EEL 4712
Midterm 1 – Spring 2012
VERSION 1

Name: ____________________

UFID: ____________________

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:

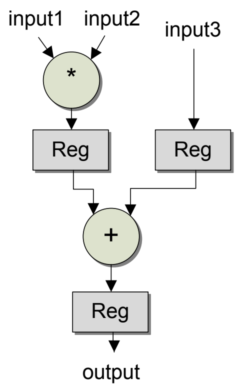
Problem#:	Points
1 (14 points)	
2 (14 points)	
3 (14 points)	
4 (14 points)	
5 (14 points)	
6 (15 points)	
7 (15 points)	

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






begin  -- BHV

  process(clk, rst)


  begin
    if (rst = '1') then









    elsif (rising_edge(clk)) then














    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 BHV 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(in4));
      temp1_out <= add1_out;
      temp2_out <= add2_out;
    end if;
  end process;

  output <= std_logic_vector(unsigned(temp1_out)+unsigned(temp2_out));

end BHV;
```

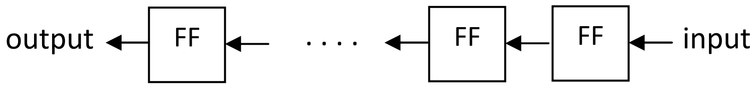
3) (14 points) Identify the violations (if any) of the *synthesis coding guidelines for combinational logic*, and the effect on the synthesized circuit.

```
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 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 BHV;
```

4) (14 points) Circle the following architectures that will correctly simulate an adder with overflow:

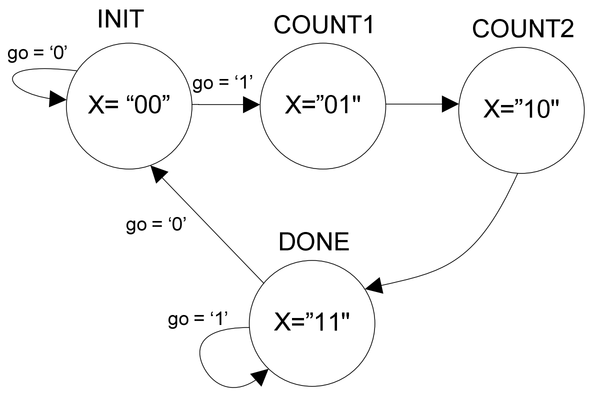
```
entity ADD is
  port (
    input1, input2 : in  std_logic_vector(15 downto 0);
    output         : out std_logic_vector(15 downto 0);
    overflow       : out std_logic);
end ADD;

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
  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));
  end process;

  overflow <= std_logic(temp(16));

end BHV2;


architecture BHV3 of ADD is
  signal temp : unsigned(16 downto 0);
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
  signal 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 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. Assume you are given the shown FF component. 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:



```
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;




begin




  U_DELAY : for i in 0 to cycles-1 generate

    U_FF : entity work.ff
      port map (









       );

  end generate U_DELAY;









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 rising 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




    elsif (clk'event and clk = '1') then




    end if;
  end process;

  process(                                    )
  begin
  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 increasing propagation delay.

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