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 2002

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

VHDL AND LOGIC SYNTHESIS

Friday, 26 April 10:00 am

There are SIX questions on this paper.

Answer FOUR questions.

Corrected copy (JVA)

Q6 10.15
diag

Q5 wording 10.20
Q2b wording 10.25

Time allowed: 3:00 hours

Examiners responsible:

First Marker(s): Clarke,T.J.W.

Second Marker(s): Cheung,P.Y.K.

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

Information for candidates: None.

1. *Figure 1.1* shows the entity of a hardware sine function generator `sinegen` with input x, output y, and generic parameters n, m. The two's complement signed number x has n bits to the left, and m bits to the right, of its fixed point. It can thus represent a real number in the range $(2^{n-1} - 2^{-m})$ to -2^{n-1}. The `sinegen` entity implements the sine function over this range by using one instance of the entity `sine_lookup`, also in *Figure 1.1*. `Sine_lookup` is a combinational block you are given that implements the sine function over the limited range 0-2π. The `sinegen` entity operates as in *Figure 1.2*. The input x is first changed to its absolute value, `xabs`, and then, by subtracting the appropriate multiple of 2π, to a number `xbase` in the range 0-2π, which is passed to the input of `sine_lookup`. The output y is driven from the output of `sine_lookup`, or its negation, as shown in *Figure 1.2*. Fixed point arithmetic, with *m* bits to the left of the fixed point, is used throughout to represent real numbers, and the implementation of `sinegen` is purely combinational.

a) Assuming that the output of `sine_lookup` is the sine of its input, explain how `sinegen` implements y = sin(x), and why the lengths of `xabs`, `xbase`, y are as specified in *Figure 1.2*.

[5]

b) Let *i* be any integer > 0. If u_i is a number in the range 0 to $2^i.2\pi$, explain why the output of a comparator/subtractor/multiplexor:

u_{i-1} = if $u_i < 2^{i-1}.2\pi$ then u_i else $u_i - 2^{i-1}.2\pi$

lies in the range 0 to $2^{i-1}.2\pi$. Hence draw a diagram indicating how such units, cascaded, can be used to convert `xabs` into `xbase`. How many units are required?

[5]

c) Write a synthesisable architecture for `sinegen`. You may assume that an implementation for entity `sine_lookup` is given, and that m, n are chosen such that all numeric vectors in `sinegen` may be represented by VHDL integers without overflow. Implement the numbers u_i as an array of vectors, each of length n+m. Your solution may use functions from the package `utils`, described in the lectures.

[10]

```
ENTITY sinegen IS
   GENERIC( n : INTEGER := 8;
            m : INTEGER := 8
          );
   PORT( x : IN  STD_LOGIC_VECTOR(m+n-1 DOWNTO 0);
         y : OUT STD_LOGIC_VECTOR(m+1 DOWNTO 0));
END sinegen;

ENTITY sine_lookup IS
   GENERIC( m: integer);
   PORT( x: std_logic_vector( m+2 DOWNTO 0);
         y: OUT std_logic_vector( m+1 DOWNTO 0)
       );
END sine_lookup;
```

Figure 1.1

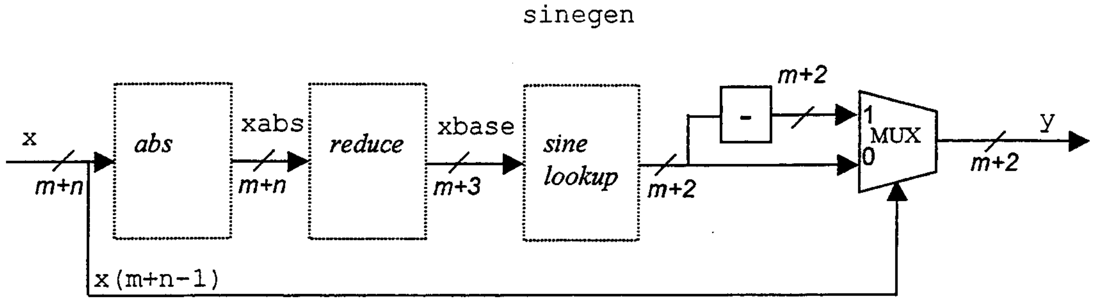


Figure 1.2

2. This question requires you to write a testbench for the entity `sinegen` defined in question 1.

   You are given a VHDL package `maths` in library `mathlib` containing functions:

```
IMPURE FUNCTION random( l: integer; h: integer) RETURN integer;
FUNCTION sin( x : real) RETURN real;
```

