UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Exam in: FYS4220/9220
Day of exam: 13 December 2016
Exam hours: 14.30 – 18.30
This examination paper consists of 8 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 indicated.

Problem 1 (25%):

a) VHDL divides the description of a module into an Entity and an Architecture section. What is the purpose of each of these two sections?

The Entity describes/provides the interface to the outside world. The Entity section specifies the name of the entity as well as the names and types of its input and output ports.

The architecture describes the functionality of the circuit. It contains either a structural or behavioral description of the entity.

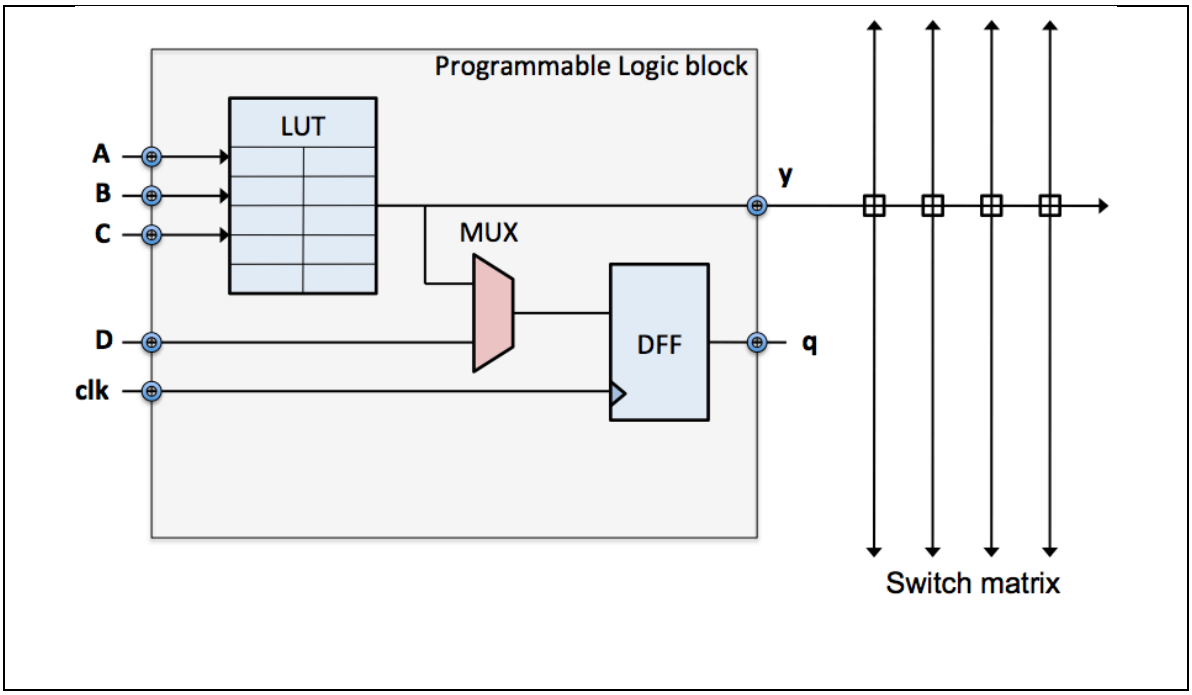
Essentially, the Entity section specifies the interface to the entity and the Architecture section specifies the functionality or composition of the entity.

b) Write the VHDL description for a process describing a d-flipflop (register) with an active high enable signal. The d-flipflop is reset independent of the enable signal using a synchronous active high reset signal.

```
p_ff : process(clk)
begin
  if rising_edge(clk) then
    if enable = '1' then
      Q <= d;
    end if;
    if reset = '1' then
      Q <= '0';
    end if;
  end if;
end process;
```

```
p_ff_alt : process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      Q <= '0';
    elsif enable = '1' then
      Q <= D;
    end if;
  end if;
end process;
```

c) An FPGA contains an array for programmable logic blocks. Draw a conceptual figure showing the most important parts of a programmable logic block.



