Examination Feedback for EEE411/6031
Autumn Semester 2007-08

Feedback for EEE411/6031 Session:2007-2008

Feedback: Please write simple statements about how well students addressed the exam paper in general and each individual question in particular including common problems/mistakes and areas of concern in the boxes provided below. Increase row height if necessary.

General Comments:

I think that some attempts could have been better. Whilst some candidates displayed a good understanding of the course content, there were a few candidates whose attempts betrayed only a superficial understanding of the course. This is quite worrying.

Question 1:

This question was done very badly by those who attempted it and, presumably, the others were put off. Some people did not seem to have grasped that in the reservation table you record storage at the end of the clock cycle (e.g. the contents of R1…R6). These people had data passing through MUX1 in one clock cycle and MUX2 in the next clock cycle. This is not possible, the devices are purely combinatorial. In fact the data would pass through MUX1 and MUX2 to be stored in R1 at the clock edge. Some people ignored some of the registers: *they all have a purpose*. y_0 enters first to R1 and moves to R2 in the next clock cycle and to R1 (having been multiplied by 2^{-1}) when x_0 is in R1. This means that in the following clock cycle x_0 and y_0*2^{-1} are both available at the input of the adder/subtractor allowing x_1 to be formed (with the value of S_0 setting the actual operation). In this same clock cycle y_0 moves back to R1 and $x_0 2^{-1}$ moves to R3. This allows the value of y_1 to be formed in this cycle. This means that the ordering of the data in the next loop will be x_1, y_1, -, β_1, γ_1 but it works just as well for this data ordering producing y_2, x_2, -, β_2, γ_2.