   The function `random`, when called, returns a new pseudo-random integer in the range `l` to `h` inclusive (`l` < `h`). The function `sin` returns the sine of its parameter `x`, which specifies an angle in radians.

a) Why is the function `random` declared IMPURE?

[2]

b) Write a testbench entity for `sinegen` that has generic `testnum`, and performs `testnum` tests with pseudo-random input stimulus. The parameters `n`, `m` should also be generic parameters of your testbench, with default values of 20 and 8 respectively. Each test should check that the output is within 2^{-m} of the value computed by the `sin` function.

[12]

c) The entity `sine_lookup` in *Figure 1.1* is implemented as a lookup table. Discuss the relative merits of pseudo-random and exhaustive testing of `sinegen`, assuming that each separate test takes 1ms to execute.

[6]

3.

a) *Figure 3.1* 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]

b) The VHDL fragment in *Figure 3.2* 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]

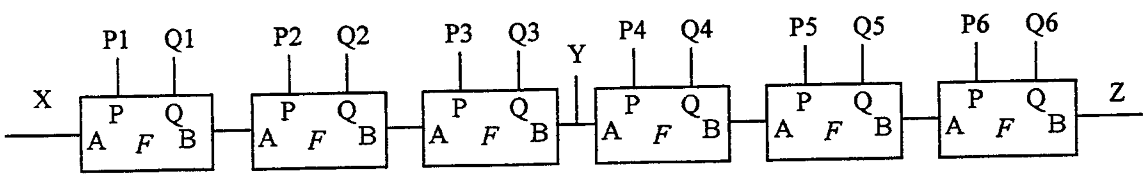


Figure 3.1

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

Figure 3.2

4. *Figure 4.1* shows the entity mult16_16 of a custom VLSI 16*16 bit unsigned multiplier. In order to use this in a synchronous circuit it is proposed to implement a positive edge triggered clocked entity mult32_32, as in *Figure 4.1*. The I/O timing for mult32_32 is shown in *Figure 4.2*. The inputs a,b are stable shortly after the rising edge of clk. The output y will be valid either 2, or 3 cycles after the corresponding inputs a,b. The output ready will be '1' during any clock cycle in which new inputs may be presented. The output y will be valid 2 or 3 cycles after a, b according to whether ready is '1' or '0' during the cycle after that in which a, b is presented. The inputs a, b are don't care for the 2nd cycle of a 3 cycle operation. The reset input provides a synchronous reset.

Mult32_32 uses three mult16_16 units to implement 32*32 bit multiplication with 32 bit result. If *al,ah* and *bl,bh* are the unsigned low and high 16 bit words of inputs a & b respectively, output *y* is calculated as:

$y = \text{floor}(2^{-16}(al*bh+bl*ah)+ ah*bh)$

The multiplication delay of mult32_32 is determined by that of each component mult16_16 unit, which has a propagation delay dependent only on the value of one of its inputs: m. If m < 2^8 the mult16_16 delay is under 1 clock cycle, otherwise it is between 1 and 2 clock cycles. You may assume that the flip-flop setup times, and all other combinational delays, are negligible, so that the mult_16_16 delay determines the required number of clock cycles for each multiplication. Data inputs and outputs of mult32_32 are registered, so providing the minimum 2 cycle delay from input to output. There are no other clocked registers in the datapath of mult32_32.

a) *Figure 4.3* shows the state diagram of an FSM that will generate the required timing, from an input *c*. The signal *c* is a function of a,b and equal to '1' when the corresponding operation must take 3 cycles. Rewrite the state diagram including ready as an output.

