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

```vhdl
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;
```

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

```vhdl
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 sel = '1' then
      output <= input2;
    end if;
  end process;
end BHV;
```
3) A. (8 points) For the following code, fill in the waveform for “output” assuming the values shown are in decimal format:

```vhdl
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 BHV 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 BHV;
```

<table>
<thead>
<tr>
<th>Input1</th>
<th>100</th>
<th>150</th>
<th>30</th>
<th>90</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input2</td>
<td>50</td>
<td>25</td>
<td>60</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Output</td>
<td>......</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

100 ns 120 ns 140 ns 160 ns
```

B. (8 points) Briefly explain the reason for the behavior of “output”, specifically mentioning when a signal’s value gets updated.
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:

```
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;

begin  -- STR

  U_SH_REG : for i in 0 to WIDTH-1 generate

    U_FF : FF port map (        clk    => clk,
                           rst    => rst,
                           input   => input,
                           output  => output(WIDTH-1 downto 0));

  end generate U_SH_REG;

end STR;
```
5) (16 points) Draw a schematic for the register-transfer-level (RTL) circuit that will be synthesized from the following VHDL code. Clearly label all components, inputs, and outputs.

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

entity BHV_TEST is
  port (
    clk, rst       : in  std_logic;
    input1, input2 : in  std_logic_vector(15 downto 0);
    input3, input4 : in  std_logic_vector(15 downto 0);
    output         : out std_logic_vector(15 downto 0));
end BHV_TEST;

architecture BHV of BHV_TEST is

  signal add1, add2 : std_logic_vector(15 downto 0);
  signal mult1 : std_logic_vector(31 downto 0);
  begin
    process(clk, rst)
    begin
      if (rst = '1') then
        add1   <= (others => '0');
        add2   <= (others => '0');
        mult1  <= (others => '0');
      elsif (clk'event and clk = '1') then
        add1  <= std_logic_vector(unsigned(input1)+unsigned(input2));
        add2  <= add1;
        mult1 <= std_logic_vector(unsigned(input3)*unsigned(input4));
      end if;
    end process;

    output <= std_logic_vector(unsigned(add2)+
                              unsigned(mult1(15 downto 0)));
  end BHV;
```
6) (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 (  
    X = "00",  
    X = "10",  
    X = "11"  
  );

  signal state : STATE_TYPE;

