Paper Number(s): **E3.06**
AM2
ISE3.5

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE
UNIVERSITY OF LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2001

MSc and EEE/ISE PART III/IV: M.Eng., B.Eng. and ACGI

(2-sided)

VHDL AND LOGIC SYNTHESIS

Friday, 11 May 10:00 am

There are SIX questions on this paper.

Answer FOUR questions.

Time allowed: 3:00 hours

Corrected Copy

Examiners: Clarke,T.J.W. and Cheung,P.Y.K.

Nine

Special instructions for invigilators: Students may bring any written or printed aids into this examination.

Information for candidates: Students may bring any written or printed aids into this examination.

1. *Figure 1a* shows a hardware block *serial_rx* that performs serial to parallel conversion. The block receives a sequence of bytes on serial bus *serial_in*, *serial_en*, and outputs bytes on *databus*. The block is controlled by *clk*, and *reset* which is active high and synchronous. All *serial_rx* signals are synchronous with the rising edge of *clk*: inputs are assumed to change just after this edge. The main sub-blocks of *serial_rx* are shown in *Figure 1a*. Except for the output data path, interconnections are omitted.

The required timing is illustrated in *Figure 1b*. Each byte is received in 8 contiguous cycles with *serial_en* high. The number of cycles with *serial_en* low between successive bytes is 1 or more. The serial data bits are received most significant bit first.

Two cycles after the last bit is received *datardy* is pulsed high for 1 cycle, as shown in *Figure 1b*. The parallel data is output on *databus* when *datardy* is high: at all other times *databus* is high impedance. If the length of the *serial_en* high pulse is not equal to 8 cycles the received data is ignored and *datardy* will remain low.

a) Write an RTL description of *serial_rx*, defining the necessary processes, signals driven in each process, and finite state machine transitions.

[10 marks]

b) Write a VHDL entity and synthesisable architecture that implements *serial_rx*.

[10 marks]

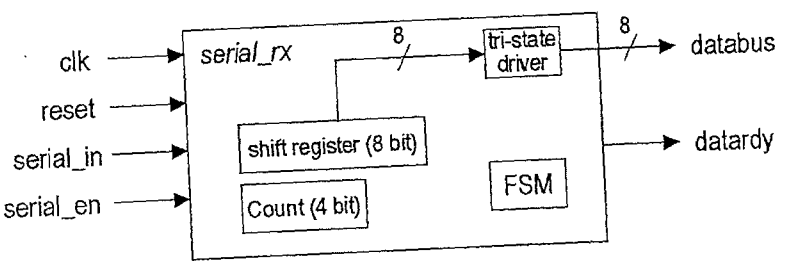


Figure 1a

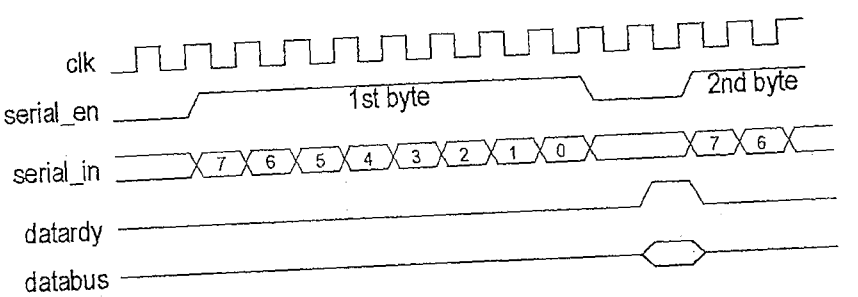


Figure 1b

2. The VHDL package in *Figure 2* is used as part of a testbench in the implementation of a stimulus generator that drives a signal *pi* of type *pix_inp_t*. The data on this signal is synchronous with the rising edge of *pclk*, and represents a raster scanned stream of video data. Data for the pixels of types luma (Y) and chroma (U & V) is multiplexed onto *pi.data* by the strobes pi.*avail_y*, pi.*avail_u*, pi.*avail_v*. The function *data_source* (whose implementation is not shown here) determines the data values of every pixel. The procedure *emit_line* generates a single line of video with length *h_size* by making a sequence of calls to *emit_pixel*.

a) Assume that the value of *data_source* is equal to its parameter, *h*, converted to *std_logic_vector*. Determine the execution time in *pclk* cycles, and draw a timing diagram that shows the waveforms generated on *pi*, by the statement:

```
emit_line( pclk, pi, 4);
```

[6 marks]

b) Explain why *pclk* and *pi* need to be defined as SIGNAL parameters in the definition of *emit_line*.

