Lab 5: Logisim

Deadline: Friday, October 8, 04:00:00 PM PT

This lab introduces Logisim - an educational tool for designing and simulating digital logic circuits. It is critical prep for Project 3. Hence, it is also slightly longer than the rest of the labs you've seen so far. **We strongly recommend starting early!**

Goals

- Learn how to design and debug basic digital logic circuits in Logisim
- Design FSMs and implement them as digital logic
- Gain more experience designing and debugging circuits with both combinational logic and state elements

---

Setup

In your `labs` directory, pull the files for this lab with:

```
$ git pull starter main
```

Please note that it may take longer to pull this set of starter updates, since Logisim Evolution, a circuit simulator, is included.

All the work in this lab will be done using **Logisim Evolution**.

Some important warnings before you begin:

- Logisim is a GUI program, so it can't easily be used in a headless environment (WSL, SSH, etc.). We recommend running it in a **local environment** with a GUI, Java 9+, and Python 3.6+. If your local system is macOS or Linux you're probably all set. If you're on Windows, use `Git Bash`, which runs on Windows with GUI support.
- Please use the version of Logisim that we distribute, since it is different from other versions on the internet (bugfixes and course-specific stuff)

- Don't move the staff-provided input/output pins; your circuit can't be tested properly if the pins move. If your circuit doesn't pass the tests and you think it is correct, check that your circuit fits in the corresponding harness in `tests/ex#_test.circ`
- Logisim doesn't auto-save your work. Remember to save (and commit) frequently as you work!

Possible Setup Errors

- "Permission Denied" trying to run Python on Windows 10
  1. Search for "app execution alias" in the Windows search prompt. Click on "Manage app execution aliases" which is a part of "System Setting"
  2. Disable the store versions of Python by toggling the corresponding executables off.

     ▶ Screenshot

- `java.lang.UnsupportedClassVersionError`

  ```
  Exception in thread "main" java.lang.UnsupportedClassVersionError: com/cburch/logisim/Ma
  ```

  You need to update to Java 9+. Follow the setup instructions from Lab 0.