[5]

b) Sketch an implementation of the datapath of mult32_32, implemented so as to minimise delay for a, b < 2^{24}. What is *c* as a function of a,b?

[5]

c) Assuming that mult16_16 is synthesisable, implement in VHDL a synthesisable architecture for mult32_32.

[10]

```
ENTITY mult16_16 IS
   PORT(
      m : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
      n : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
      p : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
      );
END mult16_16;

ENTITY mult32_32 IS
   PORT( clk   : IN  STD_LOGIC;
         reset : IN  STD_LOGIC;
         a     : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
         b     : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
         y     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
         ready : OUT STD_LOGIC;
         );
END mult32_32;
```

Figure 4.1

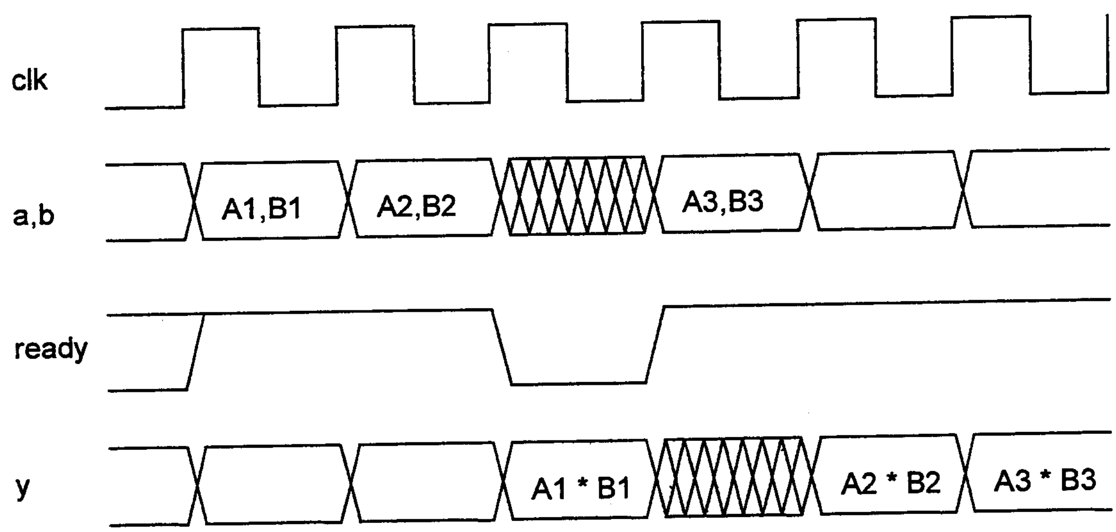


Figure 4.2

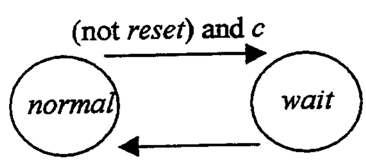


Figure 4.3

5. *Figure 5.1.* on page 8 contains a procedure `read_cycle` that communicates with process `mem_driver_proc` via shared variables `mem_request_cycle`, `mem_data`, `mem_address`, and signal `mem_ack`.

a) In VHDL both signals and shared variables can be used for inter-process communication. Why in the code of *Figure 5.1* on page 8 is `mem_ack` required to be a signal, whereas `mem_request-cycle` must be a shared variable? Discuss whether `mem_data`, `mem_address` could be signals in the cases:

(i) `read_cycle` is called in a single process.

(ii) `read_cycle` is called in multiple processes.

[10]

b) Write a VHDL procedure:

```
delay_by_clocks_and_deltas( SIGNAL clk: IN std_logic;
                            m: in INTEGER; n: IN INTEGER);
```

that when called will wait until *m* 0->1 transitions of `clk` have occurred, then wait a further *n* simulation deltas, then return. Your procedure should work correctly for all non-negative values of *m* and *n*.