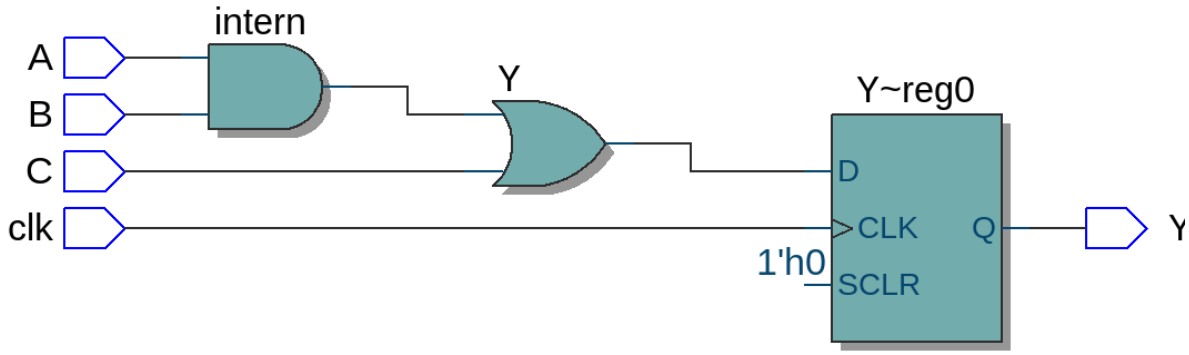
d) Draw the logical circuit diagrams for the two VHDL examples in Figure 1. Explain why the two examples generate different logic circuits. Remember to identify your diagrams by using the label name of the two processes.

```
p_A : process(clk)
  variable intern : std_logic;
begin
  if rising_edge(clk) then
    intern := A and B;
    Y      <= intern or C;
  end if;
end process;
```

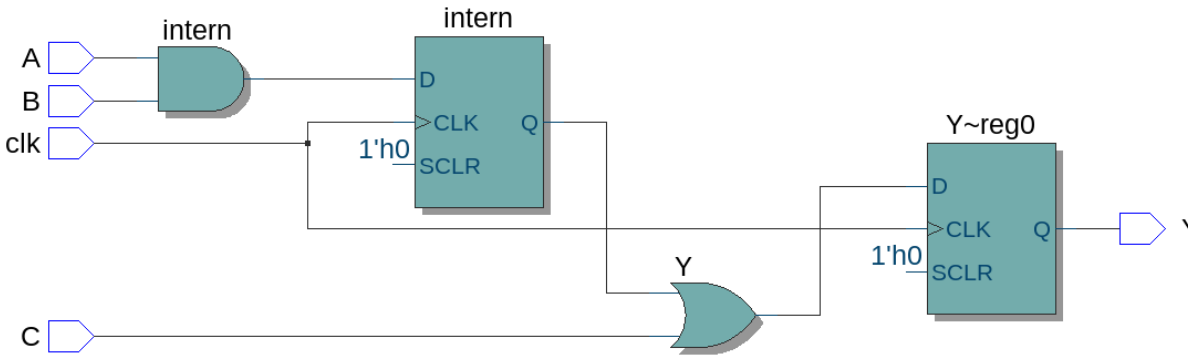
```
p_B : process(clk)
begin
  if rising_edge(clk) then
    intern <= A and B;
    Y      <= intern or C;
  end if;
end process;
```

Figure 1

p_A:



p_B:



When a signals or variable is assigned a value in VHDL, the value of the variable will be updated immediately while the value of the signal only will be updated at the end of the process when all statements have been evaluated. A signal assignment inside a clocked process will generate a register. A variable assignment will not, and therefore it will generate combinational logic only. However, if the variable is read before it is assigned a value, it will also generate a register.

e) How can you modify the VHDL description of process p_A in order to generate an identical logic behavior to process p_B while still using the process variable *intern*?

This can be done by simply changing the order of the two assignment lines. If a variable is read before it is assigned a value there is no information available about the value of the variable. The synthesizer therefore implements a register in order for the previously stored value to be used.

Problem 2 (25%):

A basic 8-bit serial transceiver protocol will be implemented in VHDL according to module entity shown in Figure 2 and the ASM state machine diagram shown in Figure 3. The task of the state machine is to read the 8-bit wide input vector *data* and transmit these 8 bits serially to the output *txd*.

- The input signal *baud_tick* is used to control the state transitions in the state machine.
- The input data is sampled and stored in an internal register *data_tx* when the both the input signals *load* and *baud_tick* are active.
- A low start bit (txd = 0) and a high stop bit (txd = 1) is used to indicate the beginning and end of the transmission.
- An internal signal *bit_cnt* is used to control which bit position of the internal register *data*_tx that should be assigned to the output *txd*. *bit_cnt* must be declared as a signal of type *integer* and with a valid range from 0 to 8.
- The state machine can be reset using the synchronous active high input signal *reset*.
- The state machine process is activated by the input clock signal *clk*.
- All the other input signals are already synchronous to the clock signal *clk*.