[4 marks]

c) Vertical blanking is defined to be time when a signal *vert_blank*, of type *std_logic*, is high. Write a procedure *emit_vert_blank* that can be used to generate a specified number of *pclk* cycles of vertical blanking.

[4 marks]

d) A video frame is defined to be a sequence of video lines and vertical blanking periods. The function *data_source* uses an integer shared variable *vertical_line_number* to determine the pixel data output by *emit_line*. A video line with number *n* is generated by a call to *emit_line* with *vertical_line_number* set to *n*. A video frame is defined by the parameters *h_size*, *v_size*, and *interlaced* and consists of lines numbered from 0 to *v_size*-1. If *interlaced* is *false*, the frame contains these lines in ascending order, followed by 20 cycles of vertical blanking. If *interlaced* is *true* the frame contains the even numbered lines in ascending order, followed by vertical blanking for 10 cycles, followed by the odd numbered lines in ascending order, followed by vertical blanking for 20 cycles. In all cases the lines are of length *h_size*.

Write a procedure *emit_frame* with header as below that generates a single video frame.

```
PROCEDURE emit_frame(
   SIGNAL pclk : IN  STD_LOGIC;
   interlaced  : IN  BOOLEAN;
   h_size      : IN  INTEGER;
   v_size      : IN  INTEGER;
   SIGNAL pi   : OUT pix_inp_t;
   SIGNAL vert_blank: OUT std_logic;
);
```

[6 marks]

```
PACKAGE pixel_pack IS

   TYPE pixel_t IS ('Y', 'U', 'V');  -- pixel type

   TYPE pix_inp_t IS
      RECORD
         data    : STD_LOGIC_VECTOR(7 DOWNTO 0);
         avail_y : STD_LOGIC;
         avail_u : STD_LOGIC;
         avail_v : STD_LOGIC;
         h       : STD_LOGIC_VECTOR(9 DOWNTO 0);
      END RECORD;

   IMPURE FUNCTION data_source( h : IN INTEGER; p : IN pixel_t)
      RETURN STD_LOGIC_VECTOR;

END PACKAGE;

PACKAGE BODY pixel_pack IS


   PROCEDURE emit_pixel(
      SIGNAL pclk : IN  STD_LOGIC;  -- clock
      SIGNAL pi   : OUT pix_inp_t;  -- driven output
      h           : IN  INTEGER;    -- horiz coord
      p           : IN  pixel_t     -- pixel type (Y, U or V)
      ) IS
   BEGIN
      pi.data    <= data_source( h, p);
      pi.avail_y <= '0'; pi.avail_u <= '0'; pi.avail_v <= '0';
      CASE p IS
         WHEN 'Y' => pi.avail_y <= '1';
         WHEN 'U' => pi.avail_u <= '1';
         WHEN 'V' => pi.avail_v <= '1';
      END CASE;
      pi.h       <= conv_std_logic_vector(h, 10);
      WAIT UNTIL pclk'EVENT AND pclk = '1';
      pi.avail_y <= '0'; pi.avail_u <= '0'; pi.avail_v <= '0';
   END PROCEDURE;

   PROCEDURE emit_line(
      SIGNAL pclk : IN  STD_LOGIC;
      SIGNAL pi   : OUT pix_inp_t;
      h_size      : IN  INTEGER
      ) IS
      VARIABLE y_hpos, uv_hpos : INTEGER;
   BEGIN
      y_hpos  := 0;
      uv_hpos := 0;
      WHILE y_hpos < h_size LOOP
         emit_pixel( pclk, pi, y_hpos, 'Y');
         IF y_hpos MOD 2 = 0 THEN
            emit_pixel( pclk, pi, uv_hpos, 'V');
         ELSE
            emit_pixel( pclk, pi, uv_hpos, 'U');
         END IF;
         y_hpos := y_hpos+1;
         IF y_hpos MOD 2 = 0 THEN
            uv_hpos := uv_hpos + 1;
         END IF;
      END LOOP;
   END PROCEDURE;

END PACKAGE BODY;
```

Figure 2

3. The entity *logic_function* in *Figure 3* implements a synthesisable logic function.

a) Under what values of c will the OTHERS case be executed, and why is the value assigned to y in this case appropriate? What is the affect of this case on the synthesis of the architecture?

