UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Written exam IN3160/4160 and INF3430/4431 - Digital system design
2020 SPRING
Duration: 2. June, 14:30 to 9. June, 14:30

General information:

- Your submission must be uploaded as a zip-file, in addition to filling in each question.
- Remember that your submission need to be anonymous, do not write your name in your submission.
- All examination support materials are permitted. You need to gather information from available sources, assess the information quality, and put it together in a submission based on your own processing of the content. The submission must reflect your individual level of knowledge.
- For assignments where it is relevant to use sources and citations, it is important that you do this properly so that you are not suspected of cheating. Read more about sources and citations.
- You have to read UiO's upload assignment student guide
- You have to read IFI's rules about cheating on exams.

Digital hand drawing:

- If your submission includes digital hand drawings, you are free to use your preferred tools (scanning, cellphone-camera etc) as long as everything is readable and delivered as one PDF. How to make a PDF-file.
- Check out MN's/UiO's recommended solutions for digital hand drawings spring 2020.

NB!

You cannot apply for a postponement of the exam beyond the 7 days the exam is held. If you submit a self-notification abut illness you will be able to take the continuation exam in the courses that offer it, or take the exam the next time the course is held (applies to IN1150, IN1000 and ENT1000).

See: https://www.mn.uio.no/om/hms/koronavirus/eksamen-2020.html (Paragraph 9, 10 and 11)

Contact:
User support for exams in the spring of 2020.

Messages during the exam
If any, messages to everyone during the exam will be posted at the course semester page: https://www.uio.no/studier/emner/matnat/ifi/IN3160/v20/
Please check for messages regularly.

About the exam
The exam consists of 10 exercises. In Exercise 1-3, 5 and 6 the answer shall be VHDL code. In exercise 4, 7, 8 and 9 diagrams and text. The 10th exercise is for uploading one zip file containing all your VHDL and pdf files.

Each exercise has a maximum score. The exam as a whole can have a score up to 100. The scores are noted to give you a chance to prioritize between exercises.

The 10th exercise has no maximum score, as it only serves the purpose of allowing a zip file to be uploaded. The zip file should only contain VHDL files and a pdf file containing all text and diagrams used. We ask you to both to fill in answers in exercise 1-9 separately and upload on exercise 10, for redundancy. The zip file will not be checked unless it is necessary.

To allow for the use of third party tools when working with the exercises, we recommend solving most of the tasks outside of inspera, and later move the code, text or files into each exercise. If you need figures that cannot be uploaded in the exercise (1-9), please make a note in the exercise text that refers to where in the zip file it can be found.

The full exam text can be viewed as a pdf (this document). You can complete the whole exam outside inspera, then later copy in the relevant VHDL code, text and diagrams.

Good luck!

Outline

In this exam, you will implement a serial peripheral interface (SPI) slave device that reads and writes to an 8-bit register. You will also write test bench code to verify the function of the device, create an ASM diagram that shows part of the device functionality, and answer questions in text format.

The implementation shall be able to both send and receive data from the SPI bus. The 8-bit register shall be connected to output pins on an FPGA. This allows for control of LEDs or other devices.

The SPI bus

The SPI bus was invented by Motorola in 80's, and is a common bus used to interface low speed devices. Normally an SPI interface will consist of four wires:
SCK – Serial Clock
MOSI – Master Out Slave In
MISO – Master In, Slave Out
SS – Slave Select (sometimes also called chip select (CS)).

SPI can have different modes for clock polarity and phase, but in this exam, we will only use the configuration where data must be valid on the rising edge of SCK.

