ECE 120: Introduction to Computing Spring 2019 ZJUI

due at start of section on SUNDAY 5 May 2019

Homework 10: From FSMs to Computers

1. Handling Bad Coins

Your ECE120 lab partner (for Labs 2, 3, and 4) has secretly been working on an extension to detect bad coins that might otherwise fool the ECE120 coin detection sensors into reporting that a yuan or a jiao had been inserted.

Your partner's new sensors produce a signal B that indicates that a particular coin is bad (B=1 for a bad coin, and B=0 for a real coin).

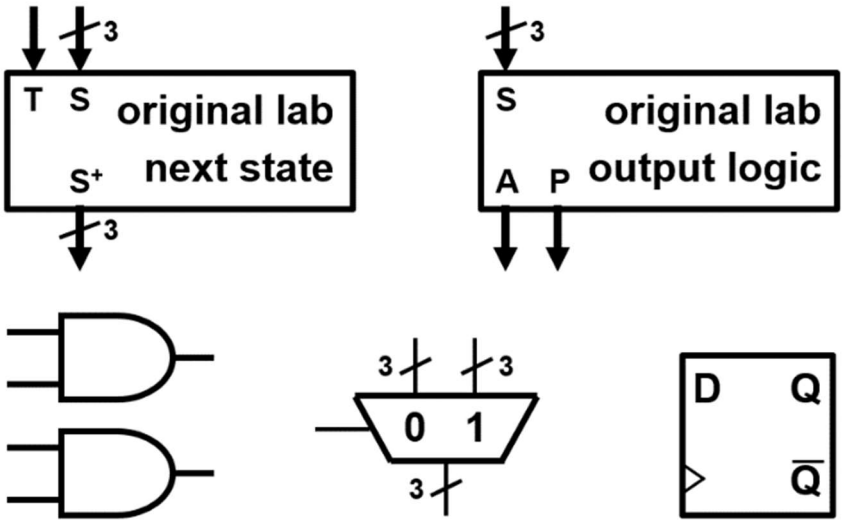
Your mission, should you choose to accept it, is to extend the lab FSM design to reject bad coins without affecting the state of the original FSM.

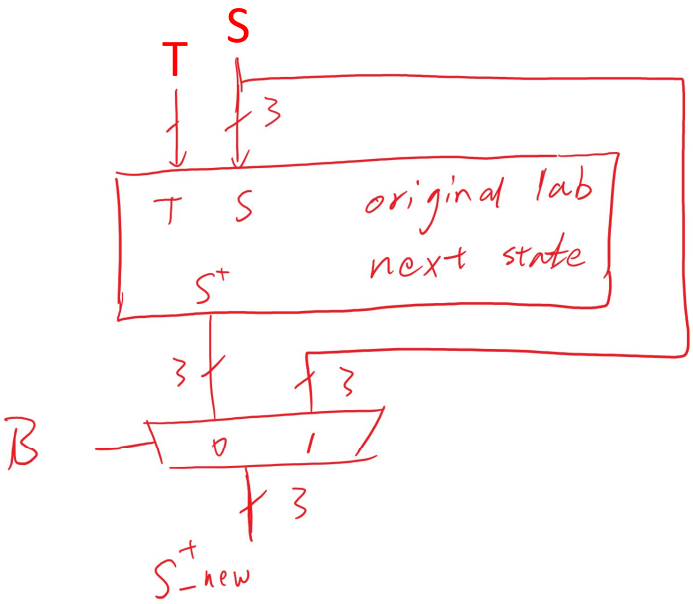
Starting with "components" that produce the correct next state, the A output, and the P output for the original lab design (be sure to include necessary inputs to these "components"!) and the additional components and gates shown below, eliminate the threat of bad coins. Be sure that no bad coin can lead to a state change in the original flip-flops, no bad coin can ever be accepted (A=1), and no bad coin can lead to the FSM indicating that payment has been received (P=1).

