IMPERIAL COLLEGE LONDON

E3.06
AC5
ISE3.5

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2005

MSc and EEE/ISE PART III/IV: MEng, BEng and ACGI

VHDL AND LOGIC SYNTHESIS

Monday, 25 April 10:00 am

Time allowed: 3:00 hours

Corrected Copy

There are FOUR questions on this paper.

Question 1 is COMPULSORY
Answer question 1 and any TWO of questions 2-4
Question 1 carries 40% of the marks, questions 2-4 each carry 30% of the marks.

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible	First Marker(s) :	T.J.W. Clarke
	Second Marker(s) :	G.A. Constantinides

This page is intentionally left blank

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*** *used in the coursework may be used freely in your implementations.*

1.

a) Determine the precise function of process `P1` in Figure 1.1. Is this process synthesisable? Describe a VHDL design technique which can be used to prevent unintentional synthesis of transparent latches from processes with complex conditional statements.

[4]

b) Figure 1.2. shows an entity *shift* for a shift register with parallel output *pout*, control input *dir*, serial data input *sin*, and output *sout*. When *dir* is 1 the register must shift 1 bit left, with *sin* providing the new LSB and *sout* equal to the old MSB. Conversely when *dir* is 0 the register will shift 1 bit right, with the new MSB equal to *sin* and *sout* equal to the old LSB. Write a synthesisable architecture for *shift*.

[4]

c) In process `P2` of Figure 1.3 determine the delay, both physical time and, if relevant, simulation delta time, between events on *clk*, and corresponding events on *a*, *b*, *c*, *d*. You may assume that *clk* changes in $\Delta(0)$. What will be the resulting circuit from synthesis of this process?

[4]

d) Write a synthesisable VHDL architecture for entity *switch* in Figure 1.4 in which *y* is a combinational function of *x*:

$y(i) = x(i+1)$ xor $x(i-1)$, when i is odd,

$y(i) = 0$ when i is even.

[4]

e)

(i) Write a synthesisable entity and architecture describing combinational logic that compares an unsigned 8 bit input *x* represented by a `std_logic_vector` signal with a `GENERIC` integer *n* to generate outputs *equal* and *less*, which are `'1'` if $x = n$, $x < n$ respectively. Your architecture should terminate with a failure `ASSERT` if simulated with a value of *n* greater than 255 or less than 0.

[2]

(ii) Determine how the hardware synthesised from this architecture will simplify when $n = 0$.

[2]

```
P1:PROCESS(clk, din, rst)
BEGIN
  IF rst = '1' THEN
    dout <= "00000";
  ELSIF clk = '0' THEN
    dout <= din;
  END IF;
END PROCESS P1;
```

Figure 1.1

```
ENTITY shift IS
PORT(
      clk, sin, dir: IN std_logic;
      sout: OUT std_logic;
      pout: OUT std_logic_vector(15 DOWNTO 0)
);
END shift;
```

Figure 1.2

```
P2:PROCESS(clk, a, b)
  VARIABLE x : std_logic;
BEGIN
  a <= clk;
  b <= a;
  x := clk;
  d <= x;
  c <= b AFTER 10 ps;
END PROCESS P2;
```

Figure 1.3

```
ENTITY switch IS
PORT(
      x: IN std_logic_vector( 1000 DOWNTO 0);
      y: OUT std_logic_vector( 1000 DOWNTO 0)
);
END switch;
```

Figure 1.4

2.

a) The times t_p, t_h, t_l of a signal *clk* are illustrated in Figure 2.1. The VHDL process CLKMON must print out a warning error if any of the following conditions are violated:

$$t_a < t_p < t_b$$

$$|t_h - t_l| < t_{skew}$$

Where t_a, t_b, t_{skew} are times defined by VHDL constants TA, TB, TSKEW respectively which you are given. You may assume the existence of a signal *clk1* which is identical to *clk* but delayed 1 simulation Δ. Using the VHDL LAST_EVENT signal attribute, where x'LAST_EVENT returns the time elapsed from the most recent event on signal *x*, or otherwise, write CLKMON.