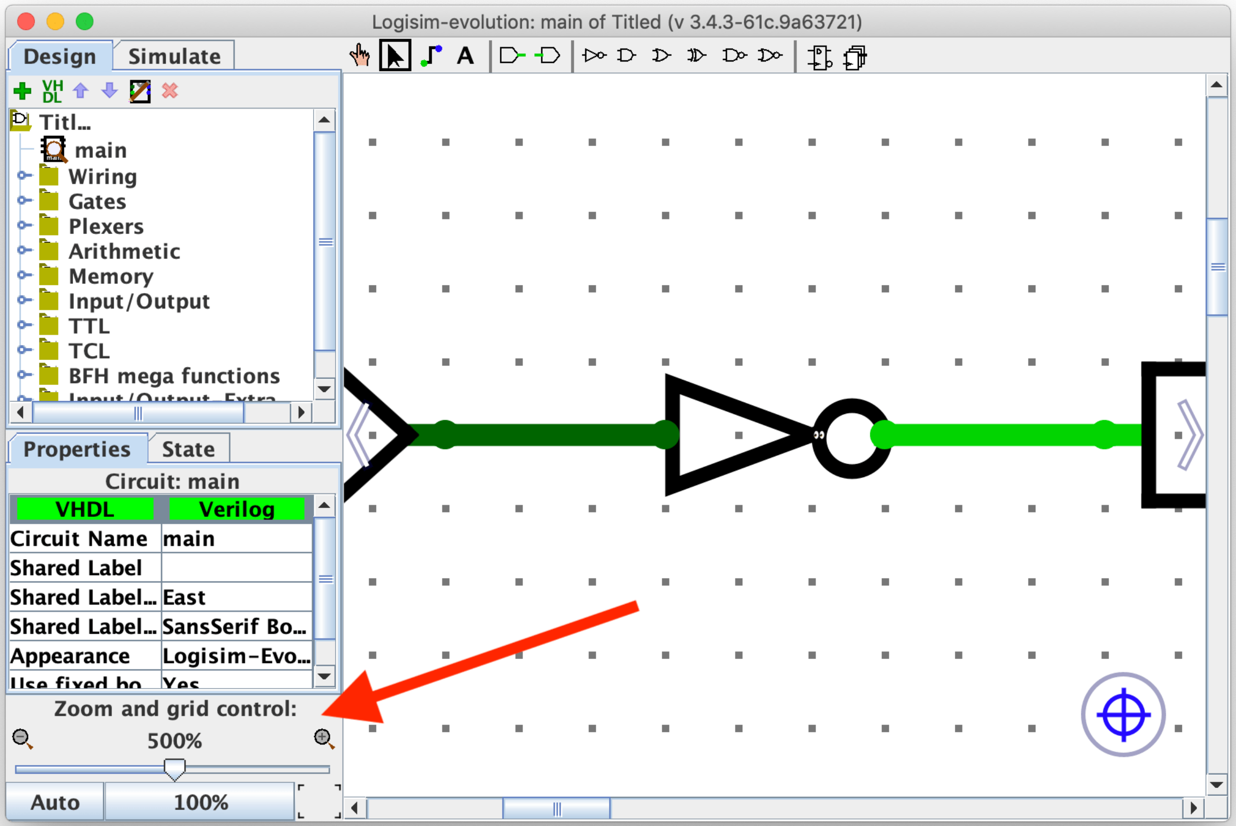
Exercise 1: Introduction

Like Venus, you can run Logisim from inside the `lab05` folder with,

```
$ java -jar ../tools/logisim-evolution.jar # If in a different folder, use the corresponding
```

After a short startup sequence, a slightly ancient-looking window should appear. If not, check for errors in your terminal.

We'll begin by creating a very simple circuit just to get the feel for placing gates and wires. Before you start, take note of a **useful feature: the zoom function!** It's in the bottom left corner, and will make your life much easier for the next few weeks. Zoom Zoom Zoom.

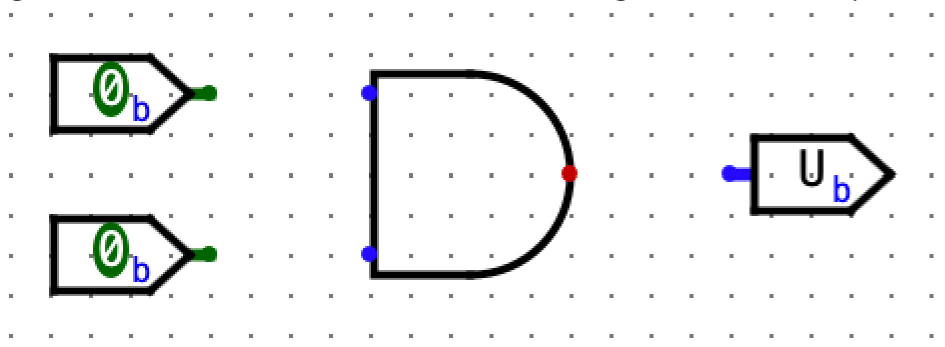


Circuit Name	main
Shared Label	
Shared Label...	East
Shared Label...	SansSerif Bo...
Appearance	Logisim-Evo...
Use fixed bo...	Yes

Building Our Circuit

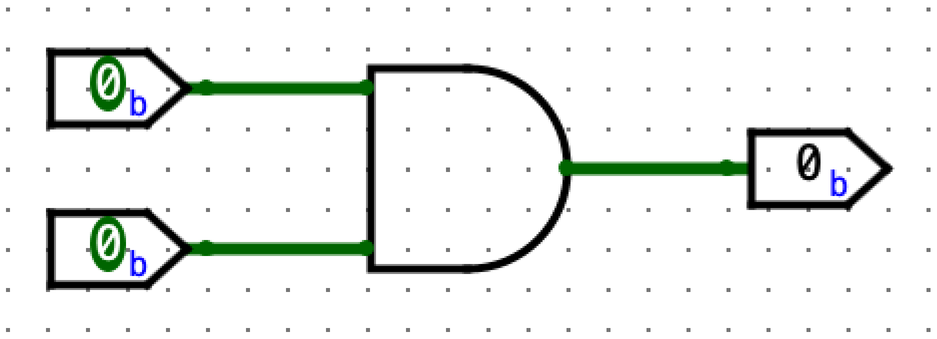
Companion video

1. Start by clicking the `AND` gate button. This will cause the shadow of an `AND` gate to follow your cursor around. Click once within the main schematic window to place an `AND` gate.
2. Click the `Input Pin` button. Now, place two input pins somewhere to the left of your `AND` gate.
3. Click the `Output Pin` button. Then place an output pin somewhere to the right of your `AND` gate. Your schematic should look something like this at this point:

