IN3160 2022 Sensor guidelines

NOTE: These are sensor **guidelines**, the criterias used are not always 100% applicable to every answer for every candidate. *Ex: Code that are largely unfinished cannot get scored for readability, and there are many standards for readability, but everyone cannot be listed in bullet form*.

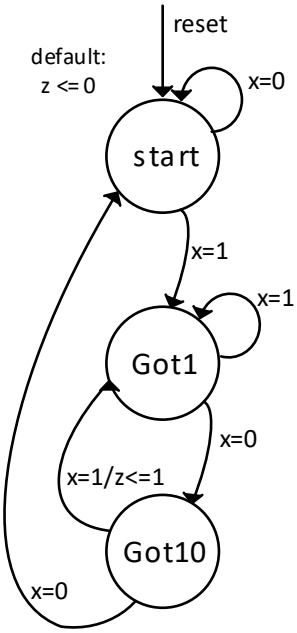
In the end, the grading is meant to reflect how well the candidate does meet the (minimum) learning outcome description found on the course webpage *:*

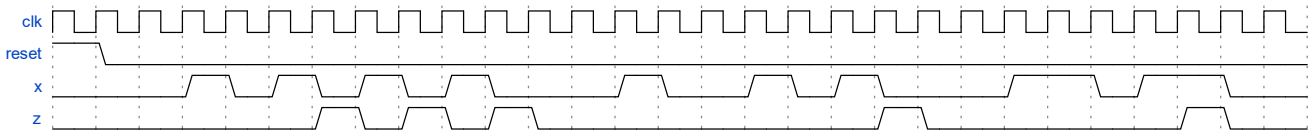
- *understand important principles for design and testing of digital systems*
- *understand the relationship between behavior and different construction criteria*
- *be able to describe advanced digital systems at different levels of detail*
- *be able to perform simulation and synthesis of digital systems.*

The learning outcome takes precedence in cases there are doubt whether the suggested ruling is suited to score an exercise or a specific answer.

Exercise 1 (5p)

The bubble state diagram above describes the behavior of a 101 sequence detector circuit.
Which of the following diagrams correspond to the behavior described in the state diagram?