[10]

6. *Figure 5.1* on page 8 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`, and interfaces to a RAM through address and read data busses, as illustrated in *Figure 6.1*.

a) Initially `mem_request_cycle` is false. 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.

[15]

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

[5]

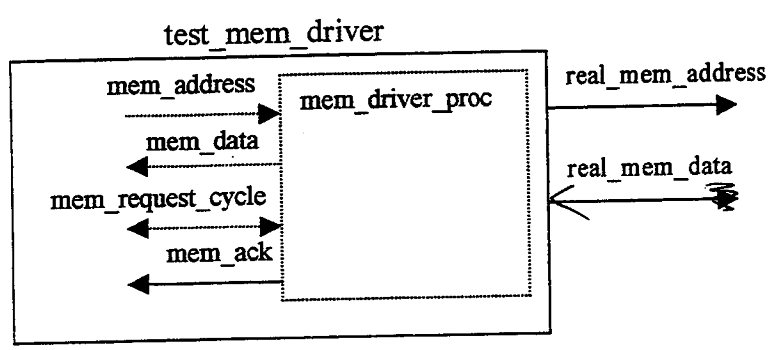


Figure 6.1

```
ENTITY test_mem_driver IS
   PORT (real_mem_address : OUT INTEGER;
         real_mem_data    : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0);
        );
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 DOWNTO 0);
   BEGIN
      WAIT UNTIL clk'EVENT AND clk = '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;

END behav;
```

Figure 5.1 (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 5.1 (continued from previous page)

E3.06
2002

ANSWERS - VHDL & Logic Synthesis

Question 1.

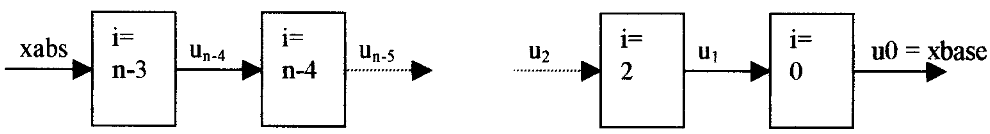
a)

If the input is negative, it is negated and the output negated to compensate. The *reduce* stage subtracts a multiple of 2pi and hence does not alter answer. Finally the output of *reduce* is in the correct range for `sine_lookup` to return the correct answer.

xabs is always positive, its max value (1.000) uses what was the sign bit in *x*, hence length *xabs* also n+m. *xbase* ranges unsigned up to 2π => 3 bits to left of point, m+3 bits overall. Finally *y* must range from -1 to 1, hence *m*+2 bits.

[5]

b) If the *then* part is taken, u_{i-1} is < 2^{i-1}.2pi. Otherwise, the subtraction forces u_{i-1} to be in the required range. *xabs* has max value 2^{n-1}. Hence ceiling(n-1-$\log_2$(2pi)) = n-3 stages are needed to reduce value to max 2pi.



[5]

c) *Using the definitions given in the architecture declaration section of Figure 1.2, write a synthesisable architecture for* `sinegen`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE work.utils.ALL;
USE work.sine_lookup;

ARCHITECTURE rtl OF sinegen IS
   CONSTANT cnum : INTEGER := n-3;
   CONSTANT pi: real := 3.1415926
   TYPE x2_arr IS ARRAY (0 TO cnum-1) OF STD_LOGIC_VECTOR(x'RANGE);
   SIGNAL x2_pre : x2_arr;
   SIGNAL xabs : STD_LOGIC_VECTOR(x'RANGE);
   SIGNAL x2 : STD_LOGIC_VECTOR(m+2 DOWNTO 0);
   SIGNAL x4  : STD_LOGIC_VECTOR(m+1 DOWNTO 0);
BEGIN


 stage1 : PROCESS(x)
   BEGIN
      IF SIGNED(x) < 0 THEN
         xabs <= -SIGNED(x);
      ELSE
         xabs <= x;
      END IF;
   END PROCESS stage1;



