**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 (8 points)</td>
<td></td>
</tr>
<tr>
<td>2 (8 points)</td>
<td></td>
</tr>
<tr>
<td>3 (16 points)</td>
<td></td>
</tr>
<tr>
<td>4 (16 points)</td>
<td></td>
</tr>
<tr>
<td>5 (16 points)</td>
<td></td>
</tr>
<tr>
<td>6 (16 points)</td>
<td></td>
</tr>
<tr>
<td>7 (16 points)</td>
<td></td>
</tr>
<tr>
<td>8 (8 points)</td>
<td></td>
</tr>
</tbody>
</table>

**Total:**

**Regrade Info:**
1) (8 points) For the entity given below, explain how the generic WIDTH gets its value:

library ieee;
use ieee.std_logic_1164.all;

entity ALU is
  generic (WIDTH : positive := 16);
  port (input1, input2 : in std_logic_vector(WIDTH-1 downto 0);
        sel : in std_logic_vector(3 downto 0);
        output : out std_logic_vector(WIDTH-1 downto 0));
end ALU;

When a component of type ALU is instantiated, a generic map defines WIDTH. If no generic map is used, WIDTH gets the default value of 16.

2) (8 points) Describe the violation of the synthesis guidelines for combinational logic in the following example:

library ieee;
use ieee.std_logic_1164.all;

entity MUX is
  port (input1, input2 : in std_logic_vector(15 downto 0);
        sel : in std_logic;
        en : in std_logic;
        output : out std_logic_vector(15 downto 0));
end MUX;

architecture BHV of MUX is
begin -- BHV

process(input1, input2, sel)
begin
  if en = '1' then
    output <= (others => '0');
  elsif sel = '0' then
    output <= input1;
  else
    output <= input2;
  end if;
end process;

end BHV;
4) (16 points) Fill in the code provided below to create a shift register with generic width. You must use a structural architecture with the provided generate loop that connects together the flip-flop (FF) components. The circuit should look like this:

```
FF  ............  FF  FF
  
output(WIDTH-1) output(1) output(0)
```

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

entity SH_REG is
generic (WIDTH : positive := 16);
port (    clk, rst : in std_logic;
          input : in std_logic;
          output : out std_logic_vector(WIDTH-1 downto 0));
end SH_REG;

architecture STR of SH_REG is

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

signal temp : std_logic_vector(WIDTH downto 0);

begin -- STR
    temp(0) <= input;
    U_SH_REG : for i in 0 to WIDTH-1 generate
        U_FF : FF port map (    clk => clk,
                                rst => rst,
                                input => temp(i),
                                output => temp(i+1))
    end generate U_SH_REG;
    output <= temp(WIDTH downto 1);
end STR;
```
(16 points) Fill in the skeleton code to implement the following Moore finite state machine, 
using the 1-process FSM model. Assume the "Clr" has a higher priority than "En", so that if "Clr" 
is asserted, the "Clr" edges are taken instead of the "En" edges. For other possible conditions 
that are not shown, assume that each state transitions back to itself (e.g. en='0'). Use the next 
page if extra room is needed.

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

entity FSM is
  port (
    clk, rst, en, clr : in std_logic;
    x : out std_logic_vector(1 downto 0));
end FSM;

architecture BHV of FSM is
  type STATE_TYPE is (STATE_0, STATE_1, STATE_2);
  signal state : STATE_TYPE;

begin
  process (clk, rst)
  begin
    if (rst='1') then
      state <= STATE_0;
      x <= "00";
    elsif (clk'event and clk='1') then
      case state is
        when STATE_0 =>
          x <= "001";
          if (en='1') then
            state <= STATE_1;
          else
            state <= STATE_0;
        when STATE_1 =>
          x <= "101";
          if (clr='1') then
            state <= STATE_0;
        when OTHERS =>
          state <= STATE_1;
      end case;
  end process;
end BHV;
```
7) a. (8 points) Define each carry bit of a 3-bit carry lookahead adder in terms of the propagate and generate functions, and the carry in \( c(0) \).

\[
\begin{align*}
\text{\( c(1) \)} &= g(0) + p(0) \cdot c(0) \\
\text{\( c(2) \)} &= g(1) + p(1) \cdot g(0) + p(1) \cdot p(0) \cdot c(0) \\
\text{\( c(3) \)} &= g(2) + p(2) \cdot g(1) + p(2) \cdot p(1) \cdot g(0) + p(2) \cdot p(1) \cdot g(0) \cdot c(0)
\end{align*}
\]

b. (8 points) For each graph below, circle the number that most closely represents a carry lookahead adder.

![Graph](image)

8) (6 free points) In the space below, describe every difference between numeric\_std and std\_logic\_arith. Alternatively, leave blank for full credit.