ECE 120: Introduction to Computing Spring 2019 ZJUI

due at start of section on Friday 19 April 2019

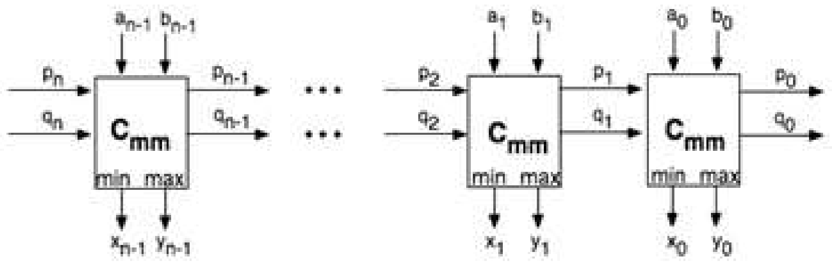
Homework 8: Finite State Machines

Please note that this homework is somewhat shorter than usual so that you can focus on preparing for the midterm on 16 April. The first problem partially covers material included in the midterm (**part (a)** and analysis of **part (a)** in **part (c)**). The remaining problems are midterm 3 material; the programming problem will help you prepare for Lab 4 (demos on 26 April).

1. Min-max using Serialization

Shown below is a min-max circuit. Given unsigned n-bit integer inputs A = $a_{n-1}...a_1a_0$ and B = $b_{n-1}...b_1b_0$, the min-max circuit produces outputs X and Y such that

$$X = x_{n-1}...x_1x_0 = \text{minimum }(A, B)$$
$$Y = y_{n-1}...y_1y_0 = \text{maximum }(A, B)$$



The C_{mm} bit-slice is similar to the comparator discussed in the notes and in class, but passes information from the most significant bit (on the left) towards the least significant bit (on the right). The values passed between bit slices are defined as follows:

$$pq = x0 \text{ iff } A = B$$
$$pq = 11 \text{ iff } A < B$$
$$pq = 01 \text{ iff } A > B$$

a. Implement the C_{mm} bit slice using gates.

We can draw K-maps, or we can try to reason through the design. Consider q_k, which should be 1 iff $a_{n-1}\ldots a_k \neq b_{n-1}\ldots b_k$. For that to be true, we need either $a_{n-1}\ldots a_{k+1} \neq b_{n-1}\ldots b_{k+1}$, in which case $q_{k+1}=1$, or $a_k \oplus b_k=1$ (or both alternatives can be true). We can thus write

$$q_k = q_{k+1} + a_k \oplus b_k.$$

Now consider p_k, which should equal to p_{k+1} whenever $q_{k+1}=1$, which gives $p_{k+1}q_{k+1}$. When $q_{k+1}=0$, p_k should be 1 if $a_k'b_k$ ($A < B$ starting at the current bit). So we can write

