Answers to Specimen Paper (new format) 2003

Question 1

a) Write a VHDL entity and architecture that implements a multi-input OR gate with an arbitrary length std_logic_vector x as input and output y.

[5 marks]

```
ENTITY orgate IS
PORT(
   x: std_logic_vector;
   y: std_logic
   );
END orgate;

ARCHITECTURE rtl OF orgate IS
   VARIABLE v := '0';
BEGIN
   OR: PROCESS(x)
   BEGIN
         FOR j IN x'RANGE LOOP
               v := v or x(j);
         END LOOP;
   END PROCESS OR;
   y <= v;
END ARCHITECTURE rtl;
```

b) Describe clearly ways of writing a synthesisable clocked process with and without a sensitivity list, and give an example of a process that could implement a 7 bit negative edge triggered counter with asynchronous clear and synchronous set.

Either use sensitivity list containing all process inputs and inside process:

IF (clock edge condition) THEN <body> END IF;

OR no sensitivity list and first statements of process is:

WAIT UNTIL (clock edge condition)

-- q,d defined as 7 bit std_logic_vector

```
PROCESS(clk, d, clear, set)
BEGIN
      IF clear='1' THEN
            q <= (OTHERS=>'0');
      ELSIF clk'EVENT AND clk='0' THEN
            IF set='1' THEN
                  q <= (OTHERS=>'1');
            ELSE
                  q <= d;
            END IF;
      END IF;
END PROCESS;
```

[5 marks]

c) Describe precisely the hardware synthesized from each *and*, *or*, *xor*, =, +, - operator in the process shown in *Figure 1.1*.

[5 marks]

See code for hardware

```
ENTITY clocked IS
GENERIC( n: INTEGER := 4);
PORT (     a,b: IN std_logic_vector(7 DOWNTO 0);
           x,y: OUT std_logic_vector(7 DOWNTO 0);
           z: OUT std_logic);
END ENTITY clocked;

ARCHITECTURE rtl OF clocked IS
BEGIN
P2: PROCESS(a,z)
  VARIABLE j: INTEGER;
BEGIN
  x <= signed(a) (- 1 + (n MOD 3)); -- no hardware synthesized from this
(all static)
  y <= b xor conv_signed(255,8); -- 8 invertors
  FOR i IN 0 TO n LOOP
      j := i - 1; -- no hardware (static -)
      z(i) <= a(i) and (b(i) or c(j)); --n+1 ands and n+1 ors
  END LOOP;
END PROCESS P2;
END ARCHITECTURE rtl;
```

d) Write a synthesizable architecture for entity *compare* in *Figure 1.2* such that, if *a*,*b* are interpreted as signed integers and *c*,*d* as unsigned integers:

$x = a > b$

$y = a < c$

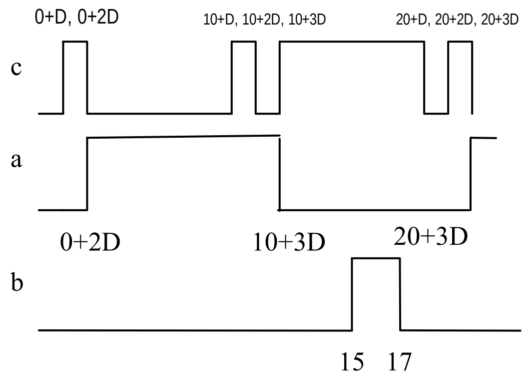
$z = (a{=}b)$ and $(c{=}d)$

w = 4 LSB of c if $a > 0$, otherwise 4 MSB of d.

```
ARCHITECTURE rtl OF compare IS
BEGIN
   P1 PROCESS(a,b,c,d)
   BEGIN
   x <= '0'; y <= '0'; z <= '0';
   IF signed(a)>signed(b) THEN
        x<='1';
   END IF;
   IF signed(a)<unsigned(c) THEN
        y <='1';
   END IF;
   IF (a=b) AND (c=d) THEN
        z <='1';
   END IF;
   w <= d(7 DOWNTO 4);
   IF signed(a) > conv_signed(0,8) THEN
        w <= c(3 DOWNTO 0);
   END IF;
   END PROCESS P1;

END ARCHITECTURE rtl;
```

[5 marks]

e) The architecture in *Figure 1.3* is part of a testbench and generates signals *a*,*b*,*c*,*d*. Draw a dimensioned timing diagram showing the waveforms and simulation times of events on signals *a*,*b*,*c*,*d* for the first 20ns of the simulation.



