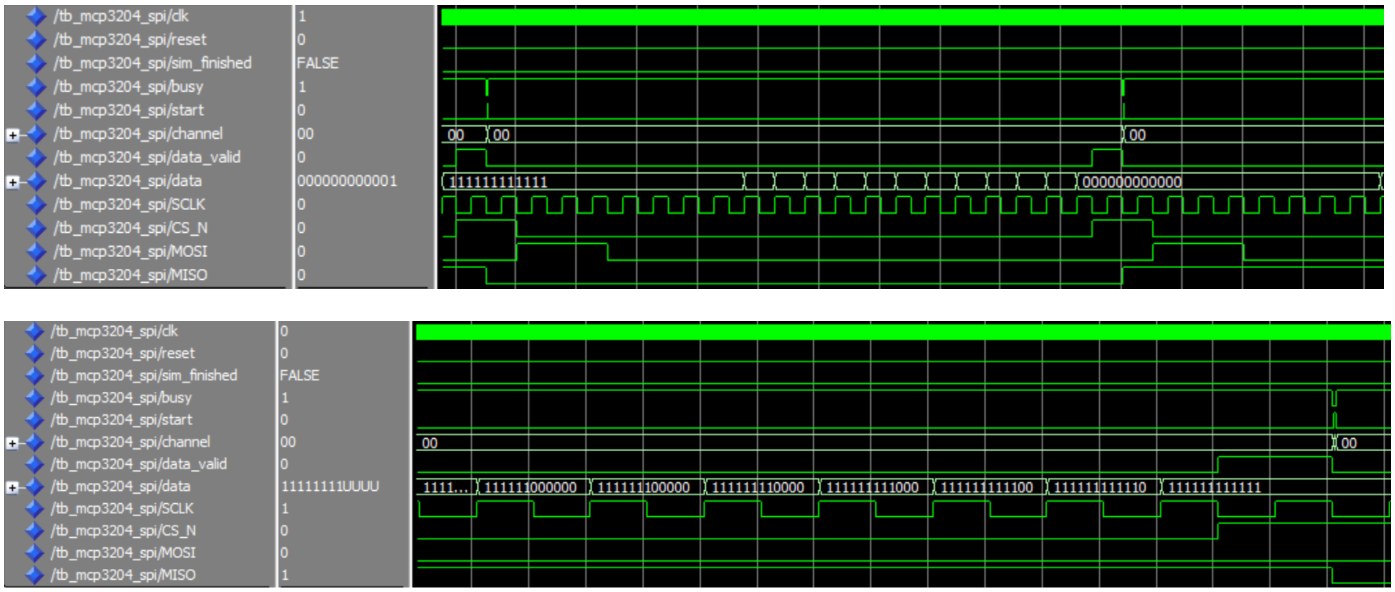

Figure 7: Full System Simulation

We could not have a running demo to demonstrate our design due to a circuit problem in our extended board in J14/J15 pins, we concluded that by using the Logic Analyzer.

Conclusion

This first laboratory of ”Projet de Systems-on-Chip CS-309” course allows us to familiarize ourselves with the design of a specific FPGA-based programmable interface and the MCP3204 ADC with the communication protocol SPI. We understood that it is very important to respect each step correctly when designing and implementing a specific programmable interface on FPGA.

If we must learn a main lesson by this laboratory, we have to remember for the future that when designing a programmable interface, the code is not the most important. First of all, it is important to take time to think on the design and to define different modules that compose our full system, to have a clear overview of what parts are missing, the right way to establish communication between them which will help a lot to detect errors afterwards. We also learnt that it is very important to have clear documentation available such that different member of the project can fully understand the code provided.