[3 marks]

b) Explain clearly how the times at which statements are executed in this architecture during simulation result in a waveform on y which is a combinatorial logic function of the waveforms on c, and a. What is the simulation delay from a to y, and c to y?

[4 marks]

c) Explain what functions the operations *unsigned()* and *unsigned'()* perform.

[4 marks]

d) Rewrite the entity and architecture of *logic_function* deriving the comparison threshold, "011", from a generic parameter, and allowing arbitrary lengths of a. Your code must be synthesisable.

[5 marks]

e) An architecture *rtlnew* is identical to *rtl* except that the signal c is omitted from the sensitivity list of process *logic*. What precisely will be the simulated behaviour of *rtlnew*? Explain, giving your reasons, in what way, if any, post-synthesis and pre-synthesis simulation of *rtlnew* will differ.

[4 marks]

4.

a) In *Figure 3* the entity *logic_function* computes y as a function of a and c. Express y as a boolean function of c, $a(2)$, $a(1)$ and $a(0)$. Derive the reduced ordered binary decision diagram for y with variable order: $(c, a(0), a(1), a(2))$.

[10 marks]

b) Design a logic implementation of a single BDD node using only 2-input NAND gates. Use this and your answer from part a) to derive a 2-input NAND gate circuit diagram that implements the logic function y. Simplify your answer by propagating constant values, identifying gates with identical inputs, and removing two inversions in series. Use only 2-input NAND gates, and do NOT use any other logic optimisations.

[10 marks]

```
ENTITY logic_function IS
   PORT(
      a: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
      c: IN STD_LOGIC;
      y: OUT STD_LOGIC
   );
END logic_function;



ARCHITECTURE rtl OF logic_function IS
BEGIN
   logic : PROCESS(a, c)
   BEGIN
      y <= '0';
      CASE c IS
         WHEN '1'    =>
            IF unsigned(a) > unsigned'("011") THEN
               y <= '1';
            END if;
         WHEN '0'    =>
            IF unsigned(a) >= unsigned'("011") THEN
               y <= '1';
            END IF;
         WHEN OTHERS =>
            y    <= 'X';
      END CASE;
   END PROCESS logic;
END rtl;
```

Figure 3

5. The diagram *Figure 5a* shows a structural implementation of a combinatorial 8X8 modified Booth parallel signed multiplier, using entities *booth_adder* and *booth_mux*, defined in *Figure 5b*. The output q is the result of multiplying x and y. All numbers are two's complement signed. The block "-1" implements two's complement negation. Where necessary bit numbers are indicated in the format $a:b$ where a is the most significant bit.

a) Write a VHDL entity *booth_multiplier* and an architecture corresponding to *Figure 5a* using the entities defined in *Figure 5b*.

[14 marks]

b) A DSP system uses an array of 16 *booth_multiply* units to multiply two 16 element 8-bit arrays, *vx* and *vy*, and drive a 16 element 16-bit array, *vq*. Write appropriate type and signal definitions to represent these arrays, and a VHDL fragment using a GENERATE loop to implement the multiplications.

[6 marks]

6. The entity *booth_adder* defined in *Figure 5b* (see opposite page) performs the logic function specified in *Figure 4* (below).

a) Write a synthesisable architecture for this entity.

[6 marks]

b) Write a testbench for *booth_adder* that will read a file each line of which specifies inputs x, mx, c and si, and the corresponding output so. The testbench must check that *booth_adder* generates the correct output for each set of inputs, and print an error message for each error that is found.

[8 marks]

c) Discuss the merits of exhaustive and random verification techniques as applied to entity *booth_adder*. You may assume that a testbench as specified in part b) above is to be used, that each distinct set of input values takes 1ms to read and test, and that all tests are conducted with generic n=8. Devise an effective verification strategy, evaluating its likely performance, and making no assumptions about the integrity of the arithmetic operations used by *booth_adder*. How would your strategy change if n=16?