To demonstrate this, the following is a VHDL implementation of the design:

```
--------------------------------------------------------------------------------
----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- cordic component
-- reset initialises it as does start being set to 1
-- theta is the angle in the form s0.00000000 16 bits long (bit 14 = 1.0)
-- sin/cos are outputs in form s0.0000000.. 30 bits long (bit 29 = 1.0)
-- ready goes to 1 when completed
entity cordic is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           start : in  STD_LOGIC;
           ready : out  STD_LOGIC;
                    theta : in  STD_LOGIC_VECTOR (15 downto 0);
           sin : out  STD_LOGIC_VECTOR (29 downto 0);
           cos : out  STD_LOGIC_VECTOR (29 downto 0)
           );
end cordic;
```

```
architecture Behavioral of cordic is

-- internal signal definitions
-- specifically named inputs to registers/muxes
signal r1_in, r3_in, r4_in, Mux2_in: std_logic_vector(15 downto 0);
-- internal registers
signal r1, r2, r3, r4, r5, r6, initial,
       arctan: std_logic_vector(15 downto 0);
-- loop counter and shift value
signal i, shft: std_logic_vector(5 downto 0);
-- state variable for contol
signal state: std_logic_vector(4 downto 0);
-- specific control signals
signal op, s, UpdateS, Mux1, firstloop: std_logic;
signal Mux2: std_logic_vector(1 downto 0);
-- temporary signals to allow multiply operation
signal t1, t1z: std_logic_vector(17 downto 0);
signal t2: std_logic_vector(20 downto 0);
signal t3: std_logic_vector(25 downto 0);
signal t4: std_logic_vector(27 downto 0);
signal t5: std_logic_vector(29 downto 0);

begin

-- each group of statements represents a functional part of the
-- cordic processor


-- multiplier (by K=0.607252935 = 0100110110111101)
-- to do this note the pattern 11 repeats 3 times and
-- 1101 repeats 2 so add up partial products to find answer
-- unsigned values used and so sign extension done manually
-- with 1 bit overhead to show process

-- calculate *11 first
-- t1=18 bits long = r4*11
t1 <= (r4(15)&r4&'0') + (r4(15)&r4(15)&r4);

-- shift it left by two places and add r4 to it to make *1101
-- t2=21 bits long = r4*1101
t2 <= (t1(17)&t1&"00")+(r4(15)&r4(15)&r4(15)&r4(15)&r4(15)&r4);

-- put a pipeline stage in to cut down the critical path
process (clk)
begin
      if clk'event and (clk = '1') then
            if reset = '1' then
                  t3 <= (others => '0');
                  t1z <= (others => '0');
            else

                  -- shift this left by 4 bits and add to
                  -- *1101 value to make *11011101
                  -- t3=26 bits long = r4*11011101
                  t3 <= (t2(20)&t2&"0000")+
                        (t2(20)&t2(20)&t2(20)&t2(20)&t2(20)&t2);
```

```
                -- have to delay t1 as well because it is used after
                -- after the pipeline stage
                t1z <= t1;

                -- should need to delay r4 as well but don't bother -
                -- use r5 instead to calculate t5
            end if;
      end if;
end process;

-- shift *11 value left by 9 places and add to *11011101 to make
-- *11011011101
-- t4=28 bits long = r4*11011011101
t4 <= (t1z(17)&t1z&"000000000") + (t3(25)&t3(25)&t3);

-- shift *1 value left by 13 places and add to *110111011101 to make
-- * 10011011011101
-- t5=30 bits long = r4*10011011011101
t5 <= (r5(15)&r5&"0000000000000") + (t4(27)&t4(27)&t4);
-- end of multiplier t5 has the result

-- arctan lookup - values in s0.000 16 bit format
arctan <= "0011001001000011" when i = "000000" else
          "0001110110101100" when i = "000001" else
          "0000111110101101" when i = "000010" else
          "0000011111110101" when i = "000011" else
          "0000001111111110" when i = "000100" else
          "0000000111111111" when i = "000101" else
          "0000000011111111" when i = "000110" else
          "0000000001111111" when i = "000111" else
          "0000000000111111" when i = "001000" else
          "0000000000011111" when i = "001001" else
          "0000000000001111" when i = "001010" else
          "0000000000000111" when i = "001011" else
          "0000000000000011" when i = "001100" else
          "0000000000000001" when i = "001101" else
          "0000000000000000";
-- end of arctan lookup

-- Multiplexer 3
-- controls the shift input to the mux
-- that sets the barrel shift amount
-- should be Mux3 but is same as Mux1
shft <= i when Mux1 = '0' else
              "000000";
-- end of Multiplexer 3

-- 16 bit barrel shifter
-- sign copying done manually
r3_in <= r1(15 downto 0) when shft = "000000" else
         r1(15)&r1(15 downto 1) when shft = "000001" else
         r1(15)&r1(15)&r1(15 downto 2) when shft = "000010" else
         r1(15)&r1(15)&r1(15)&r1(15 downto 3) when shft = "000011" else
         r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 4) when shft = "000100" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 5) when shft = "000101" else
```

```
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 6) when shft = "000110" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 7) when shft = "000111" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 8) when shft = "001000" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 9) when shft = "001001" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 10) when shft = "001010" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 11) when shft = "001011" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 12) when shft = "001100" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 13) when shft = "001101" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15 downto 14) when shft = "001110" else
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&
         r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15)&r1(15);
-- end of barrel shifter


-- set the signal that controls add or subtract (1=subtract)
op <= state(1) when i(0) = '0' else
            state(2);

-- this signal changes the order of add/subtract when dealing with beta
UpdateS <= state(4);

-- controllable adder subtractor
-- order of variables is swapped dependent on state of updateS
-- when s xor op is 1 subtraction is done (by inverting b operand
-- adding a and b together and adding 1)
process(r1, r3, s, op, updateS)
variable opa, opb: std_logic_vector(15 downto 0);
variable addsub: std_logic;
begin

      -- combinatorial so all values updated in every pass through

      -- swap operands is updateS=1
      if updateS = '0' then
            opa := r1;
            opb := r3;
      else
            opa := r3;
            opb := r1;
      end if;

      -- find true op
```

```
   addsub := s xor op;

      -- invert b is subtract
      if addsub = '1' then
            opb := not opb;
      else
            opb := opb;
      end if;

   -- fixed add operation with correction term if
   -- subtract (a-b) = a + (not b) + 1
      r4_in <= opa + opb + ("000000000000000"&addsub);

end process;
-- end of adder subtractor

-- switch the first mux over to let the ATAN value
-- into the loop at the right point
Mux1 <= state(3);

-- Multiplexer 1
mux2_in <= initial when Mux1 = '0' else
              arctan;
-- end of Multiplexer 1


-- control the second mux to feedback R2 or R6 or let atan/initial
-- values in
Mux2 <= state(1)&(state(3) or (firstloop and (not state(1))));

-- Multiplexer 2
r1_in <=  r6 when Mux2 = "00" else
                  mux2_in when Mux2 = "01" else
                  r2;
-- end of Multiplexer 2


-- this clocked process is the controller that
-- sets the state of all the control signals
-- each loop consists of 5 clock cycles during which
-- the data rotating around the loop formed by R1,3,4,5,6 is
-- x,y,-,beta,atan (following loop will be y,x,-,beta,atan)
-- based on 5 bit state variable 1 hot encoded e.g.
-- 00001 -> 00010 -> 00100 -> 01000 -> 10000 -> 00001, etc.
-- if these states are 0,1,2,3,4 then
-- combinatorial signals are set up as follows:
--   Mux1 Mux2 UpdateS op
-- 0  0    00     0    0
-- 1  0    10     0    1e0o
-- 2  0    00     0    0e1o
-- 3  1    01     0    0
-- 4  0    00     1    0
--
-- 1e0o means 1 when loop counter is even, 0 when odd
-- Mux2 has different values in the first loop 0=01, 3=01
--
-- also other values are latched at specific points
```

```
--    s Sin Cos ready firstloop initial
-- 0  X
-- 1
-- 2     Xo  Xe
-- 3     Xe  Xo           X
-- 4              X               X
--
-- where signals are latched at the end of the clock cycle in which
-- the state is as shown and Xe means latched when loop counter is even
process (clk)
begin
   -- basic clock construct with synchronous reset
      if clk'event and clk = '1' then

         -- all stored values initialised at reset or
            -- restart
            if reset = '1' or start = '1' then

                  -- initialise i so that it rolls over to
                  -- all 0s for the first loop
                  i <= (others => '1');

                  -- set a flag that is true for
                  -- the first loop and 0 thereafter
                  firstloop <= '1';

                  -- initialise the state of s to add for the
                  -- first loop
                  s <= '0';

                  -- initialise the pipeline registers
                  r1 <= (others => '0');
                  r2 <= (others => '0');
                  r3 <= (others => '0');
                  r4 <= (others => '0');
                  r5 <= (others => '0');
                  r6 <= (others => '0');

                  -- initialise the completion output
                  ready <= '0';

                  -- initialise the outputs
                  sin <= (others => '0');
                  cos <= (others => '0');

                  -- start state in 4 (ready for the first loop)
                  state <= "10000";

                  -- this is a value holding the initial input
                  initial <= (others => '0');
            else

                  -- set the default state of signals used in the
                  -- process
                  initial(14) <= '0';

                  -- advance state unless finished
```

```
-- if going back to state1, also increment the loop
-- counter
if state(4) = '1' then
      if (i(4 downto 0) = "01111") then
            ready <= '1';
      else
            state <= "00001";
            i <= i + 1;
      end if;
else

      -- roll state on
      state <= state(3 downto 0) & state(4);
end if;

-- set the initial input to 1.0 at the beginning
if state(4) = '1' then
      initial(14) <= '1';
end if;

-- update the value of s (beta bigger than theta = 1)
if state(0) = '1' then
      if r4 < theta then
            s <= '0';
      else
            s <= '1';
      end if;
end if;

-- clear the flag that says that this is the
-- first loop
if state(3) = '1' then
      firstloop <= '0';
end if;

-- update the sin and cos outputs at each loop
-- more complicated because order changes on each
-- consecutive loop
if state(3) = '1' then
      if i(0) = '1' then
            sin <= t5;
      else
            cos <= t5;
      end if;
end if;

if state(4) = '1' then
      if i(0) = '0' then
            sin <= t5;
      else
            cos <= t5;
      end if;
end if;

-- update all of the registers in the pipeline
r1 <= r1_in;
r2 <= r1;
```

```
                r3 <= r3_in;
                r4 <= r4_in;
                r5 <= r4;
                r6 <= r5;

            end if;
        end if;
end process;

end Behavioral;
```