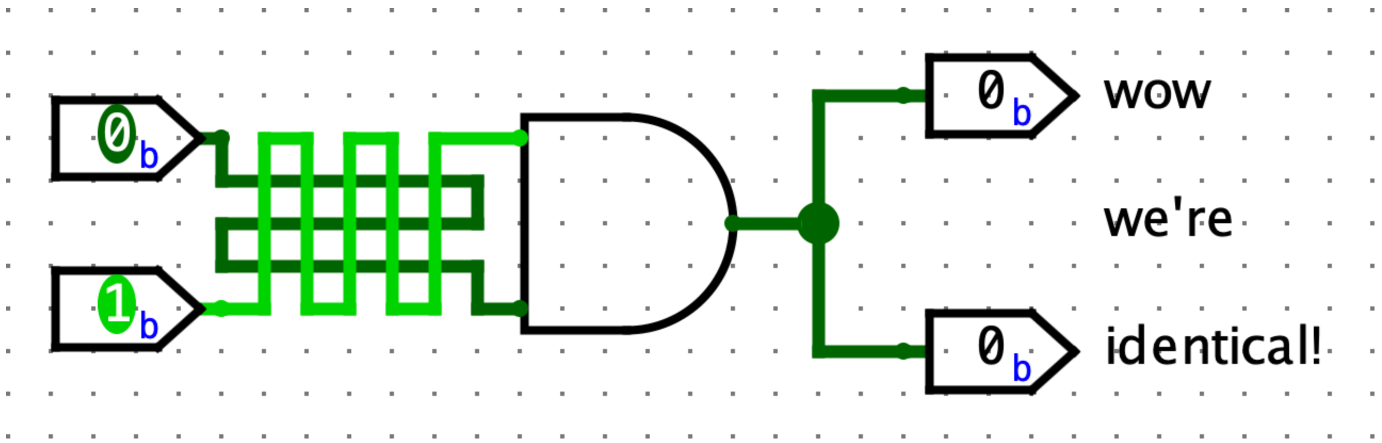


4. Click the `Select tool` button. Click and drag to connect the 2 input pins to the 2 inputs on the left side of the `AND` gate. This will take several steps, as you can only draw vertical and

horizontal wires. Just draw a wire horizontally, release the mouse button, then click and drag starting from the end of the wire to continue vertically. Repeat the same procedure to connect the output on the right side of the `AND` Gate to the output pin. After completing these steps your schematic should look similar to this:



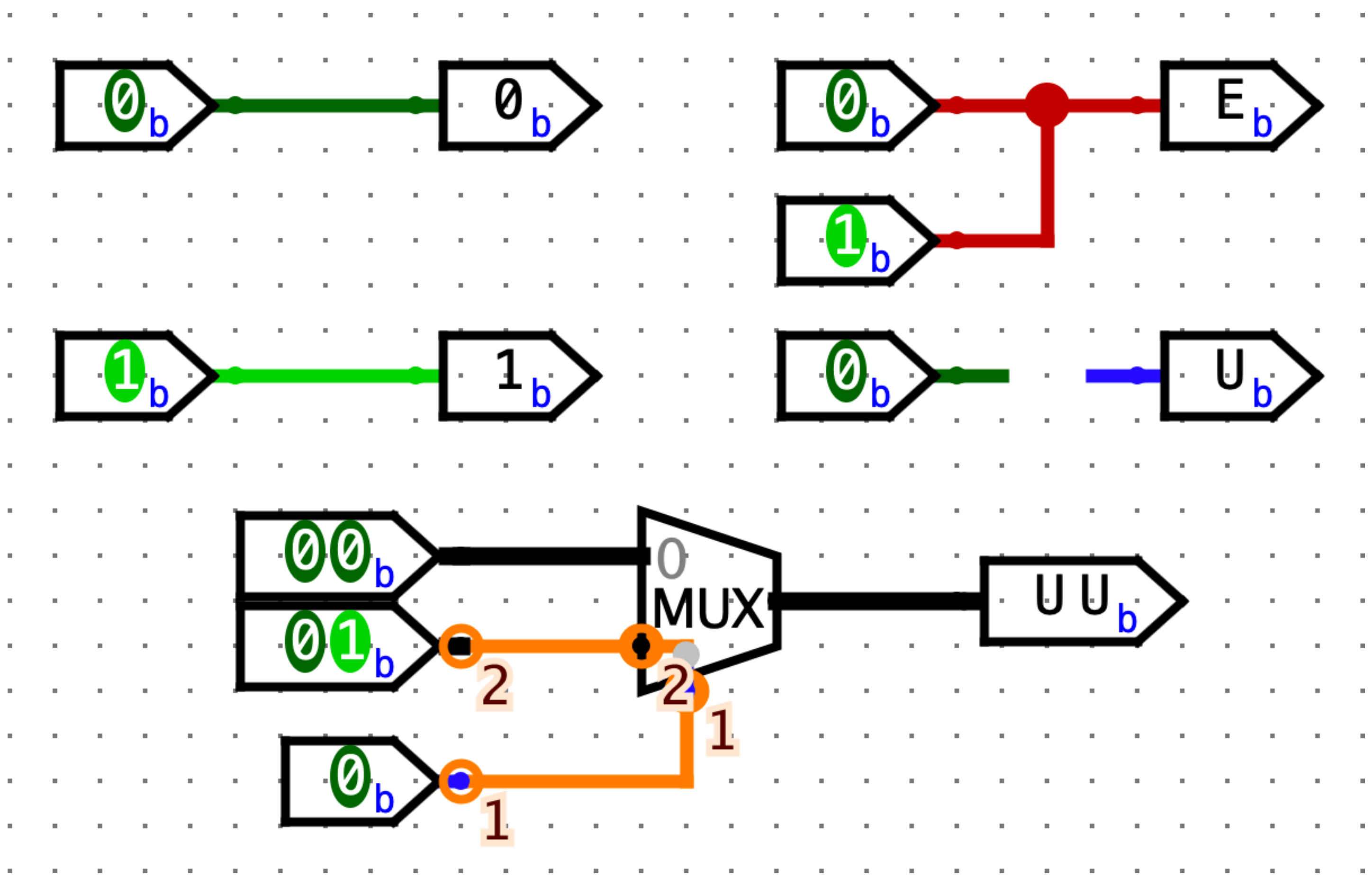
5. Finally, click the `Poke` tool and try clicking on the input pins in your schematic. Observe what happens. Does the output match with what you think an `AND` Gate should do? Now, try poking a wire directly. The current value on that wire should pop up; this will be very useful later when you build more complex circuits.
6. Now, delete the wires, and try wiring each input pin to the *other* pin on the `AND` gate, in such a way that the wires cross over. An extreme example:



If you're creating a wire and drag it **over** another wire without stopping, the wires won't connect. If you're creating a wire and stop dragging while **on top** of another wire, a junction (big circle) will be created, and the wires will connect. Make sure to pay attention to junctions when you're designing your circuits!

List of Wire Colors and Values

Please take a look at this list. It may help to try re-creating each color on your own.



Color	Meaning
Dark green	1-bit wire has a value of 0
Bright green	1-bit wire has a value of 1
Black	Multi-bit wire (many components have bit width attributes which can be configured in the attributes menu on the bottom left)
Red (values with `EEEE`)	The wire has multiple values on it (in this case, a `0` and `1` from the 2 inputs). Also, remember that a big circle appears at wire junctions.
Blue (values with `UUUU`)	The wire is floating (i.e. has no known value)
Orange	The wire is connected to components that have different bit widths. A simple example is a 1-bit input pin connected to a 2-bit output pin. In the shown example, there's a slightly hidden wire behind the `MUX` connecting the 2-bit lower data line to the 1-bit select line. Watch out for these!

Accessibility Note: In case you have a disability and find it difficult to identify these colors, please feel free to use a more convenient set of colors. To do this, go to `File -> Preferences -> Simulation Tab`. Please inform course staff of this change when you show circuits during OH or on Piazza as we often use colors as a debugging tool.