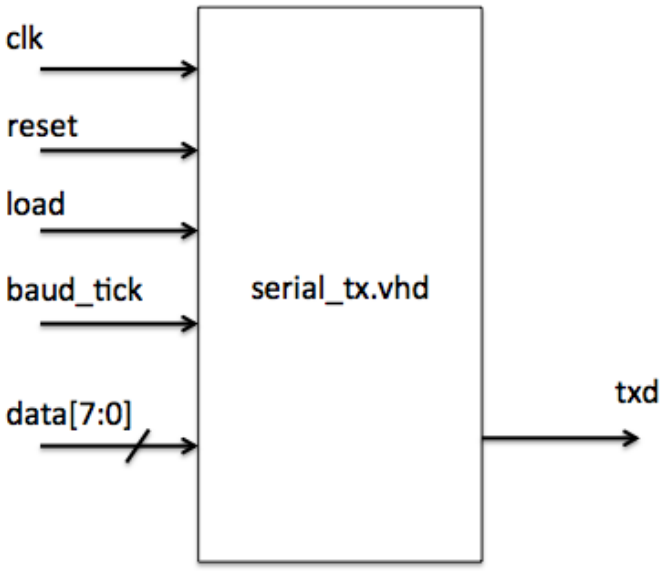


Figure 2: Top-level Entity of serial transceiver

a) (**5%**) Write the VHDL description for the Entity of the state machine. Include also relevant libraries.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity serial_tx is
  port (
    reset     : in  std_logic;
    clk       : in  std_logic;
    load      : in  std_logic;
    baud_tick : in  std_logic;
    data      : in  std_logic_vector(7 downto 0);
    txd       : out std_logic);
end serial_tx;
```

b) (**15%**) Write the VHDL description for the architecture part of the transceiver module *serial_tx*. The state machine must be implemented using a single process only.

```
architecture behavior of serial_tx is

  signal bit_cnt : integer range 0 to 8;
  signal data_tx : std_logic_vector(7 downto 0);

  type state_type is (sIDLE, sSTART, sSEND, sSTOP);
  signal state : state_type;


begin
  state_proc : process (clk)
  begin
    if rising_edge(clk) then
      case state is
        when sIDLE =>
          if baud_tick = '1' then
            if load = '1' then
              state <= sSTART;
              data_tx <= data;
            end if;
          end if;
          txd     <= '1';
          bit_cnt <= 0;

        when sSTART =>
          txd <= '0';
          if baud_tick = '1' then
            state <= sSEND;
          end if;

        when sSEND =>
          txd <= data(bit_cnt);
          if baud_tick = '1' then
            bit_cnt <= bit_cnt + 1;
            if bit_cnt < 7 then
              state <= sSEND;
            else
              state <= sSTOP;
            end if;
          end if;

        when STOP =>
          txd <= '1';
          if baud_tick = '1' then
            state <= IDLE;
          end if;

        when others =>
          null;
      end case;

      if reset = '1' then
        state <= sIDLE;
      end if;

    end if;
  end process clk_proc;