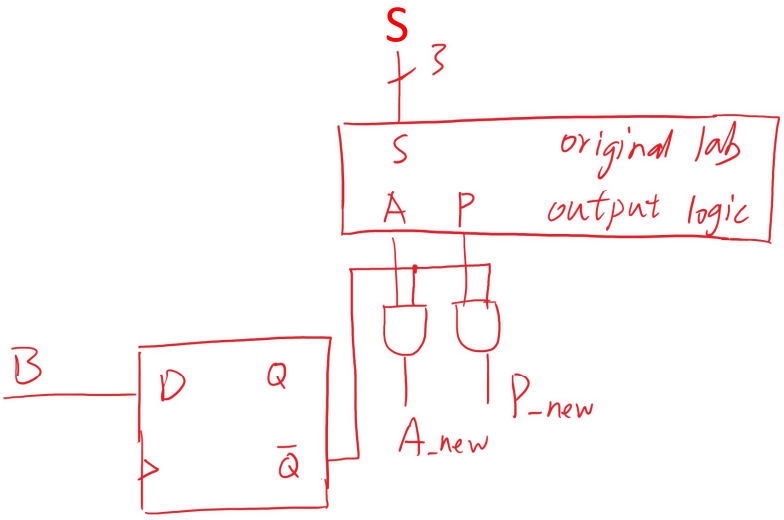
Note that you can make no changes at all to the original next-state logic nor to the original calculations of A and P. These "components" have been thoroughly implemented and debugged. You can, of course, further process the outputs of these "components."

Draw a circuit diagram illustrating how your extension integrates with the original lab design. In particular, combine the components shown below to produce the next state S^+ and the outputs A and P. As inputs, your circuit should use S to represent the current 3-bit state of the FSM, T to represent the coin type input, and B to represent the bad coin signal. **YOU MAY ONLY USE THE COMPONENTS SHOWN.**

This message will self-destruct in …




T
S
3
T S
original lab
next state
S+
3
3
B
0
1
3
S+_new


S
3
S
original lab
A P
output logic
B
D
Q
Q̄
A_new
P_new

2. Helping a Friend

A friend has developed a program, **`binary-count.c`**, intended to evaluate Prof. Lumetta's claim that binary counters built with parallel and serial gating (both terms are defined in Sec. 3.1.4 of the notes) are equivalent. Your friend uses two sets of state variables to simulate the two counter versions. Unfortunately, the parallel counter seems to be counting backward. Update your Subversion repository to obtain the hw10 subdirectory, which contains the program. Compile and run the program to see the behavior, then read the code to understand what is going wrong. Finally, turn in an explanation of your friend's mistake.

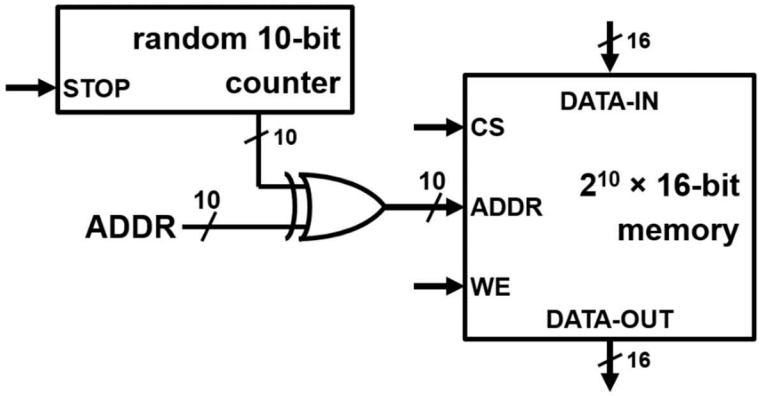
When the friend implemented parallel gating, he forgot that C works sequentially, so the next state is not calculated correctly. For example, P1, P2, and P3 make use of the new value of P0 rather than the old value. The current state bits are not recorded. Higher bits of the new state are based on current bits which are also updated, leading to incorrect state updating.

To correct the code, we can use another set of bits p0, p1, p2 and p3 to record current state and update P0 to P3 according to p0 to p3. Or just reverse the order of the updates, since P0 does not depend on P1, P2, nor P3.

```
/* Advance parallel counter. */
p0 = P0;
p1 = P1;
p2 = P2;
p3 = P3;
P0 = (p0 ^ 1);
P1 = (p1 ^ p0);
P2 = (p2 ^ (p1 & p0));
P3 = (p3 ^ (p2 & p1 & p0));
```

3. Scrambling Addresses

A friend in ECE120 has come up with the design below to randomize memory addresses.