Exercise 2: Sub-Circuits

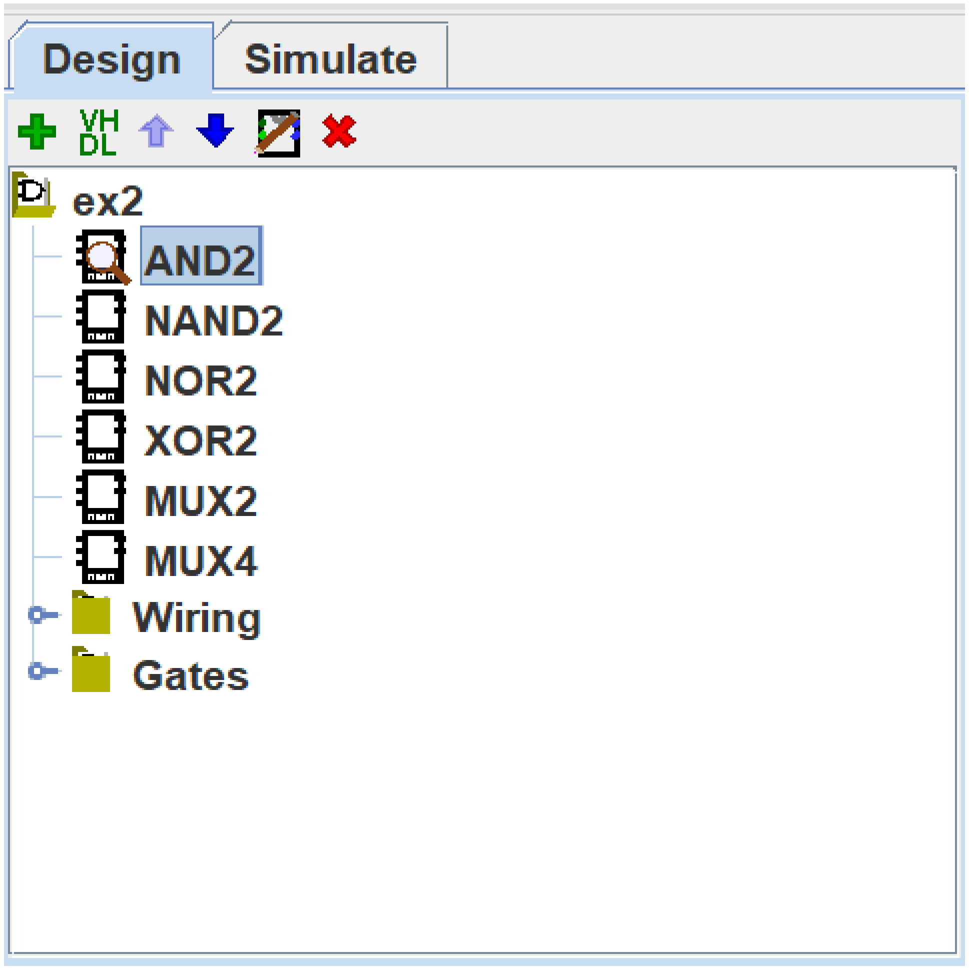
Just as C programs can contain helper functions, a schematic can contain helper subcircuits. In this part of the lab, we will create several subcircuits to demonstrate their use.

Note: Logisim Evolution generally doesn't permit names with spaces or symbols, names starting with numbers, or names that conflict with keywords (e.g. `NAND`).

Action Item

Follow the steps below. Remember to **save often**, and **avoid moving or editing the provided input/output pins**.

1. Open up the Exercise 2 schematic (`File -> Open -> lab05/ex2.circ`).
2. Open up the `AND2` sample subcircuit by double-clicking `AND2` in the circuit selector on the left side.



Note the `2` at the end; because there is a component called `AND`, we can't call it `AND`. We've created a demo circuit for your reference. It has 2 1-bit input pins, `A` and `B`, and sends the result of `A & B` to the `RESULT` output pin. This should look very similar to the practice circuit you just made.

3. Now, open up the `NAND2` subcircuit. It's time to make your own circuit! Fill in this circuit **without** using the built-in `NAND` gate from the Gates library on the left (i.e. only use the `AND`, `OR`, and `NOT` gates; they're available as little icons in the toolbar at the top of the window, or in the Gates library in the circuit selector). When you're done, similarly fill in the `NOR2`, `XOR2`, `MUX2` (2-to-1 MUX), and `MUX4` (4-to-1 MUX).
   - Please do not change the names of the subcircuits or create new ones, or your circuit may not work properly
   - Don't use any built-in gates other than `AND`, `OR`, and `NOT`. However, once you've built a subcircuit, you may (and are encouraged to) **use it to build others**. You can do this by single-clicking a subcircuit in the circuit selector, and placing it like you did for the `AND`/`NOT`/`OR` gates
   - It helps to write out a truth table for each circuit. You might also find it useful to review the lecture slides on how to build these gates
   - For the `4-to-1 MUX`, `SEL0` and `SEL1` correspond to the 0th and 1st bits of the 2-bit selector, respectively. Make sure not to switch them!