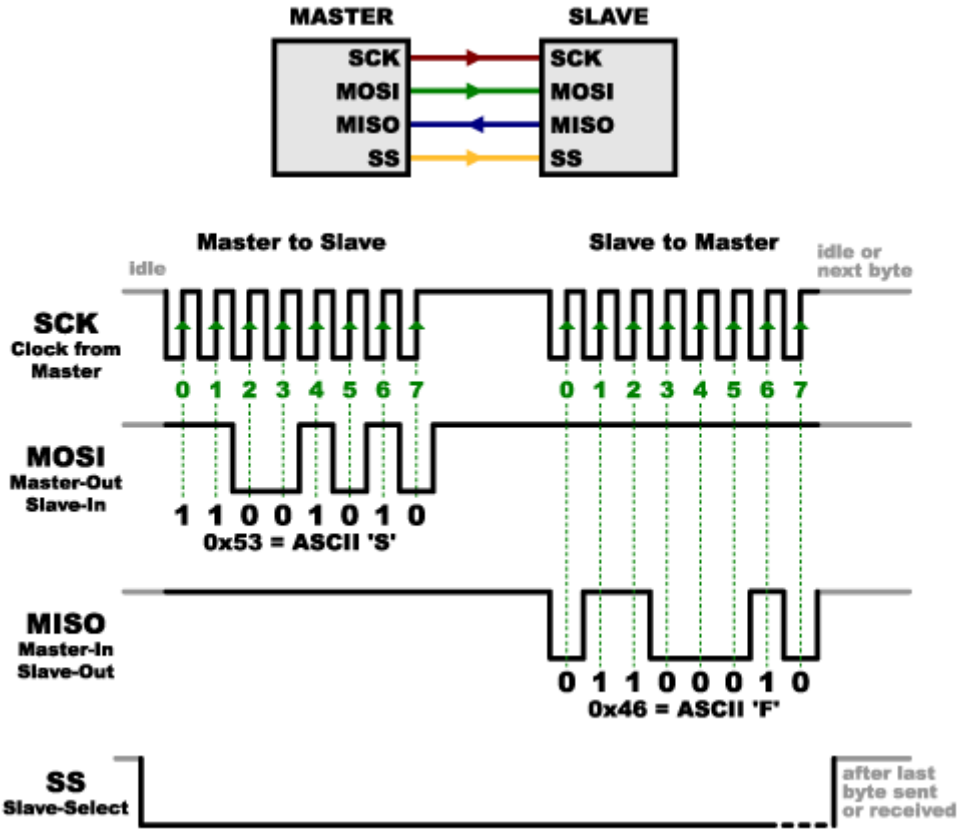

Figure 1: Simple SPI setup and signaling overview

In the simplest configuration, an SPI bus consist of two devices, a master and a slave. The master will initiate all transactions by setting slave select (`SS`) low. A short time after asserting slave select, the master will activate the SPI clock (`SCK`), and both the master and the selected slave can put data on the bus (`MOSI/MISO`).

The digital system
The system main clock (`clk`) will have a clock frequency of 100 MHz. The SPI bus master will operate `SCK` at up to 10MHz. Apart from the signaling on the SPI bus, the SPI interface will read or write 8 bits in parallel to or from another module running on the same system clock.

The top level of the design, **SPI_top.vhd,** will be a structural description to connect the required components:

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

entity spi_top is
  generic(WIDTH : natural := 8);
  port (
    clk             : in  std_logic;

    SS              : in  std_logic;
    SCK             : in  std_logic;
    MOSI            : in  std_logic;
    MISO            : out std_logic;

    data_in         : in std_logic_vector(WIDTH-1 downto 0);
    data_out        : out std_logic_vector(WIDTH-1 downto 0);
    valid           : out std_logic
    );
end entity spi_top;
architecture structural of spi_top is