end behavior;
```

c) (**5%**) Assume that the input signal *load* is coming from another clock domain with a lower frequency than the clock signal *clk*. Explain how the *serial_tx* module needs to handle the *load* signal in this case.

The signal load then needs to be synchronized to the internal clock domain. This is achieved by feeding the signal through two registers which is running on the internal clock signal clk.

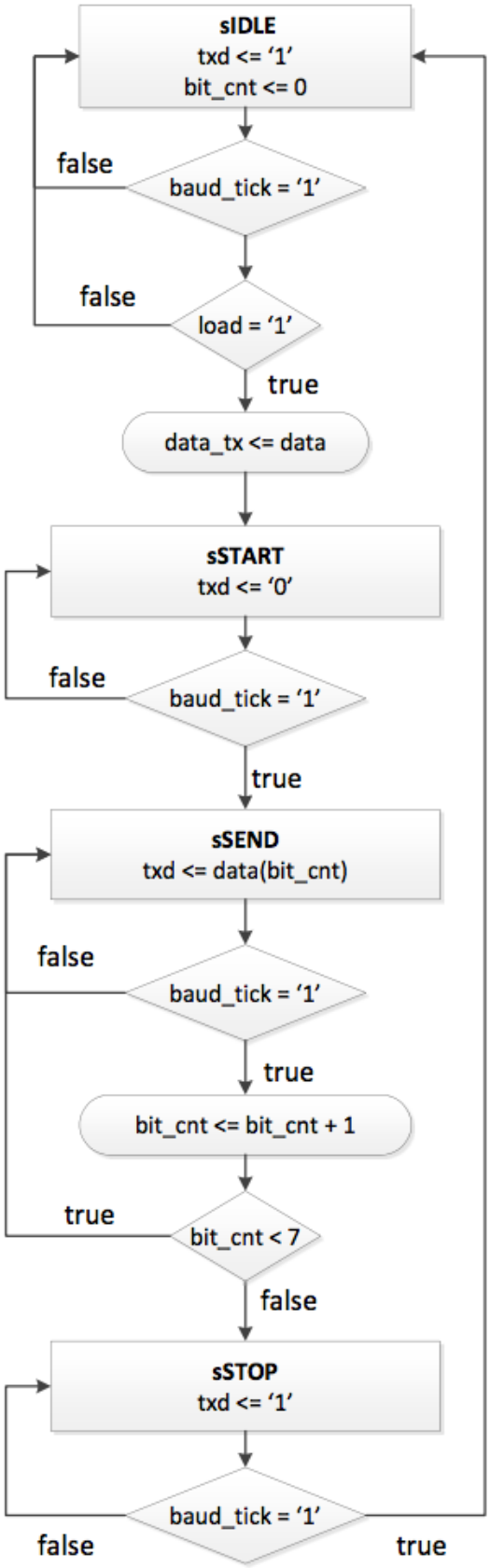


Figure 3: ASM state machine diagram for the serial transceiver

Problem 3 (25%):

a) Can you briefly explain what is meant by an embedded system?

An embedded system is a computing system designed to perform one or more specific functions with specific system constraints.

b) The NIOS-II CPU is a 32-bit general purpose RISC processor core with separate data and instruction master busses. Briefly explain the main difference between the data and the instruction busses? What is this type of architecture called?

The data bus is used to communicate data (and not CPU instructions) between to and from the connected devices.

The instruction bus is used by the CPU to fetch CPU instructions from the on-chip memory /instruction memory. This memory stores the program code to be run by the processor.

An architecture where the data and instruction bus is separated is called Harvard.

c) I2C is a two-wire interface for serial communication between components connected on the same interface. Which are the two main signal lines used for this bus interface, and how is a start of transaction and stop of transaction indicated on the I2C interface?

The two main lines are the data line (SDA) and the clock line (SCL). A start of transaction is indicated by a high to low transition on the data line while the clock line is high. A stop of transaction is indicated by a low to high transaction of the data line while the clock line is high.

d) Figure 4 shows an oscilloscope measurement of the beginning of a transaction on the two-wire I2C interface. Can you briefly explain what you see in this measurement example? What is the 7-bit address of the device being addressed in this transaction?

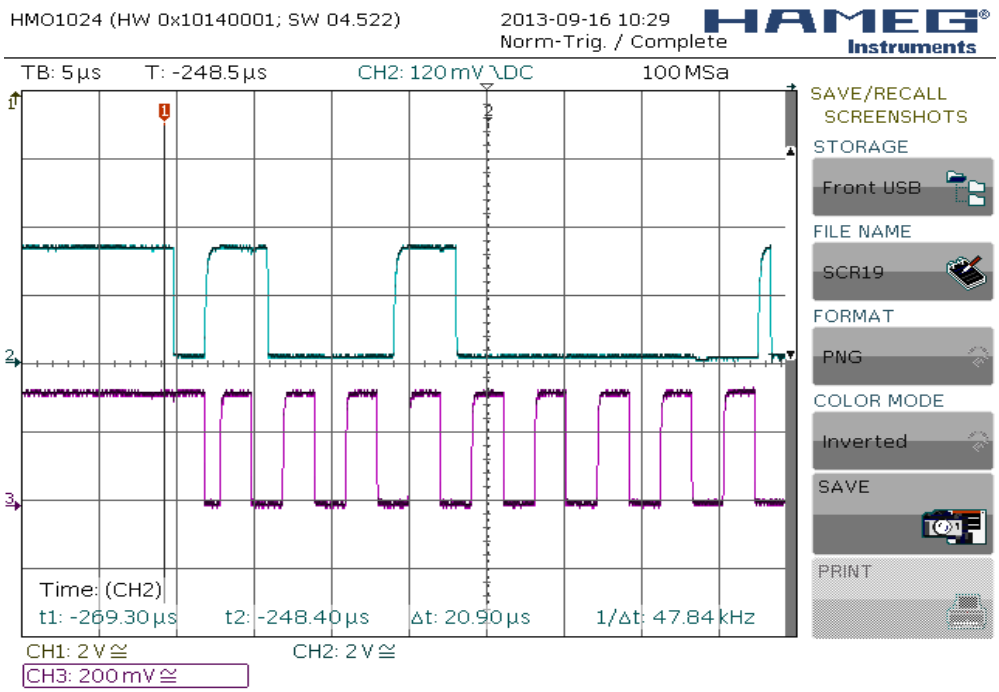


Figure 4: Oscilloscope measurement of an I2C two-wire interface

The top wave is the data line.
The bottom wave is the clock line.
This is the start of a transaction on the I2C bus. A start condition is issued by pulling the data line low while the clock line is high. The first 8 bits which are sent on the bus are the address of the I2C device (7-bit) plus an acknowledge bit. The acknowledge bit is low if the device being address is present on the bus and has correctly received and identified its address.