[10]

b) Write a behavioural architecture for VHDL entity *sig_gen* in Figure 2.2 which drives the output *x* with a pseudo-random bit-stream, so that *x* changes on the positive edge of *clk*. Your driver must randomly sample all possible bit-streams subject to the constraints that no more than 5 consecutive '1' bits or 10 consecutive '0' bits are allowed. You are given a function:
`FUNCTION random( low, high: INTEGER)RETURN INTEGER;`
The return value of RANDOM is a pseudo-random integer in the range *low* to *high*. Repeated calls to the function may therefore be used to generate pseudo-random numbers.

[10]

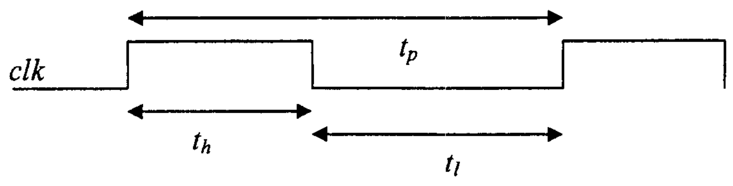


Figure 2.1

```
ENTITY sig_gen IS
PORT(
      clk: IN std_logic;
      x: OUT std_logic
);
END sig_gen;
```

Figure 2.2

3 The VHDL entity *comp_ram* in Figure 3.1 describes a clocked (synchronous) RAM with two I/O channels. The generics *wordlength* and *addrlength* respectively specify the word-length and the number of address bits in the RAM. The number of locations (words) is therefore $2^{addrlength}$. The RAM operates synchronously with the positive edge of *clk*. The two channels are labelled *a*, *b*. Each channel has identical operation and comprises an address bus, *addr*; a bidirectional data bus, *data*; and a control bus, *mode*. Entity ports have as suffix the name of the channel, thus port *addr_b* is the address bus for channel *b*. The operation of a channel is determined, each cycle, by the value of its corresponding *mode* port, as shown in Figure 3.2. There are 4 possible modes of operation: *read*, *write*, *add* and *nop*, as determined by the value of *mode*, also shown in Figure 3.2.

When *mode* is "00" the bus *data* is driven with the contents of the addressed RAM location. When the mode is "01" or "10" the addressed location is written with the value indicated in Figure 3.1. Note that in this case *Y* is the value input on port *data*, and that in these modes, as well as the "11" mode, *data* is not driven by *comp_ram*.

Each channel operates independently, except when both channels *write* or *add* to the same location. In this case a *write* prevails over an *add*, however two *add* operations are combined correctly, with the addressed RAM location's contents changed as if the two operations had happened sequentially. In case of two write operations to the same location the *write* from channel *b* is ignored. Note that RAM contents are only changed by a write on the clock edge, so a *read* from a *write* location will return the old value.

a) Explain the significance of *data_a*, *data_b* ports having VHDL mode INOUT.

[1]

b) Write a synthesisable VHDL architecture to implement *comp_ram*.

[15]

c) Determine (i) the number of flip-flops and (ii) the number and size of multiplexors synthesised from your architecture. Discuss the number of *wordlength* bit adders synthesised.

[4]

```
ENTITY comp_ram IS
  GENERIC( wordlength, addrlength : INTEGER);
  PORT(
    addr_a, addr_b : IN  STD_LOGIC_VECTOR(addrlength-1
       DOWNTO 0);
    data_a, data_b : INOUT STD_LOGIC_VECTOR(wordlength-1
       DOWNTO 0);
    mode_a, mode_b : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    clk: IN std_logic
 );
END comp_ram;
```

Figure 3.1

mode	***data***	**RAM operation**
"00" (*read*)	Output: *ram(addr)*	Read
"01" (*write*)	Input: *Y*	*ram(addr)* $\leftarrow$ *Y*
"10" (*add*)	Input: *Y*	*ram(addr)* $\leftarrow$*ram(addr)* + *Y*
"11" (*nop*)	High impedance	None

ram(addr) is the contents of ram location addressed by *addr*
Y is the external data driven onto the bus *data*

Figure 3.2

4.

a) The Boolean expression x is defined in Equation 4.1, where $\oplus$ denotes **xor**.

$$x = (a+b).(c+(d \oplus e)) \qquad (4.1)$$

For variable order a, b, c, d, e, by simplifying an ordered binary decision diagram or otherwise, derive the reduced ordered binary decision diagram (ROBDD) for x. Draw your ROBDD with 0 (1) edges to the left (right).

