ECE 368, Fall 2006, Instructor: Prof. Shantanu Dutt

Midterm Exam: Fri., Nov. 10, Time: 1:30-2:50 pm
Exam Format: Open Book (only text book and one sheet of notes), Total Points: 100

Important Note: **(1)** Write your name on the top of this question sheet and submit along with your answer book. **(2)** You need to show all your work **clearly** in deriving the answers. Just writing down the final answers is not enough.
Suggestion: Begin by reading all questions and do those first that you think you know best.

1. **Regular Circuit Description using Generate Statements**

   (a) You need to design an 8-bit equality comparator which outputs a 1 when both its input numbers X and Y are equal. Otherwise it outputs a 0. The comparator delay needs to be small, in particular, sublinear in the number of bits n of the input numbers for the general n-bit version of your design. Draw the circuit for your 8-bit comparator using only 2-input gates of one or more of the following types: and, or, xor, xnor. Assuming a delay of t_d for a 2-input gate, derive the delay for your 8-input equality comparator as well as its n-input version (assume $n = 2^k$ for some k). **25**
   Hint: Use the associativity property of part of the equality-comparison computation to design a tree-structured circuit. You may need more than one gate type.

   (b) Give a structural VHDL description of the general n-bit version of your above design assuming $n = 2^k$, and make k a generic parameter. Assume availability of entity-architectures for 2-input gates of the above types. Another generic parameter should be t_d which is the delay desired for the 2-input gates used as components in your VHDL description. The design should be described using generate statements. An example 2-input OR gate entity description is given below and you can assume similar entity descriptions of whatever 2-input gates you need to use. **25**

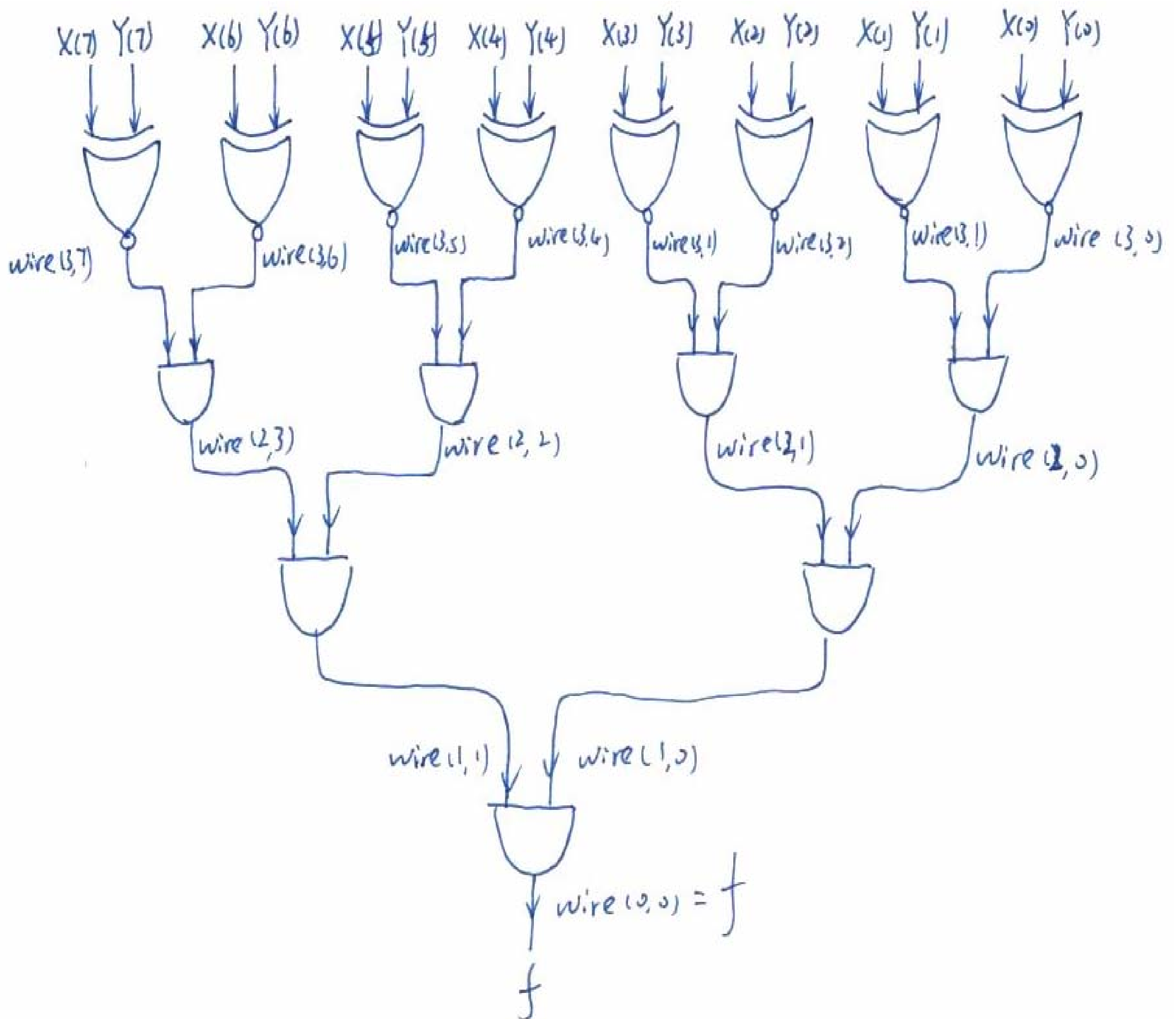
```
entity or2 is
generic (td: time := 4 ns); – propagation delay of gate
port (a, b : in std_logic; c: out std_logic);
end entity or2;
```

