Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 - Introductory Digital Systems Laboratory
Problem Set # 2

Problem 1: Fun with Combinational Logic !!!

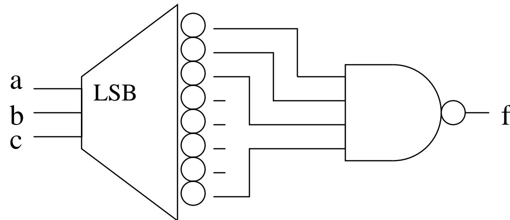
In order to find the MSP and MPS forms of the function realized by the circuit, we'll first construct the Karnaugh map.

	a	0	0	1	1
c	b	0	1	1	0
0		1	1	0	1
1		0	0	1	0

$$\text{MSP} = \bar{a}\bar{c} + \bar{b}\bar{c} + abc$$

$$\text{MPS} = (a + \bar{c})(b + \bar{c})(\bar{a} + \bar{b} + c)$$

We can construct an equivalent circuit using a 3-to-8 decoder and a `NAND` gate. If `a`, `b`, and `c` are out selectors for the decoder, then we want our output to be high when any of the four combinations that result in a high output are true. This is the equivalent of using an `OR`, and we know that a `NAND` is the same as an `OR` with inverted inputs. Therefore, if our decoder has low as the selected output, we simply connect the four true lines to the `NAND`.

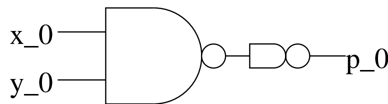


Problem 2: More Combinational Logic !!!

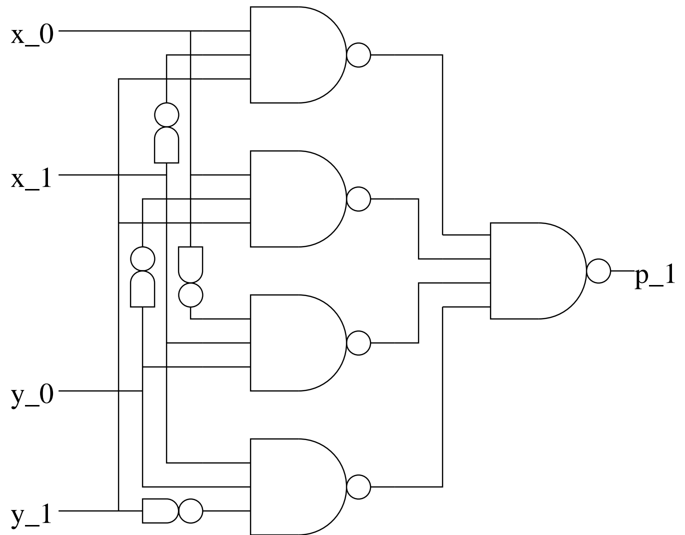
In order to design the logic circuitry for a 2-bit multiplier, we will first construct the truth table, and the Karnaugh maps for each bit. From there, we can derive the MSP forms for each bit of the product, which easily translates into a circuit of `NAND`s.

y_1	y_0	x_1	x_0	p_3	p_2	p_1	p_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

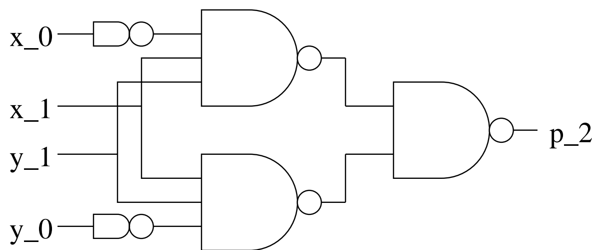
	x	0	0	1	1
y		0	1	1	0
0	0	0	0	0	0
0	1	0	1	1	0
1	1	0	1	1	0
1	0	0	0	0	0



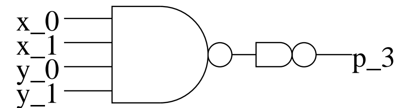
	x	0	0	1	1
y		0	1	1	0
0	0	0	0	0	0
0	1	0	0	1	1
1	1	0	1	0	1
1	0	0	1	1	0