    -- insert structural description here

end architecture structural;
```

Figure 2: VHDL Entity and architecture template, SPI_top.vhd

The implementation shall connect the following four modules:
edge_detector.vhd : an edge detector module used for SCK,
shifter.vhd : an shift register module used for IO
counter.vhd : a counter module used together with the state machine
fsm.vhd : a state machine controlling the system operation

In addition to the SPI module, a testbench shall be created.

Unless otherwise specified, all input signals are synchronous to the system clock (`clk`).

Exercise 1: 10p

In this exercise, you shall implement the VHDL edge detector module for the system (edge_detector.vhd). The edge detector shall have three inputs, the system clock (`clk`), the slave select signal (`SS`), and the SPI clock (`SCK`). The edge detector shall have one ouput, (`SCK_rise`) which shall be '1' for exactly one clock cycle when the SPI clock goes high. When the SS signal goes high, the system shall be set to the s_0 state, regardless of the `SCK` signal. All transitions shall be synchronous to the system clock.

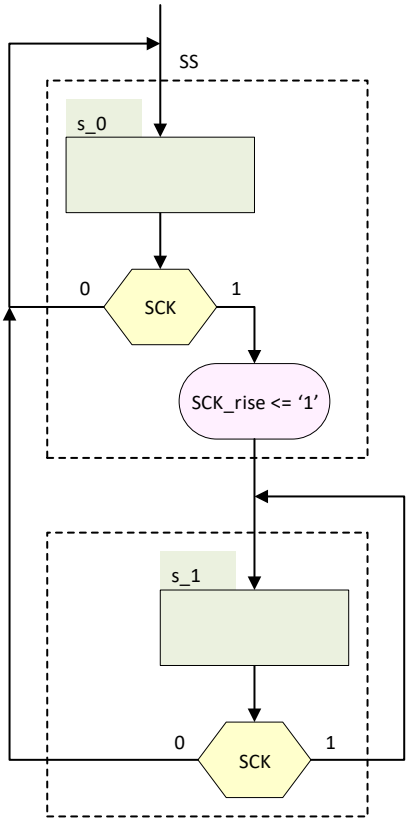

Figure 3 ASM digram of the edge detector function.

The edge detector shall be implemented as a state machine having two states as shown in Figure 3.

Exercise 2: 10p

In this exercise, you shall design a shift register as shown in Figure 4.

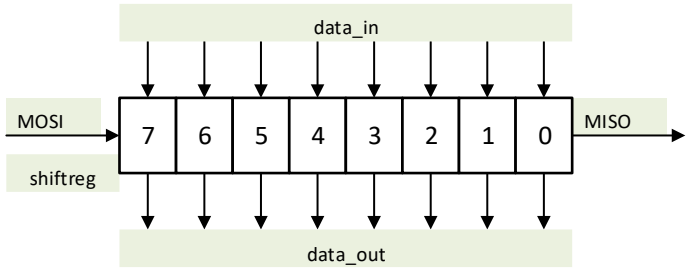

Figure 4: Shift-register for the SPI interface

```
entity shifter is
  generic(WIDTH : natural := 8);
  port (
    clk             : in  std_logic;
    SS              : in  std_logic;
    SCK_rise        : in  std_logic;
    MOSI            : in  std_logic;
    load            : in  std_logic;
    data_in         : in  std_logic_vector(WIDTH-1 downto 0);
    data_out        : out std_logic_vector(WIDTH-1 downto 0);
    MISO            : out std_logic
  );
end entity shifter;
```

Figure 5: Shifter entity declaration

The shift-register module (shifter.vhd) shall use the entity described in Figure 5, and shall be implemented as follows:

1. All output shall be synchronous to the rising edge of the system clock.
2. The shift register shall be set to zero when slave select (`SS`) is high.
3. `MISO` shall be set to bit 0 in the shift register.
4. As long as `SS` is low, the following shall happen:
    a. When `load` is high, `data_in` shall be clocked into the shift register
    b. When SCK_rise is high:
        i. the MOSI signal shall be shifted into the highest numbered bit in the shift register
        ii. All bits in the shift register shall be shifted to a lower numbered position (shifted right as shown in Figure 4)

Exercise 3: 10p
In this exercise, you will implement the counter module using the entity shown in Figure 6.

```
entity counter is
  generic(WIDTH : natural := 8);
  port (
    clk             : in  std_logic;
    reset_count     : in  std_logic;
    SCK_rise        : in  std_logic;
    mincount        : out std_logic;
    halfcount       : out std_logic
  );