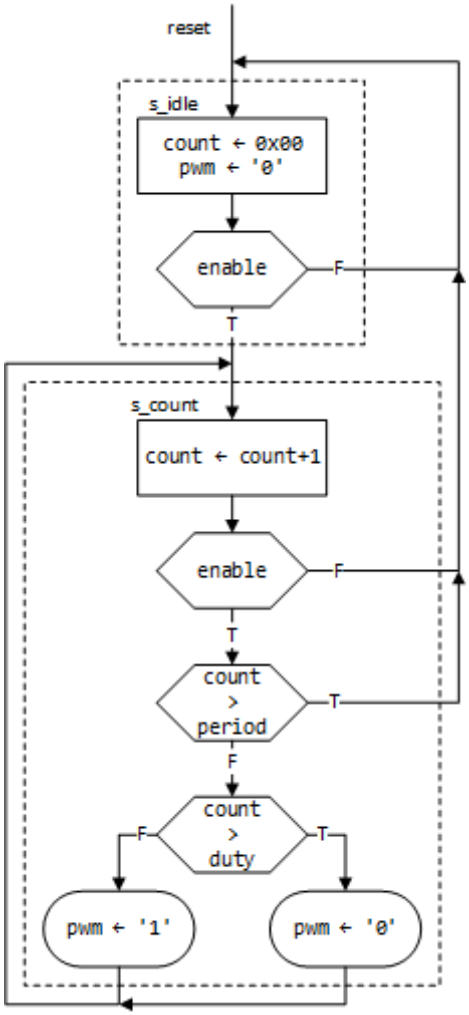


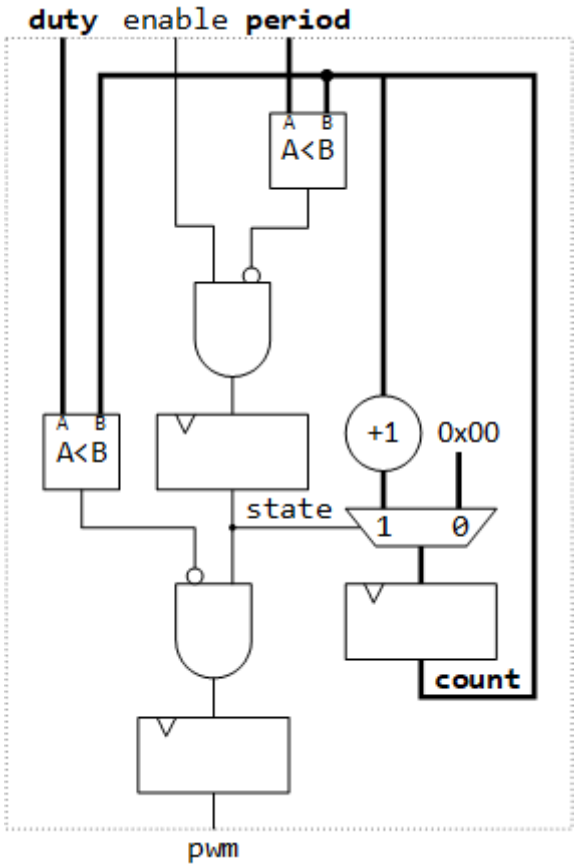


The 101 sequence detector will hoist the z flag for exactly one clock cycle, after the last ‘1’ in a “101” sequence has been completed. Eg. the sequence of 00…01010110 will flag z as follows 00…000010100.

Will be automatically corrected in Inspera

Exercise 2 (5p)





The ASMD diagram utilizes register operations for pwm and count, as indicated by their single arrow assignment "←" , thus the correct block diagram can be identified having registers for count, pwm in addition to the state registers (present in all state machines). After this assessment, the last wrong alternative can be identified by its excessive register usage, making the timing indicated in the diagram unachievable.

Will be automatically corrected in Inspera

Exercise 3 (24p)

Each statement should be individually identifiable as true or false.

Will be automatically corrected in Inspera

Exercise 4 (2p) + 5 (2p)

For a random (asynchronous) signal being stored in a register, the probability of error (P_{error}) is given by the following formula:

$$P_{error} = \frac{t_s + t_h}{t_{clk2}}$$

where t_s is the setup time and t_h is the hold time for the register used to store the value, and t_{clk2} is the period of the clock in the receiving domain.

A digital system reading a clock frequency of 50 MHz and 100ps setup and 100ps hold time.

What is the probability of error when reading a random asynchronous signal?

P_{error} = fclk2*(ts + th) = 50MHz*200ps = $50*10^6*200*10^{-12}$ =$10000*10^{-6}$ = 10^{-2} = 0,01

We use the system to read a signal from a quadrature encoder. The quadrature encoder has 1000 states per round.

What will be the average error frequency when the motor spins at 3000 rpm?

$f_{error} = P_{error}*f_{clk1}$ = 10^{-2}*3000rpm*1000/(60rpm/Hz) = 500 Hz

The autocorrect in Inspera was overridden in 2022, due to error in the autocorrect input:

- The correct answer to assignment 4 is 0,01.
    - Correct answer shall be overridden with full score which is 2 points
    - No partial scores available, (ie either 0 or 2 points)
- The correct answer to assignment 5 is 500.
    - Correct answer shall be overridden to award 2 points.
    - 1 point shall be awarded for answers to assignment for wrong answers that are 50.000 times that of assignment 4
        - While the answer technically is incorrect, the calculation is then correct, since the candidate is using the previous answer as a base for their calculation.

Exercise 6 (2p)

Brute force is acceptable for Quadrature encoder (and Gray code in general), since only one signal is allowed to change at a time, at any given time. Thus there is no n-bit problem to avoid.

This will be automatically corrected by Inspera.

