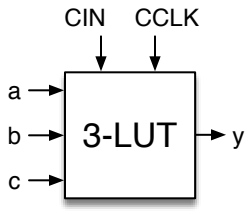
EECS 151/251A Homework 3

Due Monday, Feb 18th, 2019

Problem 1: LUT Implementation [12+3 pts]

Consider the design of a 3-input FPGA lookup table (LUT). An abstracted view of the 3-LUT is shown below. The ports, a, b, c, and y, are the data inputs and output, respectively. The input CIN stands for "configuration input", and CCLK stands for "configuration clock". The configuration clock is used at FPGA initialization time to shift in the LUT contents, one bit per clock cycle, over the CIN port.

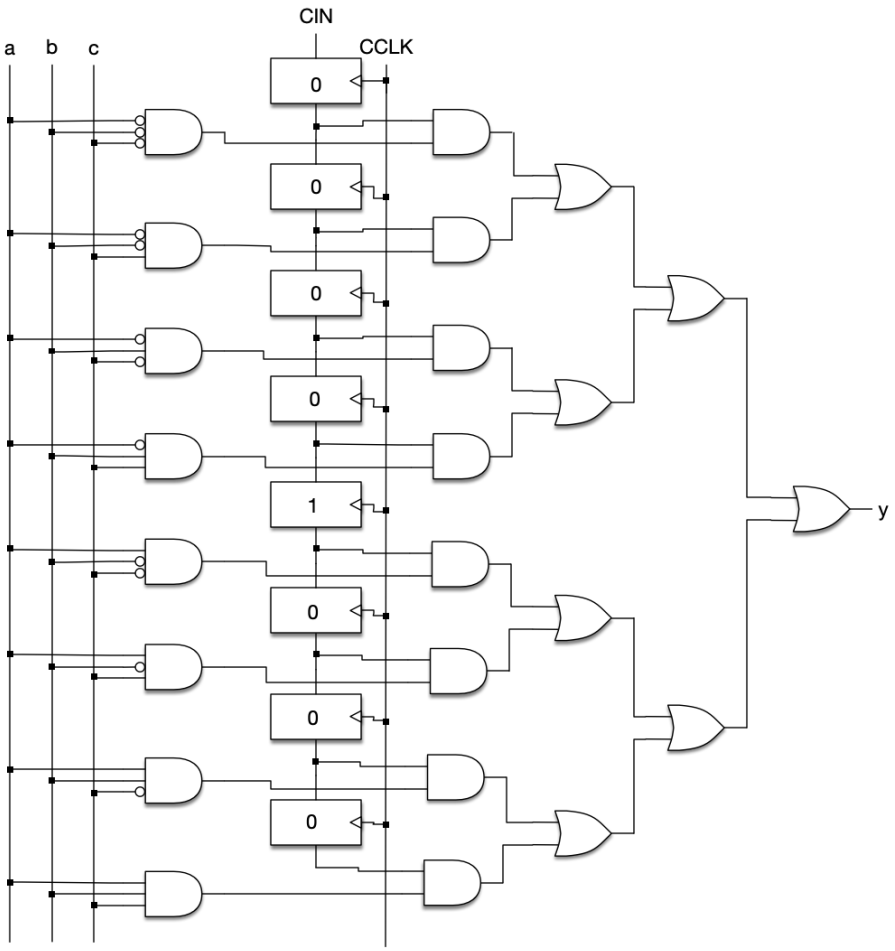


(a) Draw a circuit diagram for the internal implementation of the 3-LUT, including configuration loading. Use only the following circuit components: Inverter, 2-input and, 2-input or, Flip-flop. Remember to label all inputs and outputs.