[5 marks]

f) Write a VHDL entity and architecture *add* which implements a 4**n* bit adder with inputs *p*,*q* each 4**n* bits long and output *r* 4**n*+1 bits long, using *n* instances of the 4 bit full adder entity shown in *Figure 1.4*, which adds *p* and *q* with *cin* to generate sum *r* and carry out *cout*.

```
ENTITY add IS
GENERIC( n: INTEGER);
PORT ( p,q: std_logic_vector( 4*n-1 DOWNTO 0);
   r: std_logic_vector(4*n DOWNTO 0)
);
END add;
```

```
ARCHITECTURE rtl OF add IS
BEGIN
  FOR j IN 0 TO n-1 GENERATE
       I1: ENTITY adder4 PORT MAP(
            cin=>carry(j);
            cout => carry(j+1);
            p=>p(4*j+3 DOWNTO 4*j);
            q=>q(4*j+3 DOWNTO 4*j);
            r=>r(4*j+3 DOWNTO 4*j);
  END GENERATE;

  carry(0) <= '0';
  r(4*n) <= carry(n);
END ARCHITECTURE rtl;
```

[5 marks]

g) Explain, with reference to the entity *compare* in *Figure 1.2*, the terms *exhaustive testing* and *corner case* in test methodology.

Exhaustive testing : try every input pattern, in this case there are: $2^8 * 2^8 * 2^8 * 2^8$ tests needed.

Corner cases: "difficult" input values, in this case a=b, a=c, c=d are all possible corner cases because of the comparisons. Also all combinations of min, max values of a,b,c,d.

[5 marks]

h) Write a (non-synthesisable) function *funny* that returns TRUE if there is currently an event on its std_logic input x, and either the previous or new value of x is not '0' or '1'.

[5 marks]

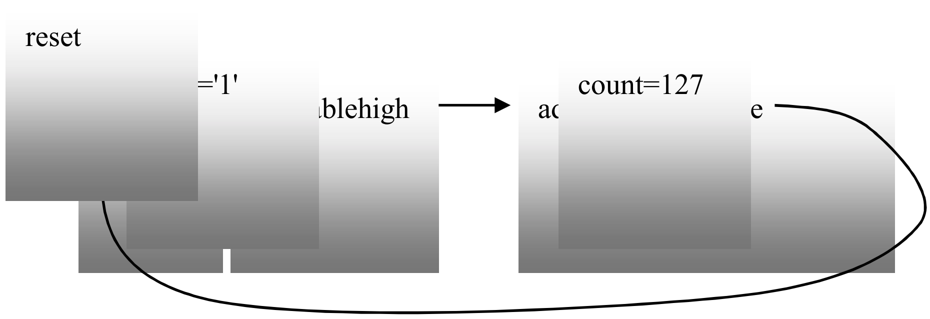
```
FUNCTION funny( SIGNAL x: IN std_logic) RETURN BOOLEAN IS
BEGIN

  IF NOT x'EVENT RETURN FALSE; END IF;

  CASE x IS
  WHEN '0' | '1' => NULL;
  WHEN OTHERS => RETURN TRUE;
  END CASE;

  CASE x'LAST_VALUE IS
  WHEN '0' | '1' => NULL;
  WHEN OTHERS => RETURN TRUE;
  END CASE;

  RETURN FALSE;

END FUNCTION funny;
```