$$p_k = p_{k+1}q_{k+1} + q_{k+1}'a_k'b_k.$$

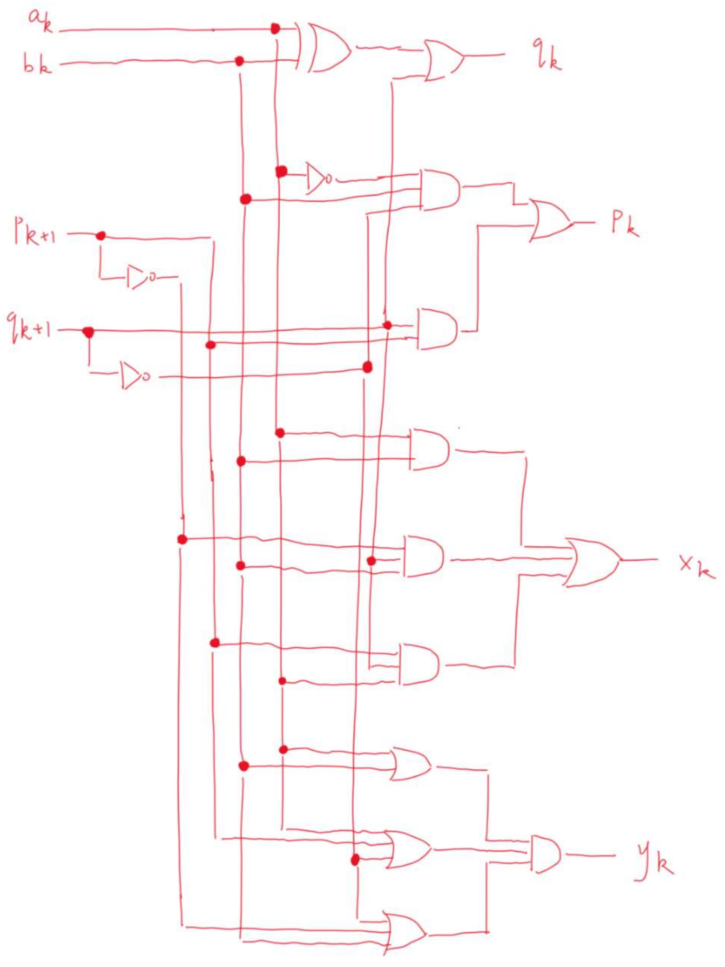
Now consider x_k. If both of the AB bits are 1 ($a_kb_k=11$), x_k must of course be 1. Otherwise, we have three possibilities based on $p_{k+1}q_{k+1}$. Case 1: $a_{n-1}\ldots a_{k+1}=b_{n-1}\ldots b_{k+1}$ ($q_{k+1}=0$), in which case the minimum of the two bits is 0 (since they're not both 1). Case 2: $a_{n-1}\ldots a_{k+1}>b_{n-1}\ldots b_{k+1}$ ($p_{k+1}q_{k+1}=01$), in which case x_k is given by b_k. And case 3: $a_{n-1}\ldots a_{k+1}<b_{n-1}\ldots b_{k+1}$ ($p_{k+1}q_{k+1}=11$), in which case x_k is given by a_k. So we write

