UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Exam in **INF3430/4431**
Day of exam: **December 1. 2017**
Exam hours: **09.00-13.00**
This examination paper consists of 11 pages.
Appendices: **1**
Permitted materials: **All printed and written, as well as calculator**

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

The task text consists of tasks 1-5, 9-10 (multiple choice tasks) to be answered in the form attached to the text and tasks 6-8 and 11 that are answered on regular sheets.

The weight of each task is shown in parentheses behind the task number.

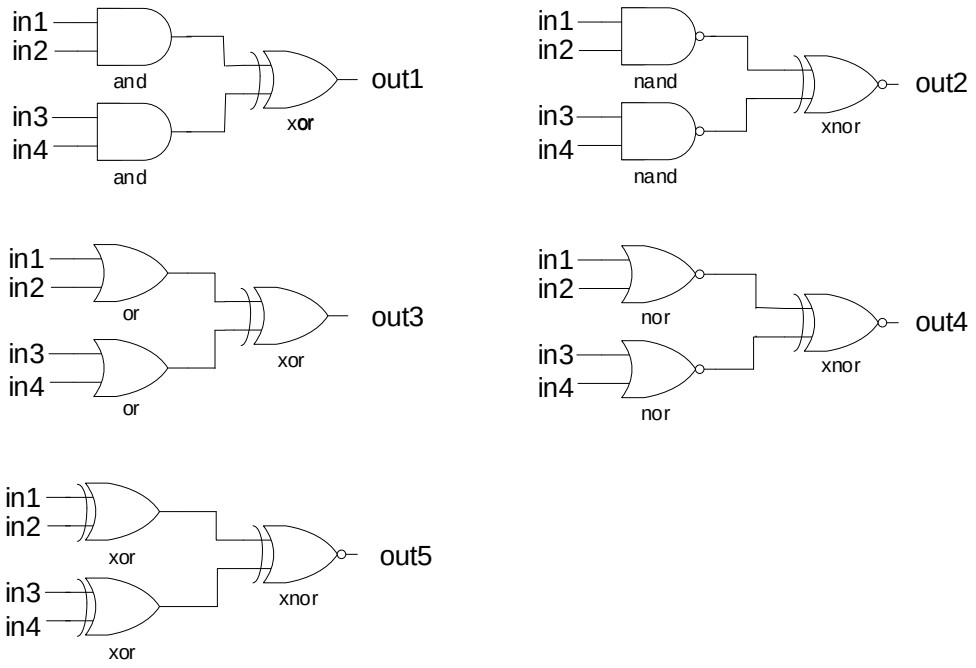
The tasks are independent so that they can be solved individually.

In the VHDL tasks, it is not necessary to repeat VHDL code in the answer which is already in the task text.

Generally for task 1-5, 9-10:
Each task consists of a theme and some statements each in an uppercase letter. The task is answered by marking a clear cross (X) in the right column for the correct answer option (i.e. one statements is true) in the form in Appendix 1. There is always at least one correct mark for each task, but there are often more correct marks. *To get the best grade you should put more crosses in a task if there are more correct statements.* There is 1 point for each cross where it should be a cross. It is given -1 points for each cross where it should not be a cross. Lack of a cross where it should be a cross also gives -1 points. You can use the right column in the task text for draft.
The form in Appendix 1 with your candidate number filled in is your answer.

Task 1 (3 %)

The figure below shows combinational circuits with out1 equal to "and-and-xor", out2 equal to "nand-nand-xnor", out3 equal to "or-or-xor", out4 equal to "nor-nor-xnor" and out5 equal to "xor-xor-xnor".



A 4-input Xilinx LUT with INIT value equal to "111E" (hex) implements:	A	and-and-xor	
	B	nand-nand-xnor	
	C	or-or-xor	
	D	nor-nor-xnor	
	E	xor-xor-xnor	

Task 2 (3 %)

FPGA technology	A	When an FPGA is configured, the state of Block RAM (BRAM) is unknown.	
	B	Xilinx Block RAMs cannot be removed from the FPGA to save space.	
	C	The Xilinx BRAM can be used as a ROM.	
	D	RAM can be made by Xilinx LUTs.	
	E	In VHDL-2008, the sensitivity list of a combinatorial process can be written as "all"; i.e. process(all).	