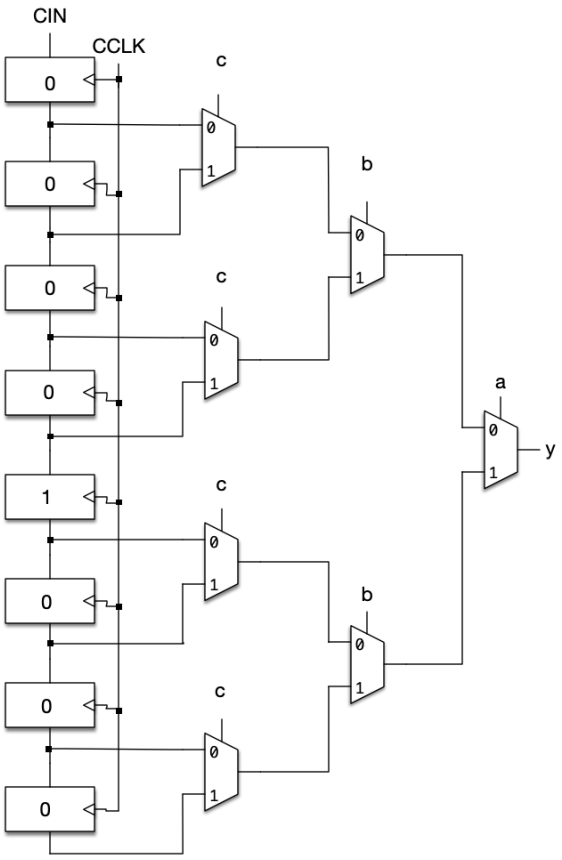
(b) Imagine now that the 3-LUT is programmed to implement the following function: `~a & b & c`. Label the appropriate nodes in your circuit diagram with the correct configuration values.

Solution:

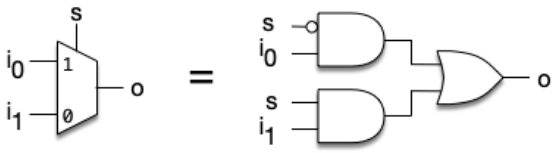
(a) One way is to use combinational logic to gate the output of each of a series of registers with the input values (which act as an index):



Another way is to use multiplexors:



Where each multiplexor is implemented logically as follows:



(b) Configuration shown in registers.

Problem 2: The "Uber" Flip-Flop [8 pts]

Write a Verilog module which implements a d-Flip-Flop with clock enable, synchronous reset, synchronous set, asynchronous preset, and asynchronous clear. Give priority to reset & clear. Give asynchronous priority over synchronous.

Synchronous Inputs:

- reset: q=0
- set: q=1

Asynchronous Inputs:

- preset: q=1
- clear: q=0

Note: The clock enable only effects synchronous inputs.

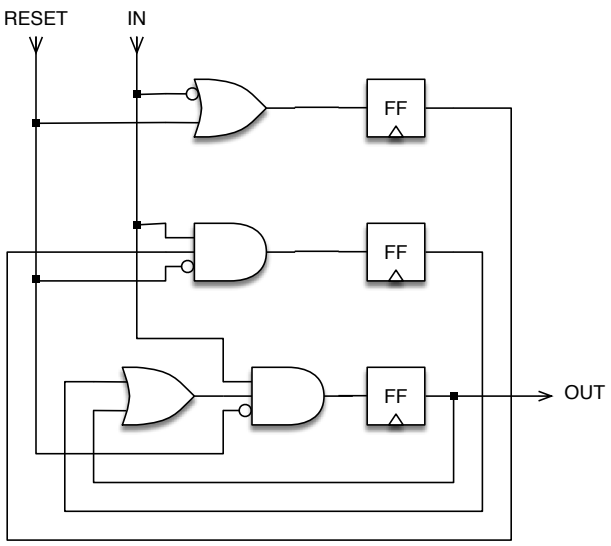
Solution:

```
module uber_flop(
  input clk,
  input d,
  input clk_en,
  input reset,
  input set,
  input preset,
  input clear,
  output reg q);

  always @(posedge preset or posedge clear or posedge clk) begin
    //async priority over sync
    if(clear) begin
            q <= 0;
    end else if(preset) begin
              q <= 1;
    end else begin
      //Sync
      if(clk_en) begin
        //Clk enabled
        if(reset) begin
          q <= 0;
        end else if(set) begin
          q <= 1;
        end else begin
          q <= d;
        end
      end
    end
  end
endmodule
```

Problem 3: Verilog Circuit Implementation [5 pts]

Write a Verilog description of the circuit shown below.

Note: A small circle drawn at an input port indicates that the input signal is inverted before passing to the gate.