$$x_k = a_kb_k + p_{k+1}'q_{k+1}b_k + p_{k+1}q_{k+1}a_k.$$

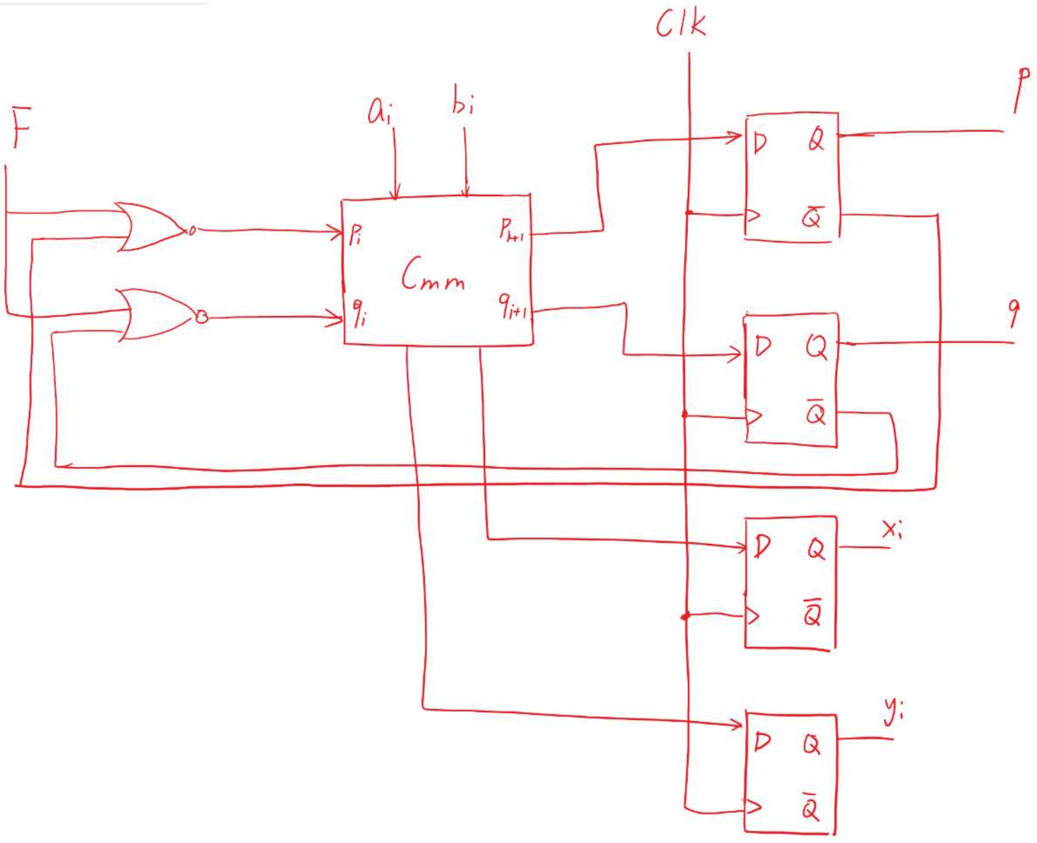
Finally, consider yk. In this case, POS is more natural. At least one of the AB bits must be 1 to produce a maximum of 1, so we start with a factor of $(a_k + b_k)$. To produce a 1, we also need either $a_{n-1}\ldots a_{k+1} \leq b_{n-1}\ldots b_{k+1}$ ($p_{k+1} + q_{k+1}'$) or $a_k=1$ (if the reason is not clear, think about the complemented logic: $a_{n-1}\ldots a_{k+1}>b_{n-1}\ldots b_{k+1}$ and $a_k=0$). Finally, to produce a 1 for the maximum bit y_k, we need either $a_{n-1}\ldots a_{k+1} \geq b_{n-1}\ldots b_{k+1}$ ($p_{k+1}' + q_{k+1}'$) or $b_k=1$. Putting the three factors together gives

$$y_k = (a_k + b_k)(a_k + p_{k+1} + q_{k+1}')(b_k + p_{k+1}' + q_{k+1}')$$

The K-map approach may be a little easier, but the answers are the same.



b. Use the C_{mm} bit slice and the approach discussed in class and in the notes to design a serial version of the min-max circuit. Draw a circuit showing how your design could be used to find the min/max of two 4-bit numbers. Be sure to label all inputs and connect them appropriately. The p_i bit actually needs no selection logic, since F=1 implies q_i=0, in which case p_i is a don't care, but we've included the extra NOR gate for clarity.

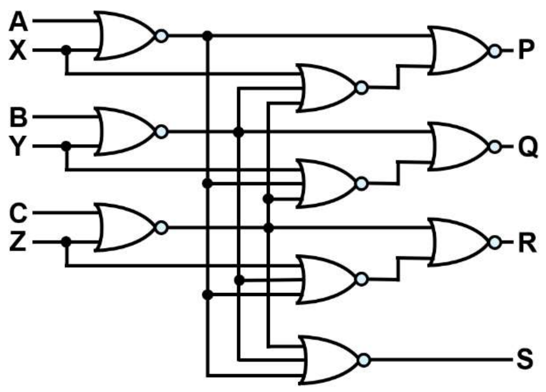


c. Compare the bit-sliced and serial designs in terms of area and delay as a function of the number of bits **N**.