Exercise 7 (10p)

In this assignment, the module compute_seq which computes *a+b+c+d* shall be implemented. The signals *a,b,c* and *d* are synchronous to the clock signal. The entity *compute_seq* is listed below.
The computation shall be using unsigned arithmetic operation on the operands *a, b, c* and *d*. The output result shall have the required number of bits such that the result is always correct, and overflows do not occur.
Implement the architecture to the module compute_seq entity as listed under as a sequential process in synthesizable VHDL. The implementation shall use one clock cycle to compute the result. Change the signal result in the entity *compute_seq* such that the declaration of *std_logic_vector* has the correct number of bits.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity compute_seq is
   ¶  port
  (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(XX downto 0)); -- XX shall be changed
  end entity compute_seq;
```

```
architecture rtl of compute_seq is
  signal next_result : unsigned(17 downto 0);
begin
  next_result <=
    ("00" & unsigned(a)) + ("00" & unsigned(b)) +
    ("00" & unsigned(c)) + ("00" & unsigned(d));
  result <= std_logic_vector(next_result) when rising_edge(clk);
end architecture rtl;

-- alternative using process:
architecture rtl of compute_seq is
begin
 process (clk) is
   variable next_result: unsigned(17 downto 0);
 begin
   if rising_edge(clk) then
     next_result :=
        ("00" & unsigned(a)) + ("00" & unsigned(b)) +
        ("00" & unsigned(c)) + ("00" & unsigned(d));
     result <= std_logic_vector(next_result);
   end if;
 end process;
end architecture rtl;
```

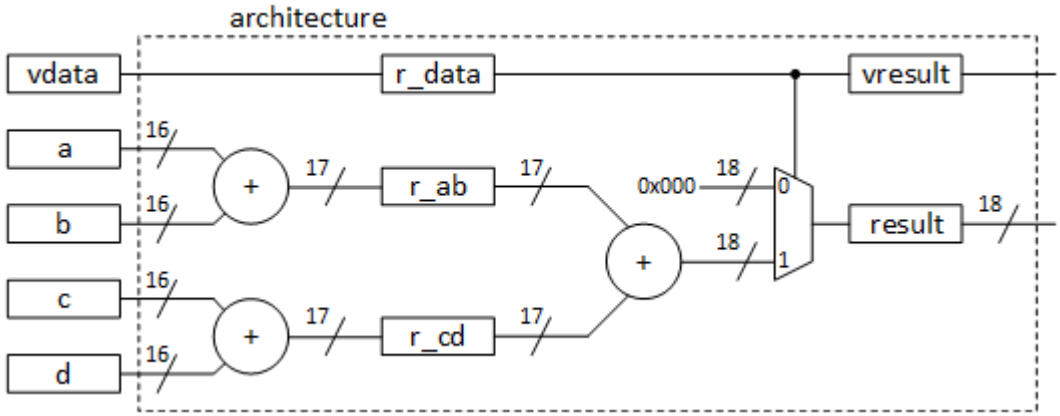
/2p Assignment operator
/2p Correct usage of unsigned/1p converting to result to std_logic vector
/1p Correct number of bits for XX:
/1p padding using “00”
/1p Single clock cycle
/1p correct usage of sensitivity in processes or just “when rising_edge(clk)” for single statement reg.
/2p synthesizable VHDL

Exercise 8 (10p)

The compute_seq implementation does not meet timing closure with the selected technology and the required clock frequency. Therefore the design has to be pipelined.
In additon, the signal vdata determines when the inputs *a, b, c* and d has valid data, i.e.. *vdata='1'* when *a,b,c* and *d* are valid. The output signal *vresult* shall be '0' when the output result is not valid and '1' when the output signal result is valid. When the output signal result is not valid the value shall be '0'. The pipelined architecture shall allow new data each clock cycle.
Draw a datapath diagram of the of two stage pipeline architecture as according the the entity listed below.

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
entity compute_seq is
  port(
    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);
    vdata : in std_logic;
    vresult : out std_logic;
    result : out std_logic_vector(XX downto 0)); -- XX shall be changed
  end entity compute_seq;
architecture rtl of compute_seq is
-- implement the compute_seq process
end architecture rtl;
```

In this exercise you can answer with digital hand drawing. Use your own sketch sheet (distributed). See instructions for filling in the sketch sheet in the link below the task bar.



