Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 - Introductory Digital Systems Laboratory

Quiz 2 November 1, 2002

1	.... (20)
2	.... (20)
3	.... (20)
4	.... (40)
TOTAL	.... (100)

NAME ....

Indicate Your Section

( )	James	12 PM
( )	Jennifer	1 PM
( )	Neira	3 PM

This quiz is **Closed Book:** Two handwritten "crib" sheets are allowed.

Put your name on all sheets and indicate your section on this page.

Write all your answers directly on the quiz.

Show all of your work.

You are not required to use a logic template, but you must **make sure your answers are legible**.

Introduction

Problems 2, 3 and 4 of this quiz are based on three vhdl programs which are attached to the back of the quiz. You may remove those pages from the quiz for ease in referring to them. They need not be turned in with the quiz. Note that there are two copies of `lock.vhd`. The first of these must be turned in as you will be modifying it.

Problem 1: Finger Exercises

Show, on Figure 1 how a shift-and-add multiplication of -3×5 would be carried out. In particular, you should show what the contents of the accumulator register would be after each step in the shift-and-add process. Assume that the multiplicand is in a left-shift register. Start with five bit wide representations of -3 and 5. Your answer should be a five bit, two's complement number. There are two copies of the answer table, in case you make a mistake and need to start over. If you write on both of them, be sure to indicate which one has your answer as we will grade only one.

Note we are treating this as two five-bit numbers, which indicates a nine bit answer. Sign extension is required for the negative, two's complement number

-3			1	1	1	1	1	1	1	0	1	
x 5			0	0	0	0	0	0	1	0	1	
=			1	1	1	1	1	1	1	0	1	
+			0	0	0	0	0	0	0	0		
=			1	1	1	1	1	1	1	0	1	
+			1	1	1	1	1	0	1			
=			1	1	1	1	1	0	0	0	1	
+			0	0	0	0	0	0				
=			1	1	1	1	1	0	0	0	1	
+			0	0	0	0	0					
=			1	1	1	1	1	0	0	0	1	
Complement:			0	0	0	0	0	1	1	1	0	
+											1	
=			0	0	0	0	0	1	1	1	1	= 15

Figure 1: Shift-and-add table for Problem 1

Problem 2: Timing

Refer to the program `easy.vhd`, the first of three. This takes a clock and two input signals a and b. It produces three output signals: x, y and z. A rudimentary timing diagram is shown in Figure 2, in which the input signals and clock are shown. Show the output signals. Remember: you don't know anything about the input signals before the start of the timing diagram. As with Problem 1, there are two copies of the blank timing diagram, in case you make a mistake. If you write on both of them, be sure to indicate which one has your answer as we will grade only one.

Note the output signals are and's of a combinatoric and registered signals. It is important to note when you don't know the output signal and when signals change

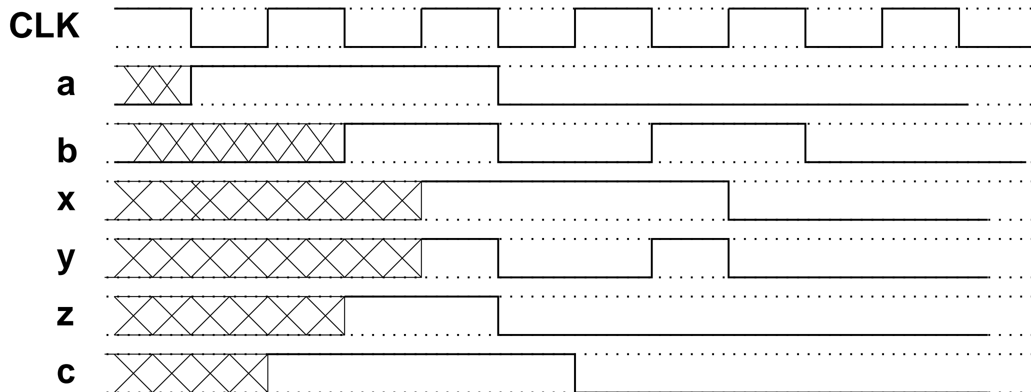

Figure 2: Timing Diagram for Problem 2: copy 1

Problem 3: Unknown Machine

The second program attached to the back, `odd.vhd`, does something which you should be able to figure out.

