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.

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.

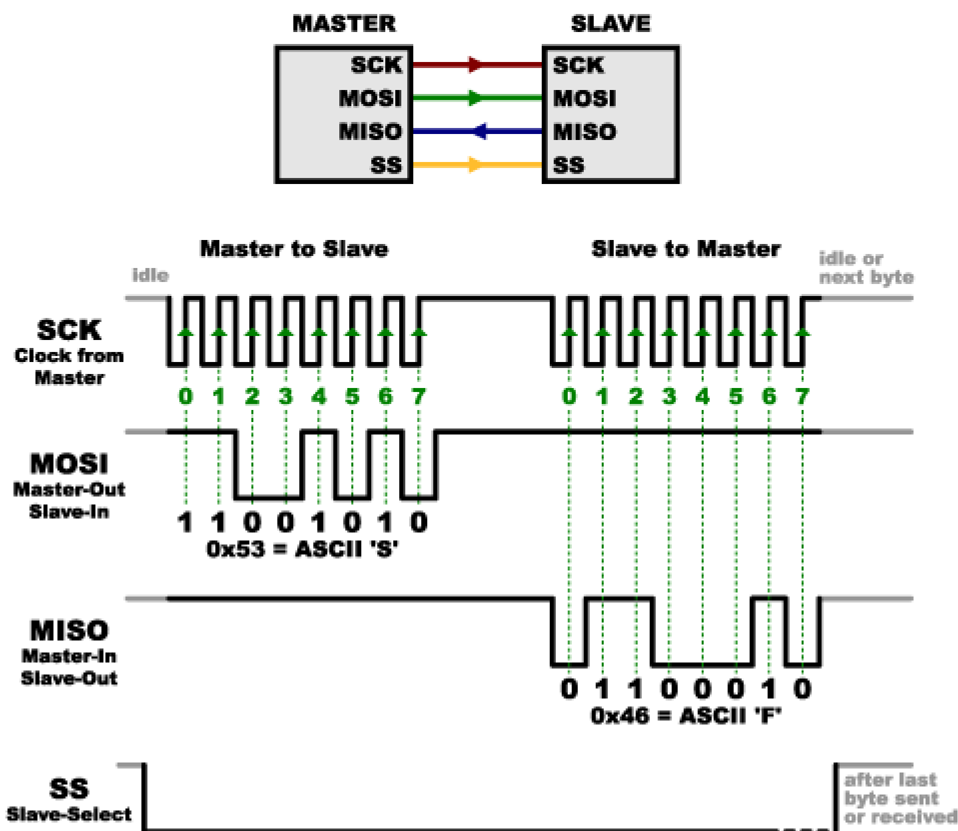


Figure 1: Simple SPI setup and signaling overview

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.

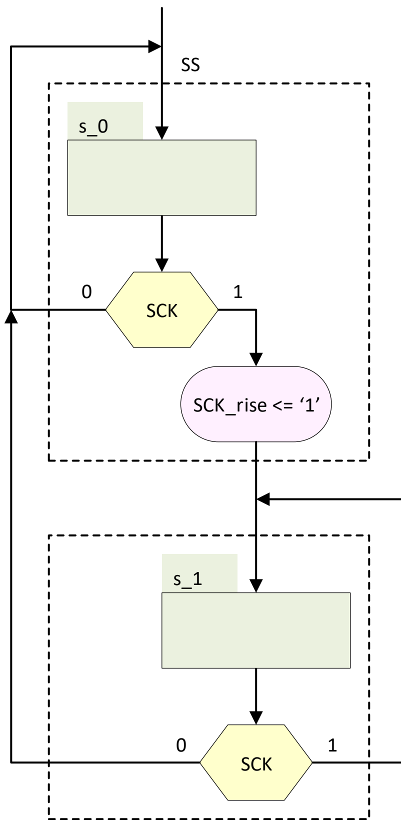


Figure 3 ASM digram of the edge detector function.