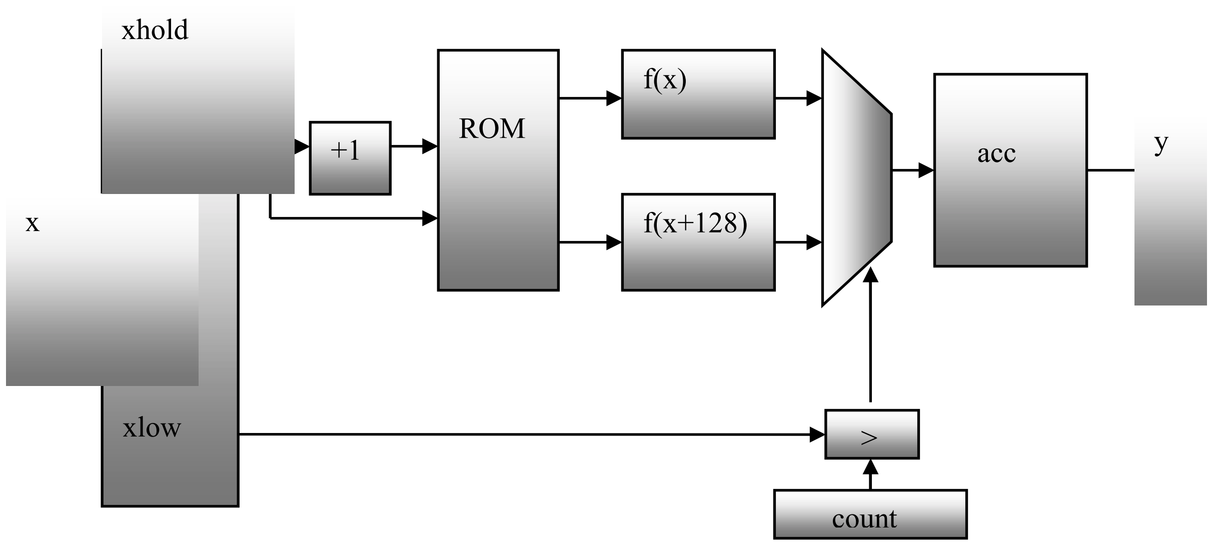
Question 2

Parts a), b) of this question test whether the student can write RTL synthesizable VHDL descriptions of hardware described at an algorithmic level. Parts c) and d) test whether the student understands how to write structural descriptions.

a)

FSM





ROM (comb) : r : = if sel='0' then f(xhigh) else f(xholdhigh+1)

XHOLD (clocked) : if start='1' then xhold := x

ACC: if init then acc := 0 else if sum then

acc := (if xholdlow >= count then f(xhold) else f(xholdhigh+1);

COUNT: if add then count := count+1 else count := 0;

b)

```
ARCHITECTURE rtl OF funcgen IS
   TYPE state IS (init, readtablehigh, sum, donestate);
   SIGNAL sel: std_logic;
   SIGNAL ss: state;
   SIGNAL xh: std_logic_vector(8 DOWNTO 0);
   SIGNAL xl, count: std_logic_vector(6 DOWNTO 0);
   SIGNAL romout: std_logic_vector(14 DOWNTO 0);
   SIGNAL acc: std_logic_vector(21 DOWNTO 0);
BEGIN
   XHOLD: PROCESS
   BEGIN
      WAIT UNTIL clk'EVENT AND clk='1';
      IF start='1' THEN
         xh <= '0' & x(14 downto 7);
         xl <= x(6 DOWNTO 0);
      END IF;
   END PROCESS XHOLD;

   ROM: PROCESS( xh, xl, count)
      VARIABLE romin: std_logic_vector(9 DOWNTO 0);
      VARIABLE sel: BOOLEAN;
   BEGIN
      sel := count > xl;
      IF sel THEN
         romin := unsigned('0' & xh)+1;
      ELSE
         romin := '0' & xh;
      END IF;

      romout <= func_table(conv_integer(unsigned(romin)));

   END PROCESS ROM;

   ACC_COUNT: PROCESS
   BEGIN
      WAIT UNTIL clk'EVENT AND clk='1';
      IF ss = sum THEN
         acc <= unsigned(acc) + unsigned(romout);
         count <= unsigned(count)+1;
      ELSE
         count <= (OTHERS=>'0');
      END IF;
   END PROCESS ACC_COUNT;

   FSM: PROCESS
   BEGIN
    WAIT UNTIL clk'EVENT AND clk='1';
    CASE ss IS
      WHEN init => IF start='1' THEN ss <= readtablehigh; END IF;
      WHEN readtablehigh => ss <= sum;
      WHEN sum => IF unsigned(count) = conv_unsigned(127,7)  THEN
                     ss <= donestate;
                  END IF;
      WHEN donestate => ss <= init;
      END CASE;
      IF reset = '1' THEN ss <= init; END IF;
   END PROCESS FSM;

   y <= acc(21 DOWNTO 7);

END ARCHITECTURE rtl;
```