1. What does this thing do? State its functioning in words.
   This thing has four loadable registers, loaded by a selector with a control word wdsel (and a write enable). At the same time it has an output which is selected in the same fashion, according to another control word rdsel

2. Draw a block diagram that describes the structure of what this thing does. Show the important elements as MUXes, Registers, Tri-States, etc.

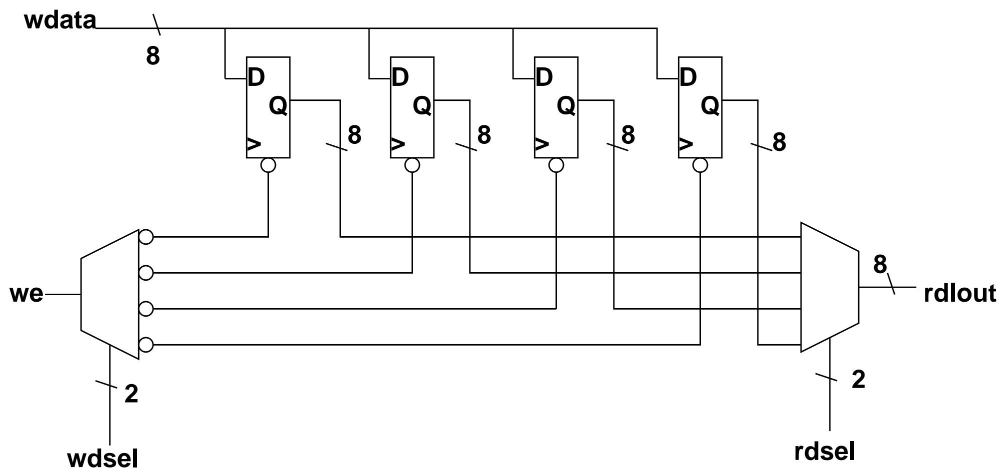

Figure 3: Functionality of the odd code

Problem 4: Problematic Lock

The third program describes a (probably faulty) digital lock. The design was inspired by a desire to discourage lock pickers. In a good example of perhaps faulty logic it was decided that, if the wrong combination was entered the lock should first warn the operator (beep) and then, if correction is not made immediately, it should blow up, thus making the would be lock picker very, very sorry. The lock is intended to be connected to an input consisting of an eight-bit number and an 'enter' key. A combination is hard-coded into the thing, but could obviously be changed just before compile time. To unlock, one enters the first number and 'enter', the second number and 'enter', the third number and 'enter' and the device should unlock. It has a provision that if a wrong number is entered it goes to an alarm state and causes an alarm to beep. At this point it must be re-set by entering all zeros. If this is not done, the next state causes a massive explosion (which is a good reason to be careful with this lock).

1. What is the combination?
   1-5. The third number is a red herring: it is not used

2. In the handy-dandy form shown in Figure 5, draw the state transition diagram for this machine, showing all inputs and outputs.

3. A number of engineers have worked with this, but none have been able to give us a description of what happens. All we know is that each of their experiments have resulted in explosions. We suspect it may be related to the relatively high clock frequency, relative to the ability of the engineers to push buttons. What is happening?
   The late engineers probably pushed the button for several clock cycles, which is too many: after getting one input number right, the second and third will not be, causing the machine to cycle directly to the massive explosion. What is needed is to produce a 'one clock cycle' signal from each button push. An equivalent circuit would be as shown in Figure 4

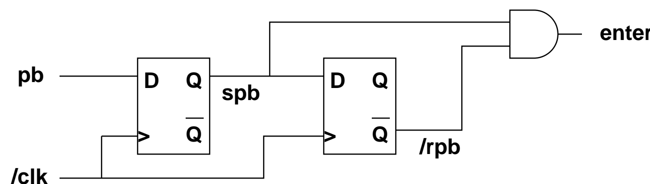

Figure 4: Equivalent Circuit for Lock Code

4. You can fix the premature explosion problem by modifying the code. Write the new code here and indicate where it goes in the VHDL program.