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

Sensorveiledning (10p):
Korrekt bruk av biblioteker: 2p (ieee.std_logic_1164) unødvendige biblioteker => 1p
Korrekt entitet: 2p (ett unødvendig signal => 1p, mangler ett => 0p)
Tilstandsmaskin korresponderer med diagram: 2 poeng
Modulen virker 2 poeng.
Koden lett å lese 2 poeng. (*skiller kombinatorikk og klokkede hendelser på en hensiktsmessig måte*)

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

entity edge_detector is
  port (
    clk             : in  std_logic;
    SS              : in  std_logic;
    SCK             : in  std_logic;
    SCK_rise        : out std_logic
    );
end entity edge_detector;

architecture RTL of edge_detector is
  type t_edge_state is (s_0, s_1);
  signal edge_state, next_state : t_edge_state;

begin
  edge_state <= next_state when rising_edge(clk);
  SCK_rise <= '1' when edge_state = s_0 and SCK = '1' and  SS = '0' else '0';

  process (all) is
  begin
    case edge_state is
      when s_0 =>
        next_state <= S_1 when SCK and not SS;
      when s_1 =>
        next_state <= S_0 when SS or not SCK;
    end case;
  end process;

end architecture RTL;
```

Typiske feil:

- Bibliotek
  - Unødvendig bruk av numeric_std
  -
- Entitet: (ingen spesielle)
- Tilstandsmaskin:
  - Asynkron reset med SS
  - Skiller ikke kombinatorikk og klokket tilordning
    - Typisk ved bruk av en-prosess maskiner- NB: kan også gjøres riktig.
- Funksjon:
  - Bruk av rising edge på flere signal i en prosess
  - Synkronisering med SCK_rise i stedet for clk
  - Bruk av buffersignaler som forsinker unødvendig
- Lesbarhet:
  - Unødvendig lange/ omstendelige prosesser. Ting som gjøres dobbelt uten at det nødvendigvis er feil, men ekstra linjer. Unødvendig bruk av process
  - Bruker process(all) der det bør være process(clk)

Exercise 2: 10p
In this exercise, you shall design a shift register as shown in .

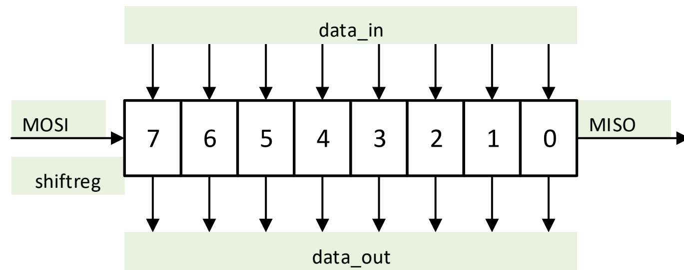


Figure 4: Shift-register for the SPI interface

```
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 , 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)

Sensorveiledning 10p
All output synkron 2 poeng
Korrekt funksjon 4 poeng i henhold til spekk
Lett å lese 4 poeng.
Løsningsforslag under:

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

entity shifter is
  generic(WIDTH : natural := 8); -- må harmonere med øvrig design
  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;

architecture RTL of shifter is
  signal next_data : std_logic_vector(WIDTH-1 downto 0);
begin

  next_data <= MOSI & data_out(data_out'high downto data_out'low + 1);
  MISO <= data_out(data_out'low);

  process (clk) is
  begin
    if rising_edge(clk) then
      if SS then
        data_out <= (others => '0');
      else
        data_out <=
          data_in when load else
          next_data when SCK_rise;
      end if;
    end if;
  end process;

end architecture RTL;
```

Typiske feil oppdaget under retting:

- Lesbarhet
  - Bruke process(all) der det skal være process(clk)
  - Navngi rtl modul for behavioral
  - (Angi numeric_std som ikke brukes.)
- Synkronitet
  - Bruke SS som asynkron reset
- Funksjon
  - Forsinke MISO og data_out ved å benytte et datasignal til å holde shiftregisteret, og så oppdatere både shiftregister, data_out og MISO på klokkeflanke i stedet for å tilordne MISO og data_out kombinatorisk. Om shiftregisteret er variabel vil det også kunne løse problemet om koden er stokket riktig.
  - Prioritere SCK_rise over load ved å droppe elsif (overlate til prioriteringsrekkefølge)

Exercise 3: 10p

In this exercise, you will implement the counter module using the entity shown in .

```
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.

Sensorveiledning 10p:

All output synkron 2 poeng

Korrekt funksjon 4 poeng i henhold til spekk

Lett å lese 4 poeng.