Task 3 (3 %)

FPGA technology	A	MicroBlaze is a hard processor core.	
	B	In Zynq-7000 circuits is it an ARM hard processor core.	
	C	A soft processor core requires less space than a hard processor core	
	D	BFM (Bus Functional Model) can only be used in testbenches.	
	E	Gigabit Transceivers is soft cores.	

Task 4 (3 %)

FPGA technology	A	It is easy to detect metastability by simulation.	
	B	After a time in metastable mode, all flip-flops will always return to '0'.	
	C	A BUFG module can only be used for clock signals.	
	D	Initial value after declaration of a signal of the type std_logic will be 'U'.	
	E	Two std_logic types with the values '0' and '1' which drives the same signal gets the value 'Z' in simulation.	

Task 5 (3 %)

Can variables in VHDL be declared in?	A	Entity	
	B	Architecture	
	C	Process	
	D	Function	
	E	Procedure	

Task 6 (8%)

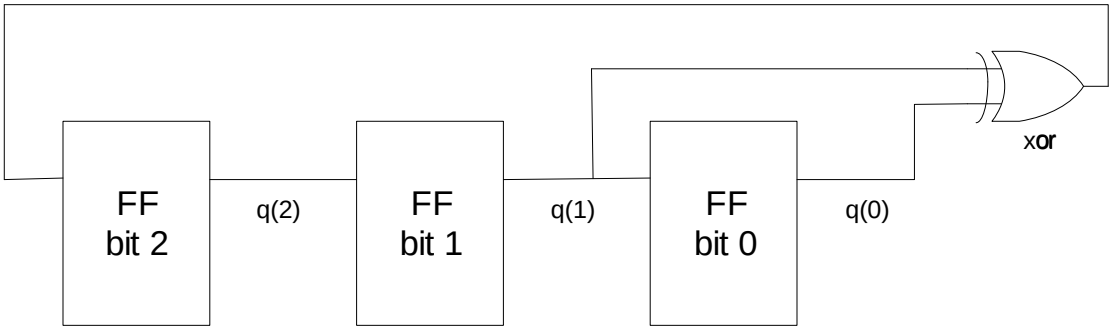
In this task, a MOORE type FSM will be created where the output q generates the numbers 4, 2, 5, 6, 7, 3, 1 and then generates the same sequence of numbers as long as the signal run is active high ('1'). When the signal run is low ('0'), the output q must keep the last generated number until the signal run becomes active high ('1') again and the generation continues. When reset signal rst is active high, output q must have the value 4 which is the first value in the sequence.

Implement the architecture of the arbitrary_sequence_gen module shown below in synthesizable VHDL.

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

entity arbitrary_sequence_gen is
  port
    (
      rst   : in  std_logic;
      clk   : in  std_logic;
      run   : in  std_logic;
      q     : out std_logic_vector(2 downto 0)
    );
end entity arbitrary_sequence_gen;

architecture rtl of arbitrary_sequence_gen is

  < Write the declarations here >

begin

  < Create state register process here >

  < Create the next state process here >

end architecture rtl;
```

Task 7 (10%)

In this task, a Linear Feedback Shift Register (LFSR) module with synchronous clocked xor feedback shall be designed with a 3 bit shift register as shown in the figure below.



Note that an xor operation is executed between bit 0 and bit 1, and the result of xor is entered at the input of bit 2 in the 3-bit shift register.

The LFSR module with this feedback will generate the same number sequence as in the previous task (i.e. 4,2,5,6,7,3,1, etc.) when the shift register starts the value commonly called "seed" equal to 4 (i.e. binary "100").

Different seed values shall be loaded into the register with the seed signal value when the signal load is active high (i.e. '1').

If the signal seed has the value zero (i.e. binary "000") which is an illegal value, a signal err will be active high until a new seed value is entered into the shift register when the load is active high again.

Implement the architecture to the LFSR module shown below in synthesizable VHDL.

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

entity lfsr is
  port
    (rst   : in  std_logic;
     clk   : in  std_logic;
     load  : in  std_logic;
     seed  : in  std_logic_vector(2 downto 0);
     run   : in  std_logic;
     q     : out std_logic_vector(2 downto 0);
     err   : out std_logic);
end entity lfsr;

architecture rtl of lfsr is

  < Declare the LFSR shift register here >

begin

  < Create the LFSR process here >

end architecture rtl;
```