5. Once the explosion issue is settled, it is noted that on occasion the thing unlocks even when a wrong number is entered. This is due to a 'glitch' arising from the crude way in which this thing was written. What is wrong?
   There could be two right answers for this: the first one is that the 'unlok' output is set when n_s is success, and that could happen if one is in state0 and the switches are in the process of being switched. The other is that a transition from state 000 to 011 (alarm) could transition through 010 (success), producing a transient glitch

6. Propose a 'fix' for this problem. Include any additional code that might be required and/or code that should be removed.
   the solution is to register the unlok signal and set it only on a clock edge or while actually in the success state

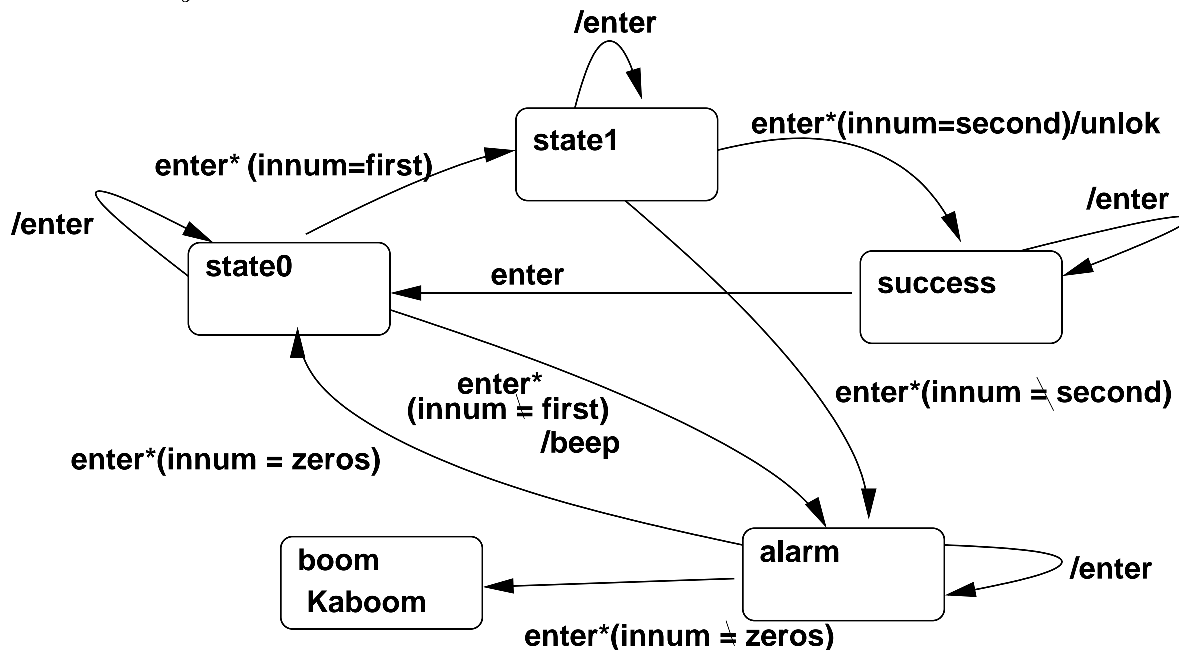

Figure 5: State Transition Diagram for Problem 3

This is lock.vhd Keep this with your quiz and turn it in: please use it to indicate where code modifications must be placed. *The required code changes are included here*

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity diglock is port
     (
          innum          : in std_logic_vector(7 downto 0);
          pb, clk        : in std_logic; -- enter will be synched pb
          beep, unlok    : out std_logic;
          Kaboom         : out std_logic := '0');
end diglock;

architecture state_machine of diglock is
        type StateType is (state0, state1, success, alarm, boom);
        signal p_s, n_s   : StateType;
        signal spb, rpb   : std_logic;
        signal enter      : std_logic;
        constant first    : std_logic_vector(7 downto 0) :="00000001";
        constant second   : std_logic_vector(7 downto 0) :="00000101";
        constant third    : std_logic_vector(7 downto 0) :="00001010";
        constant zeros    : std_logic_vector(7 downto 0) := (others => '0');
begin
       -- commented out code here is to be done better below
       -- unlok <= '1' when n_s = success else '0';
       -- beep <= '1' when n_s = alarm else '0';
        enter = spb and (not rpb); -- one clock pulse wide
        fsm: process (p_s, innum)
        begin
             case p_s is
                  when state0 =>
                       unlok <= '0';
                       beep <= '0';
                       if innum = first then
                            n_s <= state1;
                       else
                            n_s <= alarm;
                       end if;
                  when state1 =>
                       unlok <= '0';
                       beep <= '0';
                       if innum = second then
                            n_s <= success;
                       else
```