/2p Pipelining of v_result <= rdata, rdata <= vdata (names can be anything)
/2p r_ab = a+b, r_cd = c+d (or similar)
/2p result = r_ab + r_cd when r_data (eg. using mux or and)
/2p vector_sizes shown and correct
/2p Two stage pipeline with single adder sequence per stage

Exercise 9 (10p)

Implement the architecture from the previous assignment in synthesizable VHDL.

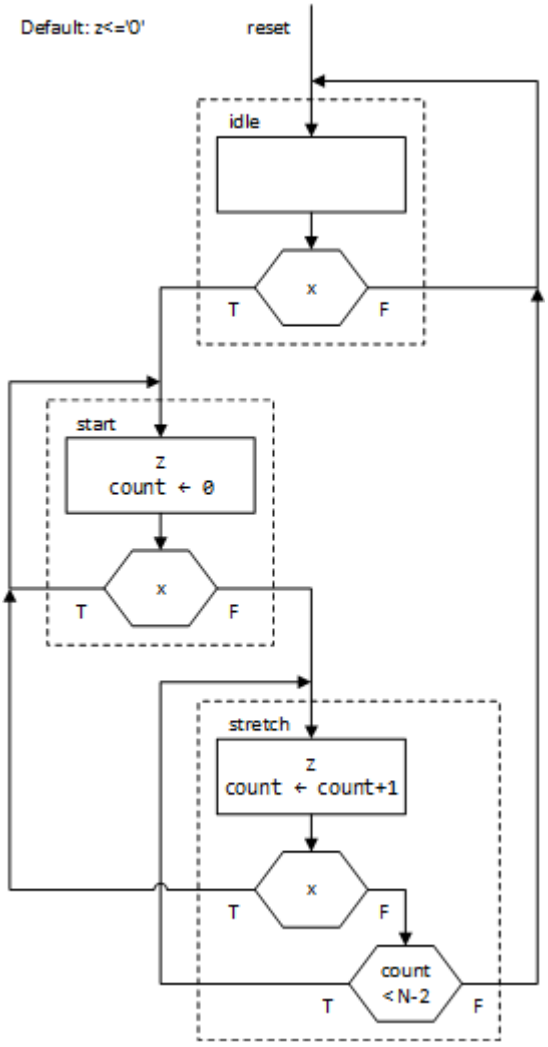
```
architecture rtl of compute_pipelined is
  signal r_data : std_logic;
  signal next_ab, next_cd, r_ab, r_cd : unsigned(16 downto 0);
  signal next_result : unsigned(17 downto 0);
begin
  --combinational logic
  next_ab <= ("0" & unsigned(a)) + ("0" & unsigned(b));
  next_cd <= ("0" & unsigned(c)) + ("0" & unsigned(d));
  next_result <= ("0" & r_ab) + ("0" & r_cd) when r_data else (others => '0');

  REG_ASSIGNMENT: process (clk) is
  begin
    if rising_edge(clk) then
      r_data <= vdata;
      vresult <= r_data;
      r_ab <= next_ab;
      r_cd <= next_cd;
      result <= std_logic_vector(next_result);
    end if;
  end process;
end architecture rtl;


-- alt. architecture using variable
architecture rtl of compute_pipelined is
  signal r_data : std_logic;
  signal r_ab, r_cd : unsigned(16 downto 0);
begin
  process (clk) is
    variable next_result: unsigned(17 downto 0);
  begin
    if rising_edge(clk) then
      r_ab <= ("0" & unsigned(a)) + ("0" & unsigned(b));
      r_cd <= ("0" & unsigned(c)) + ("0" & unsigned(d));
      next_result := ("0" & r_ab) + ("0" & r_cd) when r_data else (others => '0');
      result <= std_logic_vector(next_result);
    end if;
  end process;
end architecture rtl;
```