Task 8 (8%)

In the module computing shown below, the sum of the numbers a, b, c and d is calculated as 16 bits. The output result with16 bits is set to max value equal to x"FFFF" (i.e. all bits set to '1') when the sum is greater than x"FFFF" and the signal max is set to '1' at the same time.

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

entity compute is
  port
    (rst    : in  std_logic;
     clk    : in  std_logic;
     a      : in  std_logic_vector(15 downto 0);
     b      : in  std_logic_vector(15 downto 0);
     c      : in  std_logic_vector(15 downto 0);
     d      : in  std_logic_vector(15 downto 0);
     result : out std_logic_vector(15 downto 0);
     max    : out std_logic);
end entity compute;

architecture rtl of compute is
begin

  process (rst, clk) is
    variable result_i : unsigned(17 downto 0);
  begin
    if rst = '1' then
      result <= (others => '0');
      max    <= '0';
    elsif rising_edge(clk) then
      result_i := unsigned("00" & a) + unsigned("00" & b) +
                  unsigned("00" & c) + unsigned("00" & d);
      if result_i > "00111111111111111" then
        result <= (others => '1');
        max    <= '1';
      else
        result <= std_logic_vector(result_i(15 downto 0));
        max    <= '0';
      end if;
    end if;
  end process;

end architecture rtl;
```

It turns out that there are timing errors during implementation in selected technology and with the selected clock frequency. The architecture rtl has to be changed to a new architecture pipelined_rtl that only has 1 add operation (i.e. + operator) and 1 comparison operation (i.e. the result_i > "00111111111111111" operation) in each clock period to achieve the timing requirement. Multiple add operations can be performed in parallel in each clock period.

Implement this new architecture pipelined_rtl as shown below to the compute module (i.e. with the same entity) in synthesizable VHDL.