```
                    n_s <= alarm;
               end if;
          when success =>
               unlok <= '1';
               beep <= '0';
                    n_s <= state0;
          when alarm =>
               unlok <= '0';
               beep <= '1';
               if innum = zeros then
                    n_s <= state0;
               else
                    n_s <= boom;
               end if;
          when boom =>
                    Kaboom <= '1';   -- don't care about next state!
          when others => n_s <= state0; -- avoid trap states
     end case;
end process fsm;
syncpb: process(clk, pb)
          if rising_edge(clk)
               spb <= pb;
               rpb <= spb;
          end if;
end process syncpb;
state_clocked : process (clk)
     begin
          if rising_edge(clk) then
               if enter = '1' then
                    p_s <= n_s;
               end if;
          end if;
end process state_clocked;
end architecture state_machine;
```

This is easy.vhd

```
library ieee;
use ieee.std_logic_1164.all;
entity reg is

  port (
    a, b, clk : in std_logic;
    y, z      : out std_logic;
    x         : buffer std_logic);

end reg;
architecture top of reg is
signal c : std_logic;
begin  -- top
y <= x and b;
z <= c and b;
reg2:  process (clk)
begin  -- process
  if rising_edge(clk) then
    c <= a;
    if b='1' then
      x <= a;
    end if;
  end if;
end process;
end top;
```

This is odd.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity unknown is

  port (
    clk, we        : in  std_logic;
    rdlsel, wdlsel : in  std_logic_vector(1 downto 0);
    wdata          : in  std_logic_vector(7 downto 0);
    rdlout         : out std_logic_vector(7 downto 0));
end unknown;

architecture whatisthis of unknown is
  signal zero, one, two, three : std_logic_vector(7 downto 0);
begin  -- whatisthis
  clk_process: process (clk, one, two, zero, three, we, rdlsel, wdlsel, wdata)
  begin  -- process
    if rising_edge(clk) then
      if we = '1' then
        if wdlsel = "00" then
          zero <= wdata;
        elsif wdlsel = "01" then
          one <= wdata;
        elsif wdlsel = "10" then
          two <= wdata;
        else
          three <= wdata;
        end if;
      end if;
    end if;
    case rdlsel is
      when "00"=> rdlout <= zero;
      when "01" => rdlout <= one;
      when "10" => rdlout <= two;
      when "11" => rdlout <= three;
      when others => rdlout <= "00000000";
    end case;
  end process clk_process;
end whatisthis;
```

This is lock.vhd This is the original, problematic version.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity diglock is port
     (
           innum           : in std_logic_vector(7 downto 0);
           enter, clk      : in std_logic;
           beep, unlok     : out std_logic;
           Kaboom          : out std_logic := '0');
end diglock;

architecture state_machine of diglock is
        type StateType is (state0, state1, success, alarm, boom);
        signal p_s, n_s   : StateType;
        constant first    : std_logic_vector(7 downto 0) :="00000001";
        constant second   : std_logic_vector(7 downto 0) :="00000101";
        constant third    : std_logic_vector(7 downto 0) :="00001010";
        constant zeros    : std_logic_vector(7 downto 0) := (others => '0');
begin
        unlok <= '1' when n_s = success else '0';
        beep <= '1' when n_s = alarm else '0';

        fsm: process (p_s, innum)
        begin
             case p_s is
                  when state0 =>
                       if innum = first then
                            n_s <= state1;
                       else
                            n_s <= alarm;
                       end if;
                  when state1 =>
                       if innum = second then
                            n_s <= success;
                       else
                            n_s <= alarm;
                       end if;
                  when success =>
                            n_s <= state0;
                  when alarm =>
                       if innum = zeros then
                            n_s <= state0;
                       else
```

```
                            n_s <= boom;
                       end if;
                  when boom =>
                            Kaboom <= '1';    -- don't care about next state!
                  when others => n_s <= state0; -- avoid trap states
             end case;
        end process fsm;
        state_clocked : process (clk)
             begin
                  if rising_edge(clk) then
                       if enter = '1' then
                            p_s <= n_s;
                       end if;
                  end if;
        end process state_clocked;
end architecture state_machine;
```