c)

```
ENTITY funcgen_new IS
GENERIC( table: table_type := func_table);
PORT(
      reset, clk, start: IN std_logic;
      done: OUT std_logic;
      x: IN std_logic_vector(14 DOWNTO 0);
      y: OUT std_logic_vector( 14 DOWNTO 0)
);
END ENTITY funcgen_new;
```

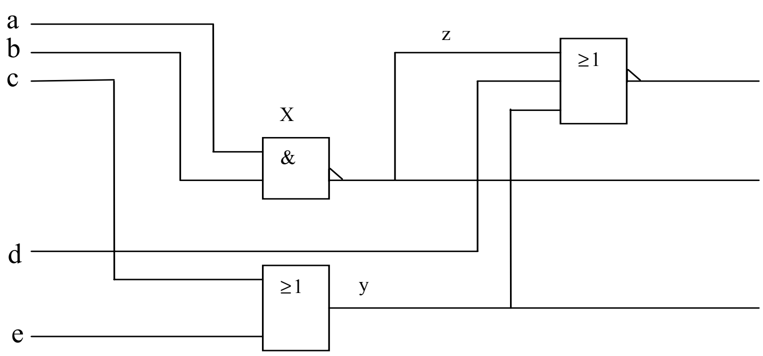
d)

```
ARCHITECTURE rtl OF mult_funcgen IS
   SIGNAL ya,yb: std_logic_vector(14 DOWNTO 0);
   SIGNAL dummy: std_logic; -- dummy signal;
BEGIN

   TA: ENTITY funcgen_new GENERIC MAP(func_table_a)
          PORT MAP(reset,clk,start, done, x1, ya);

   TB: ENTITY funcgen_new GENERIC MAP(func_table_b)
          PORT MAP(reset,clk,start, dummy, x2, yb);

   y1 <= unsigned(ya)+unsigned(yb);

   y2 <= ('0' & unsigned(ya))-(('0' & unsigned(yb))+2**14);

END ARCHITECTURE rtl;
```

Question 3

a) *Figure 3.1 shows one gate-level implementation of a circuit with 4 inputs and 3 outputs. Using transduction one of these gates can be eliminated, without altering the circuit's function. Draw the reduced circuit, and describe why the transformation is possible.*

[4]