```
architecture pipelined_rtl of compute is
begin

  < Implement the VHDL code here >

end architecture pipelined_rtl;
```

Task 9 (3 %)

How many registers (i.e. flip-flops) have the given architecture rtl in task 8	A	15 registers	
	B	16 registers	
	C	17 registers	
	D	18 registers	
	E	19 registers	

Task 10 (4 %)

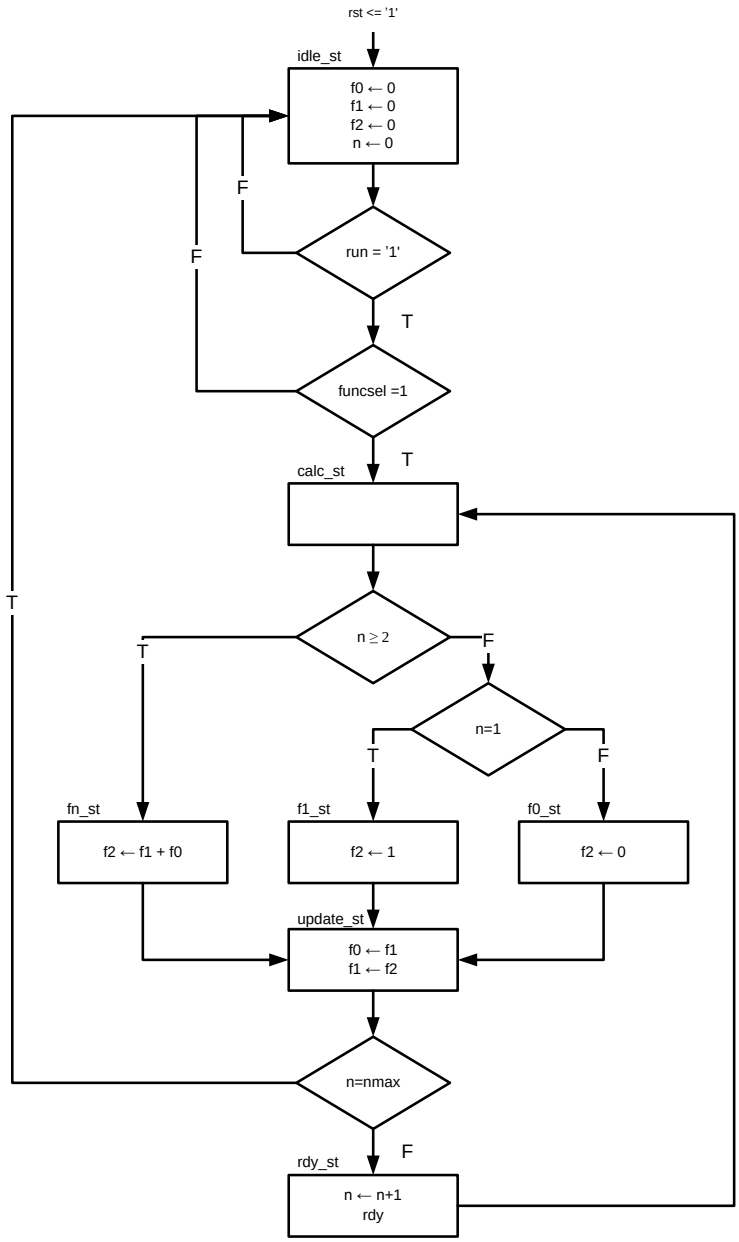
How many registers (i.e. flip-flops) have the pipelined architecture pipelined_rtl that was implemented in task 8?	A	51 registers	
	B	52 registers	
	C	53 registers	
	D	68 registers	
	E	69 registers	

Task 11

a) (15%)

The ASM-diagram below shows one possible implementation of a function which generates the wellknown Fibonacci number sequence:

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n \geq 2 \end{cases}$$



Implement the ASM-diagram above as a two-process state machine in synthesizable (RTL) VHDL. You must define your own additional internal signals after need.
The Fibonacci state machine has the following entity:

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

entity fibonacci is
  generic
    (
      FIBWIDTH : natural := 32 --# bits in the generated Fibonacci number
      );
  port
    (
      rst       : in  std_logic; --asynchronous reset
      clk       : in  std_logic; --clock
      run       : in  std_logic; --a run-puls of one clk cycle duration starts
                                 --the function generator
      funcsel   : in  std_logic_vector(2 downto 0); --function selection, "001"
                                                    --selects Fibonacci
      nmax      : in  std_logic_vector(7 downto 0); --max num of Fibonacci numbers
                                                    --which shall be generated
      inum      : out std_logic_vector(7 downto 0); --Fibonacci number index n
      rdy       : out std_logic; --a positiv puls of one clk cycle duration to
                                 --show that a new Fibonacci number n is ready
      fn        : out std_logic_vector (FIBWIDTH-1 downto 0) --Fibonacci number n
      );
end fibonacci;

architecture RTL_fibonacci of fibonacci is
  signal f0, next_f0 : unsigned(FIBWIDTH-1 downto 0);
  signal f1, next_f1 : unsigned(FIBWIDTH-1 downto 0);
  signal f2, next_f2 : unsigned(FIBWIDTH-1 downto 0);
  signal n, next_n   : unsigned(7 downto 0);

--<write more declarations here>

Begin

--<write the VHDL RTL-code here>

--Concurrent statements
  inum <= std_logic_vector(n);
  fn   <= std_logic_vector(f2);

End architecture RTL_fibonacci;
```

b) (15%)

Implement a self-checking test bench in VHDL which checks the correctness of the first 20 numbers of which the Fibonacci state machine generates

f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15	f16	f17	f18	f19
0	1	1	2	3	5	8	13	21	34	45	89	144	233	377	610	987	1597	2584	4181