begin  -- BHV

X = "00"  
En='1'  
X = "10"  
En='1'  
X = "11"  
En='1'

```

En = '1' or Clr = '1'

Clr = '1'

X = "00"  
X = "10"  
X = "11"

library ieee;
use ieee.std_logic_1164.all;

descriptive FPGA
7) a. (8 points) Define each carry bit of a 3-bit carry lookahead adder (i.e., c(1), c(2), c(3)) in terms of the propagate and generate functions, and the carry in (i.e., c(0)).

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

8) (8 free points) In the space below, describe every difference between numeric_std and std_logic_arith. Alternatively, leave blank for full credit.
1. **IEEE’s STD_LOGIC_1164**

1.1. **Logic Values**
- ‘U’ Uninitialized
- ‘X’/’W’ Strong/Weak unknown
- ‘0’/’L’ Strong/Weak 0
- ‘1’/’H’ Strong/Weak 1
- ‘Z’ High Impedance
- ‘-’ Don’t care

1.2. **Predefined Types**

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Base type</th>
</tr>
</thead>
<tbody>
<tr>
<td>STD_ULOGIC</td>
<td>Resolved STD_ULOGIC</td>
</tr>
<tr>
<td>X01</td>
<td>Resolved X, 0 &amp; 1</td>
</tr>
<tr>
<td>X01Z</td>
<td>Resolved X, 0, 1 &amp; Z</td>
</tr>
<tr>
<td>UX01</td>
<td>Resolved U, X, 0 &amp; 1</td>
</tr>
<tr>
<td>UX01Z</td>
<td>Resolved U, X, 0, 1 &amp; Z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STD_LOGIC_VECTOR</td>
<td>Array of STD_ULOGIC</td>
</tr>
<tr>
<td>STD_LOGIC_VECTOR</td>
<td>Array of STD_LOGIC</td>
</tr>
</tbody>
</table>

1.3. **Overloaded Operators**

<table>
<thead>
<tr>
<th>Description</th>
<th>Left</th>
<th>Operator</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitwise-and</td>
<td>u/l, uv, lv</td>
<td>and, nand</td>
<td>u/l, uv, lv</td>
</tr>
<tr>
<td>bitwise-or</td>
<td>u/l, uv, lv</td>
<td>or, nor</td>
<td>u/l, uv, lv</td>
</tr>
<tr>
<td>bitwise-xor</td>
<td>u/l, uv, lv</td>
<td>xor, xnor</td>
<td>u/l, uv, lv</td>
</tr>
<tr>
<td>bitwise-not</td>
<td>not</td>
<td></td>
<td>u/l, uv, lv</td>
</tr>
</tbody>
</table>

1.4. **Conversion Functions**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>u/l</td>
<td>b</td>
<td>TO_BIT(from, xmap)</td>
</tr>
<tr>
<td>uv, lv</td>
<td>bv</td>
<td>TO_BITVECTOR(from, xmap)</td>
</tr>
<tr>
<td>b</td>
<td>u/l</td>
<td>TO_STDULOGIC(from)</td>
</tr>
<tr>
<td>bv, uv</td>
<td>lv</td>
<td>TO_STDLOGICVECTOR(from)</td>
</tr>
</tbody>
</table>

2. **IEEE’s NUMERIC_STD**

2.1. **Predefined Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNREALIZED</td>
<td>(na to</td>
</tr>
<tr>
<td>REALIZED</td>
<td>(na to</td>
</tr>
</tbody>
</table>

2.2. **Overloaded Operators**

<table>
<thead>
<tr>
<th>Left</th>
<th>Op</th>
<th>Right</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>sg</td>
<td>sg</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>sg</td>
<td>sg</td>
<td></td>
</tr>
<tr>
<td>un</td>
<td>+, *, /, rem, mod</td>
<td>un</td>
<td></td>
</tr>
<tr>
<td>sg</td>
<td>+, *, /, rem, mod</td>
<td>sg</td>
<td></td>
</tr>
<tr>
<td>na</td>
<td>+, *, /, rem, mod</td>
<td>na</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>+, *, /, rem, mod</td>
<td>in</td>
<td></td>
</tr>
<tr>
<td>un</td>
<td>&gt;, &gt;=, =, /=</td>
<td>un boole</td>
<td></td>
</tr>
<tr>
<td>sg</td>
<td>&gt;, &gt;=, =, /=</td>
<td>sg boole</td>
<td></td>
</tr>
<tr>
<td>na</td>
<td>&gt;, &gt;=, =, /=</td>
<td>na boole</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>&gt;, &gt;=, =, /=</td>
<td>in boole</td>
<td></td>
</tr>
</tbody>
</table>

2.3. **Predefined Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHIFT_LEFT</td>
<td>(un, na)</td>
</tr>
<tr>
<td>SHIFT_RIGHT</td>