(ifht lett å lese- kan også beskrives som enkelhet, bruker hensiktsmessige metoder som fasiliterer få ressurser ved simulering, og som ikke sår tvil om intensjonen)

Løsningsforslag under:

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

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

end entity counter;


architecture RTL of counter is
  signal counter : integer;
  signal maxcount: std_logic;
begin

  mincount <= '1' when counter = 0 else '0';
  maxcount <= '1' when counter = WIDTH*2-1 else '0';
  halfcount <= '1' when counter = WIDTH else '0';

  process (clk) is
  begin
    if rising_edge(clk) then
      if reset then
        counter <= 0;
      elsif SCK_rise then
        counter <= 0 when maxcount else counter + 1;
      end if;
    end if;
  end process;

end architecture RTL;
```

Typiske feil

- Lesbarhet
  - Bruke process(all) der det skal være process(clk)
  - Navngi rtl modul for behavioral
  - Bruke numeric_std uten å deklarerere den
- Synkronitet
  - Bruke SS som asynkron reset
- Funksjon
  - Forsinket output pga klokket tilordning av to signaler-
    - Kan bøtes på med bruk av variabel (NB: rekkefølge)
    - Kan bøtes på med kombinatorisk tilordning av output basert på telleverk.
    -

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 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).

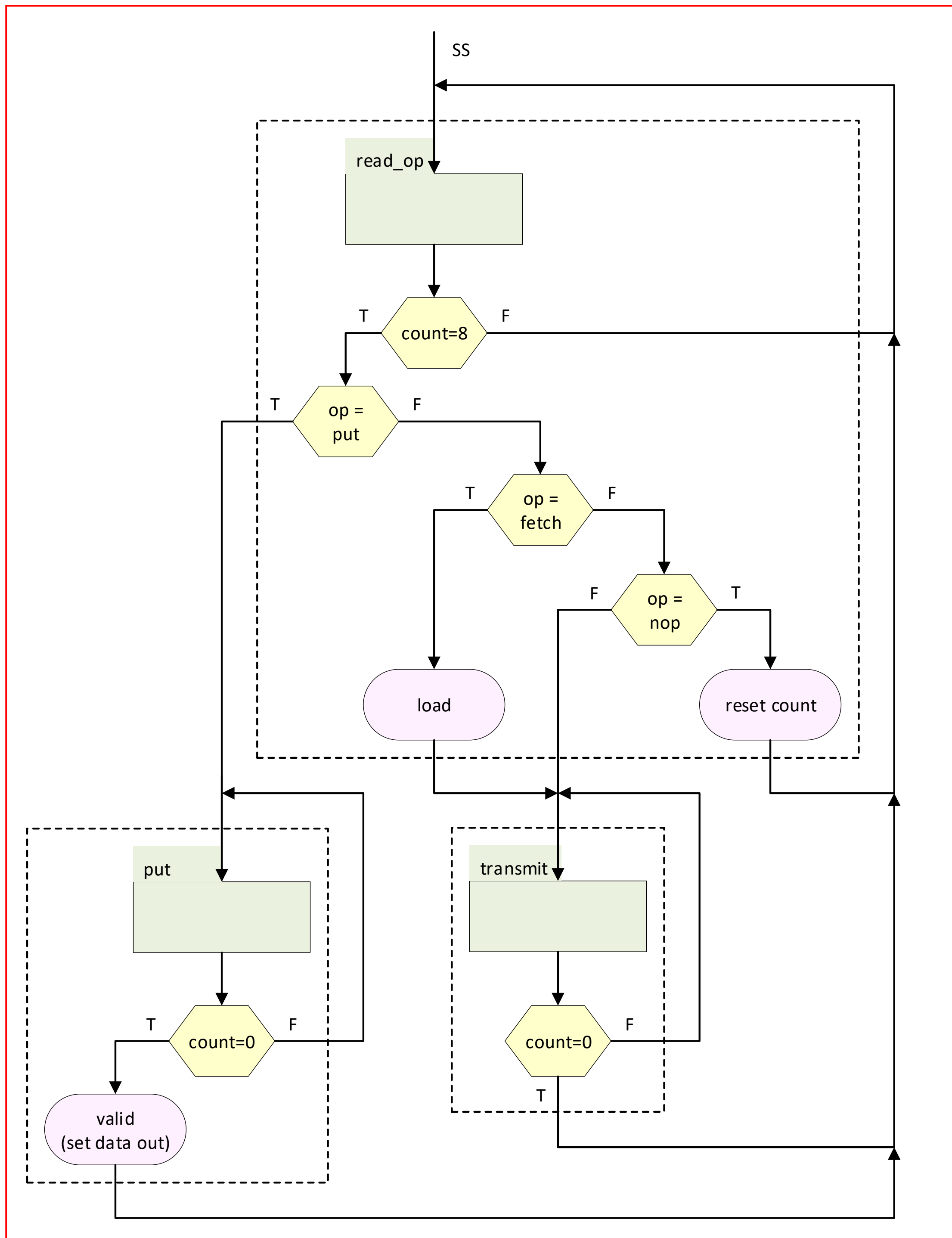
When given the Fetch command, the device shall fetch the data present at the input and send it over the SPI bus. When given the put command, the device shall put the next byte it receives over SPI, and set the valid flag when the whole byte is received. When given the Pass command, the next byte shall 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

Sensorveiledning :
Gyldig ASM: 5p, korrekt funksjon 5p, enkleste løsning 5p.



Typiske feil:

- Ikke sørge for at innganger på bokser er ovenfra
- Ikke sørge for at kun avgjørelsesbokser har utgang på siden.
- Ikke bruke defaultverdier (=> alle signaler defineres i alle tilstandsbokser)
- Bruke avgjørelsesboks for SS (unødvendig når det angis som resetsignal)

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 .

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.

Sensorveiledning:

- 1 Biblioteker (kun ieee std_logic_1164)
- 2 Komponentdeklarasjoner skal være 4
    - Sjekk at signaler er riktig
- 2 Signaler for interne signaler (navn kan variere):
    - `signal SCK_rise : std_logic;`
    - `signal load : std_logic;`
    - `signal mincount, halfcount : std_logic;`
    - `signal reset_count : std_logic;`
- 1 Generic mapping
- 2 Port-mapping
- 1 Skal ikke være kombinatorikk i denne module.
- 1 Skal ikke være unødvendige signaler

Typiske feil:

- Lage et signal "data" = unødvendig og kan medføre at data_out (fra entiteten) ikke mappes.
- Navngitt moduler for UUT (henviser til testbenkkode, «unit under test» men dette er ikke testbenk).

Mange har ikke å mappet generics, men det var ikke et krav i oppgaven