Solution:

```
module problem3(
  input IN,
  input RESET,
  input clk,
  output OUT);

  reg r0;
  reg r1;
  reg r2;

  always @(posedge clk) begin
    r0 <= (~IN)|RESET;
    r1 <= IN&r0&(~RESET);
    r2 <= IN&(r1|r2)&(~RESET);
  end

  assign OUT = r2;

endmodule
```

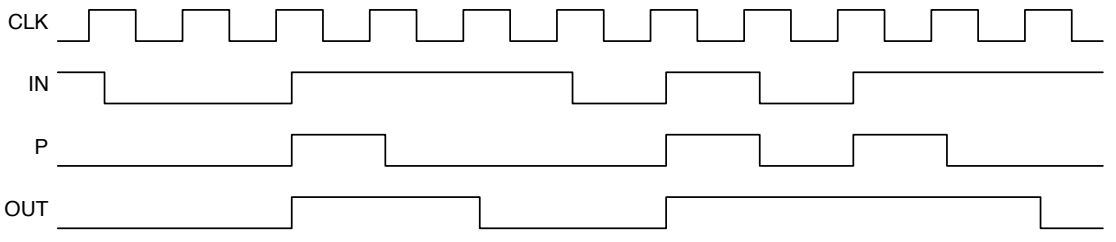
Since this is a finite state machine like you've now seen (or will soon see) in class, the preferred design pattern is to actually seperate the combinational logic in each state from the state transition itself (on a clock edge), as below:

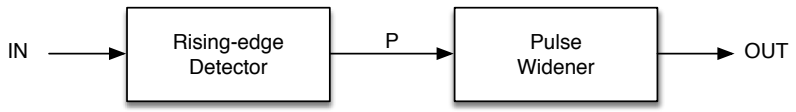
```
module problem3(
  input IN,
  input RESET,
  input clk,
```

```
  output OUT);

  reg r0, nr0;
  reg r1, nr1;
  reg r2, nr2;

  always @(*) begin
    nr0 = (~IN)|RESET;
    nr1 = IN&r0&(~RESET);
    nr2 = IN&(r1|r2)&(~RESET);
  end

  always @(posedge clk) begin
    r0 <= nr0;
    r1 <= nr1;
    r2 <= nr2;
  end

  assign OUT = r2;

endmodule
```

Problem 4: Clocked Circuits [3+3 pts]

In this problem, you will design a simple synchronous circuit that outputs a pulse of width 2 clock cycles on each rising edge of the input signal. An example input and output signal is shown below. To make things easier, we will split the design into two parts. The "Rising-edge Detector" outputs a pulse in response to a rising edge on the input, as shown below. The "Pulse Widener" stretches out the pulse for 2 cycles.



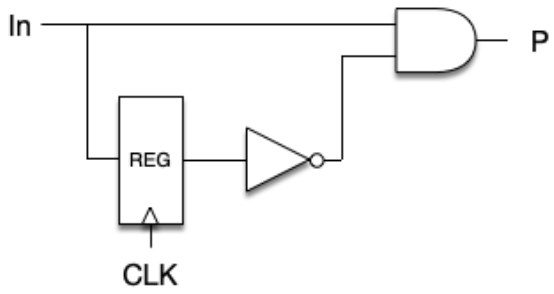


(a) Draw your circuit for the Rising-edge Detector. Use only the following circuit components: Inverter, 2-input and-gate, 2-input or-gate, flip-flop. Simpler designs are worth more points.

(b) Draw your circuit for the Pulse Widener.

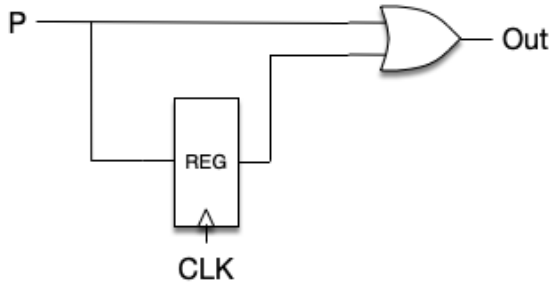
Solution:

(a) Rising Edge Detector:



Note that edge detection occurs immediately when the input changes (it is not delayed by a cycle). *Assumption: In* remains high for at least 1 positive edge of *CLK*.