	Bit-sliced Design	Serial Design
Delay	2N	11N (minimum)
Area	33N	109 = 4 + 33 + 18×4

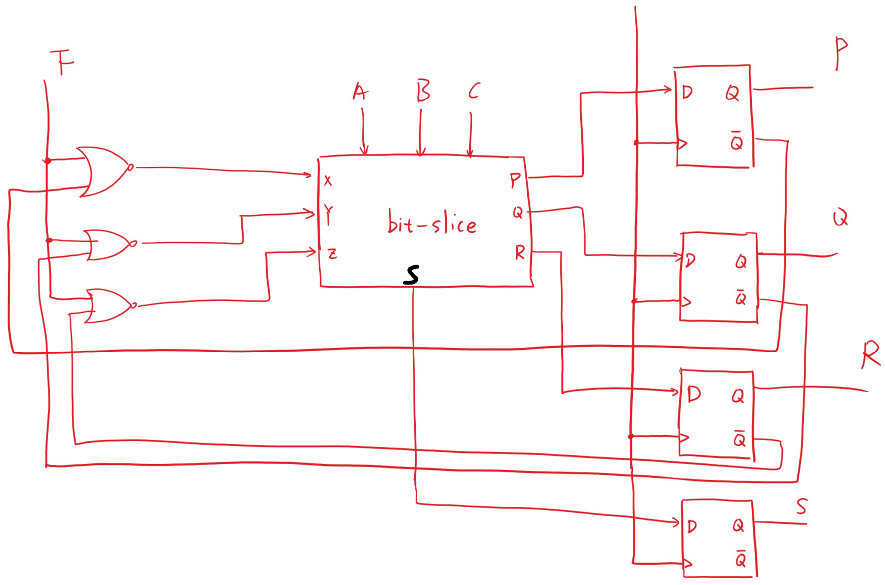
Note that if one chooses to count the inverters on the bit-slice-to-bit-slice path in the bit-sliced design, one obtains 35N – 1 area and 3N – 1 delay for the bit-sliced design (the inverters are still not necessary for the serial design).

2. **Serialization of Bit-Sliced Designs**
The bit slice shown to the right calculates … something … on three inputs. Each bit slice accepts one bit of the three values (inputs A, B, and C) and three bits from the previous bit slice (inputs X, Y, and Z). Each bit slice calculates three bits for the next bit slice (outputs P, Q, and R) as well as one bit of the final output (output S). The input to the first bit slice is XYZ=000.



a. Re-implement the bit-sliced design shown using the *serialization* approach presented in Lecture Notes Set 3.1. Use the bit-slice circuit as a building block (draw it as a rectangle—don't redraw the circuit) and add the storage and logic necessary to turn the bit-slice into a serial implementation.
b. Calculate the area and delay for an N-bit bit-sliced implementation using the bit slice shown.
c. Calculate the area and minimum delay when using your serialized implementation from **part (a)** on N-bit inputs.

a.



b. c.

	Bit-slice Design	Serialization Design
Delay	3N	12N (minimum)
Area	24N	102 = 6 + 24 + 18×4

3. Implementation of the Keyless Entry System

Refer to Lecture Notes Section 3.1.3 on pages 82-84 for the FSM design. You may also want to read about 5-variable K-maps at http://www.allaboutcircuits.com/vol_4/chpt_8/11.html. We suggest using the Gray Code K-maps.

a. Draw K-maps for outputs D, R, A and next states S_1^+ and S_0^+ as functions of U, L, P, and current states S_1 and S_0.

b. Based on these K-maps, write minimal SOP expressions (minimal area) for D, R, A, S_1^+, and S_0^+.

c. Draw a *high-level* implementation of this FSM consisting of two D flip-flops for storing current state and five boxes for computing D, R, A, S_1^+, and S_0^+ values. Clearly label all inputs and outputs and draw all connections between the boxes. Note that **we are NOT asking you to draw the detailed gate-level implementations** for your expressions in **part (b)**.

a.

