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. Design a Small Sequential Circuit

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. Use D flip-flops from the Logisim Evolution library. You must simulate your circuit to test that it behaves as expected.

Please use the systematic methodology for developing the FSM. Start with a state transition diagram, write out the truth table, then implement it. 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).

2. 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 could prove useful on your homeworks.

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. Effectively, you're saying that these two 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. 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.

3. 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 a bus with a width of 16. Connect it to an input pin, and set the width of the pin to be 16 bits, so they match. Logisim Evolution will show you width mis-matches if you have any. To then use the bus, it is often helpful to split it later. Use a splitter to pick out all 16 wires of the bus and run each one through a 1-bit wide NOT gate. Then take the outputs of the NOT gates and bundle them together to create a 16-bit bus that you connect to a 16-wide output pin.

Wide gates: As with wires, you can specify gates that are wider than 1 bit. For example, you can put in a NOT gate and then change its attributes to change its width. If it has a width of 4-bits, say, then all 4 bits into it get inverted, and the output of the NOT gate is a 4-bit bus. During placement of a gate, you can change its number of data bits by holding Alt and entering a number.

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 (0001), sub (0011), lw (1001), sw (1011), 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.

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).

	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