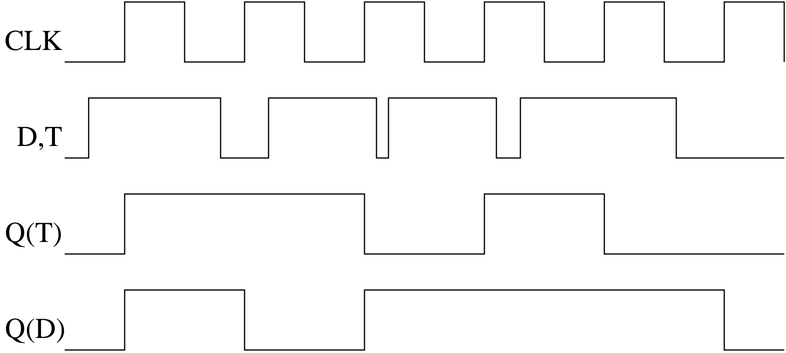
	x	0	0	1	1
y		0	1	1	0
0	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	0	1
1	0	0	0	1	1

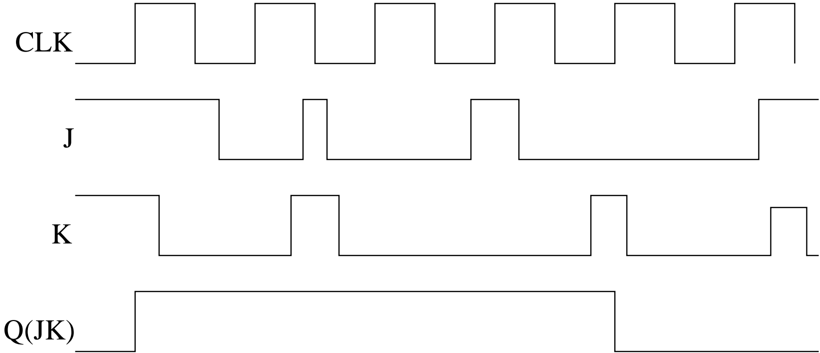


	x	0	0	1	1
y		0	1	1	0
0	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	1	0
1	0	0	0	0	0



Problem 3: Thou shall know his Flip-Flops !!!





Problem 4: Counter Theory !!!

A synchronous counter is one which changes all bits of the output which are going to change simultaneously. A ripple counter changes each of the bits in order, sequentially. A ripple counter is much more likely to be glitchy near the count, as in counting from 0111 to 1000, the progression of numbers seen will likely look like 0111, 0110, 0100, 0000, and 1000. Any one bit changing in a ripple counter happens faster than any one bit changing in a synchronous counter, but the time necessary for four bits in a ripple counter to all propagate is longer than the time needed for the synchronous counter. A ripple counter is much simpler in design, and requires less space for the same number of bits of counting.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity LS163 is
    port (clk, enp, ent, n_clr, n_ld : in std_logic;
          data_in                    : in std_logic_vector(3 downto 0);
          data_out                   : out std_logic_vector(3 downto 0);
          rco                        : out std_logic);
end LS163;

architecture behavioral of LS163 is
    signal count, allones : std_logic_vector(3 downto 0);
begin  -- behavioral
    clocked : process (clk)
    begin  -- process clocked
        if rising_edge(clk) then
            if (n_clr = '0') then
                count <= (others => '0');
            elsif (n_ld = '0') then
                count <= data_in;
            elsif (enp = '1') and (ent = '1') then
                count <= count + 1;
            end if;
        end if;
    end process clocked;
    allones <= (others => '1');
    rco <= '1' when ((ent = '1') and (count = allones)) else '0';
    data_out <= count;
end behavioral;
```

It is possible for RCO to be glitchy, so connecting RCO from one counter to the clock of another may cause the second counter to count more than once. The better was of cascading counters is to connect the RCo of one counter to the T-enable of the second counter, and give them both the same clock. RCO will have settled down by the time the second counter is clocked, so there will be no problems from glitches.

