UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Exam in: FYS4220/9220
Day of exam: 16 December 2015
Exam hours: 09.00 – 13.00
This examination paper consists of 5 pages.
Appendices: none
Permitted materials: none

Make sure that your copy of this examination paper is complete before answering.

Read all the text carefully before you start. All questions are weighted equally within a problem unless otherwise stated.

Problem 1 (25%):

a) Write the VHDL description for a d-flipflop (register) with an active high enable signal and no reset signal.

```
p_ff : process(clk)
begin
  if rising_edge(clk) then
    if enable = '1' then
      Q <= D;
    end if;
  end if;
end process;
```

b) Write a VHDL process that describes the digital circuit shown in Figure 1.? What is the main purpose/function of this circuit?

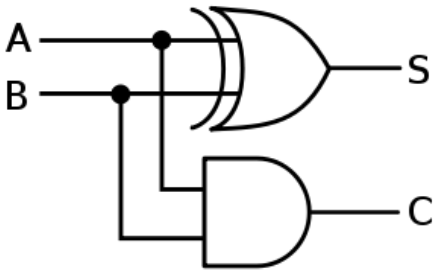


Figure 1

```
p_adder : process(A, B)
begin
  S <= A xor B;
  C <= A and B;
end process;
```

This is a half-adder with carry. It adds to bits A and B and has two outputs Sum (S) and Carry (C). The carry signal represents an overflow.

c) Explain how you make use of the process' sensitivity list for each of the two cases above, that is, the circuit in Figure 1 and the d-flipflop in *a)*.

The flip-flop is implemented using a clocked process. For a clocked process it is only the clock signal (clk) which should trigger the process and thus be in the sensitivity list.

The half-adder is implemented using a combinatorial process. For a combinatorial process all signals which are read by the process should trigger the process and thus be in the sensitivity list.

Additional information (not need for full score) In this case the process does not have a reset signal. For a clocked process with a reset signal, the reset signal should be in the sensitivity list only if it is an asynchronous reset signal, not if it is a synchronous reset signal.

d) Write the VHDL signal assignment(s) for the signal Y shown in Figure 2, in order to produce this waveform for use in a testbench.

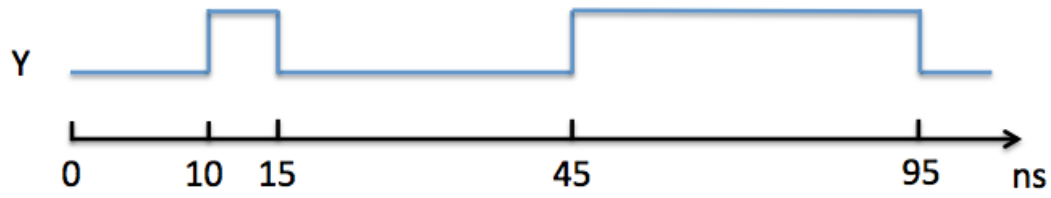


Figure 2

```
p_seq: process
begin

  -- alt1
  Y <= '0';
  wait for 10 ns;
  Y <= '1';
  wait for 5 ns;
  Y <= '0';
  wait for 30 ns;
  Y <= '1';
  wait for 50 ns;
  Y <= '0';
  wait;

  --alt 2
  Y <= '0',
       '1' after 10 ns,
       '0' after 15 ns,
       '1' after 45 ns,
       '0' after 95 ns;
  wait;

end process;
```

e) Figure 3 shows a simplified view of the programmable logic block in an FPGA. Can you explain the purpose of the Look-Up Table (LUT)?