Testing

Open a terminal session and go to the `lab05` folder. We've provided tests for each exercise, which you can run with `python3 test.py`. For each test, your circuit is run in a test harness circuit (`tests/ex2-test.circ`), and its output is compared against the reference output for that test (`tests/reference-output/ex2-test.out`). In the output file, each column corresponds to an input/output pin on the main circuit, and each row shows a set of inputs and the corresponding outputs the circuit produced. If your circuit output (`tests/student-output/ex2-test.out`) is different, you can check it against the reference output file; the `diff` command may help.

- Note that the output files are "formatted" by adding tab characters (`\t`) between each value, and they look best when tabs are displayed as 8 spaces wide. Values and column headers that are 8+ characters might mess up the alignment, so watch out for those!
- Don't modify the reference output files, or local tests might produce incorrect results
- You shouldn't need to edit the test harness circuits for this lab. However, it might be useful to take a look; we'll be doing similar things in the next lab and project 3!

Checkpoint

At this point, make sure that you are comfortable with the Logisim environment, creating sub-circuits, and re-using such circuits in other circuits.

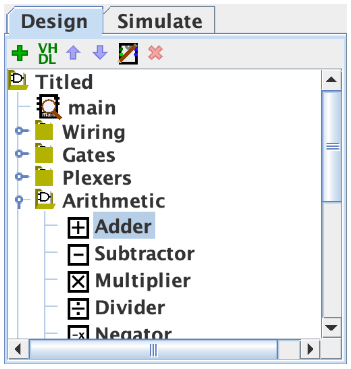
Exercise 3: Storing State

Let's implement a circuit that increments a value repeatedly. The difference between this circuit and the circuits you've built for lab so far is that it will **store** this value as a **state** in a **register**.

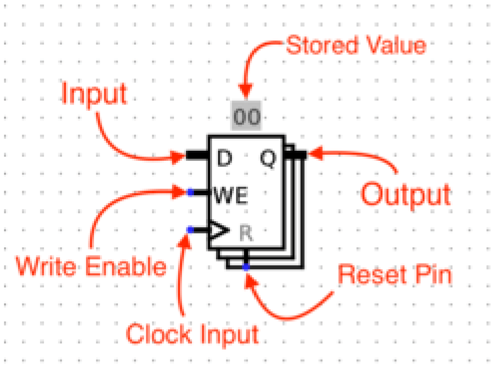
Action Item

Follow the steps below. Remember to **save often**, and **avoid moving or editing the provided input/output pins**.

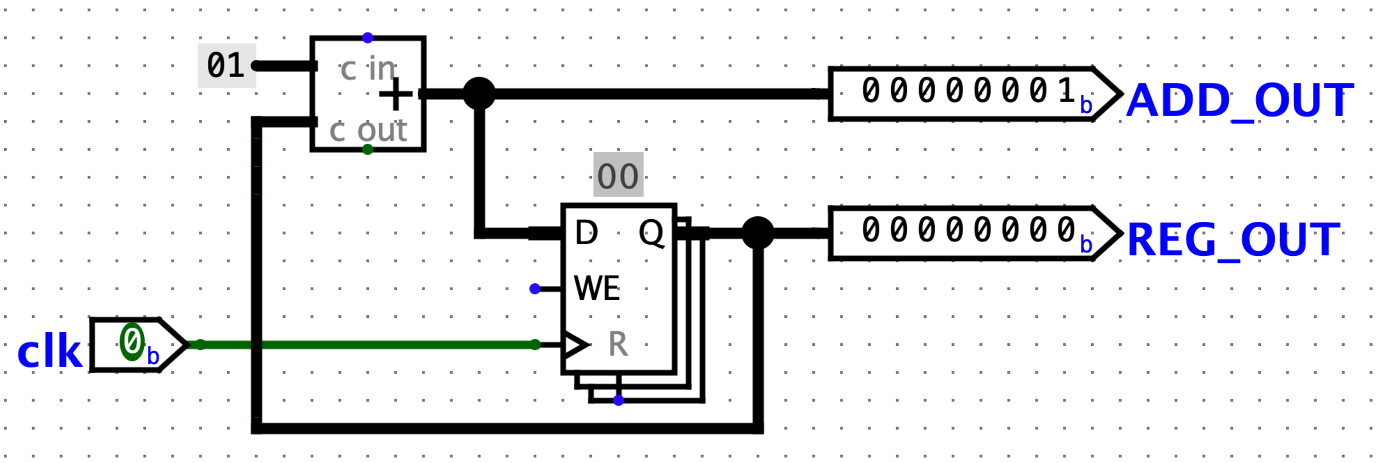
1. Open up the Exercise 3 schematic (`File -> Open -> lab05/ex3.circ`) and go to the empty `AddMachine` circuit.
2. Select the `Adder` subcircuit from the `Arithmetic` library (circuit selector on the left side) and place the adder into your `AddMachine` subcircuit.