[6]

b) VHDL Entity *add* in Figure 4.1 describes a fast adder block provided by a VLSI library. The adder block is available only in bit-widths equal to powers of 2. The generic n ($n \geq 0$) controls the adder bit-width, which equals 2^n. Multiple *add* blocks of different sizes can be connected as shown in Figure 4.2 to implement addition of arbitrary bit-width. Let m be a positive signed 32 bit integer with binary expansion $m(i)$ ($0 < i \leq 30$), where $m(0)$ is the LSB. Consider the sum:

$$M(k) = \sum_{i=0}^{k-1} m(i)2^i . \qquad (4.2)$$

Clearly:

$$\begin{aligned} m &= M(31) \\ M(i+1) - M(i) &= m(i)2^i \end{aligned} \qquad (4.3)$$

It is proposed to implement m bit addition as in Figure 4.2 using at most 1 *add* block of any given length, such that the *add* block size increases monotonically from LSB to MSB.

Show, using Equations 4.3 or otherwise, that this can be effected by using an *add* block of width 2^i to add from bits $M(i)$ to $M(i+1)$-1 only for those i such that $m(i)$=1.

You are given a VHDL function:

```
    FUNCTION mcalc(i: INTEGER)RETURN INTEGER;
```

that computes the function $M(i)$, and may assume the result of this function is a VHDL constant expression providing that its argument is also a constant expression. Complete the architecture *struct* of entity *adder* in Figure 4.3, adding to the architecture body and declaration section as necessary, to make a synthesisable VHDL implementation of *adder* which will incorporate the appropriate *add* blocks for any positive value of m.

[14]

```
ENTITY add IS
GENERIC( n: INTEGER);
PORT(
      p,q: IN std_logic_vector(2**n-1 DOWNTO 0);
      sum: OUT std_logic_vector(2**n-1 DOWNTO 0);
      cin: IN std_logic;
      cout: OUT std_logic
);
END add;
```

Figure 4.1

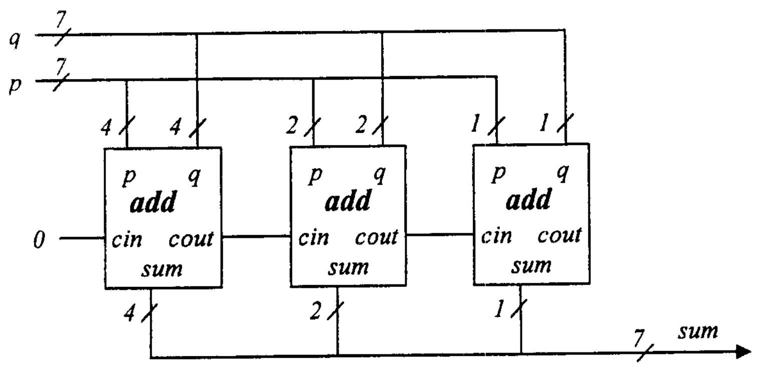


Figure 4.2

```
ENTITY adder IS
GENERIC( m: INTEGER);
PORT(
      p,q: IN std_logic_vector(m-1 DOWNTO 0);
      sum: OUT std_logic_vector(m-1 DOWNTO 0)
);
END adder;

ARCHITECTURE struct OF adder IS
  CONSTANT ms: std_logic_vector(30 DOWNTO 0) :=
     conv_std_logic_vector(m, 31);
  SIGNAL carry: std_logic_vector(31 DOWNTO 0);
BEGIN


END ARCHITECTURE struct;
```

Figure 4.3

VHDL & Logic Synthesis Master – 19.4.05
2005

Solutions

Question 1

a) This synthesises correctly and is a 5 bit transparent latch with output *dout*, active high asynchronous reset *rst* and active low clock *clk*. Setting all driven signals to default values at the start of a process will prevent unintentional latch synthesis (and also usually simplify the code).

