**IMPORTANT:**
- Please be neat and write (or draw) carefully. If we cannot read it with a reasonable effort, it is assumed wrong.
- As always, the best answer gets the most points.

**COVER SHEET:**

<table>
<thead>
<tr>
<th>Problem#</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (14 points)</td>
<td></td>
</tr>
<tr>
<td>2 (14 points)</td>
<td></td>
</tr>
<tr>
<td>3 (14 points)</td>
<td></td>
</tr>
<tr>
<td>4 (14 points)</td>
<td></td>
</tr>
<tr>
<td>5 (14 points)</td>
<td></td>
</tr>
<tr>
<td>6 (15 points)</td>
<td></td>
</tr>
<tr>
<td>7 (15 points)</td>
<td></td>
</tr>
</tbody>
</table>

**Total:**

**Regrade Info:**
ENTITY _entity_name IS
PORT( _input_name, _input_name : IN STD_LOGIC;
     _input_vector_name : IN STD_LOGIC_VECTOR(_high downto _low);
     _bidir_name, _bidir_name : INOUT STD_LOGIC;
     _output_name, _output_name : OUT STD_LOGIC);
END _entity_name;

ARCHITECTURE a OF _entity_name IS
SIGNAL _signal_name : STD_LOGIC;
BEGIN
  -- Process Statement
  -- Concurrent Signal Assignment
  -- Conditional Signal Assignment
  -- Selected Signal Assignment
  -- Component Instantiation Statement
END a;

_instance_name: _component_name PORT MAP (_component_port => _connect_port,
     _component_port => _connect_port);

WITH _expression SELECT
  _signal <= _expression WHEN _constant_value,
  _expression WHEN _constant_value,
  _expression WHEN _constant_value,
  _expression WHEN _constant_value;
  _signal <= _expression WHEN _boolean_expression ELSE
  _expression WHEN _boolean_expression ELSE
  _expression;

IF _expression THEN
  _statement;
  _statement;
ELSIF _expression THEN
  _statement;
  _statement;
ELSE
  _statement;
  _statement;
END IF;

CASE _expression IS
  WHEN _constant_value =>
    _statement;
    _statement;
  WHEN _constant_value =>
    _statement;
    _statement;
  WHEN OTHERS =>
    _statement;
    _statement;
END CASE;

<generate_label>: FOR <loop_id> IN <range> GENERATE
  -- Concurrent Statement(s)
END GENERATE;
type array_type is array(_upperbound downto _lowerbound);
1) (14 points) Fill in the following behavioral VHDL to implement the illustrated circuit. Assume that clk and rst connect to every register. Also, write the code so that only the lower half bits of the multiplier output is saved in the register. All wires in the circuit are WIDTH bits.

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

entity mult_add is
  generic (width : positive := 16);
  port (clk, rst : in std_logic;
        input1, input2, input3 : in std_logic_vector(width-1 downto 0);
        output : out std_logic_vector(width-1 downto 0));
end mult_add;

architecture BHV of mult_add is signal mul_reg, in3-reg : std_logic_vector(width-1 downto 0);

begin -- BHV
  process(clk, rst)
    variable mul_tmp : std_logic_vector(width-1 downto 0);
  begin
    if (rst = '1') then
      mul_reg <= (others => '0');
      in3-reg <= (others => '0');
      output <= (others => '0');
    elsif (rising_edge(clk)) then
      mul_tmp := std_logic_vector(unsigned(input1) * unsigned(input2));
      mul_reg <= mul_tmp (width-1 downto 0);
      in3-reg <= input3;
      output <= std_logic_vector(unsigned(mul_reg) + unsigned(in3-reg));
    end if;
  end process;
end BHV;
2) (14 points) Draw the circuit that would be synthesized for the following code. Label all inputs, outputs, and register outputs based on their signal names.

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

entity example is
  generic (width : positive := 16);
  port (clk, rst : in std_logic;
        in1, in2, in3, in4 : in std_logic_vector(width-1 downto 0);
        output : out std_logic_vector(width-1 downto 0));
end example;

architecture BNV of example is
  signal add1_out : std_logic_vector(width-1 downto 0);
  signal add2_out : std_logic_vector(width-1 downto 0);
  signal temp1_out : std_logic_vector(width-1 downto 0);
  signal temp2_out : std_logic_vector(width-1 downto 0);
begin
  process(clk, rst)
  begin
    if (rst = '1') then
      add1_out <= (others => '0');
      add2_out <= (others => '0');
      temp1_out <= (others => '0');
      temp2_out <= (others => '0');
    elsif (rising_edge(clk)) then
      add1_out <= std_logic_vector(unsigned(in1)+unsigned(in2));
      add2_out <= std_logic_vector(unsigned(in3)+unsigned(in1));
      temp1_out <= add1_out;
      temp2_out <= add2_out;
    end if;
  end process;
  output <= std_logic_vector(unsigned(temp1_out)+unsigned(temp2_out));
end BNV;
3) (14 points) Identify the violations (if any) of the synthesis coding guidelines for combinational logic, and the effect on the synthesized circuit.