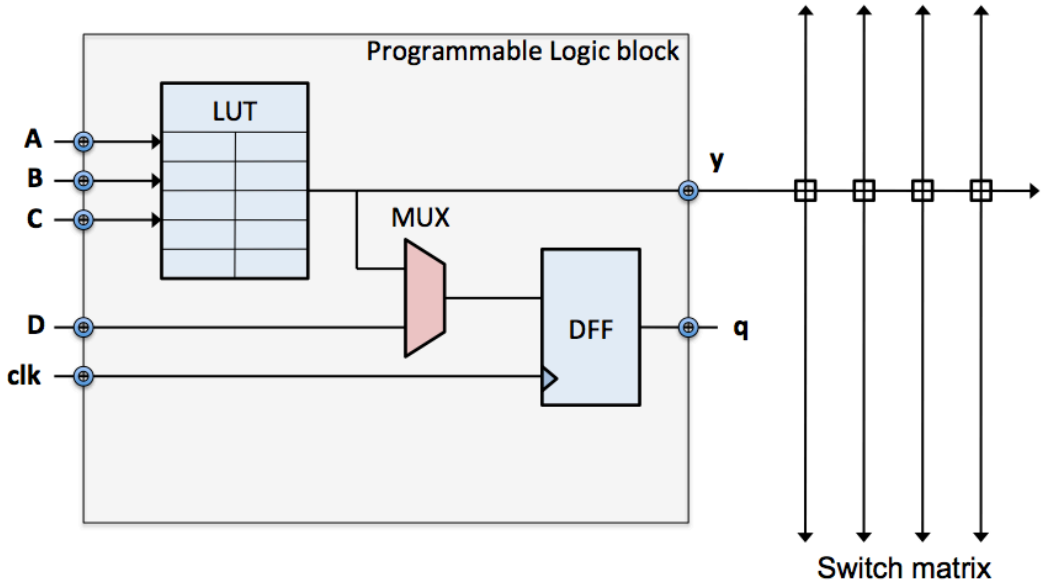


Figure 3

The LUT of a programmable block implements the combinatorial logic of your VHDL code. It is basically a table that determines what the output value is for any given input value. An LUT consist of a block of memory cells indexed by the inputs of the LUT.

Problem 2 (25%):

Pulse Width Modulation (PWM) is a common method to control the rotational speed of motors and fans. The basic principle of PWM is to have a train of pulses with a given period and duty cycle, as shown in **Figure 4**. The rotational speed is directly related to the duty cycle of the signal. A low duty cycle corresponds to a low rotational speed and a high duty cycle corresponds to a high rotational speed.

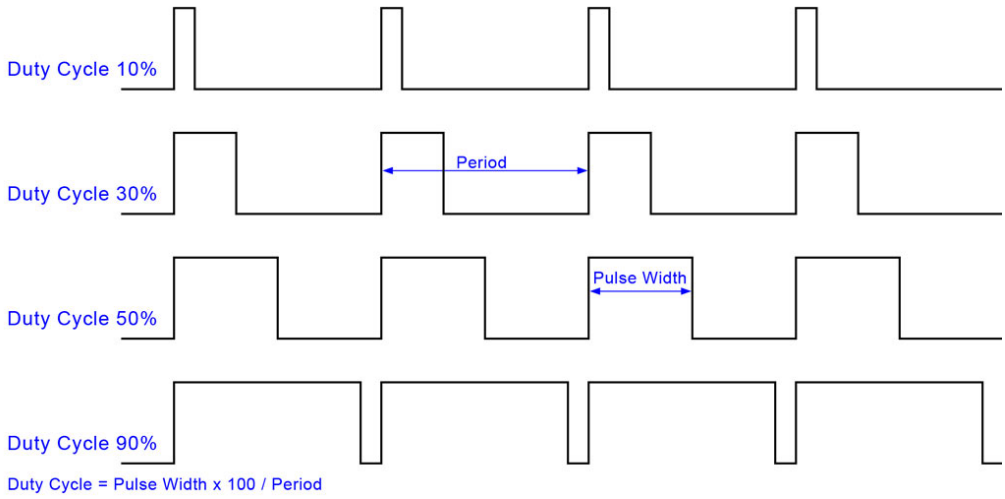


Figure 4

Figure 5 shows the top-level entity for a VHDL module for a PWM controlled fan. The signal ***fg_signal*** is a status signal from the fan which provides information about the rotational speed by sending out two positive pulses for each full rotation. Your task is to implement the VHDL description for this module according to the specification given in the following sub-problems:

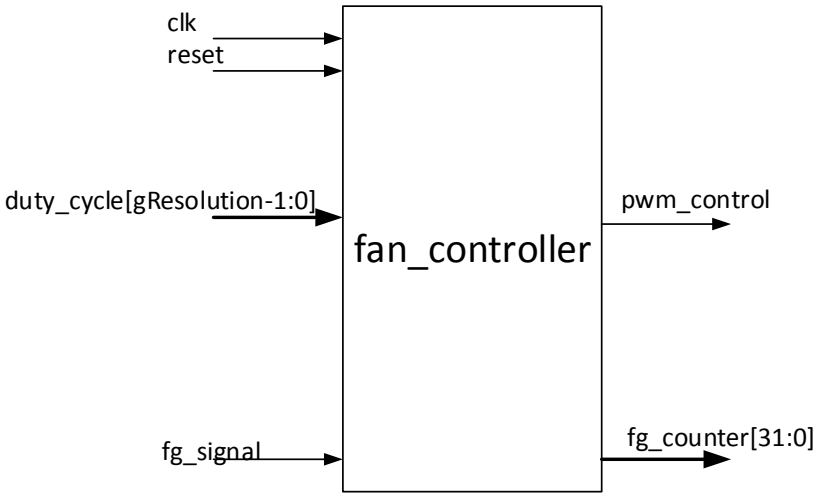


Figure 5

a) Write the VHDL code for the entity of the module **fan_controller** shown in Figure 5. All signals are of the type **std_logic** or **std_logic_vector**. **gResolution** is a **generic** input and specifies the width of the input vector **duty_cycle**. The default value of **gResolution** is 16.

```
entity fan_controller is
  generic (
    gResolution       : integer:= 16);
  port (
    -- general signals
    clk               : in    std_logic;
    reset             : in    std_logic;
    -- setting
    duty_cycle        : in    std_logic_vector(gResolution-1 downto 0);
    fg_counter        : out   std_logic_vector(31 downto 0);
    -- External outputs
    pwm_control       : out   std_logic;
    fg_signal         : in    std_logic);
end entity fan_controller;
```

b) In order to count the input pulses on the input signal **fg_signal,** this signal must be synchronized to the internal clock domain (system clock **clk**) of the **fan_controller**. Write a VHDL code for a process labelled **p_synch** that synchronizes this signal. Remember to define any additional signals you may need for this process.

```
architecture arch of fan_controller is
  signal pwm_counter            : std_logic_vector(gResolution-1 downto 0);
  signal fg_counter_i           : std_logic_vector(31 downto 0);
  signal fg_signal_sr           : std_logic_vector(2 downto 0);
  signal rising_edge_fg         : std_logic;
begin

  p_synch_fg: process (clk)
  begin
    if rising_edge(clk) then
      fg_signal_sr <= fg_signal_sr(1 downto 0) & fg_signal;
    end if;
  end process p_synch_fg;
```

c) Write the VHDL code for a process labelled **p_fg_counter** that counts the number of pulses detected on the input signal **fg_signal.** This process should be implemented with a synchronous active high reset signal. Remember to define any additional signals you may need for this process.

```
rising_edge_fg <= '1' when (fg_signal_sr(2) = '0' and fg_signal_sr(1) = '1') else '0';

p_fg_counter : process(clk)
begin
  if rising_edge(clk) then
    if (reset = '1') then
      fg_counter_i  <= (others => '0');
    elsif (rising_edge_fg = '1') then
      fg_counter_i  <= fg_counter_i + 1;
    end if;
  end if;
end process p_fg_counter;

fg_counter <= fg_counter_i;
```

d) Write the VHDL code for a process labelled **p_pwm** that

1. implements a counter **pwm_counter** which should be incremented for each clock period of the input signal **clk**. That is, a free running counter;
2. and in addition sets the output signal **pwm_control** to a high value (‘1’) when the value of the **pwm_counter** is lower than the value given by the input signal **duty_cycle**.