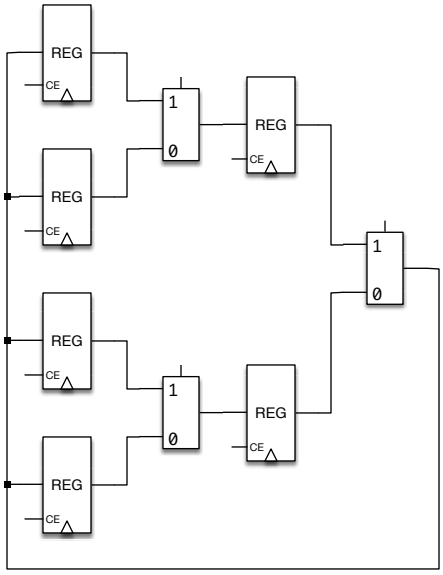
(b) Pulse Widener:



Note that the timing diagram shows that the pulse is seen immediately when P rises . It is high on the immediate edge. *Assumption:* P remains high for at least 1 positive edge of *CLK*.

Problem 5: Register Transfers [6 pts]

Given the datapath below, what is the minimum number of clock cycles needed to exchange the top-left-most register value with the bottom-left-most register value?

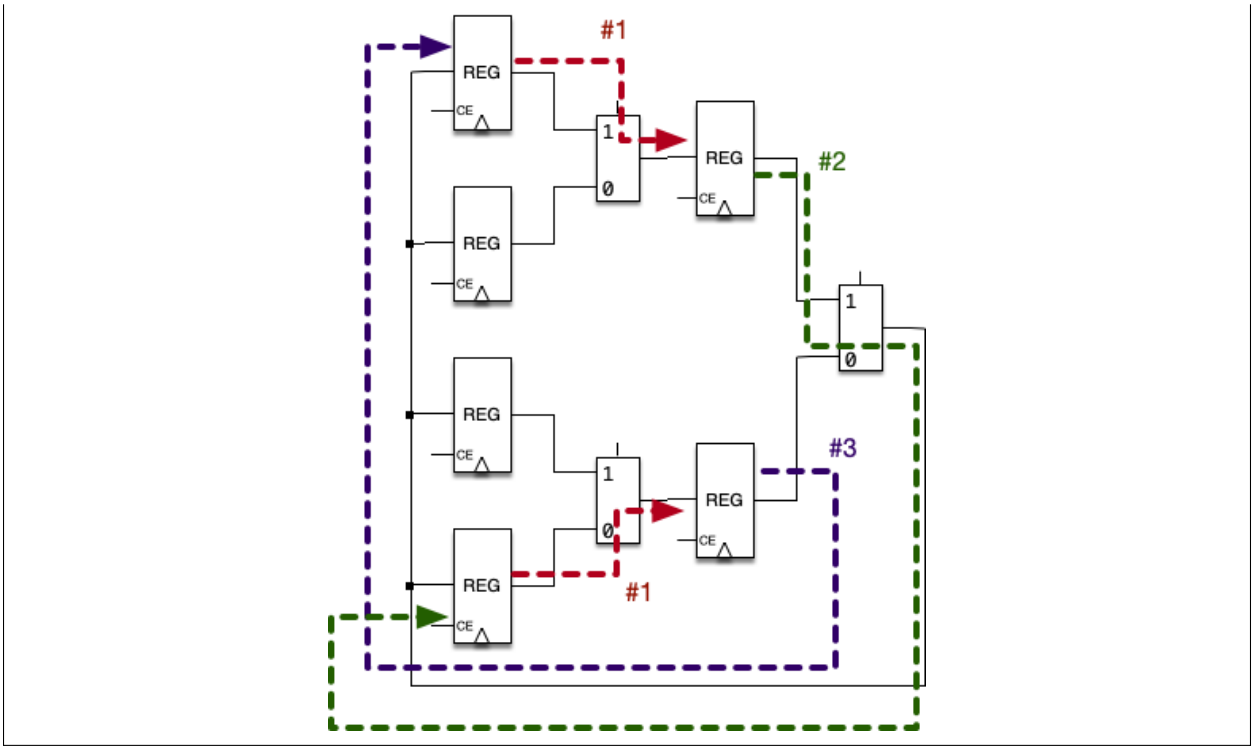


Solution:

It would take 3 cycles to swap the values in the bottom-left and top-left registers. For example, on cycle:

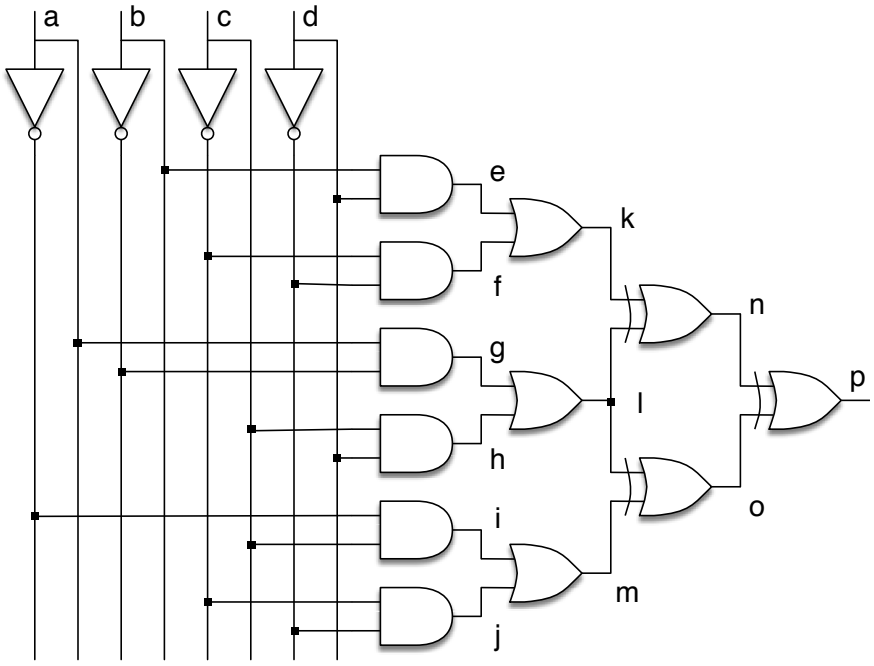
1. move values in bottom-left and top-left registers into registers in the middle of the diagram;
2. load temporary value from top-left into bottom-left;
3. load temporary value from bottom-left into top-left.

(But the operations on cycles 2 and 3 could be swapped.)



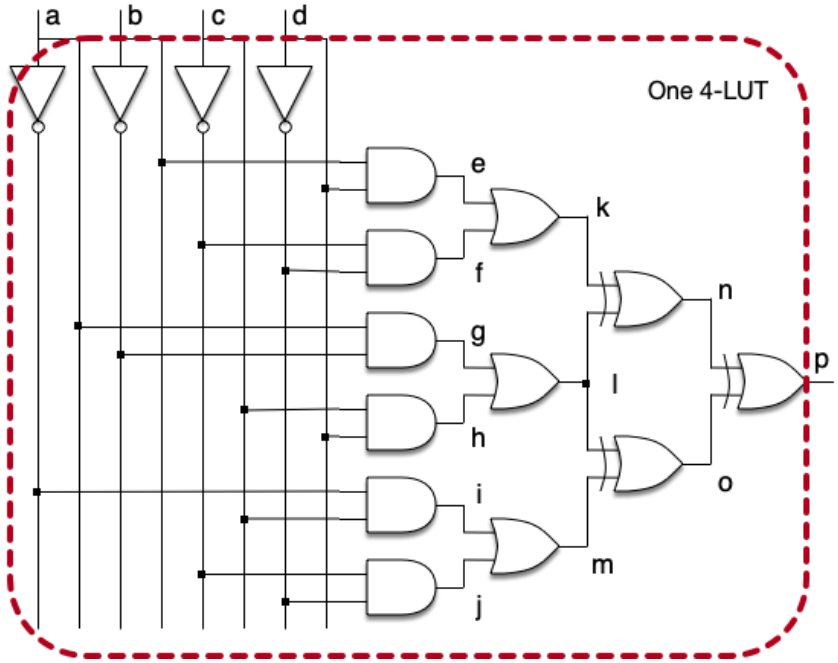
Problem 6: FPGA Mapping [4 pts]

Using only 4-input lookup tables (LUTs), partition the circuit shown below into as few LUTs as possible. *Do not attempt to simplify the gate-level circuit before mapping it to LUTs.* Indicate your answer by drawing boundaries around your partitions.



Solution:

The entire circuit would fit into a single 4-LUT. The inputs would be a, b, c and d. The output would be p.



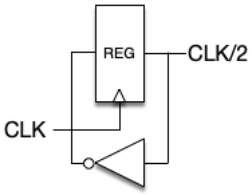
Problem 7: 251A only — *Optional* Challenge Question for 151 [8 pts]

Using only simple logic gates and flip-flops, derive a circuit whose output is a *square-wave* with 1/8 the frequency of the input clock frequency. Try to use as few flip-flops and gates as possible.