end entity counter;
```

Figure 6: The counter module (counter.vhd)

All outputs from the counter module shall be synchronous to the system clock (`clk`). The counter shall be set to zero when the `reset` signal is high, and count when `SCK_rise` is high. The counter shall count from 0 to 15 before it wraps around to 0. The signal `mincount` shall be high when the counter is 0 and `halfcount` shall be high when the counter is 8.

Exercise 4: 15p
In this exercise, you will implement an ASM diagram that shows the correct function of the VHDL code for the finite state machine (FSM) module (fsm.vhd) described in Figure 7 and Figure 8.

The state machine reads and interprets the commands sent over the SPI bus. There are four commands that the FSM interprets: Fetch, Put, Pass and No-operation (NOP).

When given the Fetch command, the device shall fetch the data present at the input and send it over the SPI bus. When given the put command, the device shall put the next byte it receives over SPI, and set the valid flag when the whole byte is received. When given the Pass command, the next byte shall be sent over the SPI bus, regardless of content. When the NOP command is sent, the device shall be ready to receive a new command on the next byte.

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

entity fsm is
  generic(WIDTH : natural := 8);
  port (
    clk             : in  std_logic;
    SS              : in  std_logic;
    halfcount       : in  std_logic;
    mincount        : in  std_logic;
    data            : in  std_logic_vector(WIDTH -1 downto 0);
    load            : out std_logic;
    valid           : out std_logic;
    reset_count     : out std_logic
  );

end entity fsm;
```

Figure 7: FSM module part I, VHDL entity

```
architecture RTL of fsm is
  constant OP_NOP   : std_logic_vector(WIDTH-1 downto 0) := x"00";
  constant OP_FETCH : std_logic_vector(WIDTH-1 downto 0) := x"01";
  constant OP_PUT   : std_logic_vector(WIDTH-1 downto 0) := x"02";
  constant OP_PASS  : std_logic_vector(WIDTH-1 downto 0) := x"03";

  type t_state is (read_op, put, transmit);
  signal fsm_state, next_state : t_state;

begin

  fsm_state <= next_state when rising_edge(clk);

  process(all)
  begin
    next_state <= fsm_state;
    if SS then
      next_state <= read_op;
    else
      case fsm_state is
        when read_op =>
          if halfcount then
            with data select next_state <=
              put      when OP_PUT,
              transmit when OP_FETCH,
              read_op  when OP_NOP,
              transmit when others;
          end if;
        when put | transmit =>
          next_state <= read_op when mincount;
        when others =>
          next_state <= read_op;
      end case;
    end if ;
  end process;

  process(all)
  begin
    load        <= '0';
    valid       <= '0';
    reset_count <= SS;
    case fsm_state is
      when read_op =>
        load        <= '1' when halfcount = '1' and data = OP_FETCH;
        reset_count <= '1' when halfcount = '1' and data = OP_NOP;
      when put =>
        valid <= '1' when mincount;
      when others =>
        null;
    end case;
  end process;

end architecture RTL;
```

Figure 8: FSM module part II, VHDL architecture

Exercise 5: 10p
In this exercise, you will create the structural architecture of the SPI top module.
The entity is for the top module is given in Figure 2.

Create a VHDL module, **SPI_top.vhd**, that connects all four modules as indicated by the names in each entity. Signals necessary to connect the modules must be created.

Exercise 6: 21p

In this exercise you shall implement a self-checking test bench that communicates with the device under test (DUT) using the SPI bus. The test bench shall stimulate the DUT, and verify that the output is correct.

Communication with the DUT shall be implemented in a procedure. The procedure shall emulate an SPI master. The procedure shall print the result of each test. The test bench shall stop with a failure condition when a test does not pass (incorrect output is observed). Use the procedure to show correct operation using `op_fetch`, `op_put` and `op_pass` commands.

Exercise 7: 6p
The SPI module will be connected to an SPI master running in a different clock domain (SPI clock still running at 10MHz). Describe with words what would be the necessary modifications to the SPI slave interface to ensure safe clock domain crossing.
(What functionality should be considered- which signals and modules should be involved, etc.)

Exercise 8 14p

In this task we assume that a number of similar slaves, all utilizing the same FPGA configuration described in this exam, all using the modifications described in Exercise 7 (thus no clock domain crossing considerations here), each running on their own circuit board, will be connected in series to the master, as shown in Figure 9. The commands the Master can issue is (as described in exercise 4 code): NOP, put, fetch and pass.

```
constant OP_NOP   : std_logic_vector(WIDTH-1 downto 0) := x"00";
constant OP_FETCH : std_logic_vector(WIDTH-1 downto 0) := x"01";
constant OP_PUT   : std_logic_vector(WIDTH-1 downto 0) := x"02";
constant OP_PASS  : std_logic_vector(WIDTH-1 downto 0) := x"03";
```

We assume that each slave is connected to its own sensor on their data_in port.