<td>(un, na)</td>
</tr>
<tr>
<td>SHIFT_LEFT</td>
<td>(sg, na)</td>
</tr>
<tr>
<td>SHIFT_RIGHT</td>
<td>(sg, na)</td>
</tr>
<tr>
<td>ROTATE_LEFT</td>
<td>(un, na)</td>
</tr>
<tr>
<td>ROTATE_RIGHT</td>
<td>(un, na)</td>
</tr>
<tr>
<td>RESIZE</td>
<td>(sg, na)</td>
</tr>
<tr>
<td>RESIZE</td>
<td>(un, na)</td>
</tr>
</tbody>
</table>

2.4. **Conversion Functions**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>un, lv</td>
<td>sg</td>
<td>SIGNED(from)</td>
</tr>
<tr>
<td>sg, lv</td>
<td>un</td>
<td>UNSIGNED(from)</td>
</tr>
<tr>
<td>un, sg</td>
<td>lv</td>
<td>STD_LOGIC_VECTOR(from)</td>
</tr>
<tr>
<td>un, sg</td>
<td>in</td>
<td>TO_INTEGER(from)</td>
</tr>
<tr>
<td>na</td>
<td>un</td>
<td>TO_UNSIGNED(from, size)</td>
</tr>
<tr>
<td>in</td>
<td>sg</td>
<td>TO_SIGNED(from, size)</td>
</tr>
</tbody>
</table>

3. **IEEE’s NUMERIC_BIT**

3.1. **Predefined Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNREALIZED</td>
<td>(na to</td>
</tr>
<tr>
<td>REALIZED</td>
<td>(na to</td>
</tr>
</tbody>
</table>

3.2. **Overloaded Operators**

<table>
<thead>
<tr>
<th>Left</th>
<th>Op</th>
<th>Right</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>sg</td>
<td>sg</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>sg</td>
<td>sg</td>
<td></td>
</tr>
<tr>
<td>un</td>
<td>+, *, /, rem, mod</td>
<td>un</td>
<td></td>
</tr>
<tr>
<td>sg</td>
<td>+, *, /, rem, mod</td>
<td>sg</td>
<td></td>
</tr>
<tr>
<td>na</td>
<td>+, *, /, rem, mod</td>
<td>na</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>+, *, /, rem, mod</td>
<td>in</td>
<td></td>
</tr>
<tr>
<td>un</td>
<td>&gt;, &gt;=, =, /=</td>
<td>un boole</td>
<td></td>
</tr>
<tr>
<td>sg</td>
<td>&gt;, &gt;=, =, /=</td>
<td>sg boole</td>
<td></td>
</tr>
<tr>
<td>na</td>
<td>&gt;, &gt;=, =, /=</td>
<td>na boole</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>&gt;, &gt;=, =, /=</td>
<td>in boole</td>
<td></td>
</tr>
</tbody>
</table>

3.3. **Predefined Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHIFT_LEFT</td>
<td>(un, na)</td>
</tr>
<tr>
<td>SHIFT_RIGHT</td>
<td>(un, na)</td>
</tr>
<tr>
<td>SHIFT_LEFT</td>
<td>(sg, na)</td>
</tr>
<tr>
<td>SHIFT_RIGHT</td>
<td>(sg, na)</td>
</tr>
<tr>
<td>ROTATE_LEFT</td>
<td>(un, na)</td>
</tr>
<tr>
<td>ROTATE_RIGHT</td>
<td>(un, na)</td>
</tr>
<tr>
<td>RESIZE</td>
<td>(sg, na)</td>
</tr>
<tr>
<td>RESIZE</td>
<td>(un, na)</td>
</tr>
</tbody>
</table>

3.4. **Conversion Functions**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>un, bv</td>
<td>sg</td>
<td>SIGNED(from)</td>
</tr>
<tr>
<td>sg, bv</td>
<td>un</td>
<td>UNSIGNED(from)</td>
</tr>
<tr>
<td>un, sg</td>
<td>bv</td>
<td>BITVECTOR(from)</td>
</tr>
<tr>
<td>un, sg</td>
<td>in</td>
<td>TO_INTEGER(from)</td>
</tr>
<tr>
<td>na</td>
<td>un</td>
<td>TO_UNSIGNED(from)</td>
</tr>
<tr>
<td>in</td>
<td>sg</td>
<td>TO_SIGNED(from)</td>
</tr>
</tbody>
</table>

© 1995-1998 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is hereby granted. See reverse side for additional information.
4. Synopsys' STD_LOGIC_ARITH

4.1. Predefined Types

UNSIGNED(na to | downto na) Array of STD_LOGIC
SIGNED(na to | downto na) Array of STD_LOGIC
SMALL_INT Integer subtype, 0 or 1

4.2. Overloaded Operators

Left    Op    Right  Return
  abs    sg    sg,lv
  -      sg    sg,lv
  un +,*,/  un    un,lv
  sg +,*,/  sg    sg,lv
  sg +,*,/c  un    sg,lv
  un +,c    in    un,lv
  sg +,c    in    sg,lv
  un +,c    u/l   un,lv
  sg +,c    u/l   sg,lv
  un <,>,<=,>=,=/  un    bool
  sg <,>,<=,>=,=/  sg    bool