Hint: Start out by first designing a circuit that outputs a square-wave with 1/2 the clock frequency, then one with 1/4, etc.

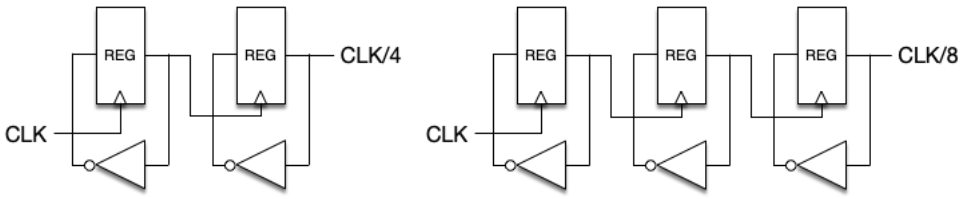
Solution:

One method is to build a clock divider and to then use its output as the clock input to subsequent stages. The first step is to build a clock divider:

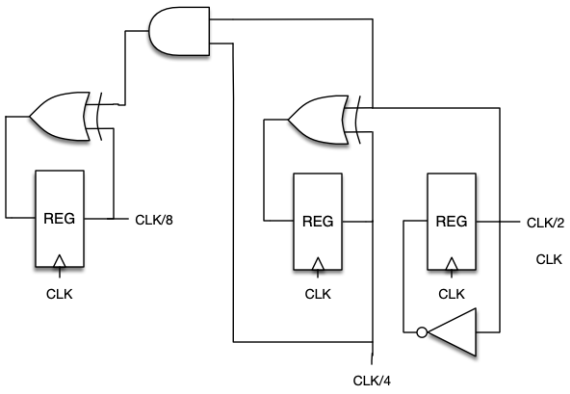


The output is a square-wave at half the frequency of the input clock signal *CLK* because state transitions only occur on its positive edge.

The next step is to chain clock dividers until the desired divider is achieved:



However, it is *bad practice* in general to use clock signals as data inputs. A more robust solution is a synchronous circuit. One such solution is based on a counter:



Problem 8: LUT Mapping [10+8 pts]

Imagine you are given a "mystery" FPGA programmed to perform a particular set of functions. We know the following about the FPGA:

1. The device contains a collection of N-LUTs (the value of N is part of the mystery). The LUTs are numbered 0, 1, 2, ... Each LUT in the FPGA has the same number of inputs (same N).
2. Each LUT has an output labeled y_i, where i is the LUT number, and inputs labeled x_{i_j}, where i is the LUT number and j is the input number.
3. For programming, the LUTs are connected in a shift register. They are programmed with a configuration bit-stream shifted in from one LUT to the next.
4. LUT 0 is programmed as an inverter, therefore $y_0 = NOT(x_{0\_0})$. Futhermore, LUT 0 is the only LUT programmed as an inverter.
5. We were able to recover a fragment of the bit stream, shown here: 0xE9AC96017F88FF55.

(a) How many LUTs would the above bit stream program?

(b) For each of the LUTs, draw a circuit using simple logic gates to represent the function it is programmed to implement. Label all inputs and outputs. Show your work.

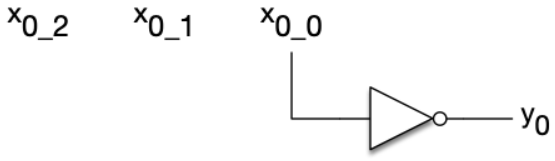
Solution:

A LUT configured as inverter for its 0-th input will have to have alternating 0s and 1s in its table entries. To see why, consider that the other inputs - which effectively form an index into the table - should not matter. Thus the pattern of flipping the value of the 0-th input must repeat.

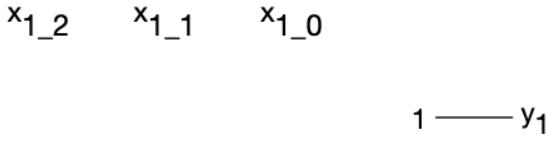
We're told that there is only one inverter. Looking at the configuration string, we find an repeated '01' pattern in the last octet: 0x55 = 01010101. We can thus surmise that the first octet configures the first LUT, that this is LUT-0 so configuration is loaded right to left, and that each LUT has 8 bits of state. Since there are 8-bits of state, we have N = $\log_2(8) = 3$.