The entity description of your equality-comparator circuit should be:

```
entity eq_comp is
generic (k: natural; td: time); – there are n = 2^k bits in each input X, Y
port (X, Y: in std_logic_vector(2**k - 1 downto 0);
f: out std_logic – circuit output);
end entity eq_comp;
```

1. Regular Circuit Description using Generate Statements

(a): The idea is: if and only if all 8 bits of X, Y are the same, then the output can be 1. We can use a 2-input xnor gate to determine whether two 1-bit is equal or not. If equal, output 1, otherwise, output 0. Now that we get 8 outputs of the 8 2-input xnor gate, if any one is 0, then the final output of the circuit is 0 which means not equal. Obviously, we should use 2-input and gate to make this judgment. The tree circuit is as followed:



Concerning n-input version, there will be k+1 levels. So the delay of n-input version is 2*(k+1) ns. Thus the delay of 8-input equality comparator should be 8 ns.

At the first level, we need n 2-input xnor gates. At the following levels, only 2-input and gates are needed. The number a of the 2-input and gates are:

$a = 1 + 2 + 4 + \ldots.. + 2^{k-1} = 2^k - 1;$

(b) VHDL CODE:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;

entity eq_comp is
generic (k: natural; td : time);    -- there are n = 2**k bits in each input X, Y
port (X, Y : in std_logic_vector(2**k - 1 downto 0);
           f : out std_logic);    -- circuit output
end eq_comp;

architecture structural of eq_comp is

component and2 is
generic (td : time := 4 ns); -- propagation delay of gate
port (a, b : in std_logic;
        c: out std_logic);
end component;

component xnor2 is
generic (td : time := 4 ns); --propagation delay of gate
port (a, b : in std_logic;
        c: out std_logic);
end component;

type matrix is array (k downto 0, 2**k - 1 downto 0) of std_logic;
signal wire: matrix;

