

Specimen Paper - new format 2003

VHDL & LOGIC SYNTHESIS

Examiner: T. J. W. Clarke

There are FOUR questions. Answer Question 1 and any TWO of Questions 2,3,4.

Question 1 carries 40% of the total mark, all other questions carry 30% of the total mark.

1. All eight parts of this question carry equal marks.

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]

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.

[5 marks]

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

[5 marks]

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:

$$x = a > b$$

$$y = a < c$$

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

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

[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 *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*.

[5 marks]

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

[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]

```

ENTITY clocked IS
GENERIC( n: INTEGER := 4);
PORT (      a,b: IN std_logic_vector(7 DOWNT0 0);
        x,y: OUT std_logic_vector(7 DOWNT0 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));
    y <= b xor conv_signed(255,8);
    FOR i IN 0 TO n LOOP
        j := i - 1;
        z(i) <= a(i) and (b(i) or c(j));
    END LOOP;
END PROCESS P2;
END ARCHITECTURE rtl;

```

Figure 1.1

```

ENTITY compare IS
PORT (      a,b,c,d: IN std_logic_vector(7 DOWNT0 0);
        x,y,z: OUT std_logic;
        w: std_logic_vector(3 DOWNT0 0);
    );
END compare;

```

Figure 1.2

```

ARCHITECTURE behave OF testbench IS
  SIGNAL a,b,c: std_logic := '0';
BEGIN
  P3: PROCESS
  BEGIN
    c <= not c;
    WAIT FOR 0 ns;
    c <= not c;
    a <= c; WAIT FOR 10 ns;
    c <= not c;
    b <= c;
    WAIT FOR 0 ns;
    b <= TRANSPORT '1' AFTER 5 ns;
    b <= TRANSPORT '0' AFTER 7 ns;
  END PROCESS P3;
END ARCHITECTURE behave;

```

Figure 1.3

```

ENTITY adder4 IS
PORT( p,q: IN std_logic_vector( 3 DOWNT0 0);
      cin: IN std_logic;
      r: OUT std_logic_vector( 3 DOWNT0 0);
      cout: IN std_logic;
      );
END ENTITY adder4;

```

Figure 1.4

2. This question concerns the implementation and use of the ROM-based VHDL function generator entity *funcgen* shown in *Figure 2.1*. Operation of this entity is as follows. The entity has positive edge triggered clock *clk*. The 15 bit unsigned input *x* is read when *start* is '1' (for a single *clk* cycle). After some number of *clk* cycles the unsigned 15 bit output *y* will be equal to the required output $f(x)$, during a single *clk* cycle in which output *done* is '1'. The number of cycles between *start* high, and the corresponding *done* high is not specified, but may be assumed to be fixed. Synchronous signal *reset* initialises the entity on power-up and is held '0' at all other times.

The ROM table *func_table* defining function *f* has type *table_type* defined as follows:

```
TYPE table_type IS
    ARRAY (0 TO 512) OF std_logic_vector(14 DOWNTO 0);
```

This table is used to define values of the function *f* as follows:

$$f(n*128) = func_table(n) \quad (0 \leq n \leq 512).$$

Intermediate values of the function *f* are computed by linear interpolation between $f(n*128)$ and $f((n+1)*128)$, so that splitting *x* into high and low order bits, $x = x_{high}*128 + x_{low}$ ($0 \leq x_{low} < 128$):

$$f(x) = (f(x_{high})*(128 - x_{low}) + f(x_{high} + 1)*x_{low} + 64)/128.$$

(Note:- $f(x)$ is rounded down to the nearest integer)

- a) A slow but space-efficient implementation of *funcgen* is proposed in which a single constant lookup table is used in successive clock cycles to compute $f1 = f(x_{high})$ and $f2 = f(x_{high} + 1)$ and the result *y* is computed in constant time from $128 - x_{low}$ successive additions of *f1* and *xlow* successive additions of *f2* to a single accumulator register, set initially to 64. After these additions the result may be obtained from the appropriate higher-order accumulator bits. Write down the necessary FSM and RTL hardware blocks to implement this operation, indicating the (possibly state-dependent) operation of each block.

[10 marks]

- b) Write a VHDL architecture to implement *funcgen*, assuming that *func_table* and *table_type* are defined in a package *funcgen_pkg* which you are given.

[10 marks]

- c) Rewrite entity *funcgen* as *funcgen_new*, in which the ROM array is an optional generic parameter, with default value *func_table*.