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

entity alu_en is
  generic (width : positive := 16);
  port (en, sel : in std_logic;
        input1, input2 : in std_logic_vector(width-1 downto 0);
        output : out std_logic_vector(width-1 downto 0));
end entity alu_en;

architecture BHV of alu_en is
begin
  process(input1,input2,sel)
  begin
    case sel is
      when '0' =>
        if (en = '1') then
          output <= std_logic_vector(unsigned(input1)+unsigned(input2));
        end if;
      when '1' =>
        if (en = '1') then
          output <= std_logic_vector(unsigned(input1)-unsigned(input2));
        end if;
      when others => null;
    end case;
  end process;
end architecture;
```

2) output not defined when en = '0' infers latch
4) (14 points) Circle the following architectures that will correctly simulate an adder with overflow:

entity ADD is
port(
  input1, input2 : in std_logic_vector(15 downto 0);
  output, overflow : out std_logic_vector(15 downto 0);
end ADD;

architecture BHV1 of ADD is
signal temp : unsigned(16 downto 0);
begin
process(input1, input2)
begin
  temp <= unsigned("0"&input1) + unsigned("0"&input2);
  output <= std_logic_vector(temp(15 downto 0));
  overflow <= std_logic(temp(16));
end process;
end BHV1;

architecture BHV2 of ADD is
begin
process(input1, input2)
begin
  temp <= unsigned("0"&input1) + unsigned("0"&input2);
  output <= std_logic_vector(temp(15 downto 0));
end process;
  overflow <= std_logic(temp(16));
end BHV2;

architecture BHV3 of ADD is
begin
process(input1, input2)
begin
  temp <= unsigned("0"&input1) + unsigned("0"&input2);
end process;
  output <= std_logic_vector(temp(15 downto 0));
  overflow <= std_logic(temp(16));
end BHV3;

architecture BHV4 of ADD is
begin
process(input1, input2)
variable temp : unsigned(16 downto 0);
begin
  temp := unsigned("0"&input1) + unsigned("0"&input2);
  output <= std_logic_vector(temp(15 downto 0));
  overflow <= std_logic(temp(16));
end process;
end BHV4;

architecture BHV5 of ADD is
begin
  temp <= unsigned("0"&input1) + unsigned("0"&input2);
  output <= std_logic_vector(temp(15 downto 0));
  overflow <= std_logic(temp(16));
end BHV5;
5) (14 points) Fill in the code provided below to create a series of flip flops (FFs) to delay an input by a fixed number of cycles. The length of the series is specified by the generic cycles. You must use a structural architecture with the provided generate loop. The circuit should look like this:

Assume you have the shown FF component:

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

entity delay is
  generic (
    cycles : positive := 8);
  port (
    clk, rst, input : in std_logic;
    output : out std_logic);
end delay;

architecture STR of delay is

  component ff
    port (
      clk, rst, input : in std_logic;
      output : out std_logic);
  end component;

  signal temp : std_logic_vector(width downto 0);

begin
  temp(0) <= input;

  U_DELAY : for i in 0 to cycles-1 generate
    U_FF : entity work.ff
      port map (
        clk => clk,
        rst => rst,
        input => temp(i),
        output => temp(i+1)
      );
  end generate U_DELAY;

  output <= temp(cycles);
end STR;
```
6) (15 points) Fill in the skeleton code to implement the following Moore finite state machine, using the 2-process FSM model. Assume that if an edge does not have a corresponding condition, that edge is always taken on a clock edge. Assume that INIT is the start state. Use the next page if extra room is needed.

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

entity fsm is
  port (                                    
    clk, rst, go : in  std_logic;
    x : out std_logic_vector(1 downto 0));
end fsm;

architecture PROC2 of fsm is
  type STATE_TYPE is (INIT, COUNT1, COUNT2, DONE);
signal state, next_state : STATE_TYPE;
begin
  process(clk, rst)
  begin
    if (rst = '1') then
      state <= INIT;
    elsif (clk'event and clk = '1') then
      state <= next_state;
    end if;
  end process;
  process (go, state)
  begin
    case state is
      when INIT =>
        x <= "00",
        if (go = '1')
          next_state <= COUNT1;
        else
          next_state <= INIT;
        end if;
```
when COUNT = 7
  x <= "01";
  next_state <= COUNT;

when COUNT = 7
  x <= "10";
  next_state <= DONE;

when DONE = 7
  x <= "11";
  if igo = '0' then
    next_state <= INIT;
  else
    next_state <= DONE;
  end if;
end case;

end process;
end PROC2;
7)  a. (5 points) True/false. A hierarchical carry-lookahead adder reduces area overhead compared to a single-level carry-lookahead adder without sacrificing propagation delay.

false

b. (5 points) Define the 4th carry bit \( c_4 \) of a carry-lookahead adder in terms of each propagate bit \( p_i \), each generate bit \( g_i \) and the carry in \( c_0 \).

\[
c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0
\]

c. (5 points) What advantage does a ripple-carry adder have over a carry-lookahead adder?

area increases linearly with bit width