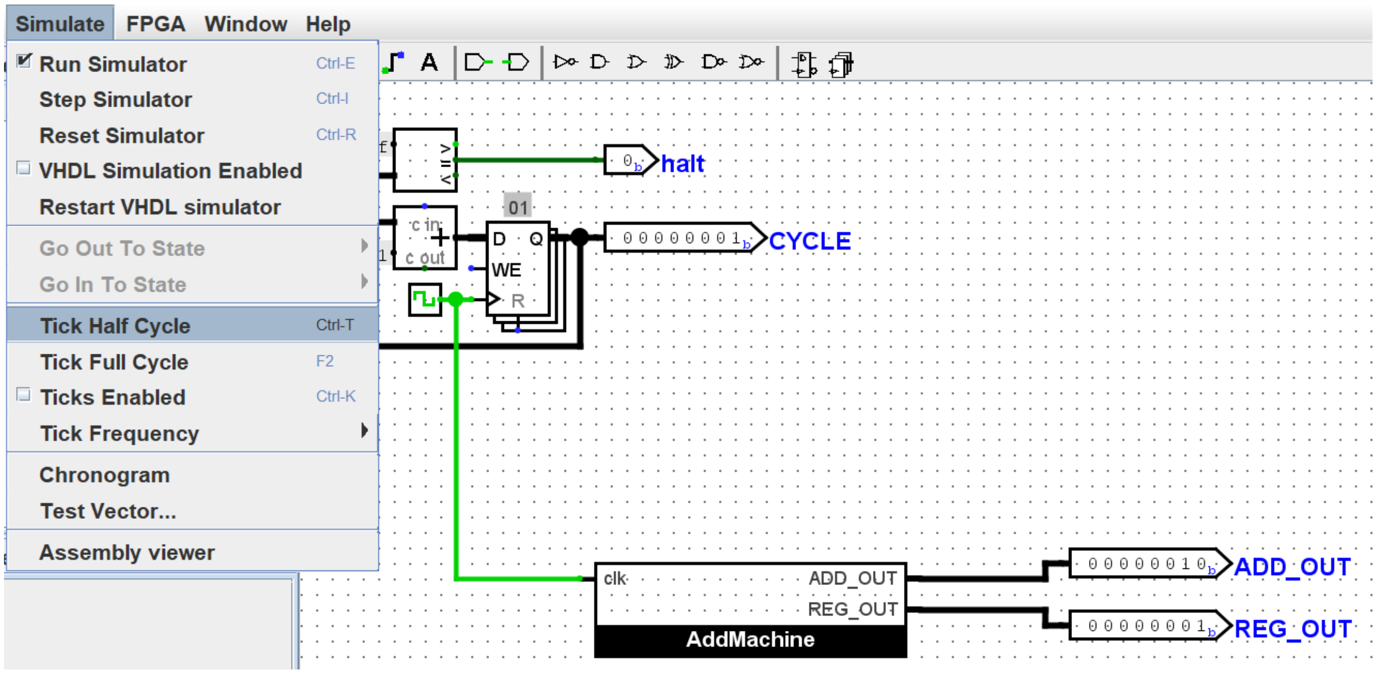
3. Select the Register from the Memory library and place one register into your subcircuit. Below is an image illustrating the parts of a register.