(a) There are 64 bits of information in the configuration string. Since each LUT needs 8, the bit stream programs 64 / 8 = 8 LUTs.

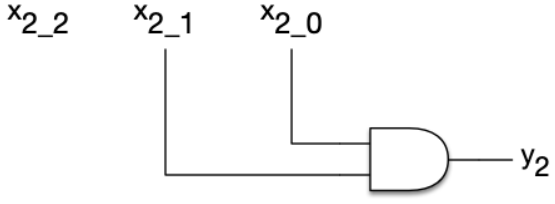
(b) LUT-0; 0x55 = 01010101
An inverter.



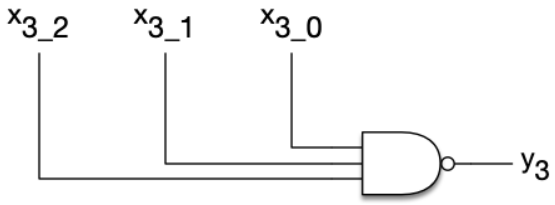
LUT-1; 0xFF = 11111111



LUT-2; 0x88 = 10001000

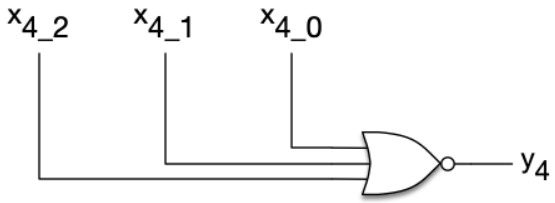


LUT-3; 0x7F = 01111111



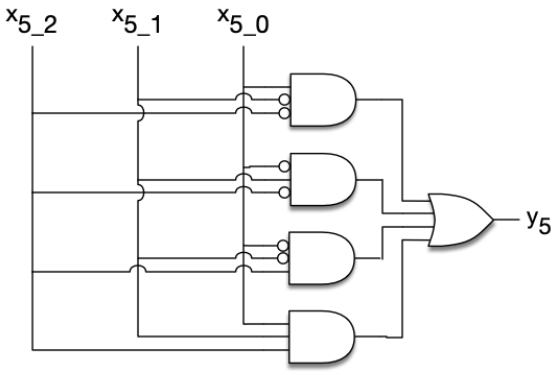
LUT-4; 0x01 = 00000001

x_{4_2}	x_{4_1}	x_{4_0}	y_4
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	



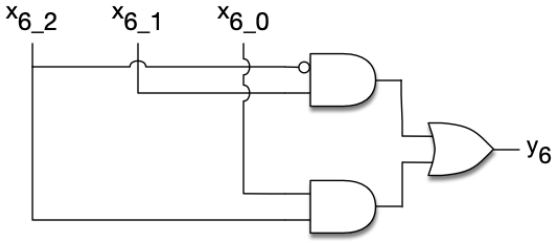
LUT-5; 0x96 = 10010110

x_{5_2}	x_{5_1}	x_{5_0}	y_4
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



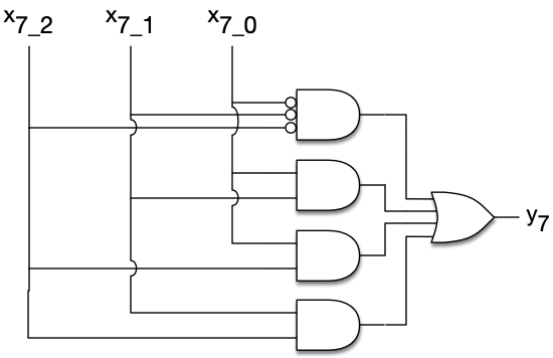
LUT-6; 0xAC = 10101100

x_{6_2}	x_{6_1}	x_{6_0}	y_4
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



LUT-7; 0xE9 = 11101001

x_{7_2}	x_{7_1}	x_{7_0}	y_4
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