begin
   outer_loop: for j in k downto 0 generate
      inner_loop: for i in 0 to 2**j-1 generate
         first_level: if j = k generate
            xnor2_gate: xnor2 generic map (td) port map(X(i), Y(i), wire(j, i));
                 end generate;
         next_level: if j < k generate
            and2_gate: and2 generic map (td) port map(wire(j+1, 2*i), wire(j+1, 2*i+1),
wire(j, i));
                 end generate;
      end generate;
   end generate;
   f <= wire(0,0);
end architecture structural;
```

2. **FSM Design**

Design a Moore FSM equality comparator with 2 1-bit serial input lines x, y. The FSM is supposed to determine the equality or non-equality of every disjoint (non-overlapping) group of 5 bits obtained on the x, y lines. If it detects equality of the current group of 5 bits, it should output a 1 for 3 ccs on its output line f. Assume that new bits on the x, y lines are available every clock cycle.

Remember to label the reset state as such. The example below gives 3 groups of 5 bits and what the output should be. Note that a 1 output on the f line naturally appears (i.e., you do not have to explicitly design for it) 1cc after the appearance of the 5th bits of two equal 5-bit numbers, and the example below shows that.

x bits	0	1	1	0	1	0	1	1	1	0	0	0	1	0	1	0 ...
y bits	0	1	1	0	1	1	1	1	0	0	0	0	1	0	1	0 ...
f	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	1 ...

Note that the difference between the equality comparator in Problem 1 and the equality comparator for this problem, is that the Problem 1 comprator has all n input bits available simultaneously, and hence it can be a combinational circuit, whereas in this problem the bits are available serially (one after the other on the same line) and hence the comparator has to be a sequential circuit (FSM) since it has to “remember” the previous bits. **25**

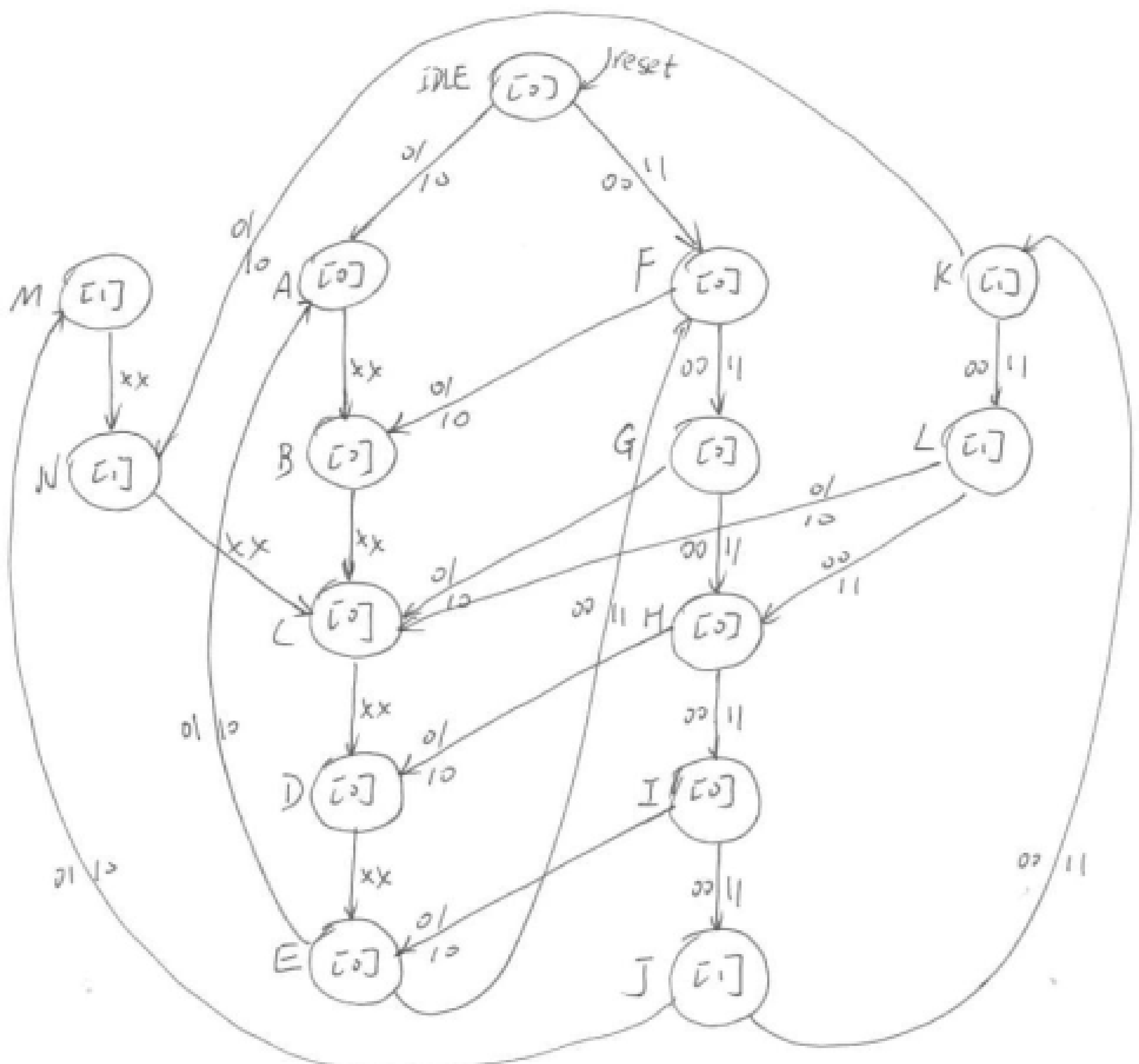
2. FSM Design

The FSM is supposed to determine the equality of *every disjoint (non-overlapping)* group of 5 bits obtained on the x, y lines. So it has to “remember” the previous bits, in other word, each bit of the disjoint 5-bit group need to have a state.

If we detect that the i_{th} bit of x and y are not equal, x and y are not equal. So we can ignore the following bits of x and y and safely output 0. For example, if we detect the 1th bits of x and y are not equal, then the following 4 states outputs are all 0 regardless of their value.

If the FSM detects equality of the current group of 5 bits, it required 3 ccs high output. As we have done in the lab, we need 4 other analog states which are the same as the originals except the output is 1.

The FSM diagram is as followed (00/11 stands for 1 bit of x and y equal, 01/10 stands for 1 bit of x and y not equal, xx stands for don’t care):



The VHDL code is:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity fsm_eq_comp is
    port (x, y, clk, reset: in std_logic;
                                fout: out std_logic);
end entity fsm_eq_comp;


architecture behav of fsm_eq_comp is

type states is (IDLE, A, B, C, D, E, F, G, H, I, J, K, L, M, N);
signal next_state, curr_state: states:=IDle;

begin

ff_proc: process is                     -- 2 ns delay
begin
    wait until (clk'event and clk='1') or (reset='1');
    if reset='1' then
        curr_state <= IDLE after 2 ns;
    else
        curr_state <= next_state after 2 ns;