```vhdl
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

  component edge_detector
    port (
      clk             : in  std_logic;
      SS              : in  std_logic;
      SCK             : in  std_logic;
      SCK_rise        : out std_logic
    );
  end component;

  component shifter
    generic(WIDTH : natural := 8);
    port (
      clk             : in  std_logic;
      SS              : in  std_logic;
      SCK_rise        : in  std_logic;
      load            : 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)
    );
  end component;

  component counter
    generic(WIDTH : natural := 8);
    port (
      clk             : in  std_logic;
      reset           : in  std_logic;
      SCK_rise        : in  std_logic;
      mincount        : out std_logic;
      halfcount       : out std_logic
    );
  end component;
```

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

    signal SCK_rise              : std_logic;
    signal load                  : std_logic;
    signal mincount, halfcount   : std_logic;
    signal reset_count           : std_logic;

begin
  edge: edge_detector
    port map(
      clk => clk,
      SS => SS,
      SCK => SCK,
      SCK_rise => SCK_rise
    );

  shift: shifter
    port map(
      clk => clk,
      SS => SS,
      SCK_rise => SCK_rise,
      MOSI => MOSI,
      MISO => MISO,
      load => load,
      data_in => data_in,
      data_out => data_out
    );

  count: counter
    port map(
    clk => clk,
    reset => reset_count,
    SCK_rise => SCK_rise,
    mincount => mincount,
    halfcount => halfcount
    );


  state: fsm
    port map(
      clk => clk,
      SS => SS,
      load => load,
      mincount => mincount,
      halfcount => halfcount,
      reset_count => reset_count,
      data => data_out,
      valid => valid
    );

end architecture structural;
```

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.

Sensorveiledning:
1. Import av biblioteker 1p
2. Tom entitet 1p
3. Architecture skal være behavior eller annet. Det skal ikke være RTL/structural. 1p
4. Komponent-instansiering 2p
5. Signaldeklarasjon 2p
6. Port map. 2p
7. Riktig oppsett av klokke 3p
8. Bruk av prodedure 3p
9. Bruk av assert 3p
10. Bruk av report med failure 3p

Typiske feil:

- Biblioteker:
    - Glemt/ feil bibliotek
- Signaldeklarasjon
    - Sette defaultverdi på ouptuts fra enheten (maskerer flytende signaler ved startup).
- Bruke én prosedyre som som kjøres én gang i en prosess tyder på at man har misforstått oppgaven.
- Mange viser at de ikke har forstått designet ved at de forsøker å sende kommandoer over datalinjene og ikke SPI bussen. (Dette gir dysfunksjonell kode for å teste enheten).