  stage2 : PROCESS(x2_pre, xabs)
      VARIABLE i   : INTEGER;
      VARIABLE tmp : STD_LOGIC_VECTOR(x'RANGE);
   BEGIN
      FOR i IN 0 TO cnum-1 LOOP
         tmp := xabs;
         IF i /= cnum-1 THEN
            tmp := x2_pre(i+1);
         END IF;
         IF UNSIGNED( tmp) > INTEGER(REAL(2**(i+m))*2.0*pi) THEN
            x2_pre(i+1) <= UNSIGNED(tmp) -
                INTEGER(REAL(2**(i+m))*2.0*pi);
         ELSE
            x2_pre(i+1) <= tmp;
         END IF;
      END LOOP;

      xbase <= x2_pre(0)(m+2 DOWNTO 0);
   END PROCESS stage2;
```

```
   stage4: ENTITY sine_lookup GENERIC MAP(m=>m)
                  PORT map(x=>xbase,y=>x4);



   p_out: PROCESS(x,x4)
      begin
         IF x(x'left) = '1' THEN
            y <= 0-unsigned(x4);
         else
            y <= x4;
         END IF;

   END process p_out;



END rtl;
```

[10]

Question 2.

This question requires you to write a testbench for the entity `sinegen` *defined in question 1.*

a) *Why is the function* `random` *declared IMPURE?*

It must return a different value each time it is called, using a shared variable to store state. Therefore it must be declared inpure.

[2]

b) *Write a testbench entity for* `sinegen`.

```
LIBRARY ieee;
LIBRARY mathlib
   USE ieee.std_logic_1164;
USE ieee.std_logic_arith;
USE mathlib.maths;

ENTITY sinetest IS
   GENERIC( testnum : INTEGER;
            m       : INTEGER := 20;
            n       : INTEGER := 8
            );
END sinetest;


ARCHITECTURE behav OF sinetest IS
  SIGNAL x_i: std_logic_vector(m+n-1 downto 0);
  SIGNAL y_i: std_logic_vector(m+1 downto 0);
BEGIN

    dut: ENTITY sinegen(x_i, y_i);

   dotest: PROCESS
   BEGIN
      FOR i = 1 TO testnum LOOP
         x_i <= random(0, 2**(n+m)-1);
         WAIT FOR 100 ns;
         y_real := REAL(conv_integer(SIGNED(y_i)))/REAL(2**m);
         x_real := REAL(conv_integer(SIGNED(x_i)))/REAL(2**m);
         ASSERT ABS(y_real - sin(x_real)) < (1.0 / REAL(2**m))
            REPORT "Bad output : x = " & REAL'IMAGE(x_real) & ", y = " & REAL'IMAGE(y_real);
         SEVERITY warning;
      END LOOP;
      REPORT "Test finished";
      WAIT;
   END PROCESS dotest;