The address of the device is 0x48. The first 7 seven bit are the address and the 8th bit is the acknowledge bit.

e) Figure 5 and Table 1 shows the register interface of an IP module that is memory mapped to the address space of a NIOS-II microcontroller system. The base address of the module is already defined as:

#define I2C_AVALON_ADDR 0x00011000

This register interface is used control the communication with an external I2C device.

Control register (addr. 0x0)

Bits	31-4	3	2	1	0
Description	**Not in use**	**No. bytes**		**rnw**	**CMD**

Address register (addr 0x1)

Bits	31 - 16	15 - 8	7	6 - 0
Description	**Not in use**	**I2C register address**	**Not in use**	**I2C device address**

Data register (addr. 0x2)

Bits	31-16	15 - 0
Description	**Read data (from I2C)**	**Write data (to I2C)**

Figure 5: Register map

Table 1: Register map description

Register	Address	Type	Description
Control register[31:0]	0x0		Control register
[0]		Write	cmd: Write '1' to this bit position to start the transaction on the I2C bus. Automatically reset after one clock cycle.
[1]		Write	rnw = 1: Read from I2C device, rnw = 0: write to I2C device
[3:2]		Write	Number of bytes to be written to or read from I2C device.
[31-4]			Not in use
Address register[31:0]	0x1		Address register
[6:0]		Write	7-bit device address of the I2C device
[7]			Not in use
[15:8]		Write	Register address internal to the I2C device
[31:16]			Not in use
Data register[31:0]	0x2	Read	Data register
[15:0]		Write	Data to be written to I2C device
[31:16]		Read	Data read from I2C device

You will write the c-code for the function *write_to_i2c_device()*. The function is defined below:

void write_to_i2c_device(**alt_u32** i2c_device_addr, **alt_u32** i2c_reg_addr, **alt_u32** no_bytes, **alt_u32** data)

where:

- *i2c_device_addr*: is the 7-bit device address of the I2C device
- *i2c_reg_addr*: is the address of the register internal to the I2C device
- *no_bytes*: is the number of bytes to write to the I2C device
- *data*: is the data to be written to the I2C device

The IORD and IOWR macros in Table 2, which are provided by the NIOS-II Hardware Abstraction Layer (HAL), cab be used to communicate with memory-

mapped modules.

Table 2: Definition of IORD and IORW macros functions.

IOWR(BASE, REGNUM, DATA)	Write the value DATA to the register at the offset REGNUM in a device with base address BASE. Registers are assumed to be offset by the address width of the bus.
IOWR(BASE,REGNUM)	Read the value of the register REGNUM in a device with base address BASE. Registers are assumed to be offset by the address with of the bus.

You only need to write the c-code for the function write_to_i2c_device(). No other code is needed to answer the question.