[6 marks]

c(2)	*c(1)*	*c(0)*	*so*
0	0	0	si
0	0	1	$si+x$
0	1	0	$si+x$
0	1	1	$si+2x$
1	0	0	$si+2mx$
1	0	1	$si+mx$
1	1	0	$si+mx$
1	1	1	si

Figure 4

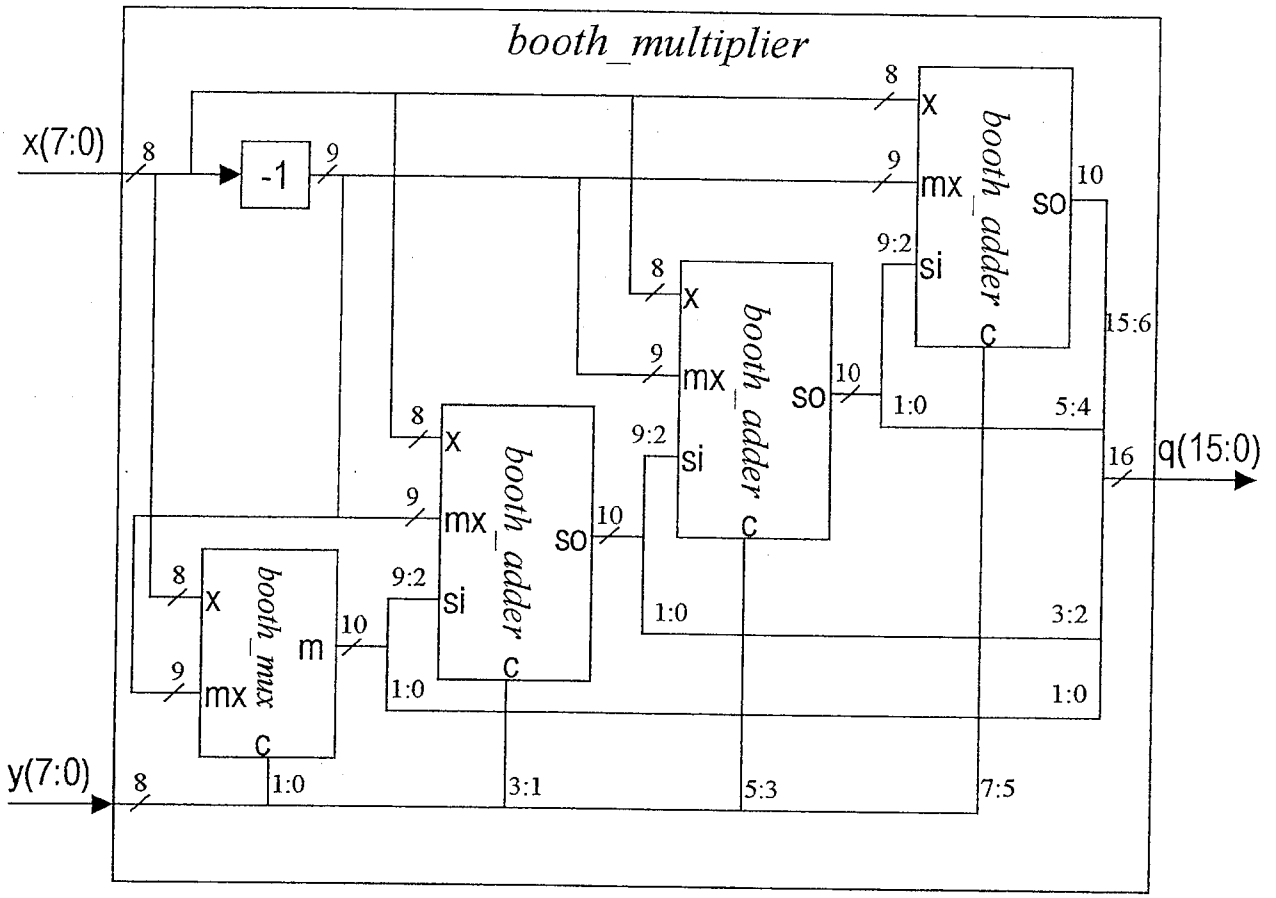


Figure 5a

```
ENTITY booth_adder IS
   GENERIC(
      n        :      INTEGER
      );
   PORT(
      x    : IN   SIGNED(n-1 DOWNTO 0);
      mx   : IN   SIGNED(n DOWNTO 0);
      si   : IN   SIGNED(n-1 DOWNTO 0);
      so   : OUT  SIGNED(n+1 DOWNTO 0);
      c    : IN   STD_LOGIC_VECTOR(2 DOWNTO 0)
      );
END booth_adder;


ENTITY booth_mux IS
   PORT(
      c     : IN   STD_LOGIC_VECTOR(1 DOWNTO 0);
      x     : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
      mx    : IN   STD_LOGIC_VECTOR(8 DOWNTO 0);
      m     : OUT  STD_LOGIC_VECTOR(9 DOWNTO 0)
      );
END booth_mux;
```

Figure 5b

Solution to Question 1

This question tests whether the students can design an RTL hardware description, and express it in correct synthesisable VHDL.

