Special Information for Invigilators: none.

Information for Candidates

VHDL language reference and course notes can be found in the booklet VHDL Exam Notes.

Unless otherwise specified assume VHDL 1993 compiler.

Library functions from the VHDL package ***utility_pack*** *from the coursework may be used freely in your implementations.*

The Questions

Question 1 is COMPULSORY

1. a)

(i) Determine the precise behaviours of processes `P1 & P2` in *Figure 1.1*, assuming that z is initially '0'. Is P1 synthesisable?

(ii) You may assume that synthesis of a VHDL process depends only on its precise behaviour in simulation. State, giving reasons, whether P2 is synthesisable?

[4]

b) *Figure 1.2* shows a random access memory entity *ram* with data output *dout* and control input *write,* address input *addr* and data input *din*. In *ram* what is the width of one memory location and the number of memory locations? Write a synthesisable architecture for entity *ram* implementing the function shown in *Figure 1.2*.

[4]

c) Calculate the ROBDD for the Boolean expression:

X **xor** Y **xor** Z **xor** W

with variable order X, Y, Z, W

[4]

d) You may assume that *a*, *b*, *c*, *d* in *Figure 1.3* are all '0' at the start of simulation. Draw the first 20ns of the waveforms on these signals when driven by process P3, annotating each signal transition with its physical time and indicating the simulation Δ of the transition if this is not 0.

[4]

e) Entity *iobuff* in *Figure 1.4* has ports *za, zb* which are both inputs and (tri-state) clocked outputs. Its behaviour is described in *Figure 1.5* where the notation x_i denotes the value on signal *x* during clock cycle *i*. Write an architecture for *iobuff* consistent with *Figure 1.5*.

[4]

```
P1:PROCESS
BEGIN
  WAIT UNTIL clk'EVENT;
  z <= not z;
END PROCESS P1;
```

```
P2:PROCESS
BEGIN
  WAIT UNTIL clk'EVENT;
  IF clk='1' THEN
    z <= not z;
  END IF;
END PROCESS P2;
```

Figure 1.1

```
ENTITY ram IS
PORT(
     write, clk: IN std_logic;
     addr: IN std_logic_vector(3 DOWNTO 0);
     din: IN std_logic_vector(7 DOWNTO 0);
     dout: OUT std_logic_vector(7 DOWNTO 0)
);
END ram;
```

write	*Operation on next clock positive edge*	*Dout in current cycle*
1	ramloc[addr] := din	ramloc[addr]
0	n/a	ramloc[addr]

Figure 1.2

```
P3:PROCESS
  VARIABLE xv : std_logic;
BEGIN
  WAIT FOR 10 ns;
  xv := a;
  a <= not xv;
  b <= not a;
  WAIT FOR 0 ns;
  c <= b;
  d <= b AFTER 5 ns;
END PROCESS P3;
```

Figure 1.3

```
ENTITY iobuff IS
  PORT(dir,clk:IN std_logic;
      za,zb:INOUT std_logic
);
END iobuff;
```

Figure 1.4

dir_n	za_n	zb_n
1	zb_{n-1}	High impedance
0	High impedance	za_{n-1}

Figure 1.5

Students must answer TWO out of Questions 2-4.

2. *Figure 2.2* shows an implementation of entity *transpose* in *Figure 2.1*. This uses a 16 word RAM *tram* to implement 4X4 matrix transposition. The circuit is synchronous with the *negative* edge of *clk*. The operation is initiated by a '1' on *reset* and controlled by finite state machine *fsm* with three states as in *Figure 2.3*. Note that the double vertical lines indicate where clock cycles have been omitted.

   During *state1*, the16 matrix elements each of width 16 bits are input sequentially on *din* in row order and written to the RAM. In *state2* the 16 matrix elements are output in column order. As this happens all 16 elements are summed in block *alureg*. In *state3*, lasting 1 cycle, the sum is output. The address inputs of *tram* are driven by a block a*mux* which permutes address inputs in state 2 to implement the transposed element order. Block *muxd* selects the appropriate signal for *dout*.

   a) Write appropriate VHDL to define type *word* as a 16 bit vector which can be used in entities instantiating *transpose* as well as the entity and architecture of *transpose* and *tram*.