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.TextIO.all;

entity tb_spi_top is
end tb_spi_top;

architecture behavior of tb_spi_top is

  -- Component Declaration

  component spi_top
  generic(WIDTH : natural := 8); -- denne er viktig at matcher shiftregisteret
    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 component;

  -- 100 MHz clk

  -- 10 MHz SPI_clk
  --- Sende spi kommando og data - prosedyre
  --- Sette interface data inn   - (filoperasjon?)

  -- Teste spi data ref kommando
  -- Teste spi data vs. data
  -- Bruke assertions og skrive utdata til fil.
  -- konkludere etter kjøring, om data



  -- generics common to design
  constant WIDTH          : natural := 8;

  -- timing constants
  constant HALF_PERIOD     : time := 5 ns;
  constant SPI_HALF_PERIOD : time := 50 ns;

-- Slave commands/operations
  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";

  -- signals driven by TB
  signal clk             : std_logic := '0';
  signal SS              : std_logic := '1';
  signal SCK, MOSI       : std_logic := '0';
  signal data_in         : std_logic_vector(WIDTH-1 downto 0)  := (others => '0');

  -- Signals driven by design (shall not have default value)
  signal MISO            : std_logic;
  signal valid           : std_logic;
  signal data_out        : std_logic_vector(WIDTH-1 downto 0);
```

```vhdl
begin
  UUT: spi_top
  port map(
      clk  => clk,
      SS   => SS,
      SCK  => SCK,
      MOSI => MOSI,
      MISO => MISO,
      data_in  => data_in,
      data_out => data_out,
      valid    => valid
  );

  -- master clock generation
  clk <= not clk after HALF_PERIOD;

  -- stimuli
  STIMULI: process is
    procedure send_spi_byte(byte: std_logic_vector) is
    begin
      for i in byte'reverse_range loop
        SCK <= '0';
        MOSI <= byte(i);
        wait for SPI_HALF_PERIOD;
        SCK <= '1';
        wait for SPI_HALF_PERIOD;
      end loop;
    end procedure;

    procedure fetch_data(data: std_logic_vector) is
    begin
      SCK <= '0';
      wait for SPI_HALF_PERIOD;
      SS <= '0';
      data_in <= data;
      send_spi_byte(OP_FETCH);
      send_spi_byte(OP_NOP);
      SS <= '1';
      wait for SPI_HALF_PERIOD*3;
    end procedure;

    procedure pass_data(data : std_logic_vector) is
    begin
      SCK <= '0';
      wait for SPI_HALF_PERIOD;
      SS <= '0';
      send_spi_byte(OP_PASS);
      send_spi_byte(data);
      send_spi_byte(OP_NOP);
      SS <= '1';
      wait for SPI_HALF_PERIOD*3;
    end procedure;

    procedure put_data(data: std_logic_vector) is
    begin
      SCK <= '0';
      wait for SPI_HALF_PERIOD;
      SS <= '0';
      send_spi_byte(OP_PUT);
      send_spi_byte(data);
      SS <= '1';
      wait for SPI_HALF_PERIOD*3;
    end procedure;
```

```
begin
    wait for HALF_PERIOD*10;
    fetch_data("11110000");
    fetch_data("11001100");
    put_data("10101010");
    put_data("01001000");
    pass_data("11111111");
    pass_data("10110111");
    report ("Testing finished!");
  std.env.stop;
  end process;

  REPORT_PUT:
  process is
  begin
    wait until valid = '1';
      report("                                             ",
          "  data_out when valid = ", to_string(data_out));
    wait until valid = '0';
  end process;