The RTL description should specify all processes providing:

- Is the process combinatorial or sequential?
- list of signals driven.
- pseudocode specifying the values driven signals are driven to.

The code below shows one answer.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;


ENTITY serial_rx IS
   PORT(
      reset     : IN  STD_LOGIC;
      clk       : IN  STD_LOGIC;
      -- input bus
      serial_in : IN  STD_LOGIC;
      serial_en : IN  STD_LOGIC;
      -- output bus
      databus   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
      datardy   : OUT STD_LOGIC
      );
END serial_rx;

ARCHITECTURE rtl OF serial_rx IS

   --fsm type
   TYPE fsmstate IS (
      waiting, receiving, dataready
      );

   -- fsm signals
   SIGNAL state, next_state : fsmstate;

   SIGNAL si_count : STD_LOGIC_VECTOR( 3 DOWNTO 0);
   -- shift register to store received data
   SIGNAL dsi      : STD_LOGIC_VECTOR( 7 DOWNTO 0);


BEGIN

   -- main rx shift register
   sireg : PROCESS
   BEGIN
      --unusually, don't need to initialise this register
      WAIT UNTIL clk'EVENT AND clk = '1';
      IF serial_en = '1' THEN
         dsi <= dsi(6 DOWNTO 0) & serial_in;
      END IF;
```

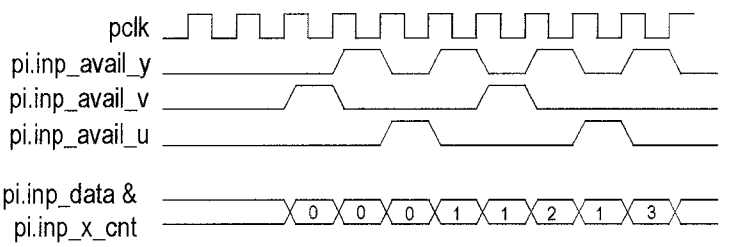
```
   END PROCESS sireg;

   -- bit count register
   sicount : PROCESS
   BEGIN
      WAIT UNTIL clk'EVENT AND clk = '1';
      IF reset = '1' THEN
         si_count <= (OTHERS => '0');
      ELSIF serial_en = '1' THEN
         --stop count wrap-around by holding count at 12
         IF si_count(3 DOWNTO 2) /= "11" THEN
            si_count <= UNSIGNED(si_count)+1;
         END IF;
      ELSE
         si_count <= (OTHERS => '0');
      END IF;
   END PROCESS sicount;

   -- FSM state register
   fsm : PROCESS
   BEGIN
      WAIT UNTIL clk'EVENT AND clk = '1';
      IF reset = '1' THEN
         state <= waiting;
      ELSE
         state <= next_state;
      END IF;
   END PROCESS fsm;

   -- FSM transitions and outputs
   fsmtrans : PROCESS(state, serial_en, si_count, dsi)
   BEGIN
      next_state <= state;
      datardy    <= '0'; databus <= (OTHERS=>'Z');
      CASE state IS

         WHEN waiting =>
            IF serial_en = '1' THEN
               next_state <= receiving;
            END IF;

         WHEN receiving =>
            IF serial_en = '0' THEN
               IF si_count = conv_std_logic_vector(8, 4) THEN
                  next_state <= dataready;
               END IF;
               next_state <= waiting;
            END IF;

         WHEN dataready =>
            datardy <= '1';
            next_state <= waiting;
            databus <= dsi;
      END CASE;
   END PROCESS;
END rtl;
```

PYKC
TOWL

Solution to Question 2

This question tests the student's understanding of procedural abstraction in VHDL testbenches.

a)



b)
pclk provides timing information ('EVENT attribute) and therefore must be a signal of IN mode. The result of calling *emit_line* is to drive *pi* with a time-varying waveform, hence it must be a signal of OUT mode.

c)

```
PROCEDURE emit_vert_blank(
      n                          : IN  INTEGER;
      SIGNAL pclk                : IN  STD_LOGIC;
      SIGNAL vert_blank          : OUT STD_LOGIC
      ) IS
   BEGIN
      vert_blank <= '1';
      FOR i IN 1 TO n LOOP
         WAIT UNTIL pclk'EVENT AND pclk = '1';
      END LOOP;
      vert_blank <= '0';
   END PROCEDURE;
```

PYKC
TJWL

d)

