Logisim Bonus Lab

Exercise 1: Multiply

You decide that you want to implement multiply using repeated addition (ex: you want to multiply 5*3, so you could perform 5*3 = 5 + 5 + 5). Since we will be using the output of a cycle in the next cycle (i.e. use some sort of feedback) in this repeated-adder-multiplier, we will need to use a register and will only perform one addition per cycle. Since this multiply circuit will be within our ALU we need to output not only the product of the two numbers, but a signal specifying whether or not the CPU needs to stall while we finish computing the product of the two numbers.

Open MultRepeatedAdder.circ and take a look at the mult circuit. The inputs A and B are expected to be multiplied via repeated addition and their product should be placed in the "Product" output pin. Before the product at the "Product" output pin displays the correct output, the pin "Stall" needs to be 1 so the ALU knows to stall its pipeline.

The approach we are going to take is the following:

To multiply A*B we will repeatedly add A, B # of times, (Ex: 3*4 = 3 + 3 + 3 + 3, we added +3 repeatedly four times)

When the RESET input is 1, we are going to setup our registers to begin performing the multiplication (this means zeroing counters, capturing the values of A and B into registers, etc.)

We will need a counter to know the number of times we have added A. Once this counter == B we are done. This counter will be implemented in the "CountUpTo" sub-circuit. This circuit has the number/limit that we want to count up to as an input and outputs a signal specifying whether or not we have reached that limit yet (note: the counter outputs just a boolean value, not the count itself).

In the mult circuit we will need to:

- Store the values of A and B into registers when the RESET bit is 1 (hint: think about the enable bit of a register). We store the input values because we will need to use those values across multiple cycles and we want to make sure those values don't change during that time.
- We need an adder with A and the current sum from a register as inputs. This register should be connected to the output value (Make sure the register resets at the "reset" signal and holds its value if we are not stalling).

- We will need a means of using the counter subcircuit we made to create a control signal that will determine if we should still be adding.
- Set the output pins "Product" and "Stall Pipeline" to their correct values. Note that you don't need to implement any efficiency tricks; it should always take B cycles for the value to be computed.

To correctly use this circuit to multiply you must set inputs A and B and have the clock ticking in the main circuit. Then poke the reset pin to 1 and leave it for a clock cycle (this should cause the counter to reset and inputs A and B to be saved into registers). Then poke the reset pin back to 0 and watch the output after each cycle be incremented by A.

TIPS: Take time to understand how a register's RESET and ENABLE pins function. When RESET is 1, the register will reset to 0. While the ENABLE input bit is explictly 0, the register will not change it's stored value even if the clock rises.

Checkoff

- *Show your TA your "Mult" and "Counter" subcircuits.*

Exercise 2: Fibonacci Numbers

So far we have built circuits that are either 1) purely combinational and require no clock or 2) have memory but run infinitely. In this exercise, we want to build a circuit that will compute the N^{th} Fibonacci number. As a quick review the Fibonacci numbers are defined by $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$.

1. Start with just the infinite Fibonacci computation. Hopefully this should be pretty intuitive based on previous exercises that you have done before. How many registers do you need? What arithmetic blocks do you need? Where should the output be attached? Make sure you figure out a way to properly initialize the registers for the computation to work.
2. Your sub-circuit will compute up to the 15^{th} Fibonacci number and we will assume our input $N > 0$ is an unsigned number. How many input bits do you need and how many output bits do you need to represent the 15^{th} Fibonacci number?
3. Now to prevent the circuit from computing the Fibonacci numbers to infinity, we will make use of the Enable bit (lower left) on registers. When unset or pulled high, the registers will continue to load their inputs on the rising edge of the CLK. If Enabled is pulled low, then the registers will NOT load their inputs. We need to create a signal that will properly go low when we want the circuit to stop computing Fibonacci numbers.

4. Create an additional part of the circuit that halts the original circuit after N computations. For this exercise you may find the following blocks useful: Counter (Memory), Comparator (Arithmetic), and Probe (Wiring). Make sure that it properly stops on the N^{th} Fibonacci number (watch for off-by-one errors) and that it actually stays there (run it for at least 20 clock cycles).

Your output should be in binary, but you can use a probe to display the value in decimal. Make sure you check the attributes of any counter and comparator you use.

Extension: With proper register initialization, you can get this circuit fib15 to work properly for N > 0. But register initialization is annoying and must be repeated every time you reset your circuit. With a clever addition, you can get this fib15 to work for N >= 0 without the need for register initialization (Hint: it involves a MUX).

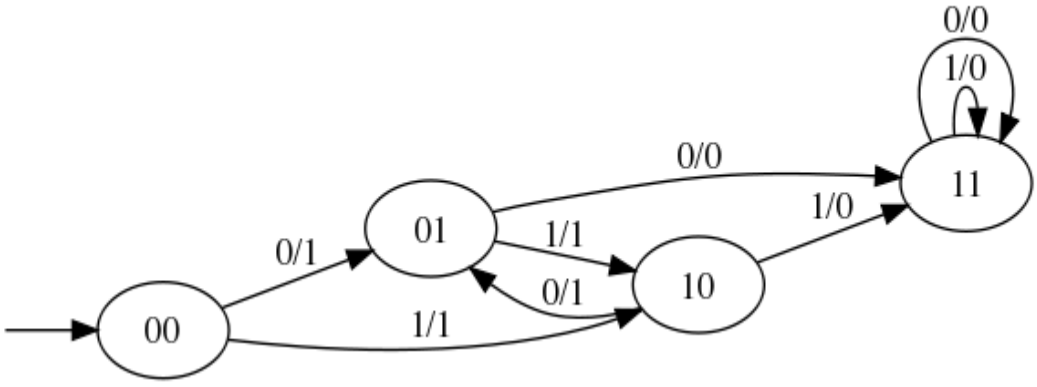
Checkoff: Show your TA your fib15 circuit and verify that for N = 15 after some time it will return Out = 610 indefinitely.

Exercise 3: FSMs to digital logic

Now we're ready to do something really cool; translate an FSM into a digital logic circuit:

If two ones in a row or two zeroes in a row have ever been seen, output zeros forever. Otherwise, output a one.

1. Note that the FSM is implemented by the following diagram:



2. Observe that the following is a truth table for the FSM:

st1	st0	input		next st1	next st0	output
0	0	0	|	0	1	1
0	0	1	|	1	0	1
0	1	0	|	1	1	0
0	1	1	|	1	0	1
1	0	0	|	0	1	1
1	0	1	|	1	1	0
1	1	0	|	1	1	0
1	1	1	|	1	1	0

3. We've provided you with a starter Logisim circuit to start out. Download here.

4. Note that the top level of the circuit looks almost exactly the same as our previous adder circuit, but now there's a FSMLogic block instead of an adder block. FSMLogic is the combinational logic block for this FSM. We have handled the output bit for you, as it's the most complicated to simplify and implement. You should complete the circuit by completing the StateBitOne and StateBitZero subcircuits, which produces the next state bits.

   You **could** go from the truth table to SOP to a circuit, or you could notice that for each state bit, there are only two situations in which it is zero. This could make your life easier if you think a bit outside the box...

Checkoff

- *Show your StateBitZero circuit to your TA and demonstrate that it behaves correctly.*
- *Show your StateBitOne circuit to your TA and demonstrate that it behaves correctly.*