ECE/CS 250 – Prof. Bletsch
Recitation #5
Advanced Logic Design with Logisim Evolution

Objective: In this recitation, you will learn how to design more sophisticated digital logic and use Logisim Evolution for the design and simulation of digital circuits.

Complete as much of this as you can during recitation. If you run out of time, please complete the rest at home.

1. Useful Features in Logisim Evolution

Logisim Evolution has many features that can be very handy. Please experiment with as many of these features now as you can. They will prove useful on your homework!

1) Sub-circuits: If you have not yet created sub-circuits (which was part of the previous recitation), this is an extremely useful feature. Hierarchy is your friend. It'll keep your schematics from becoming unreadable messes.
2) Tunnels: You can "connect" point A to point B with a tunnel. Every tunnel with the *same name* is connected to each other. Effectively, you're saying that these points are wired together, but without having to draw the wire. Tunnels can enable you to keep your schematics much less messy and easier to read.
3) Probes: A probe allows you to observe the value on an internal wire. You can change the format to hex or decimal, too. For debugging, this is very helpful!
4) Flipping gate orientation: To keep your schematics clean, it can be handy to flip the orientation of a gate so that it faces one way instead of another. This can be done with the "facing" attribute, or by using the arrow keys are a shortcut while placing.

Demonstrate to yourself that you've mastered all of the above, and ask for help if you have any questions.

TIP: Don't breeze through this. You'll be spending a lot of time in Logisim – take this chance to get good at it!

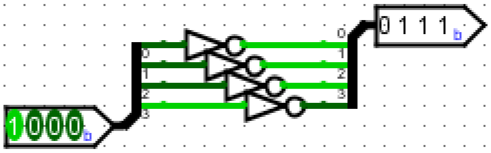
2. Using Buses, Splitters, and Wide Gates

Buses and Splitters: Logisim Evolution has support for "buses", groups of 1-bit wires that are bundled together for convenience (and to make the schematics less messy). Each wire/bus has an attribute that is its width.

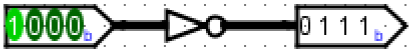
Create an input pin with a width of 4 bits. Create a splitter and connect it to the input pin. The default splitter width is 2 bits, so an orange mismatch condition will appear. Change the "bit width in" and "fan out" of the splitter to 4 to resolve this.

If you look closely, you'll see the output pins of the splitter are numbered. Bit 0 is the least significant bit (right-most when it's written out as a binary number); Bit N is the most significant (left-most when written out).

Now that we've split our 4-bit input into four separate wires, put down four NOT gates for each. Then use another 4-bit splitter to bundle the results together, and hook to a 4-bit output pin. Result:



Wide gates: Boy, it was a pain to lay down and wire each of those NOT gates. Well good news: As with wires, you can specify gates that are wider than 1 bit. Rip up the two splitters and the four single-bit NOT gates and replace them with just one 4-bit NOT gate. Result:



These multi-bit gates will do their operation on each bit separately, so this is precisely equivalent to the prior circuit. During placement of a gate, you can change its number of data bits by holding Alt and typing a number.

3. Design a Small Finite State Machine (FSM)

Design and simulate a sequential circuit with one input (and a clock input) and the following behavior. If the input has been equal to 1 for the three previous cycles, then the output is 1. Otherwise, the output is 0. Make it a Moore machine (meaning the output depends solely on the current state, not on inputs; i.e., output can be written on states in the state transition diagram).

Please use the systematic methodology for developing the FSM:

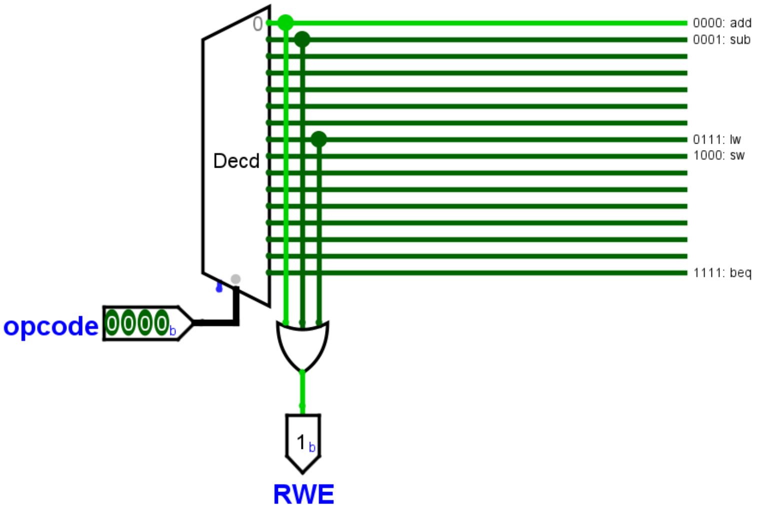
1. Write a state transition diagram (bubbles and arrows).
   ^ *Have a TA check this over before proceeding.*
2. Convert it to a truth table.
3. Convert the truth table to expressions (simplifying is optional).
4. Implement the logic circuit, then plumb in D Flip-Flops for the D’s and Q’s.
5. Simulate your circuit to test that it behaves as expected.