/2p Correct usage of signals and variables (if any)
/2p Correct type conversion and vector sizes
/2p Code according to schematic in assignment 8 (or assignment text)
/2p Readability : indentation, spacing, comprehensible code sequence
/2p Single clk edge check used for signals that are registers only

Exercise 10 (10p)



/4p Valid ASM or ASMD
* only decision box splits path.
* Single entry for each state
* state exit paths goes into state box
* correct drawing: all state entry from top, decision box exit on side or below.
/1p Moore type state machine (No mealy boxes present)
/2p Solves the task for N=2 (N=3 solution => 1 point)
/2p Solves the task for any N (ie counter based solution) –N+-1 is 1 point
/1p Optimal Moore solution (3 states, 4 decisions)

Note: This diagram works for N>=2. If this was intended for N=1, a check for N=1 would be required in the start state -to allow the state to exit directly to idle when x is not active and N<2.

Exercise 11 (10p)

Suggested solution:

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

entity pulse_stretch is
  generic( N: integer := 2);
  port(
    clk, reset : in std_logic;
    x : in std_logic;
    z : out std_logic);
  end entity pulse_stretch;

architecture rtl of pulse_stretch is
  type state_type is (idle, start, stretch);
  signal state, next_state : state_type;
  signal count, next_count : integer;
begin

  REG_ASSIGNMENT: process (clk) is
  begin
    if rising_edge(clk) then
      state <= idle when reset else next_state;
      count <= 0 when reset else next_count;
    end if;
  end process;

  STATE_ASSIGNMENT : process(all) is
  begin
    case state is
      when idle =>
        next_state <= start when (x = '1') else idle;
      when start =>
        next_state <= start when (x = '1') else stretch;
      when stretch =>
        next_state <= start when (x = '1') else stretch when (count < N-2) else idle;
      when others =>
        next_state <= idle;
    end case;
  end process;

      -- can be done with STATE_ASSIGNMENT for two-process FSM.
  STATE_BASED_ASSIGNMENT: process(all) is
  begin
    next_count <= 0;
    z <= '0';
    case state is
      when idle => null; -- use default
      when start =>
        z <= '1';
      when stretch =>
        z <= '1';
        next_count <= count + 1;
      when others => null; -- use default
    end case;
  end process;
end architecture rtl;
```

/2p Libraries correspond to usage (normally IEEE - std_logic and numeric_std)
/2p Synchronous reset (correct asynch reset = 1p, no or useless reset implementation = 0p)
/2p Two/ three process state machine (valid one process = 1 p)
/1p Generic N implemented correctly (integer/natural or unsigned is OK)
/1p Correct implementation (ie deduct for N+1 or N-1, or other flaws)
/1p Readability: reasonable layout, indentation
/1p Synthesizable (simulation only code => 0, syntax is not an issue as long as it can be understood)

Exercise 12 (10p)

Code suggestion:

```
library ieee;
  use ieee.std_logic_1164.all;
  use std.env.stop;

entity tb_pulse_stretch is
end entity tb_pulse_stretch;

architecture behavioral of tb_pulse_stretch is

  procedure send_pulse(
    signal clk  : in std_logic;
    test_vector : in std_logic_vector;
    signal x    : out std_logic
  ) is
  begin
    for i in test_vector'range loop
      wait until rising_edge(clk);
      x <= test_vector(i);
    end loop;
  end procedure;

  component pulse_stretch is
    generic (
      N : integer := 2
      );
    port
      (clk   : in  std_logic;
       reset : in  std_logic;
       x     : in  std_logic;
       z     : out std_logic);
  end component;

  signal clk   : std_logic := '0';
  signal reset : std_logic;
  signal x     : std_logic;
  signal z     : std_logic;

  constant N : integer := 2;
  constant HALF_PERIOD : time := 20 ns;

begin
  UUT : pulse_stretch
    generic map (
      N => N
      )
    port map (
      clk   => clk,
      reset => reset,
      x     => x,
      z     => z
      );