The only complicated bits of this design are the multiply by K (0.6…) section and the control. The rest of it is pretty much as shown in Figure 1 from the question. The design uses 16 bit internal representation of data (although the multiplier extends this to 30 bits). Given than the barrel-shifted value becomes 0 at 2^{-16}, the number of iterations has been limited to 15.

Having not come to grips with the first part, the second part, which could be answered entirely independently, was not really attempted. Given that β is iterating towards a value of θ, you can either have sufficient iterations to get arbitrarily close or monitor the value of β to find out when it has got close enough.

Question 2:

This question was generally well done although a *few* candidates, strangely, mistook the content of the question entirely. The code snippets presented bigger hurdle than I expected given that we had undertaken a similar exercise in the class a few weeks previous.

Question 3:

This question was generally OK. Candidates could (in the main) quote the probability expression and could calculate the probability (if they did not put the wrong numbers in!). However, the part of the question that asked for the average time to transmit a message was less well done. Some candidates interpreted the retry time as being 1.5μs for the initial contended message, 1.5μs gap, and 1.5μs to resend the message. I was happy to work with this although what I intended was that contention would be detected at the outset and the loser would wait 1.5μs (whilst the winner transmitted) the transmitting would take a further 3μs – not 4.5μs as some candidates believed. However, few came up with the real solution which was that, a retried message still was not guaranteed to be transmitted and so the real cost would be: $P_A T_{mess}+(1-P_A)(2P_A T_{mess}+(1-P_A)(3P_A T_{mess}+(1-P_A)(\ldots))) = T_{mess}/P_A$ in the limit.

In part c where the number of output ports is reduced, clearly, candidates recognized that the probability of contention increased. However, in c.ii. nobody identified that the rate at which messages could be transmitted (and received) fell below $3x10^5$. This, in fact, invalidated the value of P_A. I did not expect candidates to calculate the new equilibrium point (where the rate at which messages are being generated matches the rate at which they can be accepted). However, I would have hoped that candidates might see that there was a problem.

Question 4:

This question was well answered. The descriptive parts were well addressed and most candidates made a good attempt at the implementation of the program. Some attempts seemed overly complex and a few were flawed but virtually all of them showed understanding of what was going on.