4. Build a (simplistic) Instruction Decoder

For your next homework, you'll be building a processor in Logisim Evolution. One aspect of processor design is building an instruction decoder that takes an N-bit opcode (bits $x_N \ldots x_0$) and generates signals to control the processor's datapath. For this recitation, assume that you have 4 control signals: RWE (1-bit), ALUop (2-bit), DWE (1-bit), and MuxA (1-bit). Assume that there are the following instructions with their opcodes in parentheses: add (0000), sub (0001), lw (0111), sw (1000), and beq (1111). Implement logic to decode the instructions such that each one produces the signals as specified in the table below. Remember: your circuit has 4 bits of input and 5 bits of output.

	RWE	ALUop	DWE	MuxA
add	1	00	0	0
sub	1	10	0	0
lw	1	00	0	1
sw	0	00	1	1
beq	0	11	0	0

Hint: You could do this with the regular truth table approach we've learned, but try using a *decoder* component to turn the opcode into one-hot representation (see the slide "Control Logic using a Decoder (one-hot representation)" from the "Datapath and Control" lecture). An example of how you might set up this approach is shown below, with RWE computed for you. Text labels like those shown at the right can help you keep track of things.



<u>ALL DONE?</u>

Nice! Don't head out, though. Work on the current homework and talk to the TAs for help. You can leave if your homework tester shows all passing and you've turned in all the written & code materials.

5. Bonus exercise: Working with a Clock and Keyboard and TTY Display

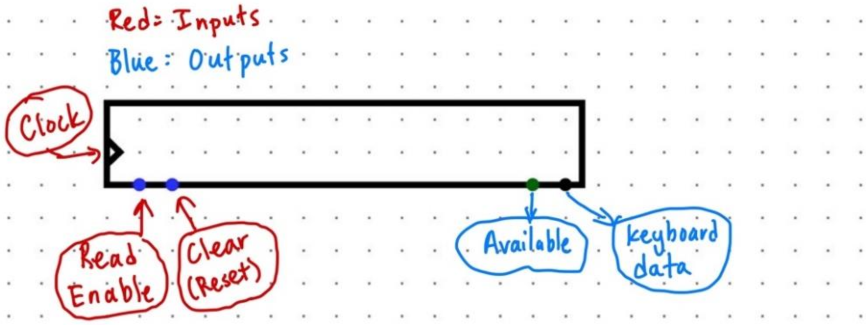
(Note: this exercise is presented in case you want to practice using these components, but it's not expected that you reach it during recitation)

Logisim has components to simulate IO devices; let's practice them. Make a circuit that uses a **clock**, **keyboard**, and **TTY** display component to take user input and display the letters typed on the TTY display. The keyboard should be falling-edge triggered and the TTY display should be rising-edge triggered. The TTY display should have 13 rows and 80 columns. You can change the properties of each component by clicking on them and selecting the properties on the bottom-left side of Logisim. For some of the inputs to the keyboard and/or TTY display, you will need to connect an input pin.

Once you have built the circuit, you can test to see if input you enter in the keyboard shows up on the TTY Display.

1. Reset the simulation (Simulate menu -> Reset simulation, or CMD+R/Ctrl+R)
2. Use the poke tool 👆 to click on the keyboard component
3. Type letters into the keyboard; they will fill the keyboard's *buffer* (a queue of keys to output as 7-bit signals, one character at a time)
4. Poke the clock and watch the data from the keyboard disappear and reappear on the TTY display

Here are the inputs and outputs of the Keyboard:



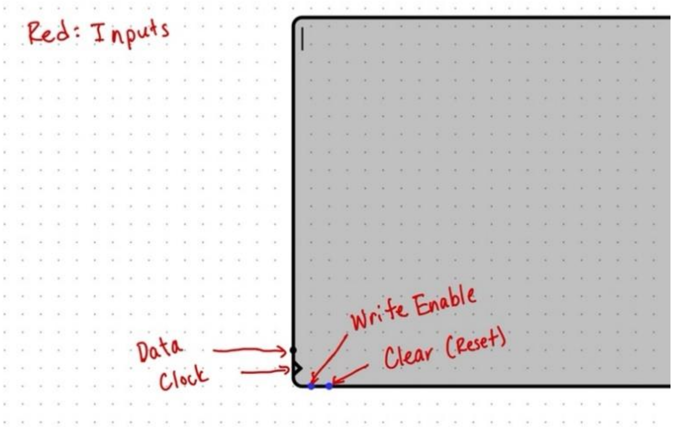
Inputs:

- **Clock**: 1 bit input – connected to the clock
- **Read Enable**: 1 bit input – tells the keyboard to read a letter from the keyboard
- **Clear**: 1 bit input – clears the keyboard

Outputs:

- **Available**: 1 bit output – ON when there is data in the keyboard
- **Keyboard Data**: 7 bit output – Contains the ascii value of the letter read from the keyboard

Here are the input and outputs of the TTY Display:



Inputs:

- **Clock**: 1 bit input – connected to the clock
- **Read Enable**: 1 bit input – tells the keyboard to write a letter to TTY Display
- **Clear**: 1 bit input – clears the TTY Display
- **Data**: 7 bit input – Contains the ascii value of the letter to be written to the TTY Display

An example solution will be shown in Recitation and is linked on the course site.