  clk <= not clk after HALF_PERIOD;
```

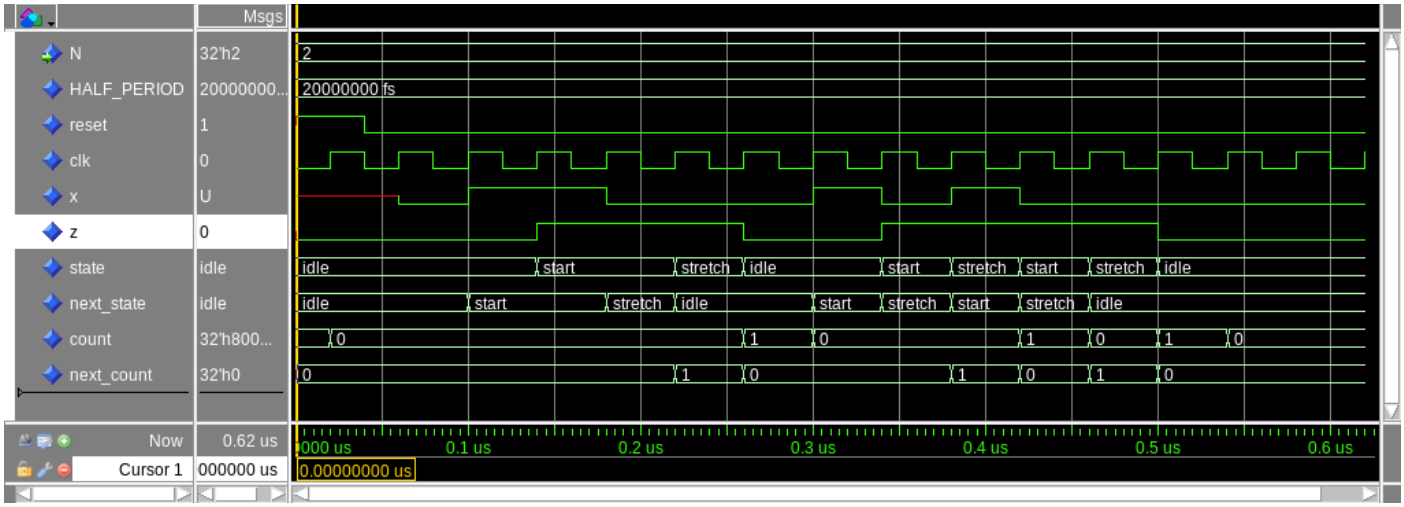
```
  p_check : process
    variable counter : integer := 0;
  begin
    wait until rising_edge(clk);
    if (x = '1') then
      counter := N;
    elsif counter > 0 then
      counter := counter -1;
      assert (z = '1') report(
        " Z not asserted in cycle: " & integer'image(counter) &
        ", with N= " & integer'image(N)
      ) severity error;
    else
      assert (Z='0') report("Z asserted too long") severity error;
    end if;
  end process;

  p_stimuli : process
  begin
    report("starting simulation");
    reset <= '1';
    wait until falling_edge(clk);
    reset <= '0';
    send_pulse(clk, "01100010100000", x);
    report("end simulation");
    wait until rising_edge(clk);
    stop;
  end process;

end architecture behavioral;
```

Note: The procedure may well be defined in p_stimuli, simplifying procedure declaration.

/1p Library references
/1p Empty entity
/1p Architecture name: (sim, tb, or something signifying that it is not RTL or structural
/1p Component declaration
/1p Signal declarations
/1p Port map/ UUT instantiation (choice of abbreviations (UUT/DUT/etc) is not important)
/1p Clock generation
/1p Procedure / function for tests
/1p assertions used (correctly)
/1p Checks the correct input waveform



```
run -all
# ** Note: starting simulation
#    Time: 0 fs  Iteration: 0  Instance: /tb_pulse_stretch
# ** Note: end simulation
#    Time: 580 ns  Iteration: 0  Instance: /tb_pulse_stretch
```