```
PROCEDURE emit_frame(
   SIGNAL pclk       : IN  STD_LOGIC;
   interlaced        : IN  BOOLEAN;
   h_size            : IN  INTEGER;
   v_size            : IN  INTEGER;
   SIGNAL pi         : OUT pix_inp_t;
   SIGNAL vert_blank : OUT STD_LOGIC;
   SIGNAL v          : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0)
   ) IS

   VARIABLE vpos : INTEGER;

BEGIN
   vpos := 0;

   WHILE vpos < v_size LOOP
      vertical_line_number := vpos;
      emit_line( pclk, pi, h_size);
      IF interlaced THEN
         vpos := vpos + 2;
      ELSE
         vpos := vpos + 1;
      END IF;
   END LOOP;

   emit_vert_blank( 10, pclk, vert_blank);

   IF interlaced THEN
      emit_vert_blank( 10, pclk, vert_blank);
      vpos := 1;
      WHILE vpos < v_size LOOP
         vertical_line_number := vpos;
         emit_line( pclk, pi, h_size);
         vpos := vpos+2;
      END LOOP;
      emit_vert_blank( 10, pclk, vert_blank);
   END IF;

   emit_vert_blank(10, pclk, vert_blank);

END PROCEDURE emit_frame;
```

Solution to Question 3

a)
The OTHERS case will be executed if *c*= 'X', 'U', '-', 'H','L','W','Z' i.e. any value in std_logic other than '0' or '1'. Assigning 'X' (undefined) to y in these cases is reasonable since 'weak' logic values ('H' and 'L') are not normally used and therefore *y* need only have a well defined value when inputs to the logic function are '0' or '1'. Synthesis ignores all input values except '0' and '1'. Therefore this case is irrelevant to synthesis. NB – this case is required by the VHDL compiler, which does not allow incomplete case statements.

b)

This process has sensitivity list *(a,c)* and no waits. Hence the statements in it execute in 0 time, after
a) the start of simulation
b) any change in the value of *a* or *c*.
The statements in the process assign a value to y under all possible inputs. Note that the first assignment, *y*<='0', has no effect because it is overridden by a later assignment in all cases. The value of *y* changes 1 simulation delta after the change in *a* or *c* that initiated the process, hence the delay from *a* or *c* to *y* is 1 delta.
After execution of the process reaches the end of the process body execution is suspended until the next transition of *a* or *c*.

c)
unsigned() implements type conversion from (in this case) *std_logic_vector* to *unsigned*. The latter type is equipped with *unsigned* arithmetic operations by library *std_logic_arith* that allow unsigned numeric comparison of *unsigned* vectors. This function is necessary to allow *std_logic_arith* arithmetic on *a*.

unsigned'() selects the type of the literal constant to be u*nsigned*. This type is ambiguous, since the literal could be either *std_logic_vector*, *unsigned*, or *signed*. Without this operation the VHDL compiler will not be able to resolve the type ambiguity and fail, since the > and >= operations are defined on multiple types.

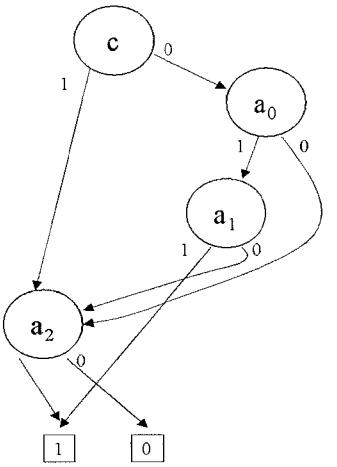
d)

```
ENTITY new_logic_function IS
   GENERIC(
      v :      INTEGER
      );
   PORT(
      a :      STD_LOGIC_VECTOR;
      c : IN   STD_LOGIC;
      y : OUT STD_LOGIC
      );
END new_logic_function;