```
library ieee;
use ieee.std_logic_1164.all;

entity cascadeLS163 is
  port (clk, enp, ent, n_clr, n_ld : in  std_logic;
        data_in                    : in  std_logic_vector(7 downto 0);
        data_out                   : out std_logic_vector(7 downto 0);
        rco                        : out std_logic);
end cascadeLS163;

architecture behavioral of cascadeLS163 is
  signal rco_connect : std_logic;
  component LS163
    port (
      clk                   : in  std_logic;
      enp, ent, n_clr, n_ld : in  std_logic;
      data_in               : in  std_logic_vector(3 downto 0);
      data_out              : out std_logic_vector(3 downto 0);
      rco                   : out std_logic);
  end component;
begin  -- behavioral
  LS163_first :  LS163 port map (
    clk, enp, ent, n_clr, n_ld, data_in(3 downto 0),
    data_out(3 downto 0), rco_connect);
  LS163_second : LS163 port map (
    clk, enp, rco_connect, n_clr, n_ld, data_in(7 downto 4),
    data_out(7 downto 4), rco);
end behavioral;
```

Problem 5: Shifters !!!

Our one-bit shift cell is equivalent to a 4-input multiplexor, where the output is buffered with a D-flipflop. The two selector bits of the bit shifter at the selector bits for the multiplexor, and the three other inputs, along with the output of the D-flipflop looped back around, are the inputs to the multiplexor. The output of the D-flipflop is the output of the cell.

```
library ieee;
use ieee.std_logic_1164.all;

entity oneBitShiftCell is
  port (
    CLK, SR, SL, DI : in  std_logic;
    S               : in  std_logic_vector(1 downto 0);
    DO              : out std_logic);
end oneBitShiftCell;

architecture behavioral of oneBitShiftCell is
begin  -- behavioral
  process (CLK)
  begin  -- process
    if rising_edge(CLK) then
      case S is
        when "00" => DO <= SL;          -- Shift SL in from left when number
                                        -- shifts to the right.  SL -> ## ->
        when "01" => DO <= SR;          -- Shift SR in from right when number
                                        -- shifts to the left.  <- ## <- SR
        when "11" => DO <= DI;          -- Load data from input.
        when others => NULL;            -- Maintain old value of data.
      end case;
    end if;
  end process;
end behavioral;
```

For the 4-bit shift register, we need to connect the outputs of the single bit shifters to the `SL` and `SR` inputs of the adjacent cells, so the value will be passed from one cell to the next. In addition, when the third selector bit is high, we need to override the values we would be shifting left or right, so we instead shift in the two default values. Since the operations for when the second and third selectors are high isn't specified, we'll simply leave them the same as when the third selector is low.

```
library ieee;
use ieee.std_logic_1164.all;

entity fourBitShifter is
  port (
    CLK, SR, SL : in  std_logic;
    DI          : in  std_logic_vector(3 downto 0);
    S           : in  std_logic_vector(2 downto 0);
    DO          : out std_logic_vector(3 downto 0));
end fourBitShifter;

architecture behavioral of fourBitShifter is
  signal DO_temp, SR_temp, SL_temp : std_logic_vector(3 downto 0);
  component oneBitShiftCell
    port (
      CLK        : in  std_logic;
      SR, SL, DI : in  std_logic;
      S          : in  std_logic_vector(1 downto 0);
      DO         : out std_logic);
  end component;
begin  -- behavioral
  SR_temp <= "1000" when S(2) = '1' else DO_temp(2 downto 0) & SR;
  SL_temp <= "0000" when S(2) = '1' else SL & DO_temp(3 downto 1);

  bit_zero  : oneBitShiftCell port map (CLK, SR_temp(0), SL_temp(0), DI(0),
                                        S(1 downto 0), DO_temp(0));
  bit_one   : oneBitShiftCell port map (CLK, SR_temp(1), SL_temp(1), DI(1),
                                        S(1 downto 0), DO_temp(1));
  bit_two   : oneBitShiftCell port map (CLK, SR_temp(2), SL_temp(2), DI(2),
                                        S(1 downto 0), DO_temp(2));
  bit_three : oneBitShiftCell port map (CLK, SR_temp(3), SL_temp(3), DI(3),
                                        S(1 downto 0), DO_temp(3));
  DO <= DO_temp;
end behavioral;
```