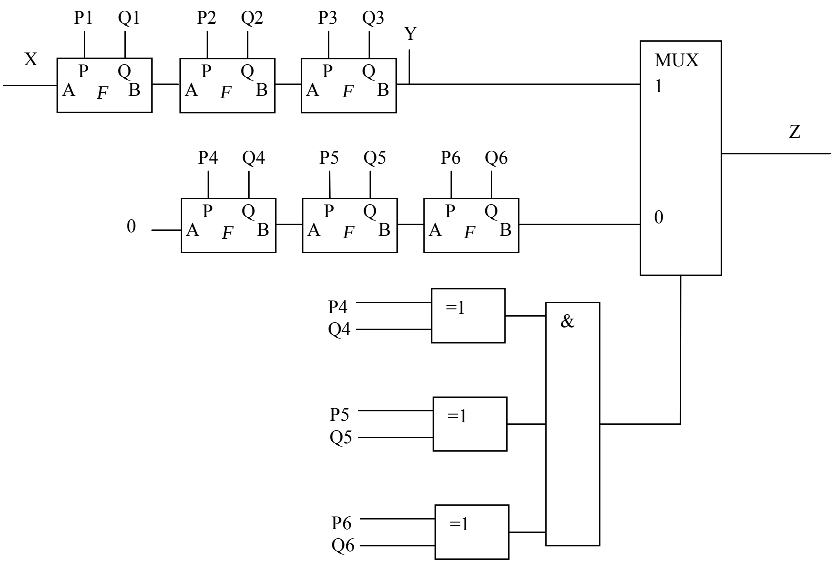


When c is 0, the gate X is equivalent to the original 3 input NAND gate. When c is 1, y is 1, hence z is don't care.

b) *Figure 3.2 shows a critical path from X to Z in a circuit. Each of the blocks F is defined by: B = P.Q + P.A + Q.A. By applying controllability factoring at point Y, derive an equivalent circuit with reduced critical path length. What is your control function C?*

[8]

C= (P4 xor Q4).(P5 xor Q5).(P6 xor Q6).



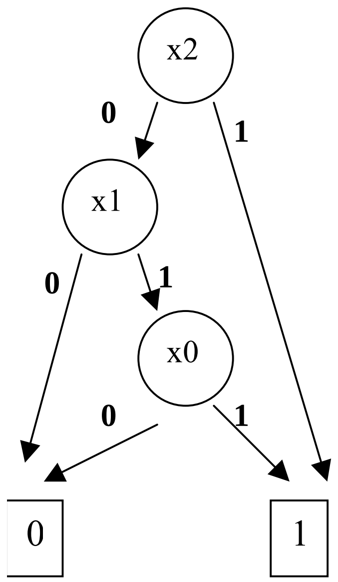
c) *The VHDL fragment in Figure 3.3 defines y as a boolean function of x, where x has type* `std_logic_vector( 2 downto 0)`. *Write a truth table for* `y`, *and compute two ROBDDs for* `y` *using variable orders:* `x(0),x(1),x(2)`, *and* `x(2),x(1),x(0)` *respectively.*

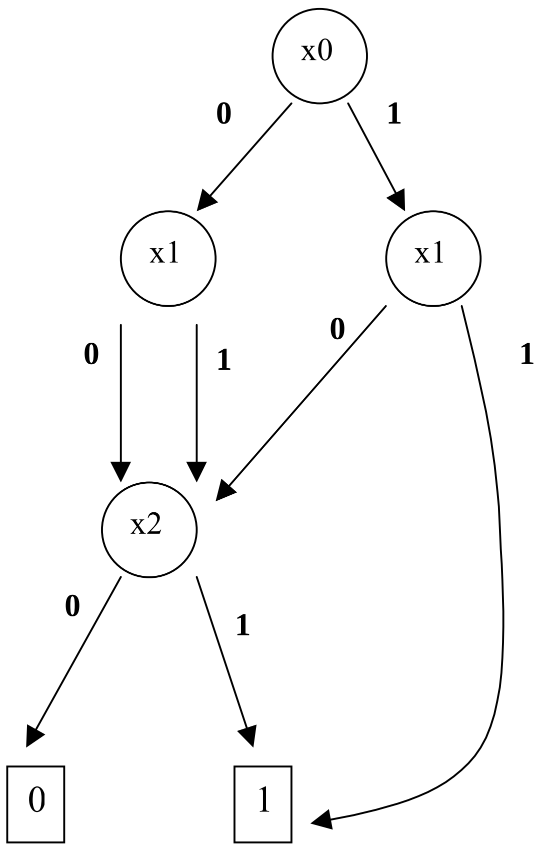
[8]