```
void write_to_i2c_device(alt_u8 i2c_device_addr, alt_u8 i2c_reg_addr,alt_u8 no_bytes,alt_u32 data)
{
 //write to address register of i2c wrapper
  IOWR(I2C_AVALON_ADDR, 0x1, i2c_reg_addr << 8 | i2c_device_addr );

  //write to the data register of the i2c wrapper
  IOWR(I2C_AVALON_ADDR, 0x2, data & 0xffff);

  //Set the corresponding bits of the ctrl register to enable the i2c communication
  IOWR(I2C_AVALON_ADDR, 0x0, (no_bytes << 2) | 0x1);
}
```

Problem 4 (25%):

a) What is the most important difference between an embedded system with real-time requirements and an embedded system without real-time requirements, and what is a common misconception about real-time systems?

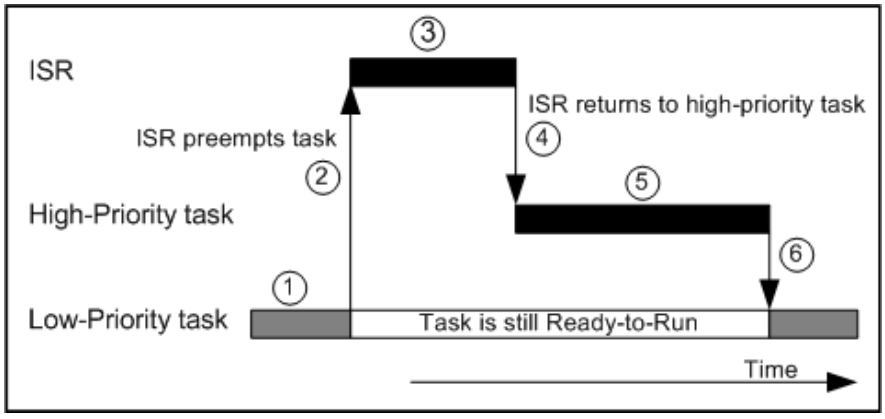
Real-time requirements means that the correctness of the system does not only depend on the logical result of the computation but also on the time when the result is generated.
A common misconception about real-time systems is that they provide better performance or are faster than other systems. This is not true. The objective of a real-time system is to meet the individual timing requirements of each task rather than being fast.

b) 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.

c) Explain the term preemption in relation to a real-time system using text and a simple diagram.

Preemption is the act of temporarily interrupting a task being carried out by the real-time kernel. If a task of higher priority is made ready to run, the current task running will be suspended, and the higher priority task is immediately given control of the CPU. When the higher priority task is completed, the execution of the lower priority task is resumed. This is also known as a context switch.



d) Explain what is meant by deadlock and how it can be avoided?

A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function. For example:

- Task T1 has exclusive access to Resource R1
- Task T2 has exclusive access to Resource R2
- If T1 needs exclusive access to R2 and T2 needs exclusive access to R1, neither task can continue.

Deadlock can be avoided by
- Acquiring all resources before proceeding
- Specifying a timeout when acquiring a semaphore
-

e) What is the difference between a semaphore and mutex?

Answer:
There are two main difference between a semaphore and a mutex:
- Ownership: a mutex can only be released by the tasks that owns it /acquired it. A semaphore can be released by any task. A semaphore can therefore be used to synchronize tasks.
- Priority inheritance: Since a mutex knows its current owner, it is possible to promote the priority of the owner whenever a higher priority task starts waiting for the mutex. A mutex can therefore be used to reduced the priority inversion problem.

f) Figure 6 shows two tasks which has been implemented as part of a uC/OS-II system. Can you explain what these tasks are doing and how they are related? Which two system services that have been used in this code?

The code reads data from an ADC with a repetition rate of 1 reading per 1000 system ticks, or when ever a message is received from task1. The channel address to read is increased everytime task1 is executed. The execution of task1 is synchronized to another

The code shows two tasks, task 1 and task2:
- Task1 waits for a synchronization semaphore new_addr_sem.
- When the synchronization semaphore is received, it increments the channel address and sends the pointer to the channel address variable as a message in an uC/OS-II mailbox.
- If the channel address reaches 10, it is reset to 0.
- Task2 uses a mailbox to synchronize/control its execution.
- When a message is received, it updates a local variable with the new channel address and calls the read_adc function.
- If the mailbox does not receive a new message within the specified timout of 1000 system ticks, the OSMboxPend returns, and the task continues by calling the read_adc function using the old/previous channel address.

The two system services which has been used is the
- semaphore
- message mailbox

```
void task1(void *pdata)
{
  INT8U err;
  int channel_addr = 0;
  while(1)
    {
      OSSemPend(new_addr_sem,0,&err);
      channel_addr = channel_addr + 1;
      err = OSMboxPost(msg_box,(void *)&channel_addr);
      if (channel_addr == 10)
        channel_addr = 0;
    }
}


void task2(void *pdata)
{
  INT8U err;
  int channel_addr;
  int *msg;
  int timeout = 1000;
  int data;

  while(1)
    {
      msg = (INT32U *) OSMboxPend(msg_box,timeout,&err);
      if (error_code != OS_NO_ERR)
        {
          channel_addr = *msg;
        }
      data = read_adc(channel_addr);
    }
}
```

Figure 6: uC/OS-II code example