4.3. Predefined Functions

SHL(un, un)    un    SHR(un, un)    un
SHL(sg, un)    sg    SHR(sg, un)    sg
EXT(lv, in)    lv    zero-extend
SEXT(lv, in)    lv    sign-extend

4.4. Conversion Functions

From    To    Function
  un,lv  sg    SIGNED(from)
  sg,lv  un    UNSIGNED(from)
  sg,un  lv    STD_LOGIC_VECTOR(from)
  un,sg  in    CONV_INTEGER(from)
  in,un,sg,u  un    CONV_UNSIGNED(from, size)
  in,un,sg,u  sg    CONV_SIGNED(from, size)
  in,un,sg,u  lv    CONV_STD_LOGIC_VECTOR(from, size)

5. Synopsys' STD_LOGIC_UNSIGNED

5.1. Overloaded Operators

Left    Op    Right  Return
  +      lv    lv
  lv +,*,/  lv    lv
  lv +,c    lv    lv
  lv +,c    u/l   lv
  lv <,>,<=,>=,=/  lv    bool
  lv <,>,<=,>=,=/c  in    bool

5.2. Conversion Functions

From  To    Function
  lv    in    CONV_INTEGER(from)

6. Synopsys' STD_LOGIC_SIGNED

6.1. Overloaded Operators

Left    Op    Right  Return
  abs    lv    lv
  +,*    lv    lv
  lv +,*,/  lv    lv
  lv +,c    in    lv
  lv +,c    u/l   lv
  lv <,>,<=,>=,=/  lv    bool
  lv <,>,<=,>=,=/c  in    bool

6.2. Conversion Functions

From    To    Function
  lv    in    CONV_INTEGER(from)

7. Synopsys' STD_LOGIC_MISC

7.1. Predefined Functions

AND_REDUCE(lv | uv)    u/l
OR_REDUCE(lv | uv)    u/l
XOR_REDUCE(lv | uv)    u/l

8. Cadence's STD_LOGIC_ARITH

8.1. Overloaded Operators

Left    Op    Right  Return
  u/l    +,*,/    u/l    u/l
  lv +,*,/  lv    lv
  lv +,c    lv    lv
  lv +,c    u/l   lv
  lv <,>,<=,>=,=/  lv    bool
  lv <,>,<=,>=,=/c  in    bool

8.2. Predefined Functions

SH_LEFT(lv, na)    lv
SH_LEFT(uv, na)    uv
SH_RIGHT(lv, na)    lv
SH_RIGHT(uv, na)    uv
ALIGN_SIZE(lv, na)    lv
ALIGN_SIZE(uv, na)    uv
ALIGN_SIZE(u/l, na)    lv,uv

8.3. Conversion Functions

From  To    Function
  lv,uv,u/l  in    TO_INTEGER(from)
  in    lv    TO_STDLOGICVECTOR(from, size)
  in    uv    TO_STDLOGICVECTOR(from, size)

9. Mentor's STD_LOGIC_ARITH

9.1. Predefined Types

UNSIGNED(na to | downto na) Array of STD_LOGIC
SIGNED(na to | downto na) Array of STD_LOGIC

9.2. Overloaded Operators

Left    Op    Right  Return
  abs    sg    sg
  -      u/l   u/l
  un +,*,/  un    un,lv
  sg +,*,/  sg    sg,lv
  sg +,*,/c  un    sg,lv
  un +,c    in    un,lv
  sg +,c    in    sg,lv
  un +,c    u/l   un,lv
  sg +,c    u/l   sg,lv
  un <,>,<=,>=,=/  un    bool
  sg <,>,<=,>=,=/  sg    bool
  un <,>,<=,>=,=/c  un    bool
  sg <,>,<=,>=,=/c  sg    bool
  un <,>,<=,>=,=/c  u/l   un,lv
  sg <,>,<=,>=,=/c  u/l   sg,lv
  un <,>,<=,>=,=/c  u/l   bool
  sg <,>,<=,>=,=/c  u/l   bool

9.3. Predefined Functions

ZERO_EXTEND(uv | lv | un, na)    same
SIGN_EXTEND(u/l, na)    sg
AND_REDUCE(uv | lv | un | sg)    u/l
OR_REDUCE(uv | lv | un | sg)    u/l
XOR_REDUCE(uv | lv | un | sg)    u/l

9.4. Conversion Functions

From  To    Function
  u/l,uv,lv,un,sg  in    TO_INTEGER(from)
  u/l,uv,lv,un,sg  in    CONV_INTEGER(from)
  bool    u/l    TO_STDLOGIC(from)
  na    un  TO_UNSIGNED(from, size)
  na    un  TO_SIGNED(from, size)
  in    sg    TO_SIGNED(from, size)
  in    sg    TO_UNSIGNED(from, size)
  na    lv    TO_STDLOGICVECTOR(from, size)
  na    uv    TO_STDLOGICVECTOR(from, size)

© 1995-1998 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is hereby granted.

Qualis Design Corporation
Elite Consulting and Training in High-Level Design
Phone: +1-503-670-7200  FAX: +1-503-670-0809
E-mail: info@qualis.com  Web: http://www.qualis.com
Also available: VHDL Quick Reference Card
Verilog HDL Quick Reference Card