You can assume that you have the first 20 (n=0-19) Fibonacci numbers stored in the test bench internal signal ***myfasit*** defined as:

```
constant MY_NMAX : natural:= 20;
type fibfasit is array (0 to MY_NMAX-1) of natural;
signal myfasit : fibfasit := (0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,
610,987,1597,2584,4181);
```

c) (6%)

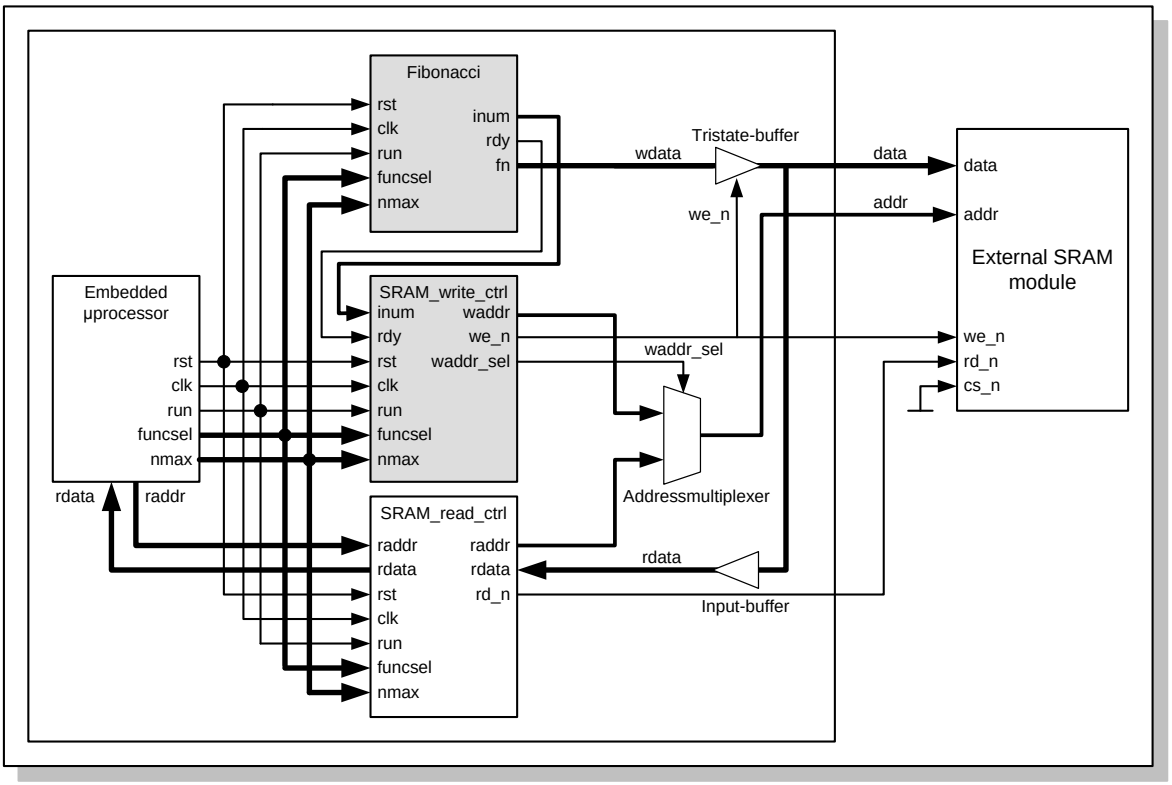
Modify the ASM-diagram to the state machine described in a) to a Mealy machine. Doing this shall reduce the latency in the state machine with two clock periodes.

d) (10%)