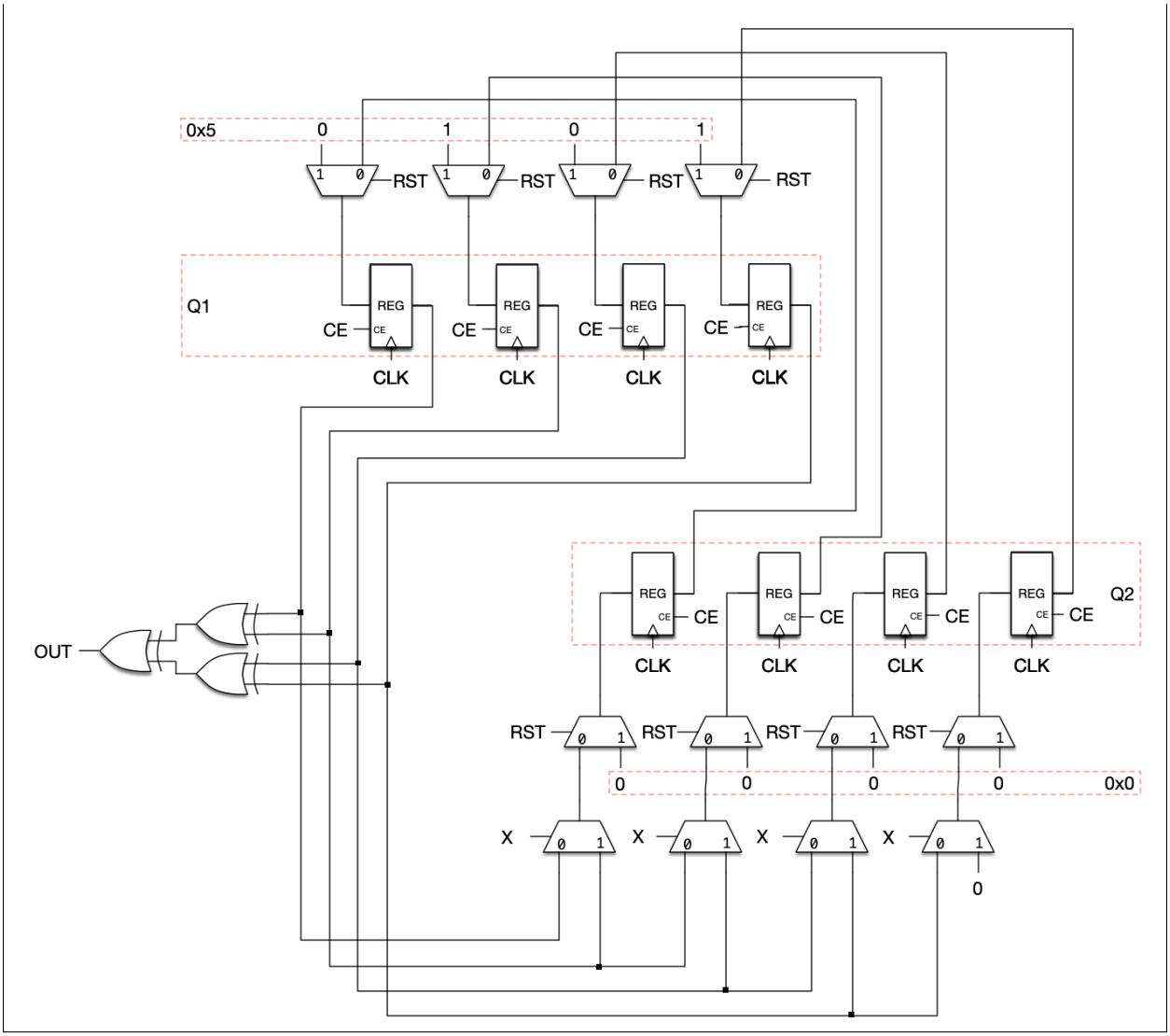


Problem 9: Interpreting Verilog [5 pts]

Neatly sketch the circuit that corresponds to the following Verilog code:

```
module foo (CE, X, CLK, RST, OUT);
   input CE, CLK, X, RST;
   output OUT;
   reg [3:0] Q1;
   reg [3:0] Q2;
   assign OUT = ^Q1;
   always @ (posedge CLK)
     if (RST) begin
        Q1 <= 4'h5;
        Q2 <= 0;
     end
     else
       if (CE) begin
          if (X) Q2 <= Q1<<1;
          else Q2 <= Q1;
          Q1 <= Q2;
       end
endmodule // foo
```

Solution:


0x5
0
1
0
1
RST
RST
RST
RST
Q1
REG
CE
CLK
OUT
Q2
0
0
0
0
0x0
X
X
X
X
0