[2 marks]

- d) The entity *mult_funcgen* in *Figure 2.2* has two 15 bit unsigned inputs *x1*, *x2* and 16 bit unsigned outputs *y1*, *y2*. In *mult_funcgen* is proposed to use two instances of *funcgen_new* to implement two different functions *fa* and *fb* from corresponding tables *func_table_a* and *func_table_b* both contained in *funcgen_pkg*. Write a synthesizable VHDL architecture for the entity *mult_funcgen* in which the timing of *start* and *done* is similar to that of *funcgen*, with *x1* and *x2* replacing *x*, and *y1* and *y2* replacing *y*. The outputs *y1* and *y2* are defined as follows:

$$y1 = fa(x1) + fb(x2)$$

$$y2 = fa(x1) - fb(x2) + 2^{14}$$

You may assume that integer array valued generic parameters are fully supported by the VHDL compiler and synthesis system.

[8 marks]

```
ENTITY funcgen IS
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;
```

Figure 2.1

```
ENTITY mult_funcgen IS
PORT(
reset, clk, start: IN std_logic;
done: OUT std_logic;
x1,x2: IN std_logic_vector(14 DOWNTO 0);
y1,y2: OUT std_logic_vector( 15 DOWNTO 0)
);
END ENTITY mult_funcgen;
```

Figure 2.2

3.

- a) *Figure 3.1* shows one gate-level implementation of a circuit with 5 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.

[10]

- 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 ?

[10]

- c) The VHDL fragment in *Figure 3.3* defines y as a Boolean function of $x(i)$, 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.

[10]

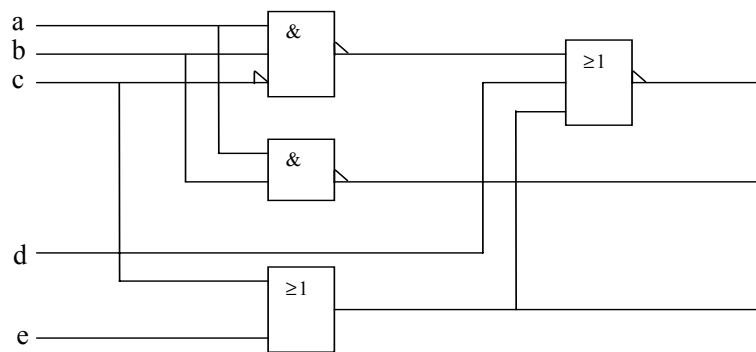


Figure 3.1

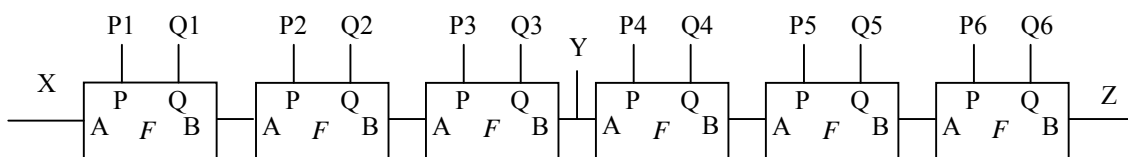


Figure 3.2

```

PROCESS(x)
BEGIN
  IF SIGNED(x) > 2 THEN
    y <= '1';
  ELSE
    y <= '0';
  END IF;
END PROCESS;

```

Figure 3.3

4. *Figure 4.1* gives VHDL source for an entity `test_mem_driver` with a behavioural architecture, and a package `comms` containing procedure `read_cycle`. The `test_mem_driver` entity has a positive edge active clock `clk`, interfaces to a RAM through address and read data busses, and has control signals `start` and `test`, as illustrated in *Figure 4.2*. In operation `start` is pulsed to 1 for 1 clock cycle, initiating a test of the memory driving code. Two tests are possible, test 1 or test 2, according to the numeric value of `test` during the clock cycle in which `start` is '1'. After some time all activity will finish. In order to initiate another test the simulation must be restarted.

a) Initially `mem_request_cycle` is false. `Start` is pulsed high for one `clk` cycle, with `test = 1`. 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 simulation deltas where relevant.

[10]

b) It is intended that a call to `read_cycle` will initiate a 1 cycle long read of the RAM, at the address specified by the value of `addr`. During what time window after a clock edge must `read_cycle` be called for this behaviour to result?

[10]

c) You may assume that in a VHDL simulation only one process may be executing at a given time, and that a process will always continue executing until it is suspended by a wait statement or wait on sensitivity list. Multiple processes scheduled to start on the same delta are therefore sequenced (in an arbitrary order). Draw a diagram indicating the timing 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.