end;
```

[12]

c) *Discuss the relative merits of pseudo-random and exhaustive testing of* `sinegen`, *assuming that each separate test takes 1ms to execute.*

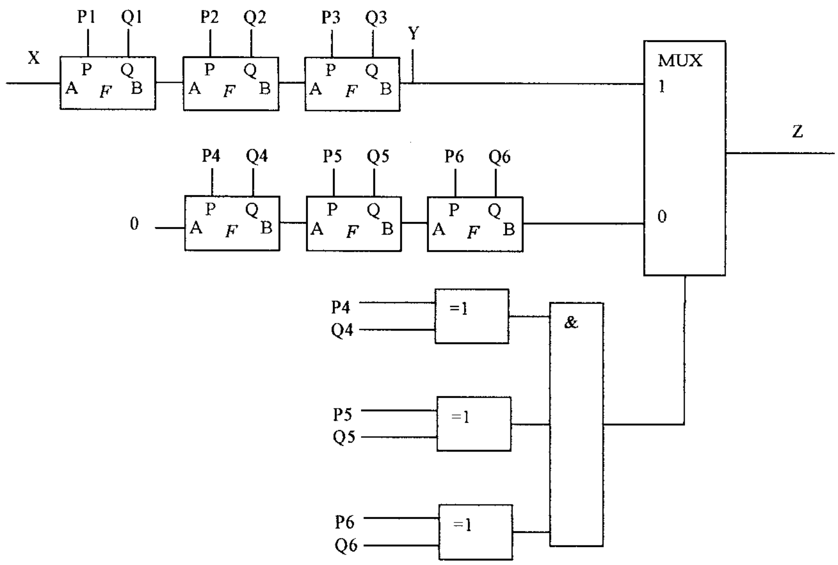
For exhaustive testing execution time is 2^{n+m} ms. This is feasible for say n+m < 25. Above this size the test time becomes unpleasantly large. Pseudo-random testing will provide a faster test, of reasonably good quality. If sine_lookup is based on a ROM lookup every location should be tested if possible => exhaustive test of all values < pi/2. Corner cases should be added: x= max negative input, max positive input, x=pi/2 (sin is max) x=-pi/2 (sin is min).

[6]

Question 3

a) *By applying controllability factoring at point Y, derive an equivalent circuit with reduced critical path length. What is your control function C?*

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



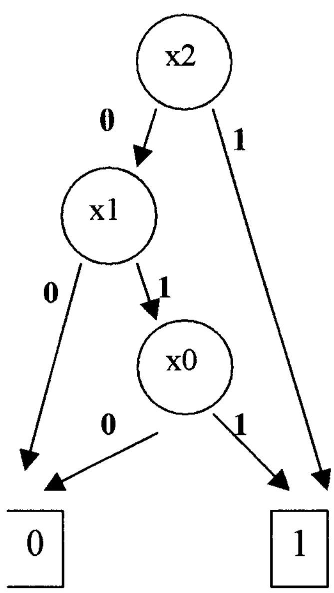
[10]

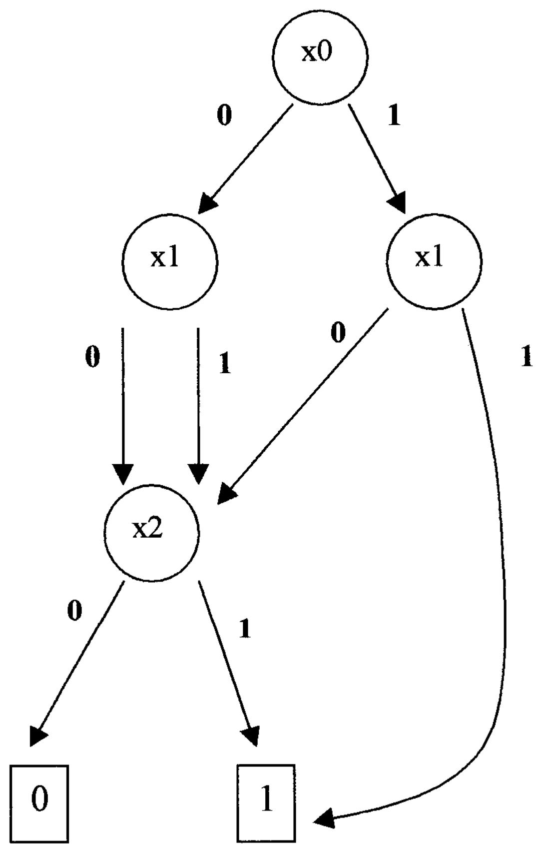
b) *Write a truth table for* `y`*, and compute two ROBDDs for* `y` *using variable orders:* `x(0),x(1),x(2),` *and* `x(2),x(1),x(0)` *respectively.*

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)



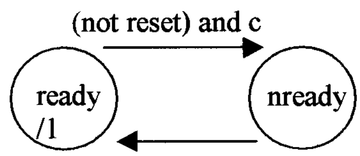


[10]

Question 4

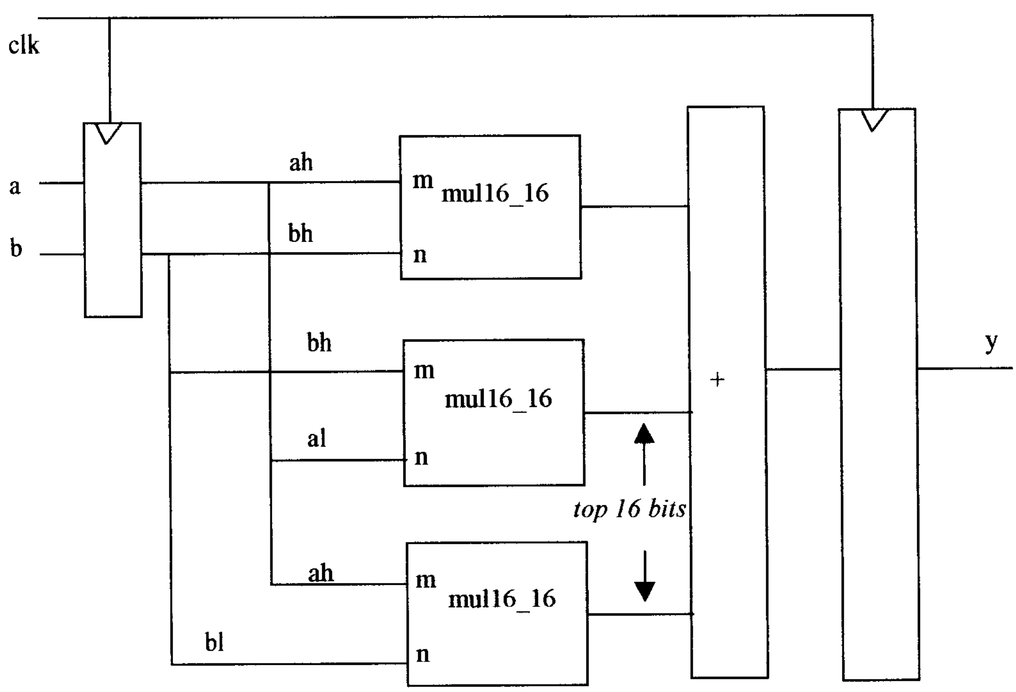
a)

Output: ready (default 0).



[5]

b)



c will be 0 iff a < 2^{24} and b < 2^{24}.

[5]

c) Implement in VHDL the architecture of *mult32_32*.

[10]