ARCHITECTURE rtl OF new_logic_function IS
   CONSTANT m: UNSIGNED(a'range) := conv_unsigned( v, a'length);
BEGIN
   logic : PROCESS(a, c)
   BEGIN
      y           <= '0';
      CASE c IS
         WHEN '1'     =>
            IF UNSIGNED(a) > m THEN
               y <= '1';
            END IF;
         WHEN '0'     =>
            IF UNSIGNED(a) >= m THEN
               y <= '1';
            END IF;
         WHEN OTHERS =>
            y     <= 'X';
      END CASE;
   END PROCESS logic;
END rtl;
```

e)
Synthesis
The sensitivity list will be automatically completed, with a warning message, hence no change in the synthesised code.
Simulation.
The output will be updated when *a* changes, but not *c*. This corresponds to unrealisable hardware in which any edge on a updates a latched value of *c*. The output is then derived from the latched *c* and *a*. Thus pre-synthesis simulation is different from post-synthesis simulation.

Solution to Question 4

This question tests whether the students understand how combinatorial logic is implemented, simulated, and synthesised, in VHDL.

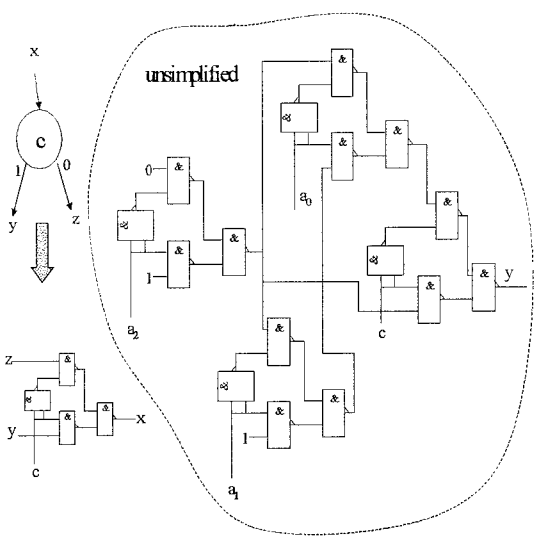


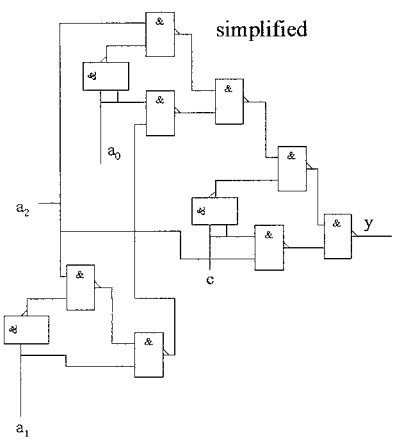
a)

' denotes boolean negation

$y = c * a_2 + c'*(a_2 + a_1*a_0)$

b)





Solution to Question 5

This question tests whether the students are able to write structural VHDL descriptions.

a)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE work.ALL;


ENTITY booth_multiplier IS
   PORT(
      x : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);  --signed
      y : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);  --signed
      q : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)  --signed
      );
END booth_multiplier;


ARCHITECTURE rtl OF multiplier IS

   SIGNAL minus_x : SIGNED(8 DOWNTO 0);
   SIGNAL sum0    : SIGNED(9 DOWNTO 0);
   SIGNAL sum1    : SIGNED(11 DOWNTO 2);
   SIGNAL sum2    : SIGNED(13 DOWNTO 4);
   SIGNAL sum3    : SIGNED(15 DOWNTO 6);
BEGIN

   minus_x <= -SIGNED(x(7) & x);

   m0 : ENTITY booth_mux
      PORT MAP( y(1 DOWNTO 0), x, STD_LOGIC_VECTOR(minus_x),
STD_LOGIC_VECTOR(sum0));

   b0 : ENTITY booth_adder
      GENERIC MAP(10)
      PORT MAP( SIGNED(x), minus_x, sum0, sum1, y(3 DOWNTO 1));

   b1 : ENTITY booth_adder
      GENERIC MAP(8)
      PORT MAP(SIGNED(x), minus_x, sum1(11 DOWNTO 4), sum2, y(5 DOWNTO 3));

   b2 : ENTITY booth_adder
      GENERIC MAP(8)
      PORT MAP(SIGNED(x), minus_x, sum2(13 DOWNTO 6), sum3, y(7 DOWNTO 5));

q <= STD_LOGIC_VECTOR(sum3 & sum2( 5 DOWNTO 4) & sum1(3 DOWNTO 2) &
                      sum0(1 DOWNTO 0));


END rtl;
```

b)

```
ARCHITECTURE rtl OF mul_array IS


   TYPE array_word8 IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);

   TYPE array_word16 IS ARRAY (array_word8'RANGE) OF
            STD_LOGIC_VECTOR(15 DOWNTO 0);

   SIGNAL vx, vy : array_word8;

   SIGNAL vq : array_word16;

BEGIN

   G1    : FOR i IN array_word8'RANGE GENERATE
      m1 : ENTITY booth_multiplier
           PORT MAP( vx(i), vy(i), vq(i));
   END GENERATE;

END rtl;
```