[10]


```

ENTITY test_mem_driver IS
  PORT (real_mem_address : OUT INTEGER;
        real_mem_data    : IN  STD_LOGIC_VECTOR( 7 DOWNT0 0);
        start,           : IN  STD_LOGIC;
        test             : IN  INTEGER
        );
END test_mem_driver;

ARCHITECTURE behav OF test_mem_driver IS
  SIGNAL clk      : STD_LOGIC;
  SIGNAL mem_ack  : BOOLEAN;
BEGIN

  clkgen : PROCESS
  BEGIN
    clk <= '0';
    WAIT FOR 50 ns;
    clk <= '1';
    WAIT FOR 50 ns;
  END PROCESS clkgen;

  mem_driver_proc : PROCESS
  BEGIN
    FOR i IN 1 TO 10 LOOP
      WAIT FOR 0 ns;
    END LOOP;
    IF mem_request_cycle THEN
      real_mem_address <= mem_address;
      WAIT UNTIL clk'EVENT AND clk = '1';
      mem_data         := real_mem_data;
      mem_ack          <= true;
      mem_request_cycle := false;
      WAIT FOR 0 ns;
      mem_ack          <= false;
    ELSE
      real_mem_address <= 0;
      WAIT UNTIL clk'EVENT AND clk = '1';
      WAIT FOR 0 ns;
      mem_data         := (OTHERS => 'X');
    END IF;
  END PROCESS mem_driver_proc;

  p1 : PROCESS
  VARIABLE a, b : STD_LOGIC_VECTOR( 7 DOWNT0 0);
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1' AND start = '1';
    WAIT FOR 0 ns;
    WAIT FOR 0 ns;
    read_cycle( 1, a, mem_ack, clk);
    read_cycle( 2, b, mem_ack, clk);
    WAIT;
  END PROCESS p1;

  p2 : PROCESS
  VARIABLE c : STD_LOGIC_VECTOR( 7 DOWNT0 0);
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1' AND start = '1' AND test = 2;
    WAIT FOR 0 ns;
    WAIT FOR 0 ns;
    read_cycle( 100, c, mem_ack, clk);
    WAIT;
  END PROCESS p2;
END behav;

```

Figure 4.1 (continued on next page)

```

PACKAGE comms IS

    SHARED VARIABLE mem_request_cycle : BOOLEAN := false;
    SHARED VARIABLE mem_address      : INTEGER;
    SHARED VARIABLE mem_data         : STD_LOGIC_VECTOR( 7 DOWNTO 0 );

    PROCEDURE read_cycle(
        addr          : IN  INTEGER;
        VARIABLE data : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL  ack   : IN  BOOLEAN;
        SIGNAL  clk   : IN  STD_LOGIC);

END PACKAGE comms;

PACKAGE BODY comms IS

    PROCEDURE read_cycle(
        addr          : IN  INTEGER;
        VARIABLE data : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL  ack   : IN  BOOLEAN;
        SIGNAL  clk   : IN  STD_LOGIC) IS
    BEGIN
        WAIT FOR 0 ns;
        WAIT FOR 0 ns;
        WHILE mem_request_cycle = true LOOP
            WAIT UNTIL clk'EVENT AND clk = '1';
            WAIT FOR 0 ns;
            WAIT FOR 0 ns;
        END LOOP;
        mem_request_cycle := true;
        mem_address      := addr;
        WAIT UNTIL ack;
        data              := mem_data;
    END read_cycle;

END PACKAGE BODY comms;

```

Figure 4.1 (continued from previous page)

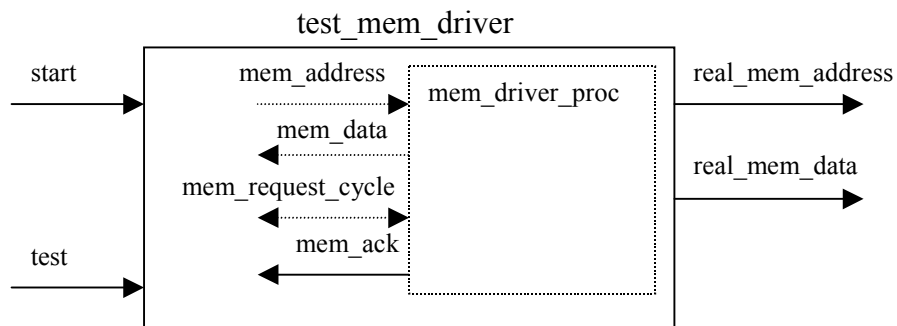


Figure 4.2