```
ARCHITECTURE rtl OF mult32_32 IS
     SIGNAL p1, p2, p3                  : STD_LOGIC_VECTOR(31 DOWNTO 0);
     SIGNAL a_big, b_big, ready_int : STD_LOGIC;
     SIGNAL al, ah, bl, bh              : STD_LOGIC_VECTOR(15 DOWNTO 0);
     SIGNAL fsmwait                     : STD_LOGIC;  --FSM state bit
     SIGNAL c                           : STD_LOGIC;  --wait condition
  BEGIN

     in_reg : PROCESS
     BEGIN
        WAIT UNTIL clk'EVENT AND clk = '1';
        IF fsm_wait = '0' THEN
           al <= a(15 DOWNTO 0);
           ah <= a(31 DOWNTO 16);
           bl <= b(15 DOWNTO 0);
           bh <= b(31 DOWNTO 15);
        END IF;
     END PROCESS in_reg;

     c <= a_big or b_big;

     m1 : ENTITY mul16_16(ah, bl, p1);
     m2 : ENTITY mul16_16(bh, al, p2);
     m3 : ENTITY mul16_16(ah, bh, p3);

     out_reg : PROCESS
     BEGIN
        y <= p3 + (p1(31 DOWNTO 16)+p2(31 DOWNTO 16));
     END PROCESS our_reg;

     compare : PROCESS(a, b)
     BEGIN
        a_big <= UNSIGNED(a(31 DOWNTO 24)) /= 0;
        b_big <= UNSIGNED(b(31 DOWNTO 24)) /= 0;
     END PROCESS compare;

     fsm : PROCESS
     BEGIN
        WAIT UNTIL clk'EVENT AND clk = '1';
        IF reset = '1' THEN
           fsmwait <= '0';
        ELSE
           fsmwait <= c AND NOT fsmwait);
        END IF;
     END PROCESS fsm;

     ready <= NOT fsmwait;