D		S_1	
		0	1
S_0	0	0	1
	1	0	1

R		S_1	
		0	1
S_0	0	0	0
	1	0	1

A		S_1	
		0	1
S_0	0	0	0
	1	1	0

S_1^+		ULP							
		000	001	011	010	110	111	101	100
S_1S_0	00	0	0	0	0	0	0	0	1
	01	0	0	0	0	0	0	0	0
	11	1	0	0	0	0	0	0	1
	10	1	0	0	0	0	0	0	1

S_0^+				ULP				
S_1S_0	000	001	011	010	110	111	101	100
00	0	1	1	0	0	1	1	0
01	1	1	1	0	0	1	1	1
11	1	1	1	0	0	1	1	1
10	0	1	1	0	0	1	1	1

b. $D = S_1$

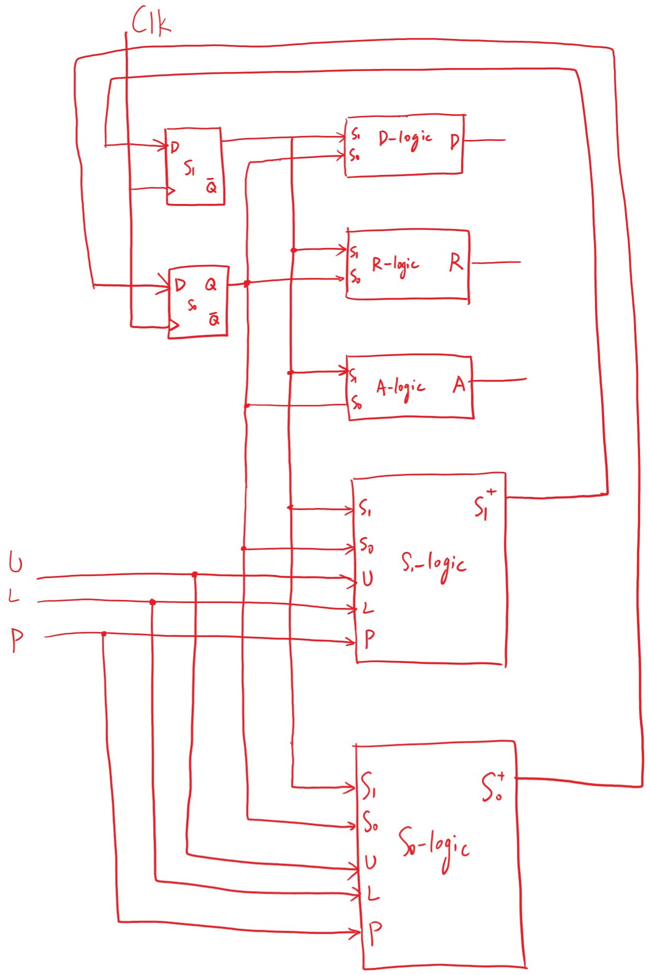
$R = S_1S_0$

$A = S_1'S_0$

$S_1^+ = S_1L'P' + S_0'UL'P'$

$S_0^+ = P + S_0L' + S_1UL'$

c.



4. FSM Simulation

Update your Subversion directory to obtain the subdirectory **`hw8dist`**, which contains the program **`FSMsimulate.c`**. The program simulates an FSM with three bits of state (S_2, S_1, and S_0), one input (T), and two outputs (A and P).

a. Use the table printed by the program to draw a state transition diagram for the FSM simulated by the code. *Hint: We strongly suggest that you use the printed table, which covers all reachable transitions, rather than trying to derive the state transition diagram from the equations in the code.*

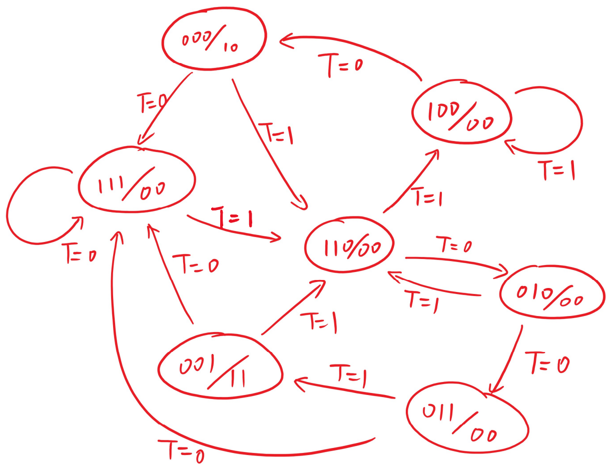
b. In what state is the FSM placed at the start of the simulation? (What are the initial values of S_2, S_1, and S_0?)