b)

```
ARCHITECTURE shift_arch OF shift IS

SIGNAL pout1: std_logic_vector(15 DOWNTO 0);

BEGIN

SHIFT: PROCESS
BEGIN
  WAIT UNTIL clk'EVENT and clk='1';
  CASE dir IS
  WHEN '0' =>
       pout1(14 DOWNTO 0) <= pout1(15 DOWNTO 1);
       pout1(15) <= sin;
       sout <= pout1(0);
   WHEN '1' =>
       pout1(15 DOWNTO 1) <= pout1(14 DOWNTO 0);
       pout1(0) <= sin;
       sout <= pout1(15);
   WHEN OTHERS => NULL; -- prevent compiler errors
  END CASE;
END PROCESS SHIFT;

pout <= pout1;

END shift_arch;
```

c)

clock -> a delay 1 delta
clock -> b delay 2 delta
clock -> c delay 10 ps 0 delta
clock -> d delay 1 delta

During synthesis the AFTER delay is ignored and nets *a*, *b*, *c*, *d* will all be connected directly to *clk*. synthesis will succeed.

d)

```
ARCHITECTURE switch_arch OF switch IS
BEGIN

P1: PROCESS(x)
    BEGIN
      FOR i IN 0 TO 1000 LOOP
        y(i) <= '0';
        IF i MOD 2 = 1 THEN
           y(i) <= x(i-1) xor x(i+1);
        END IF;
      END LOOP;
    END PROCESS P1;

END switch_arch;
```

Question 1 (contd).

e)

```
ENTITY compare IS
GENERIC(n: INTEGER);
PORT(
  x: IN std_logic_vector(7 DOWNTO 0);
  equal, less: OUT std_logic
);
END compare;

ARCHITECTURE synth OF compare IS

BEGIN

ASSERT n >= 0 and n < 256 REPORT "n is out of range" SEVERITY
       failure;

P1: PROCESS(x)
BEGIN
       equal <='0';
       less <= '0';
       IF unsigned(x) = conv_unsigned(n,8) THEN equal <= '1';
       END IF;
       IF unsigned(x) < n THEN less <= '1'; END IF;
END PROCESS P1;

END synth;
```

If n = 0 then *less* is always '0' and requires no hardware. *equal* requires an 8-input NOR gate or equivalent for the comparison with 0.

Question 2

a)

```
CLKMON: PROCESS(clk)
VARIABLE thigh, tlow: TIME;
BEGIN
   IF clk='0' THEN
      thigh :=  clk1'LAST_EVENT;
   ELSE
      tlow := clk1'LAST_EVENT;
   END IF;
   ASSERT abs(thigh-tlow) < TSKEW
      REPORT "SKEW error" SEVERITY warning;
   ASSERT thigh + tlow < TB and thigh+tlow > TA
      REPORT "Clock period error" SEVERITY warning;
END PROCESS CLKMON;
```

b)

```
ARCHITECTURE behave OF sig_gen IS
   SIGNAL x1: std_logic;
BEGIN
   P1: PROCESS
        VARIABLE x_new: std_logic;
        VARIABLE count: INTEGER:= 0;
  BEGIN
      WAIT UNTIL clk'EVENT and clk='1';
      IF random(0,1)=1 THEN -- set random next value
         x_new := '1';
      ELSE
         x_new := '0';
      END IF;
      IF (x1 xor x_new) = '1' THEN
         count := 1;
      ELSE
         count := count +1;
      END IF;
      IF ((count > 5) and (x1='1')) OR ((count > 10) and (x1='0'))
       THEN
         x_new:= not x_new; -- correct if violates conditions
         count := 1;
      END IF;
      x1 <= x_new;
   END PROCESS P1;

   x <= x1;

END behave;
```

Question 3

a) INOUT used because bidirectional bus requires input & output from data ports.

b)

```
ENTITY comp_ram IS
  GENERIC( wordlength, addrlength : INTEGER);
  PORT(
    addr_a, addr_b : IN  STD_LOGIC_VECTOR(addrlength-1 DOWNTO 0);
    data_a, data_b : INOUT STD_LOGIC_VECTOR(wordlength-1 DOWNTO 0);
    mode_a, mode_b : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    clk: IN std_logic
  );
END comp_ram;

(entity is given)
```