The 10-bit counter is simply a loop of all 1,024 10-bit patterns, produced at a rate of one pattern per clock cycle, in some random order. The STOP input allows the counter to be stopped; whenever STOP=1, the counter holds its current 10-bit pattern until STOP is lowered.

Your friend's intention is to guard against certain memory-based attacks by scrambling the address sequence used for the attack. Each address sent to the memory is XOR'd (bitwise) with the 10-bit pattern produced by the counter.

a. Your friend wants to let the counter continue to change state (STOP=0) while using the memory, but finds that memory does not work when he tries to operate in this mode. Explain why.

Since the counter continues to change state while using the memory, the same address passed to the memory at different times may be XORd with different counter outputs. So the same memory address points to different memory cells at different times.

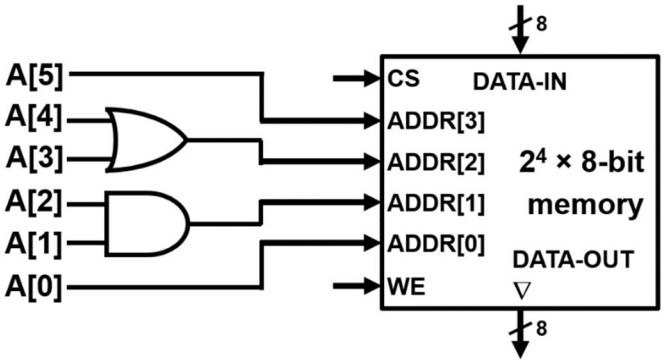
b. Your friend's ECE120 lab partner has suggested to him that he take the following approach: after turning on the power and letting the counter run for a while, stop the counter (in a random state), then make use of the memory, keeping the counter stopped until the power is turned off. Each time the memory is used (from the first use of memory until power off), the counter holds a new random value.

Your friend thinks that the memory still won't work. Is he right or wrong? Explain your answer.

He is wrong. The memory will work this time. Each time the memory is used, the bits used to mask addresses are the same, so each 10-bit address will be transformed into an exclusive corresponding address. When power is turned off, all bits in memory are lost, so the fact that the counter value will be different when the power is turned back on does not matter.

4. Memory Aliasing

Consider the memory below, which uses two gates to reduce a 6-bit address into 4 bits for use as the address delivered to a 2^4×8-bit memory. Recall that the symbol "∇" on the DATA-OUT lines means that the memory uses tri-state buffers so that these outputs produce bits only for read operations.



Complete the table below with the 4-bit ADDR values delivered to the memory's ADDR (address) input and the 8-bit patterns delivered to the memory's DATA-OUT output in response to the input in each row. If the DATA-OUT outputs are left floating, as shown in the first row—a write operation—write "Z" in the DATA-OUT column. If the values of the DATA-OUT outputs are unknown, write "bits" in the DATA-OUT column, as shown in the second row. If a specific bit pattern appears on the DATA-OUT outputs, write the bit pattern in hexadecimal notation (as seen in the DATA-IN column). The inputs in the rows are applied sequentially to the memory, starting from the top of the table, and each operation (if any) is completed before the operation specified in the next row is started.