   [3]

   b) The RAM *tram* is implemented as a separate entity. Write an appropriate entity declaration for *tram*. You do not need to write an architecture for *tram*.

   [2]

   c) Write an architecture for *transpose* which instantiates *tram* and implements blocks a*mux*, *muxd*, *fsm*, *alureg*, and *count*.

   [15]

```
ENTITY transpose IS
  PORT (
      clk, reset : IN STD_LOGIC;
      datain  : IN word;
      dataout : OUT word
    );
END transpose;
```

Figure 2.1

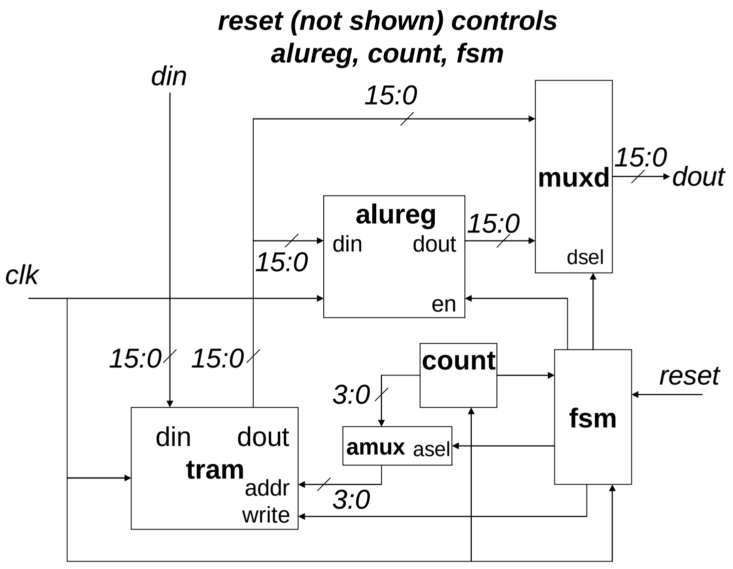

Figure 2.2

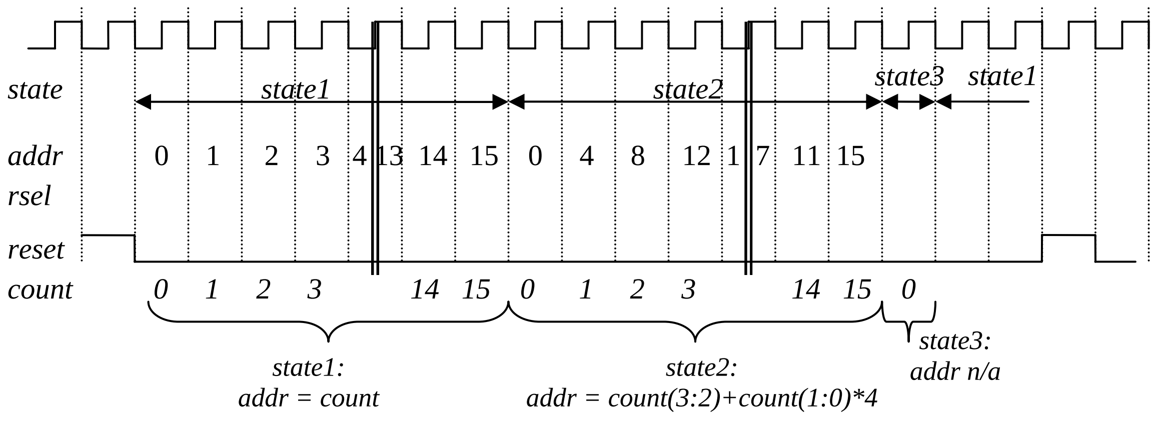

Figure 2.3

3. Figure 3.1 illustrates a hardware engine to compute the Mandelbrot fractals which implements repeatedly two complex number iterations each of the form:

$$z'_r := \textbf{round}\{(z_r*z_r - z_i*z_i)*2^{-n}\} + c_r$$
$$z'_i := \textbf{round}\{2z_i*z_r*2^{-n}\} + c_i \qquad (3.1)$$

where z_i, c_i, z_r, c_r are two's complement m-bit signed integers representing imaginary and real parts of (fixed point) complex numbers z and c respectively, and z' is the new value of z. The function **round** implements rounding to the nearest signed integer.

a) A fixed-point signed vector $x(m\text{-}1{:}0)$ with n fractional bits ($n < m$) may be rounded to the nearest fixed-point integer by adding bit $x(n\text{-}1)$ onto x at bit index n, and setting the bottom n bits to 0. Write a VHDL function:

```
FUNCTION roundn(n: INTEGER; x: SIGNED) RETURN SIGNED;
```

which rounds a signed fixed point vector x to the nearest integer.

[4]

b) *Figure 3.1* shows hardware to implement Equation (3.1). The multiplier blocks compute products z_i*z_i, z_i*z_r, z_r*z_r, which are held in *REG1*. The block *COUNT* is a one bit counter. Block *ASR* computes the new values z'_i, z'_r from these products and the complex constant c, which is output from multiplexer *MUX*. Supposing complex inputs *ca* and *cb* are constant, determine the output of *REG2* for the first four clock cycles after reset. Hence explain how the circuit computes Equation (3.1).

[4]

c) Define a VHDL type *complex* which implements a complex number as an array of two signed vectors, each of length *m*, where *m* is a constant.

[2]

d) You may assume that the VHDL operator :
* : **`signed`** $\times$ **`signed`** $\rightarrow$ **`signed`**
synthesises signed multipliers. The length of the result is the sum of the lengths of the two operands. Using the * operator and previous answers from this question write a VHDL architecture for the entity in *Figure 3.2* which implements *Figure 3.1*.

[10]

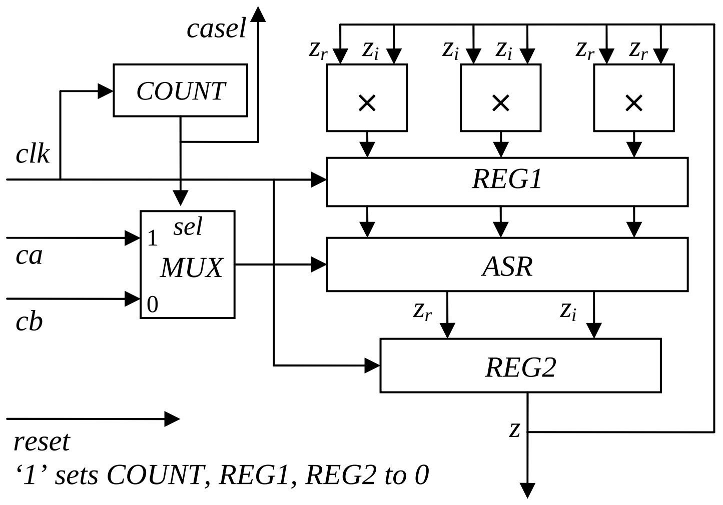

Figure 3.1

```
ENTITY mandelbrot IS
  GENERIC( n : INTEGER);  -- precision
  PORT(
    ca, cb  :   IN complex;
    z       :   OUT complex;
    reset, clk: IN std_logic;
    casel:      OUT std_logic -- '1' when ca is selected
    );
END mandelbrot;
```

Figure 3.2

4.

(a) Explain how in VHDL constant expressions are synthesised differently from signals whose value may change:

(i) as indexes of arrays

(ii) as operands of logical operators.

Specify three distinct contexts in which VHDL identifiers can represent constant expressions.