c. Which, if any, states are unreachable from the initial state? (Your answer should again be in terms of S_2, S_1, and S_0.)

d. The FSM simulated by the code is a sequence recognizer. It recognizes two input sequences of different lengths. Using your state transition diagram, identify the two sequences recognized and explain the meaning of the outputs A and P in terms of these sequences.

e. Notice that the initial state used in the code is in fact one of the recognition states for one of the two sequences. In which state should the FSM have started in order to guarantee that a full sequence has appeared in the input before the FSM reports a sequence match?

a.



b. $S_2S_1S_0$ = 000 is the starting state.

c. $S_2S_1S_0$ = 101 is an unreachable state.

d. The recognizer recognizes the sequences 110 and 1001. These sequences are not allowed to overlap. So, for example, the sequence 11001 reports only the first sequence, not the second (1001). The output A indicates that a sequence has been recognized, and the output P indicates which sequence: P=0 means 110, and P=1 means 1001.

e. The FSM should start in state $S_2S_1S_0$ = 111.

5. **Simulation of the Lab FSM**
   Read Section 3.3 of the class notes, "Design of the Finite State Machine for the Lab." In a previous homework, you calculated expressions for both the A and P output signals using only NAND as well as only NOR functions.

   a. Use the state transition diagram (or, if you prefer, the next state table) provided in the notes to calculate next-state logic expressions for the lab FSM using only NAND gates (start with optimal SOP, then transform the equations to NAND, remembering that both state variables and their complements are available directly from the flip-flops).

      In the program **`FSMsimulate.c`**, replace the output and next-state computations with your NAND expressions. Be sure that you remove all but the LSB from each variable using &1, as is already done in the program.

      Now replace the test vector input with the bits provided in the notes (under "Testing the Design"). You should include both the initialization sequence as well as the testing sequence from the notes in the new value of **`TBits`**. Note that the input value **`T`** is taken first from the LSB of **`TBits`**, so you will need to reverse the order given in the notes, then translate from binary to hexadecimal (do not use decimal). Your full test sequence should require 15 inputs, so you should have four hex digits. Change the loop to simulate for 15 cycles instead of 21.
      The file **`goldOutput`** contains the correct output (printed table) for your modified program. To compare your results with the correct one, compile your program, say to the executable **`myProgram`**, then execute your code and send the output to a file by typing, "**`./myProgram > myFile`**" (no quotes, and any unused file name will do). Finally, execute "**`diff goldOutput myFile`**" (again, no quotes) to see a list of the differences between the correct output and your output. If the **`diff`** command outputs nothing, your equations are correct!

      For **part (a)**, commit your modified code back to your Subversion repository.

   b. Repeat **part (a)** using only NOR gates (start with POS, then transform the equations to NOR). Copy your **`FSMsimulate.c`** program into a second file called **`NORsimulate.c`**, then add the new file to your Subversion repository. **NOTE: If you fail to add the file to your repository, you will earn no credit for part (b).** Follow the same steps: insert your NOR-based equations into **`NORsimulate.c`** program, compile it, and check that it produces the same table as appears in **`goldOutput`**.

      For **part (b)**, commit the modified **`NORsimulate.c`** file to your Subversion repository.

   NOTE: You will not receive solutions for this problem. If you turn in an incorrect solution, you will lose points, but you must find correct expressions for your use in building the FSM on the protoboard. Be sure that you obtain equations in both NAND and NOR form that correctly reproduce the **`goldOutput`** file.