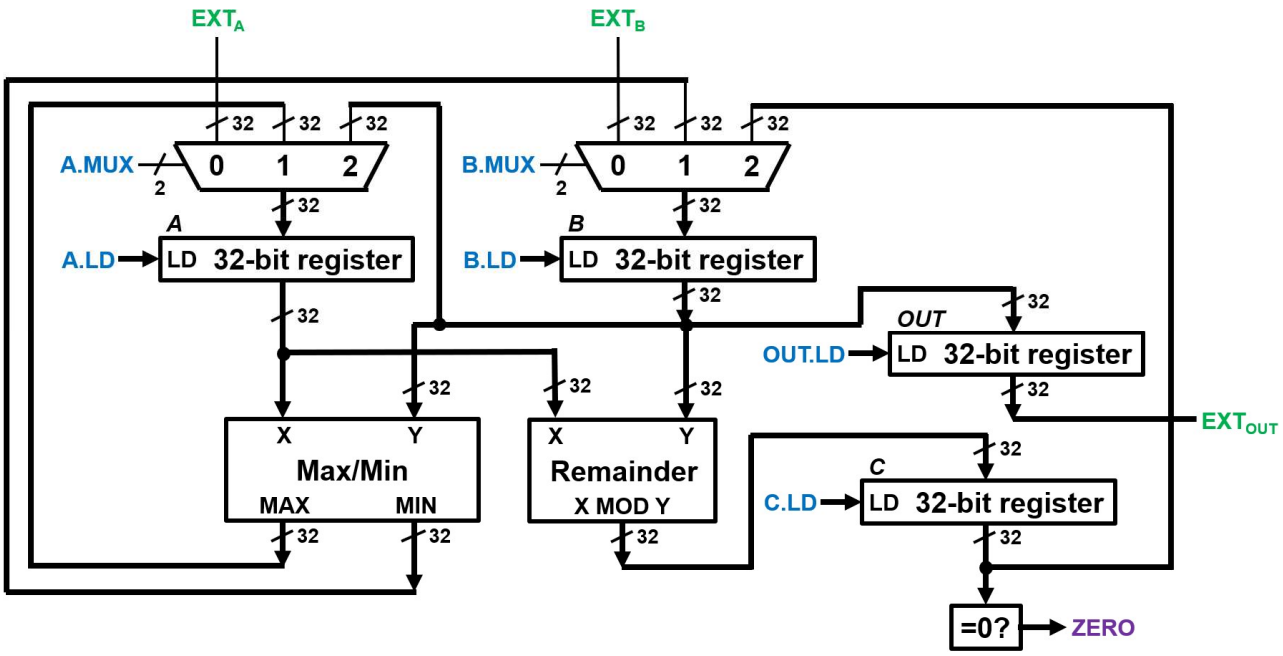
CS	A[5:0]	WE	DATA-IN[7:0]	ADDR	DATA-OUT[7:0]
1	110101	1	xC2	1101	Z (high impedance)
1	000000	0	x99	0000	bits (unknown)
0	010010	1	x3F	0100	Z
1	000100	1	xBB	0000	Z
1	111011	0	x29	1101	xC2
0	000000	0	x4F	0000	Z
1	011110	1	x17	0110	Z
1	001000	0	x11	0100	bits
1	001110	0	x2C	0110	x17

5. Euclid's Algorithm

After reading your TA's problem about Euclid's Algorithm for Midterm 1, Prof. Lumetta got excited and designed an FSM and a datapath to calculate the Greatest Common Denominator of two numbers (the largest integer that divides both numbers evenly). Unfortunately, he misplaced the transition diagram and circuit design, so all he has left is the datapath, a C function implementing the desired operation, and some vague memories about how the FSM worked.

Fortunately, Prof. Lumetta has students who know how to design FSMs!

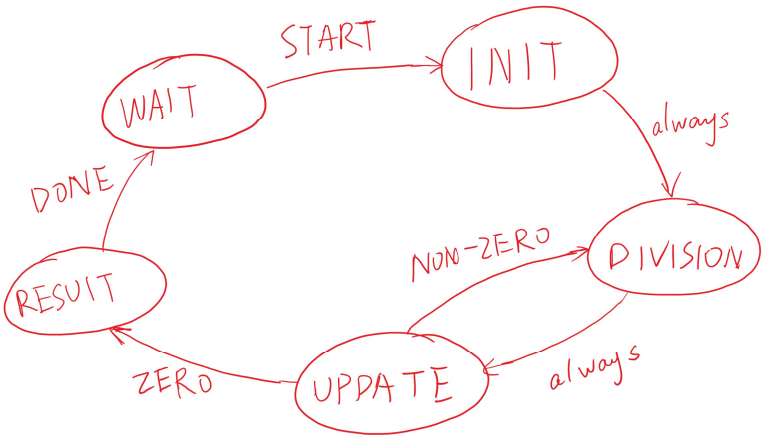
The datapath is shown below. Signals to/from external logic are written in green. Control signals are written in blue. And one status signal (in purple) goes from the datapath to the FSM. Two additional signals are not shown: while providing two unsigned numbers through the EXT_A and EXT_B inputs, a START signal is raised to 1 to begin operation of the FSM, and a DONE signal (also active high) issued by the FSM to tell the external logic that the computation is done, at which point the external logic can read the answer from the EXT_{OUT} output.