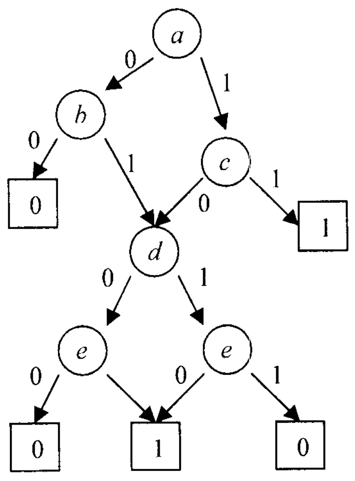
```
ARCHITECTURE synth OF comp_ram IS
  SUBTYPE ram_word IS std_logic_vector(wordlength-1 DOWNTO 0);
  TYPE ram_array IS ARRAY (0 TO 2**addrlength-1) OF ram_word;
  SIGNAL ram: ram_array;
BEGIN

BA: BLOCK (mode_a="00")
BEGIN
data_a <= GUARDED ram(conv_integer(unsigned(addr_a)));
END BLOCK BA;

BB: BLOCK (mode_b="00")
BEGIN
data_b <= GUARDED ram(conv_integer(unsigned(addr_b)));
END BLOCK BB;

PW: PROCESS
  VARIABLE a,b: INTEGER;
  VARIABLE rb: ram_word;
BEGIN
   WAIT UNTIL clk'EVENT and clk='1';
   a := conv_integer(unsigned(addr_a));
   b := conv_integer(unsigned(addr_b));

   CASE mode_b IS
   WHEN "01" => rb := data_b;
   WHEN "10" => rb := unsigned(data_b)+unsigned(ram(b));
   WHEN OTHERS => NULL;
   END CASE;
   -- keep rb in a variable to use later if required
   ram(b) <= rb;

   CASE mode_a IS
   WHEN "01" => ram(a) <= data_a;
   WHEN "10" => ram(a) <= unsigned(data_a)+unsigned(ram(a));
   WHEN OTHERS => NULL;
   END CASE;

   IF a=b and mode_a="01" and mode_b="01" THEN
      ram(a) <= unsigned(data_a) + unsigned(rb);
   END IF; -- special case
END PROCESS PW;

END synth;
```

Question 3 (contd)

`wordlength*(2**addrlength)` flip-flops. Two `(2**addrlength) -> 1` multiplexors. Probably need only two *wordlength* adders since for the *a* = *b add* case good synthesis will reuse the separate *a*, *b* adders. Code written to enable this. At most need 3 adders since one adder is explicitly reused in the code.

Question 4.

a) ROBDD:



Question 4 (contd).

b)

```
ENTITY add IS
GENERIC( n: INTEGER);
PORT(
       p,q: IN std_logic_vector; -- length must be 2**n
       sum: OUT std_logic_vector; -- length must be 2**n
       cin: IN std_logic;
       cout: OUT std_logic
);
END add;

ENTITY adder IS
GENERIC( m: INTEGER);
PORT(
       p,q: IN std_logic_vector; -- length must be m
       sum: OUT std_logic_vector; -- length must be m
);
END adder;

 (entities are given)
```

From Equation 4.3, *m* can be decomposed into the powers of 2 for which the corresponding bit *m*(*i*) is 1. Each such bit has a corresponding *add* block, when concatenated these blocks add up the *m* bits. If *m*(*i*)=1, then *M*(*i*+1)-1:*M*(*i*) has 2^i bits and corresponds to a single *add* block.

```
ARCHITECTURE synth OF adder IS
   CONSTANT ms: std_logic_vector(30 DOWNTO 0)
      := conv_std_logic_vector(m, 31);

   SIGNAL carry: std_logic_vector(31 DOWNTO 0);
BEGIN
   G1:FOR i IN 0 TO 30 GENERATE

      G2:IF ms(i)='1' GENERATE
           I1: ENTITY add
           GENERIC MAP(i)
           PORT MAP(p(mcalc(i)+2**i-1 DOWNTO mcalc(i)),
                q(mcalc(i)+2**i-1 DOWNTO mcalc(i)),
                sum(mcalc(i)+2**i-1 DOWNTO mcalc(i)),
                carry(i),
                carry(i+1)); -- instantiate the add block
         END GENERATE G2;

      G3: IF ms(i)='0' GENERATE
            carry(i+1)<= carry(i); -- bypass this stage
          END GENERATE G3;

   END GENERATE G1;

   carry(0) <= '0'; -- set the LSB carry to '0'

 END ARCHITECTURE synth;
```