    end if;
end process ff_proc;

op_proc: process (curr_state) is        -- 3 ns delay
begin

case curr_state is
    when IDLE => fout <= '0' after 3 ns;
    when A => fout <= '0' after 3 ns;
    when B => fout <= '0' after 3 ns;
    when C => fout <= '0' after 3 ns;
    when D => fout <= '0' after 3 ns;
    when E => fout <= '0' after 3 ns;
    when F => fout <= '0' after 3 ns;
    when G => fout <= '0' after 3 ns;
    when H => fout <= '0' after 3 ns;
    when I => fout <= '0' after 3 ns;
    when J => fout <= '1' after 3 ns;
```

```
        when K => fout <= '1' after 3 ns;
        when L => fout <= '1' after 3 ns;
        when M => fout <= '1' after 3 ns;
        when N => fout <= '1' after 3 ns;
end case;

end process op_proc;

ns_proc: process (curr_state,x, y) is      --5 ns delay
begin

case curr_state is
         when IDLE=>
                  if ((x='1' and y = '0') or (x = '0' and y = '1')) then
                       next_state <= A after 5 ns;
                     else
                       next_state <= F after 5 ns;
                     end if;
         when A=>
                  next_state <= B;
         when B=>
                  next_state <= C;
         when C=>
                  next_state <= D;
         when D=>
                  next_state <= E;
         when E=>
                  if ((x='1' and y = '0') or (x = '0' and y = '1')) then
                       next_state <= A after 5 ns;
                     else
                       next_state <= F after 5 ns;
                     end if;
         when F=>
                  if ((x='1' and y = '1') or (x = '0' and y = '0')) then
                       next_state <= G after 5 ns;
                     else
                       next_state <= B after 5 ns;
                     end if;
         when G=>
                  if ((x='1' and y = '1') or (x = '0' and y = '0')) then
                       next_state <= H after 5 ns;
                     else
                       next_state <= C after 5 ns;
                     end if;
```

```
        when H=>
                if ((x='1' and y = '1') or (x = '0' and y = '0')) then
                    next_state <= I after 5 ns;
                  else
                    next_state <= D after 5 ns;
                  end if;
        when I=>
                if ((x='1' and y = '1') or (x = '0' and y = '0')) then
                    next_state <= J after 5 ns;
                  else
                    next_state <= E after 5 ns;
                  end if;
        when J=>
                if ((x='1' and y = '1') or (x = '0' and y = '0')) then
                    next_state <= K after 5 ns;
                  else
                    next_state <= M after 5 ns;
                  end if;
        when K=>
                if ((x='1' and y = '1') or (x = '0' and y = '0')) then
                    next_state <= L after 5 ns;
                  else
                    next_state <= N after 5 ns;
                  end if;
        when L=>
                if ((x='1' and y = '1') or (x = '0' and y = '0')) then
                    next_state <= H after 5 ns;
                  else
                    next_state <= C after 5 ns;
                  end if;
        when M=>
                next_state <= N;
        when N=>
                next_state <= C;
end case;
end process ns_proc;
end architecture behav;
```