(Problem continues on the next page.)

The C function implementing the desired operation is given below. **Read Section 3.7 of the class lecture notes before attempting this problem.**

a. Start by drawing an abstract state transition diagram for the FSM. You need exactly five states, including a WAIT state for the FSM to occupy while the external logic provides inputs and reads outputs. Give your five states reasonably meaningful names. Draw them inside circles, and draw arcs between the states based on the inputs to the FSM (START and ZERO). Keep in mind that your FSM must work with the datapath shown above. You may want to work on **part (b)** at the same time.



b. Draw a table in which each state in your FSM occupies one row. For each state, give the RTL executed by that state in the datapath as well as the next states (with conditions for each possible next state—some states' conditions may be 'always'). See Section 3.7.4 of the notes for an example of such a table.

state	actions	condition	next state
WAIT	A ← EXT_A B ← EXT_B	START $\overline{\text{START}}$	INIT WAIT
INIT	A ← MAX(A,B) B ← MIN(A,B)	(always)	DIVISION
DIVISION	C ← A MOD B	(always)	UPDATE
UPDATE	A ← B B ← C OUT ← B	ZERO $\overline{\text{ZERO}}$	RESULT DIVISION
RESULT	(none)	(always)	WAIT

c. Next, based on your answer to **part (b)**, write a table of control signals (8 bits in total) and the output signal (DONE) for each state.

state	$S_4S_3S_2S_1S_0$	A.MUX	A.LD	B.MUX	B.LD	C.LD	OUT.LD	DONE
WAIT	1 0 0 0 0	00	1	00	1	0	0	0
INIT	0 1 0 0 0	01	1	01	1	0	0	0
DIVISION	0 0 1 0 0	xx	0	xx	0	1	0	0
UPDATE	0 0 0 1 0	10	1	10	1	0	1	0
RESULT	0 0 0 0 1	xx	0	xx	0	0	0	1

d. Finally, use a one-hot encoding—be sure to specify which state uses which bit!—to write equations for all control signals, the output signal, and the next-state logic for your FSM.

state:

$S_4^+ = S_4 \cdot \overline{START} + S_0$
$S_3^+ = S_4 \cdot START$

$S_2^+ = S_3 + S_1 \cdot \overline{ZERO}$
$S_1^+ = S_2$
$S_0^+ = S_1 \cdot ZERO$

control signals:

$A.MUX[1] = S_1$
$A.MUX[0] = S_3$
$A.LD = S_4 + S_3 + S_1$
$B.MUX[1] = S_1$
$B.MUX[0] = S_3$
$B.LD = S_4 + S_3 + S_1$
$C.LD = S_2$
$OUT.LD = S_1$
$DONE = S_0$

```
unsigned int
extOut (unsigned int extA, unsigned int extB)
{
    unsigned int A = extA;
    unsigned int B = extB;
    unsigned int C;
    unsigned int OUT;
    unsigned int swap;
    if (A < B) {
        swap = A;
        A = B;
        B = swap;
    }
    for (C = A % B; 0 != C; C = A % B) {
        A = B;
        B = C;
    }
    OUT = B;
    return OUT;
}
```

6. One from the Textbook

Do Patt & Patel Problem 4.4.

Word length defines the number of bits handled by a computer's processing unit. Larger word length means the processor is able to process data faster, but does not affect the problems that it can solve (regardless of its word size, a computer is a universal computing device). L0arger word length means larger registers and larger operands for the ALU. Extra processing to split big operands, which takes more time, is not needed.