X2	X1	X0	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Order: x(2),x(1),x(0)

Order x(0), x(1), x(2)



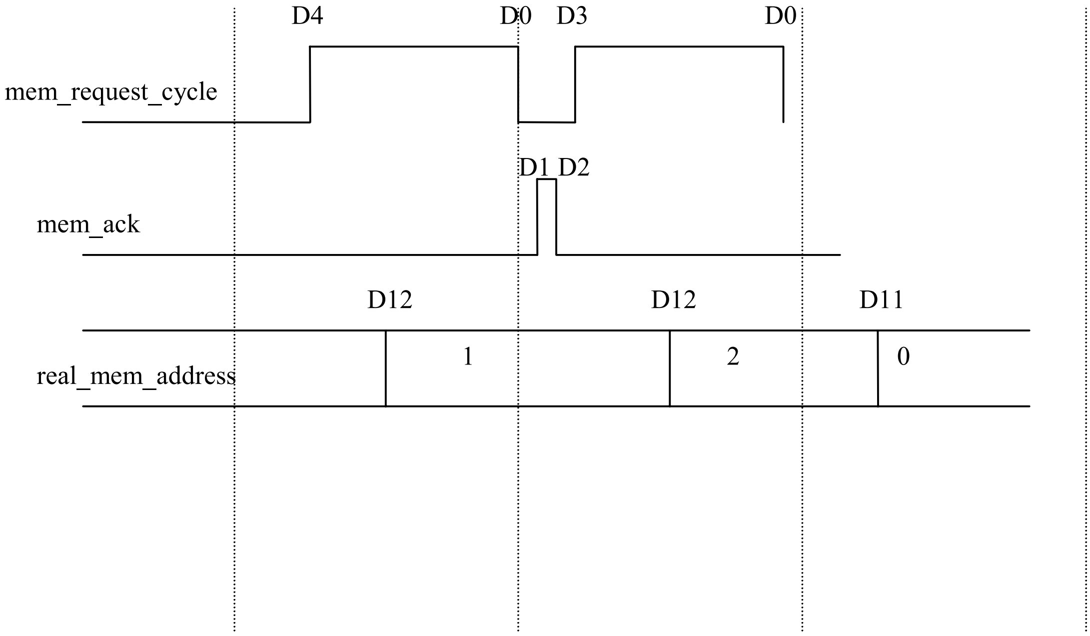


Question 4.

d) *Draw the waveforms of all signals and shared variables used in test_mem_driver, until the final (indefinite) wait statement in process p1 is executed. You must indicate precise timing of all signal and shared variable transitions, including delta delays where relevant.*

[10]

D = number of deltas from clock edge. All times are referenced from clock rising edge, at 50, 150, 250. NB clock edge is actually at delta 1 in this architecture, so add 1 for true deltas. mem_data changes on falling edge of mem_request cycle. mem_addr changes 1 delta after rising edge of mem_request_cycle.



e) *During what time window after a clock edge will read_cycle have this behaviour?*

[5]

`mem_request_cycle` is tested 11delta after the clock edge by mem_driver proc. For it to be certainly read as set, it must be set 10delta after clock => read_cycle executed 8delta after clock edge. mem_request cycle is tested by read_cycle 2delta after the call, and reset by mem_driver_proc on the clock edge. So window is clock edge to clock edge + 8delta.

f) *Draw a diagram indicating the order of call and return of each of the three* `read_cycle` *procedure calls executed during test 2. If more than one result is possible indicate all possibilities.*

[5]

Depending which of the 1st read_cycles in p1 or p2 is executed 1st, 1,100,2 or 100,1,2. The waiting read_cycle will test mem_request_cycle before the newly called one, hence these are only two orders possible. Both the 1st p1 & p2 read_cycles will be called at the same time, the one that executes first will return after 1 cycle, the other after 2. The 2nd p1 read cycle will be called immediately after the 1st cycle end and therefore take 1 or 2 cycles, terminating at the end of the 3rd cycle from the start.