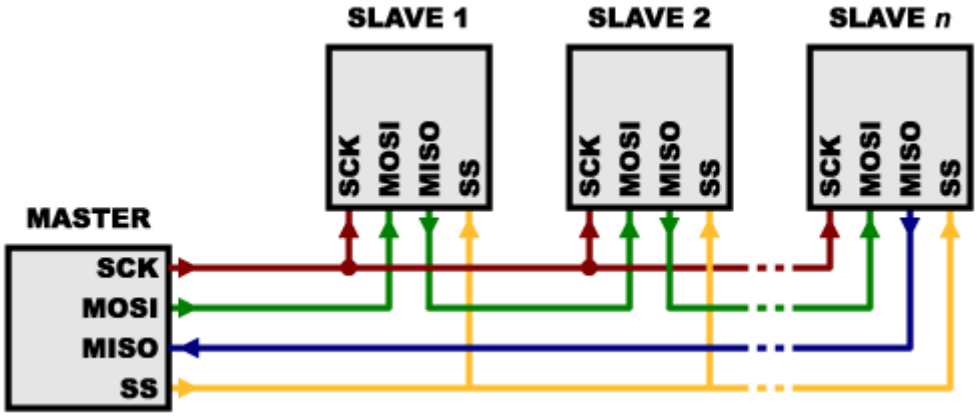

Figure 9: SPI devices in a series configuration

a) To what extent can they run without individual modifications?

i. Will this setup work electrically, or is modification necessary (if so, how)?

ii. Which commands can be used to provide consistent result? (explain) (consistent *in this context* means that we know either what will happen, or who provided the result)

We connect four devices, all sharing the same configuration, in a row similar to Figure 9, having n = 4.

b) If possible, what would be the shortest possible sequence of op-codes the master should send to…

i. … retrieve the data input from all the slaves

ii. … set the ouput on all slaves

iii. … set the output on slave 2 only

iv. … test that all slaves communicate by passing the byte 0x01

c) If all unused bits in the op_codes were used to individual addresses for the SPI devices. How many slave devices could then be connected in series and still be read individually? (We assume the Master will be able to drive the slave select signal).

Exercise 9 4p

In this task, we assume that a number of similar slaves, each having a seperate `SS`-signal, will be connected in parallel to the master, as shown in Figure 10.

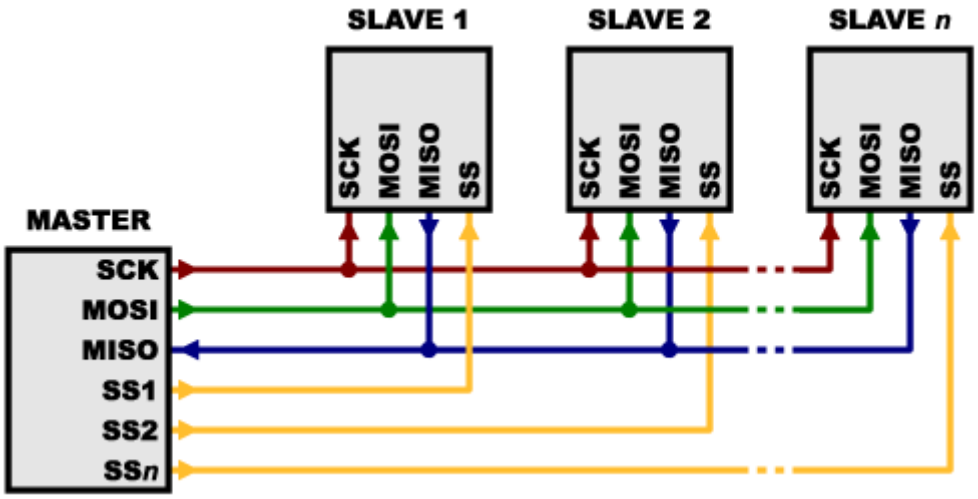

Figure 10: SPI devices in a parallel configuration

To what extent can they run without individual modifications…

a) Will this setup work electrically, or is modification necessary (if so, how)?

b) Which commands can be used to provide meaningful result?