Solution to Question 6

Part a) of the question tests whether students can design arithmetic logic functions. Parts b) and c) test whether the students can design VHDL testbenches with simple file I/O, and whether they understand how to verify hardware modules.

a) (entity given in question)

```
ENTITY booth_adder IS
   GENERIC(
      n        :      INTEGER
      );
   PORT(
      x        : IN  SIGNED(n-1 DOWNTO 0);  --signed
      mx : IN  SIGNED(n DOWNTO 0);    --signed
      si  : IN  SIGNED(n-1 DOWNTO 0);  --signed
      so : OUT SIGNED(n+1 DOWNTO 0);  --signed
      c        : IN  STD_LOGIC_VECTOR(2 DOWNTO 0)
      );
END booth_adder;

ARCHITECTURE rtl OF booth_adder IS
   SIGNAL z : SIGNED(n+1 DOWNTO 0);
BEGIN
   PROCESS(x, mx, c)
   BEGIN
      CASE c IS
         WHEN "000" | "111"          =>
            z                              <= (OTHERS => '0');
         WHEN "001" | "010"          =>
            z                              <= x(7) & (x(7) & x);
         WHEN "101" | "110"          =>
            z                              <= mx(8) & mx;
         WHEN "011"                  =>
            z                              <= x(7) & x & '0';
         WHEN "100"                  =>
            z                              <= mx & '0';
         WHEN OTHERS                 => z <= (OTHERS => 'X');
      END CASE;
   END PROCESS;

   PROCESS( sum_in, z)
   BEGIN
      so <= SIGNED(si) + SIGNED(z);
   END PROCESS;

END rtl;
```

b)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE STD.textio.ALL;

ENTITY test_booth_adder IS
END test_booth_adder;



ARCHITECTURE beh OF test_booth_adder IS

   CONSTANT n      : INTEGER := 8;
   CONSTANT tdelay : TIME    := 1 us;
   FILE testdata   : TEXT OPEN read_mode IS "testfile";

   SIGNAL x_i  : SIGNED(n-1 DOWNTO 0);  --signed
   SIGNAL mx_i : SIGNED(n DOWNTO 0);    --signed
   SIGNAL si_i : SIGNED(n-1 DOWNTO 0);  --signed
   SIGNAL so_i : SIGNED(n+1 DOWNTO 0);  --signed
   SIGNAL c_i  : STD_LOGIC_VECTOR(2 DOWNTO 0);


BEGIN

   PROCESS
      VARIABLE buf              : LINE;
      VARIABLE x, mx, si, so, c : INTEGER;
      VARIABLE line_num: INTEGER;
   BEGIN
      line_num := 0;
      WHILE NOT endfile(testdata) LOOP
         line_num := line_num+1;
         readline( testdata, buf);
         read( buf, x);
         read( buf, mx);
         read( buf, si);
         read( buf, so);
         read( buf, c);

         x_i  <= conv_signed(x, n);
         mx_i <= conv_signed(mx, n+1);
         si_i <= conv_signed(si, n);
         c_i  <= conv_std_logic_vector(c, 3);

         WAIT FOR tdelay;

         ASSERT conv_integer(so_i) = so
            REPORT "verification error : line " & INTEGER'IMAGE(line_num)
            SEVERITY error;

      END LOOP;
      REPORT "Test Completed" SEVERITY note;
   END PROCESS;

END beh;
```

c)

In all cases test data can be generated by writing a Golden C equivalent of this function and generating the testbench stimulus files from this. Naïve exhaustive testing of this block would require 2**(8+9+8+3)=2**28 distinct tests. With an elapsed time of 1ms/test this would take 256,000s ~ 80 hours. This would give excellent verification, and be easy to implement. However the 80 hour test length is inconvenient. Random testing, with all inputs randomised, would give reasonably good confidence very quickly.

A clever strategy, with coverage as good as exhaustive testing, and also quick, is modifies exhaustive testing. Notice that, if the block is working correctly, the output y depends on either *mx* or *x* but not both. Therefore for each *c* value we can exhaustively test on all inputs except the one that should be don't care. That input can be given random values throughout the test sequence.

If n=16 the time becomes 2*24 (exhaustive) or 2**16 (modified exhaustive) worse. Even the modified strategy now takes ~ 1 year. In this case the best solution is to run random tests, with all inputs randomised, and measure test coverage for example by post-synthesis VHDL line coverage (pre-synthesis line coverage would give little information about the arithmetic operations).