[4]

(b) *Figure 4.1* shows VHDL entity and architecture *permute* which has *k* bit input *a* and output *b*. Precisely what hardware will this synthesise if

(i) $k = 2, m = 1$

(ii) $k = 4, m = 2$

[2]

(c) Entity *switch* in *Figure 4.3* uses multiple *permute* blocks all with *k*=2, *m*=1. It connects 2^n inputs *a* to 2^n outputs *b* and is made up of *n* layers labelled 0 to *n*-1 each containing 2^{n-1} copies of *permute* with identical *p* input as illustrated in *Figure 4.2*, where the dots indicate repeated blocks which have been omitted. Between layers *q* and *q*+1 there is an array of 2^n signals *x*(*q*). You are given synthesisable functions **`x0(q,i)`** and **`x1(q,i)`** which determine the connections of the *permute* blocks. In layer *q*, the *i*th copy of permute (i = 0,…,2^{n-1}) is connected as in *Figure 4.4*.

Using one or more **`FOR GENERATE`** loops, complete architecture *synth* in *Figure 4.3* using synthesisable code. [14]

	index 0	*index 1*
xin	`x(q)(x0(q,i))`	`x(q)(x1(q,i))`
xout	`x(q+1)(x0(q,i))`	`x(q+1)(x1(q,i))`

Figure 4.4

```
ENTITY permute IS
GENERIC(k,m:INTEGER);
PORT (
      p    : IN std_logic_vector(m-1 DOWNTO 0);
      xin  : IN std_logic_vector(k-1 DOWNTO 0);
      xout : OUT std_logic_vector(k-1 DOWNTO 0)
    );
END permute;

ARCHITECTURE behave OF permute IS
BEGIN
     G1: FOR i IN k DOWNTO 1 GENERATE
     xout(i-1) <= xin((i*(conv_integer(unsigned((p)))+1) mod k+1)-1);
     END GENERATE G1;
END behave;
```

Figure 4.1

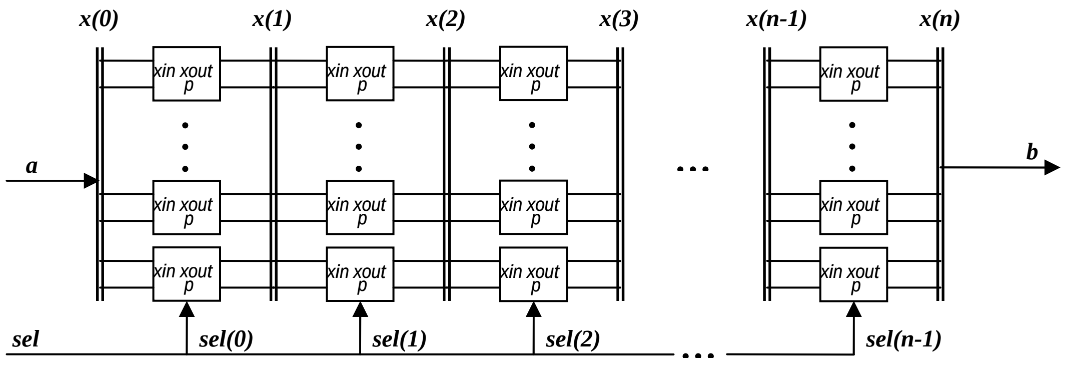

Figure 4.2

```
ENTITY switch IS
   GENERIC(n: INTEGER);
   PORT( a: IN std_logic_vector(2**n-1 DOWNTO 0);
         b: OUT std_logic_vector(2**n-1 DOWNTO 0);
         sel: IN std_logic_vector(n-1 DOWNT0 0) );
END switch;

ARCHITECTURE synth OF switch IS
  TYPE grid IS ARRAY (0 TO n+1) OF std_logic_vector(2**n-1 DOWNTO 0);
  SIGNAL x: grid;
BEGIN
END synth;
```

Figure 4.3