The process should be implemented with a synchronous active high reset signal. Remember to define any additional signals you may need for this process.
Assume that a complete period for the signal **pwm_control** corresponds to the time it takes for the **pwm_counter** to count from 0 to its maximum value (all bits equal to ‘1’).

```
-- counter for pwm
p_pwm : process (clk)
begin
  if rising_edge(clk) then
    if (reset = '1') then
      pwm_counter <= (others => '0');
      pwm_control <= '0';
    else
      pwm_counter <= pwm_counter + 1;
      if (pwm_counter < duty_cycle) then
        pwm_control <= '1';
      else
        pwm_control <= '0';
      end if;
    end if;
  end if;
end process p_pwm;
```

Problem 3 (25%):

The fan_controller module from Problem 2 will be implemented into an embedded system together with a Nios-II processor.

a) (5%) In order to add the **fan_controller** to the Nios-II based embedded system it needs to be modified into a memory mapped module. What does it mean that a module is memory mapped? Describe the modifications needed in order to make the module memory mapped to the Nios-II based system. It is not necessary to write the VHDL code to answer this question.

Memory mapped I/O access means that a modules/peripherals connected to the Avalon bus is mapped into the address space of the master (cpu). These modules can thus be accessed by writing and reading data to the respective address location in the master address space, in the same way as a standard memory.

In order to make the module memory mapped a bus wrapper must be added "around" the module. The bus wrapper contains an addressable register interface that can be connected to and communicate with the Avalon bus. This register interface is mapped into the address space of the master (cpu) and therefore addressable from the master (cpu).

b) (5%) When the **fan_controller** is added to the Nios-II processor in QSYS, should it be connected to the avalon data bus or to the avalon instruction bus, or both? Explain your answer.

It should be connected to the data bus. The instruction bus is just used between the Nios-II CPU and the memory which stores the program to be run by the processor, that is, the instruction which shall be executed. The data bus is just used to communicate data and not instructions. The fan controller does not provide any instructions.

c) (5%)Based on the register map shown in Table 1, the following driver macro functions should be implemented for the driver **fan_ctrl_regs.h** using the **#define** directive:

- FAN_CTRL_DUTY_CYLE_WR(data)
- FAN_CTRL_DUTY_CYCLE_RD
- FAN_CTRL_FG_COUNTER
- FAN_CTRL_VERSION

You can assume that the base address of the module is given by FAN_CTRL_BASE. Use the HAL functions IOWR and IORD to define the macro functions. You only need to write the specified macro function and not any other code of the header file.

Table 1

Register	Address	Type	Description
Duty_cycle[15:0]	0x1	Read/Write	This register can be read and written to. It

			sets the duty_cycle of the pwm_control signal.
Fg_counter[31:0]	0x2	Read	If read it returns value of the fg_counter
Version[7:0]	0x3	Read	If read it returns the value of the version register (0xA).

```
#define FAN_CTRL_DUTY_CYCLE_WR(data) IOWR(FAN_CTRL_BASE,DUTY_CYCLE_OFFSET,data)
#define FAN_CTRL_DUTY_CYCLE_RD       IORD(FAN_CTRL_BASE,DUTY_CYCLE_OFFSET)
#define FAN_CTRL_FG_COUNTER          IORD(FAN_CTRL_BASE,FG_COUNTER_OFFSET)
#define FAN_CTRL_VERSION             IORD(FAN_CTRL_BASE,VERSION_OFFSET)

#define DUTY_CYCLE_OFFSET 0x1
#define FG_COUNTER_OFFSET 0x2
#define VERSION_OFFSET    0x3
```

d) (10%) Assume that you have a function **read_temperature()** which returns a floating point value of the temperature from a temperature sensor. Write the c-code to adjust the rotational speed of the fan (change the duty_cycle) based on the temperature value read from the temperature sensor. If the temperature is higher than 50°C, the speed of the fan must increase by a single step. If the temperature is lower than 50°C, the speed of the fan must decrease by a single step. This should be repeated every second and never stop. Use the appropriate driver function(s) defined in c) to adjust the fan speed.

```
int main()
{
  float temperature = 0.0;
  int duty_cycle = 3;

  FAN_CTRL_RESET;

  FAN_CTRL_DUTY_CYCLE_WR(duty_cycle); //start value


  while(1)
    {
      temperature = read_temperature();
      if (temperature > 50){
        FAN_CTRL_DUTY_CYCLE_WR(++duty_cycle);

      }
      else if (temperature < 50) {
        FAN_CTRL_DUTY_CYCLE_WR(--duty_cycle);
      }
      usleep(1000000);
    }
}
```

Problem 4 (25%):

a) What is the main task of the scheduler in a Real-Time Operating System (RTOS)?