3. **VHDL: Signals and Variables**

Consider the following architecture description.

```
architecture XYZ of ABC is
signal a, b, c, d:  integer := 1;
signal clk:  bit := '1';
begin

clock:  process is
begin
clk <= '1','0' after 4 ns;
wait for 8 ns ;
end process clock;

foo:  process is
variable x, y, z:  integer := 1;
begin
-- point 1
x := y + 2*(b**3);
a <= b**2 + x;
b <= a**2 + y after 4 ns;
--point 2
wait for 2 ns;
--point 3
c <= a+b;
y := 2*a + c;
--point 4
wait until clk'event and clk='1';
d <= a + b + c;
--point 5
wait on clk;
end process foo;
end architecture XYZ;
```

Determine the simulation time, and values of the signals and variables at the following points and iterations given in the table below. Assume that simulation begins at time 0 and that the first value of the clock is '1' (and thus that the first transition on the clock will be negtive edge and will occur at time 4 ns—half the clock period). **25**

iter 1, point 1	time	a	b	c	d	x	y	z
iter 1, point 2	time	a	b	x				
iter 1, point 3	time	a	b	x				
iter 1, point 4	time	b	c	y				
iter 1, point 5	time	b	c	d				
iter 2, point 1	time	a	b	c	d	x	y	z
iter 2, point 2	time	a	b	x				

3. VHDL: Signals and Variables
Variable assignment: the value of the variable will immediately change.
Signal assignment: the value of signals will not change immediately, but after △ time, which is smaller than any real time;

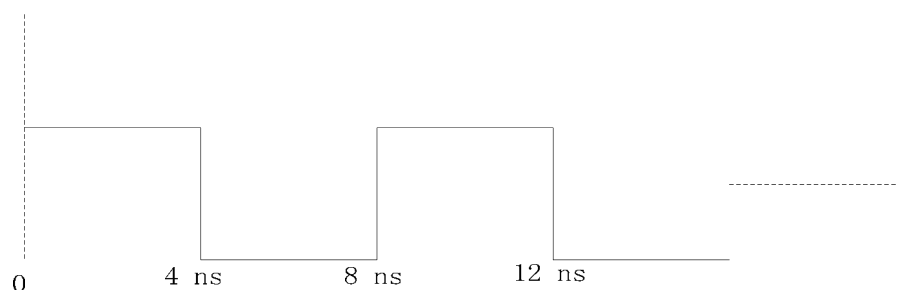
iter 1, point 1:
At the very beginning, time is 0;
a, b, c, d, x, y, z are the original values which are all 1s.

iter 1, point 2:
Variable and signal assignments cannot advance simulation time, so time is 0;
x is a variable, so x changes immediately, x =1+2*(1**2)=3;
a, b are signals, so they will not change immediately, so a = 1, b = 1;
a will change after △ and it will use the new value of x (3);
b will change after 4 ns and it will use the old value of a (1) because a doesn't change at current time;

iter 1, point 3:
wait for statement will advance simulation time, so time is 2 ns now!
Now the value of a changes. a = 1**2+3=4;
x will not change, so x is still 3;
The current time is 2 ns, so b haven't changed and its value is 1.

iter 1, point 4:
variable and signal assignment cannot advance simulation time, so time is 2 ns;
b doesn't change so b = 1;
c will change after △ so now c is still 1; c will use the new value of a (4) and old value of b (1);
y will change immediately. Here a is 4, c is the old value 1 as c haven't changed yet. So y = 2*4+1=9;

iter 1, point 5:
wait until statement will advance the simulation time. The waveform of the clk should be like this:



so the first positive edge of clk will come at 8 ns, thus the current time is 8 ns;

The value of b will change to 2, b = 1**2 + 1 = 2;
c will take effect which c = 4+1 = 5;
d will not change immediately, so d is still 1;

The "wait on clk;" statement will advance the simulation time to 12 ns as clk will change at 12 ns;
Now d will change to a+b+c= 4+2+5=11;

iter 2, point 1:
the current time should be 12 ns;
Given the above result: a = 4, b = 2, c = 5, d = 11, x = 3, y = 9, z = 1;

iter2, point 2:
use the same analysis given in iter1, point 2.
time = 12 ns;
Now x = 25; a, b will not change: a = 4, b = 2;

So the answer is as followed:

iter 1, point 1	time	a	b	c	d	x	y	z
	0 ns	1	1	1	1	1	1	1
iter 1, point 2	time	a	b	x				
	0 ns	1	1	3				
iter 1, point 3	time	a	b	x				
	2 ns	4	1	3				
iter 1, point 4	time	b	c	y				
	2 ns	1	1	9				
iter 1, point 5	time	b	c	d				
	8 ns	2	5	1				
iter 2, point 1	time	a	b	c	d	x	y	z
	12 ns	4	2	5	11	3	9	1
iter 1, point 2	time	a	b	x				
	12 ns	4	2	25				