  END ARCHITECTURE rtl;
```

Question 5.

a) mem_ack is waited on by read_cycle, hence must be signal. mem_cycle_request is driven in both read_cycle and mem_driver_proc, hence must be shared variable. mem_data could be a signal, the extra 1 delta delay would not matter since mem_ack has similar delay. mem_address could be a signal, although to be safe at all times a 1 delta delay would need to be added to mem_driver_proc, between checking mem_cycle_request and reading mem_address. However if read_cycle were called from multiple processes this would not be possible.

[10]

b)

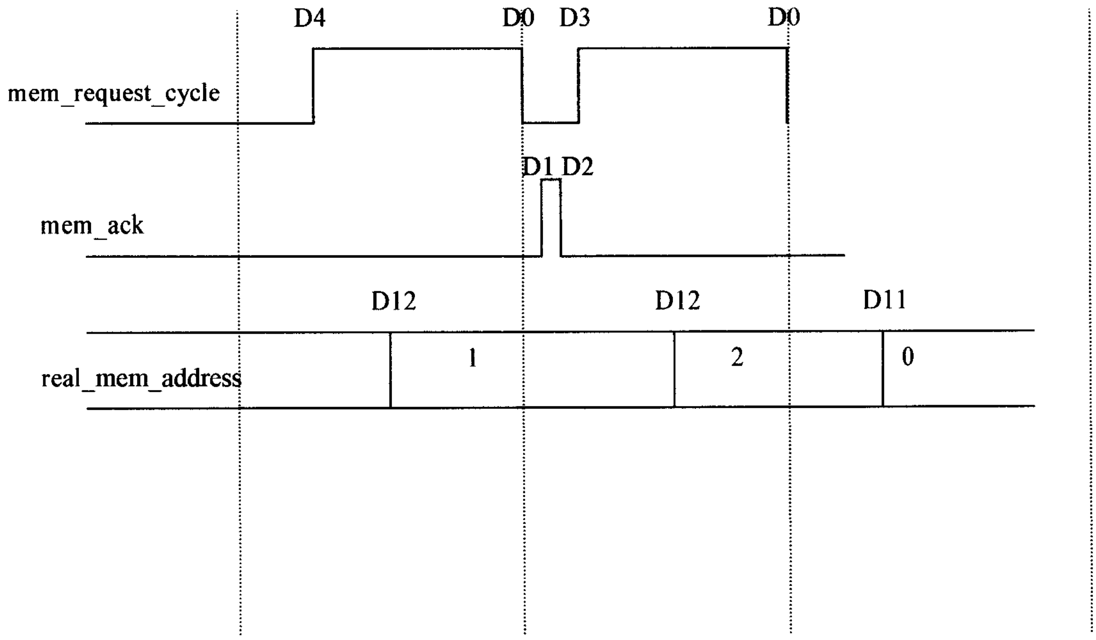
```
PROCEDURE delay_by_clocks_and_deltas( SIGNAL clk : IN STD_LOGIC;
                                      m          : IN INTEGER;
                                      n          : IN INTEGER) IS
BEGIN

   FOR i IN 1 TO n LOOP
      WAIT UNTIL clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0';
   END LOOP;

   FOR i IN 1 TO m LOOP
      WAIT FOR 0 ns;
   END LOOP;
END PROCEDURE delay_by_clocks_and_deltas;
```

[10]

Question 6.

a) *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.*

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*.



[15]

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

`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 10 delta after clock => read_cycle executed 8 delta after clock edge. mem_request cycle is tested by read_cycle 2 delta after the call, and reset by mem_driver_proc on the clock edge. So window is clock edge to clock edge + 8 delta.

[5]