ANNEXE: VHDL CODE OF MCP3204_SPI

```vhdl
-- MCP3204 SPI interface.
-- Author: Phil mon Favrod [philemon.favrod@epfl.ch]
-- Author: Sahand Kashani-Akhavan [sahand.kashani-akhavan@epfl.ch]
-- Author: Imad MAROUF (<imad.marouf@epfl.ch>)
-- Author: Eser CAN (<eser.can@epfl.ch>)
-- Revision: 1
-- Last modified : 18/03/2020
-- ##############################

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

library work;
use work.all;

entity mcp3204_spi is
    port(
        -- 50 MHz
        clk        : in  std_logic;
        reset      : in  std_logic;
        busy       : out std_logic;
        start      : in  std_logic;
        channel    : in  std_logic_vector(1 downto 0);
        data       : out std_logic_vector(11 downto 0);
        data_valid : out std_logic;

        -- 1 MHz
        SCLK : out std_logic;
        CS_N : out std_logic;
        MOSI : out std_logic;
        MISO : in  std_logic
    );
end mcp3204_spi;

architecture rtl of mcp3204_spi is
    signal reg_clk_divider_counter : unsigned(4 downto 0) := (others => '0'); -- need to be able to count until 24
    signal reg_spi_en : std_logic := '0'; -- pulses every 0.5 MHz
    signal reg_rising_edge_sclk : std_logic := '0';
    signal reg_falling_edge_sclk : std_logic := '0';

    signal reg_sclk : std_logic := '0';

    signal busy_tmp : std_logic := '0';
    signal data_valid_tmp : std_logic := '0';
    signal data_tmp : std_logic_vector(data'range) := (others => '0');
    signal cs_n_tmp : std_logic := '1';
    signal mosi_tmp : std_logic := '0';

    signal data_inversed : std_logic_vector(12 downto 0);
    signal data_counter : unsigned(3 downto 0) := (others => '0');
```

```vhdl
    type FSM is (STATE_INIT, STATE_START, STATE_SGL, STATE_D2, STATE_D1, STATE_D0, STATE_WAIT, STATE_WRITE, STATE_END);
    signal state : FSM := STATE_INIT;

begin
    clk_divider_generation : process(clk, reset)
    begin
        if reset = '1' then
            reg_clk_divider_counter <= (others => '0');
        elsif rising_edge(clk) then
            reg_clk_divider_counter <= reg_clk_divider_counter + 1;
            reg_spi_en            <= '0';
            reg_rising_edge_sclk  <= '0';
            reg_falling_edge_sclk <= '0';

            if reg_clk_divider_counter = 24 then
                reg_clk_divider_counter <= (others => '0');
                reg_spi_en            <= '1';

                if reg_sclk = '0' then
                    reg_rising_edge_sclk <= '1';
                elsif reg_sclk = '1' then
                    reg_falling_edge_sclk <= '1';
                end if;
            end if;
        end if;
    end process;

    SCLK_generation : process(clk, reset)
    begin
        if reset = '1' then
            reg_sclk <= '0';
        elsif rising_edge(clk) then
            if reg_spi_en = '1' then
                reg_sclk <= not reg_sclk;
            end if;
        end if;
    end process;


    STATE_LOGIC : process(clk, reset)
    begin

      CS_N<= cs_n_tmp;
      MOSI<= mosi_tmp;
      data<= data_tmp;
      data_valid<= data_valid_tmp;
      SCLK<= reg_sclk;
      busy<= busy_tmp;

        if reset = '1' then
      data_valid_tmp <= '0';
      data_counter   <= (others => '0');
      state          <= STATE_INIT;
      cs_n_tmp       <= '1';
      busy_tmp       <= '0';
      mosi_tmp       <= '0';

        elsif rising_edge(clk) then
        if ( start = '1') then
          busy_tmp <= '1';
        elsif reg_falling_edge_sclk = '1'
    then
            case state is
                when STATE_INIT =>
                  if busy_tmp = '1' then
                     state <= STATE_START;
                     mosi_tmp  <= '1';
                     cs_n_tmp  <= '0';
                  end if;

                when STATE_START =>
                  mosi_tmp  <= '1';
                  state <= STATE_SGL;

                when STATE_SGL =>
                  state <= STATE_D2;
                  mosi_tmp  <= '1';

                when STATE_D2 =>
                    mosi_tmp  <= '0';
                  state <= STATE_D1;

                when STATE_D1 =>
                  mosi_tmp  <= channel(1);
                  state <= STATE_D0;

                when STATE_D0 =>
                  data_counter <= to_unsigned
    (12, data_counter'length);
                    mosi_tmp  <= channel(0);
                  state <= STATE_WAIT;

                when STATE_WAIT =>
                  state <= STATE_WRITE;

                when STATE_WRITE =>
                  if data_counter /= 0 then
                     data_counter <=
    data_counter - 1;
                  else
                     cs_n_tmp  <= '1';
                     data_valid_tmp <= '1';
                     state  <= STATE_END;
                  end if;

                when STATE_END =>
                  state  <= STATE_INIT;
                  busy_tmp  <= '0';
                  data_valid_tmp <= '0';
           end case;
        end if;
      end if;
    end process;

    counter_to_write : process(clk, reset)
    begin
        if reset = '1' then
            data_tmp <= (others => '0');

        elsif rising_edge(clk) then
            if reg_rising_edge_sclk = '1'
    then
                if state = STATE_WRITE then
                    data_inversed(to_integer
    (data_counter)) <= MISO;
                end if;
            end if;
        end if;
    data_tmp <= data_inversed(11 downto 0);
    end process;
end architecture rtl;
```