  REPORT_MISO:
  process is
    variable count : integer;
    variable MIS, MOS   : std_logic_vector(WIDTH-1 downto 0);
  begin
    count := 0;
    MIS  := (others => '0');
    MOS  := (others => '0');
    while count < WIDTH loop
      wait until SCK = '1';
      wait until clk = '0';
      wait until clk = '1';
        MIS(count) := MISO;
        MOS(count) := MOSI;
        count := count + 1;
      wait until SCK = '0' or SS = '1';
    end loop;
    report ("                                             ",
      " MOSI = ", to_string(MOS),
      " MISO = ",     to_string(MIS),
      " data_in = ",  to_string(data_in),
      " data_out = ", to_string(data_out));
  end process;

end architecture behavior;
```

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.)

(The SCK signal and the SS would need to synchronized using two flip-flops (2-FF). The 2-FF version of SS should be used throughout the design, and the 2-FF SCK should be used for input to the edge detector.
All synchronization shall occur only one place.)

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.

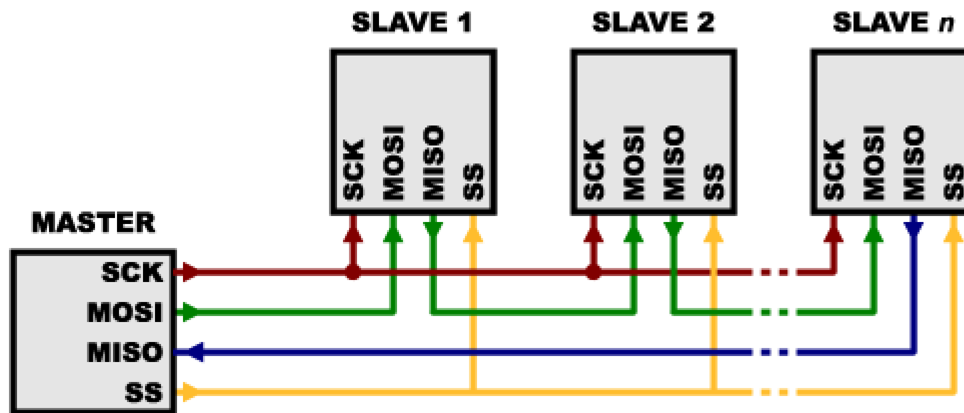


Figure 9: SPI devices in a series configuration

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

i. Will this setup work electrically, or is modification necessary (if so, how)?
(yes without modification (other than CDC from 7.)) 2p

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)
(put will set the same output on all devices, pass will be predictable, fetch will lead to random results, since data_in will be passed rather than the op_code. Data can also be an op-code) 2p

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 (impossible) 2p

ii. … set the ouput on all slaves (0x02, <putbyte>, 0x00, 0x00, 0x00) 2p

iii. … set the output on slave 2 only (impossible) 2p

iv. … test that all slaves communicate by passing the byte 0x01
(0x11, 0x01, 0x00, 0x00, 0x00, 0x00) 2p

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).
(64 slaves 6 bit => 2^6) 2p

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 .

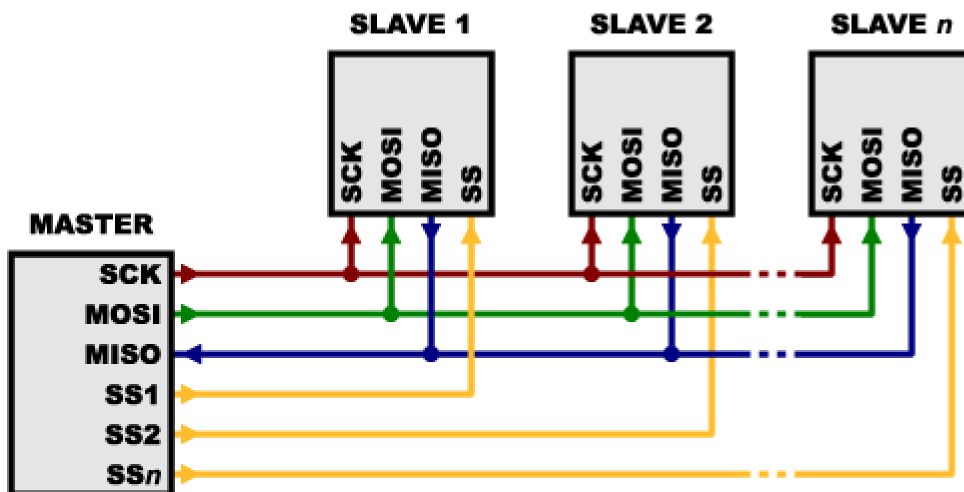


Figure 10: SPI devices in a parallel configuration

To what extent can they run without individual modifications…

a) Will this setup work electrically, or is modification necessary?
   No, MISO will need tristate functionality. 2p
b) Which commands can be used to provide meaningful result?
   All commands can be used. NOP is ok, but pointless 2p