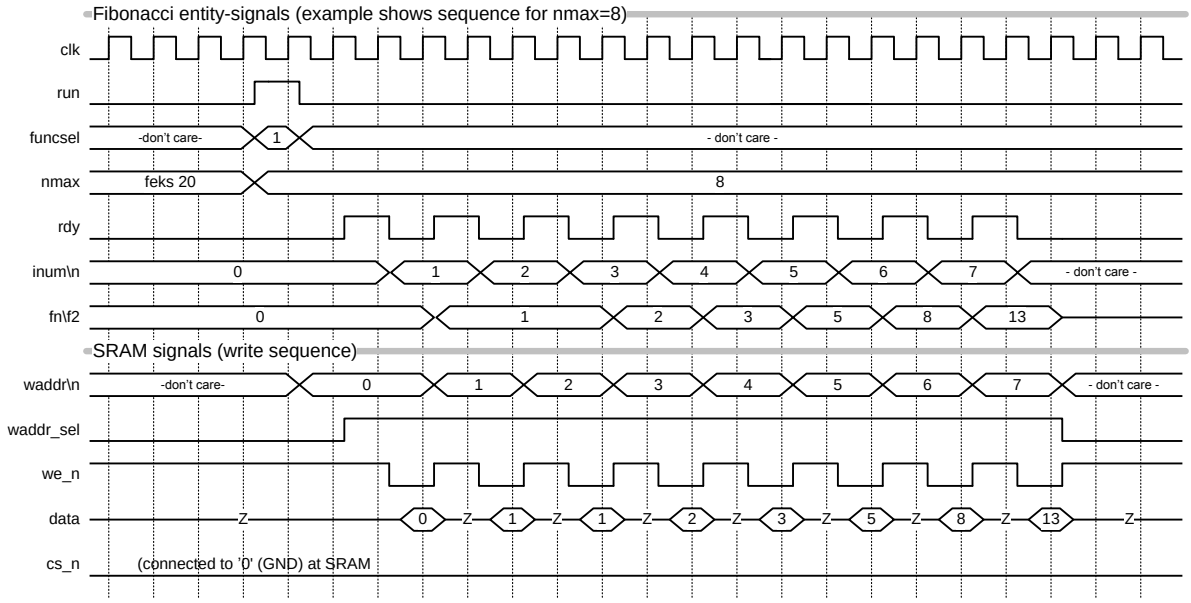
Make an ASM diagram as a Mealy state machine for the module ***SRAM_write_ctrl,*** see the figure below.

SRAM_write_ctrl ‘s main task is to store the Fibonacci numbers into a SRAM when they are generated. The Fibonacci index number ***inum,*** is used as the SRAM write address

Fibonacci, SRAM_write_ctrl and ***SRAM_read_ctrl*** (which should not be implemented) are all parts in system to make optimized functions in an embedded microprocessor system.



The signals ***rd_n***, ***we_n*** and ***cs_n*** are all active low.



Please notice that the above timing diagram follows the timing of the state machine described in 11c)

The behavior of SRAM_write_crtl can be summarized as follows:

1. The write-function is started when SRAM_write_ctrl checks that the input-signals ***run*** and ***funcsel*** are '1' and «001» respectively. Then the signal ***waddr_sel*** is set to active '1'. ***waddr_sel*** controls the address multiplexer such that ***waddr*** becomes the SRAM active address (please look at the figure on the page 10). It is very important that ***waddr_sel*** is active until all the Fibonacci numbers are stored.
2. Then it waits for the ***rdy***-signal to be active high as an indication of that a new Fibonacci number is ready to be stored
3. In the clock period after that ***rdy*** has gone active high, ***we_n*** shall go active low for one clock period. The data is then stored in the address ***waddr(addr)*** at the rising edge of ***we_n.*** In this state change the write address ***waddr***, is assigned ***inum*** (e.g. the next write address, see the timing diagram for details)
4. Item 2) og 3) is then repeated until all the Fibonacci numbers are stored, i.e. ***nmax*** number of write operations.

e) (6%)

Implement a process in VHDL which describes the tristate-buffer in the figure on page 10. You shall use ***wdata*** and ***we_n*** signals shown in the figure as inputs.

Appendix 1.
INF3430/INF4431. Answers for candidate #: _____

Oppgave	A	B	C	D	E
1					
2					
3					
4					
5					
9					
10					