The main task of the scheduler is to determine which task is to run next.

b) What is the main difference between priority-based pre-emptive scheduling and Round Robin scheduling? Which of these two scheduling types is used by the uC/OS-II Real Time kernel?

In an RTOS with priority-based pre-emptive scheduling the order of task execution is determined by the task's priority. That is, a task with a high priority can pause/pre-empt a task with a lower priority, even if the lower priority task is not completed.
In an RTOS with Round Robin scheduling the tasks are executed in a cyclic order. Time slices are assigned to each task in equal portions handling all tasks without priority. A task is interrupted if it is not completed by the given available time, and it is resumed next time a slot is assigned to the task.

c) Can you explain the concept of priority inversion with respect to the uC/OS-II RTOS?

Priority inversion means that a high priority task has is pre-empted by a medium priority task effectively inverting the relative priorities of the two tasks. This is caused by a lower priority task which is blocking the usage of a shared resource (using a semaphore) also requested by the higher priority task. When a medium priority is scheduled to run, and this task is not requesting the use of the shared resource, it pre-empts the low priority task and effectively behaves as it has a higher priority relative to the high priority task.

d) What is a semaphore and why is it used in the example code shown in Figure 6?

A semaphore can be viewed as a key which is used to protect a shared resources. In order for a task to use a shared resource, it must first acquire the key. If the key is not available it has to wait until the key is available, that is, another task has completed to use the shared resource, before it can get access to it.
A semaphore can also be used to synchronize to events. By having one task posting the semaphore and a second task waiting for the semaphore, the execution of the second task can be synchronized with the first task.
In the example code of Figure 6 the semaphore is used to protect the access to the shared resource JTAG UART.

e) Make the necessary modifications to **task1** in order to post the temperature value using a mailbox in uC/OS-II. Assume that a mailbox already has been declared and created outside of the task. The mailbox stores the pointer to a floating point object.

```
//Declaration area:
//OS_EVENT *msg_box;
//In main:
//msg_box= OSMBoxCreate((void*)NULL);

void task1(void *pdata)
{
  INT8U return_code = OS_NO_ERR;
  float temperature = 0.0;

  while(1)
    {
      temperature = read_temperature();

      return_code = OSMboxPost(msg_box,(void*)&temperature);

      OSSemPend(shared_print_sem,0,&return_code);
      printf("Temperature: %.2f\n",temperature);
      OSSemPost(shared_print_sem);
      OSTimeDlyHMSM(0,0,1,0);
    }
}
```

f) Write the c-code for another task **task2** which receives the message from the mailbox and uses this to control the rotational speed of the fan_controller. Apply the same driver functions as in Problem 3. If the temperature is higher than 50°C, the speed of the fan must be increase by a single step. If the temperature is lower than 50°C, the speed of the fan must decrease by a single step.

```c
void task2(void *pdata)
{
  INT8U return_code = OS_NO_ERR;
  float temperature = 0.0;
  int duty_cycle = 3;

  FAN_CTRL_RESET;
  FAN_CTRL_DUTY_CYCLE_WR(duty_cycle); //start value

  float msg_rx;
  while(1)
    {

      msg_rx = (float*)OSMboxPend(msg_box,0,&return_code);
      temperature = *msg_rx;

      if (temperature > 50){
        FAN_CTRL_DUTY_CYCLE_WR(++duty_cycle);

      }
      else if (temperature < 50) {
        FAN_CTRL_DUTY_CYCLE_WR(--duty_cycle);
      }

      OSSemPend(shared_print_sem,0,&return_code);
      printf("Temperature: %.2f\n",temperature);
      OSSemPost(shared_print_sem);
    }
}
```

```c
void task1(void *pdata)
{
  INT8U return_code = OS_NO_ERR;
  float temperature = 0.0;

  while(1)
    {
      temperature = read_temperature();
      OSSemPend(shared_print_sem,0,&return_code);
      printf("Temperature: %.2f\n",temperature);
      OSSemPost(shared_print_sem);
      OSTimeDlyHMSM(0,0,1,0);
    }
}
```

Figure 6