4. Wire up the clk input pin to the clock pin on your register. It's generally better to have all components in a circuit use the same clock so that everything is in sync. In this case, the testing harness uses a clock for its register, so it passes that clock signal in through the clk pin here for your circuit's registers. In the future, if you're working on a circuit that doesn't have an existing clock signal, feel free to make your own by using a new Clock from the Wiring library.
5. Connect the output of the adder to the input of the register, and the output of the register to the input of the adder.
   - You may get an "Incompatible widths" error when you try to connect components. This means that your wire is trying to connect two pins with different bit widths. If you click on a component with the Selection tool (mouse pointer icon on the toolbar at the top), you will notice that there is a Data Bits property in the bottom left field of the window. This value determines the number of bits the component's input and output have. Make sure that both the adder and register have a data bit width of 8, and the "Incompatible widths" error should be resolved.
6. Wire an 8-bit constant with a value of 1 to the second input of the adder. You can find the Constant circuit element in the Wiring library. To change it's value to 1, simply type 1 for the Value property and hit Enter. You should now see the value as 0x1 (Logisim automatically converts the decimal value you enter to hex).
7. Connect the two output pins to your circuit so that you may monitor what comes out of the adder and the register. The output of the adder should be connected to ADD_OUT and the output of the register to REG_OUT. Thus, by the end, your circuit should look similar to:

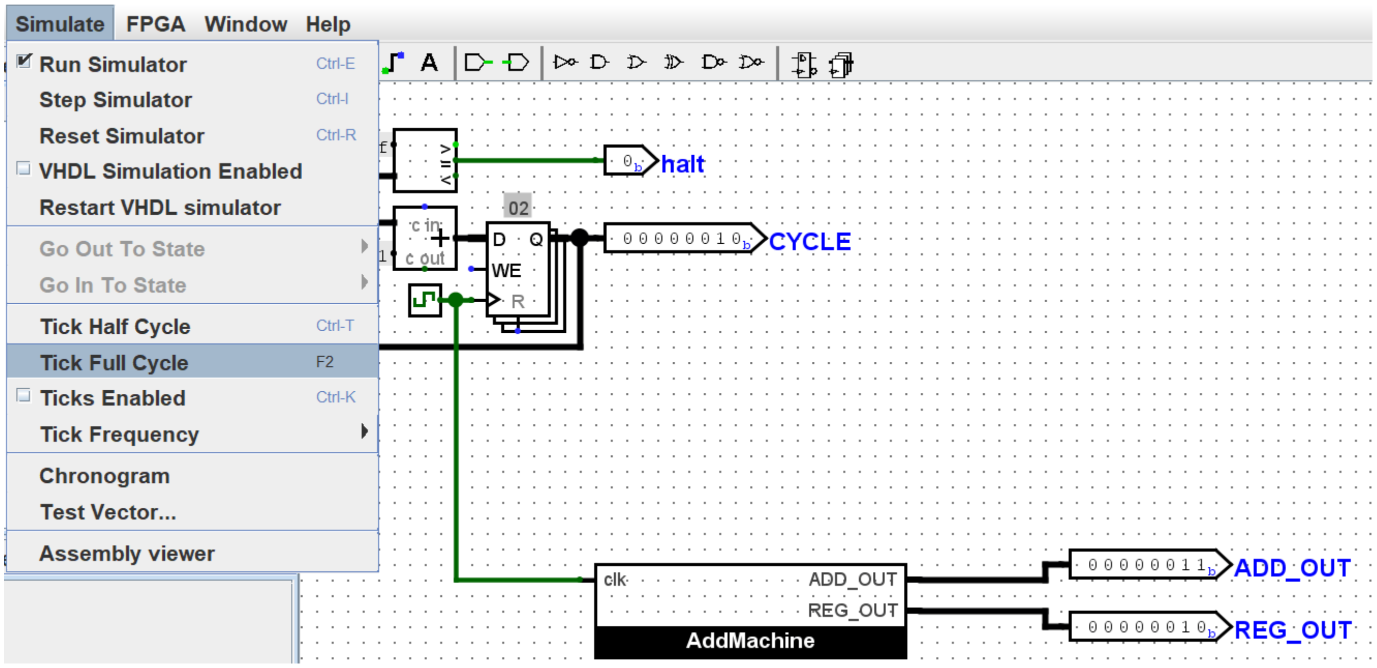


8. Now, open up the test circuit for this exercise (`lab05/tests/ex3-test.circ`). At the top left, we have a small circuit (like your `AddMachine`) keeping track of the current cycle. Below that, you should see your `AddMachine` circuit hooked up to the clock and some output pins.
9. Tick your circuit once by going to `Simulate -> Tick Half Cycle` (`Command/Control + T`). The outputs from your `AddMachine` should have gone up! Additionally, the clock should now be bright green.

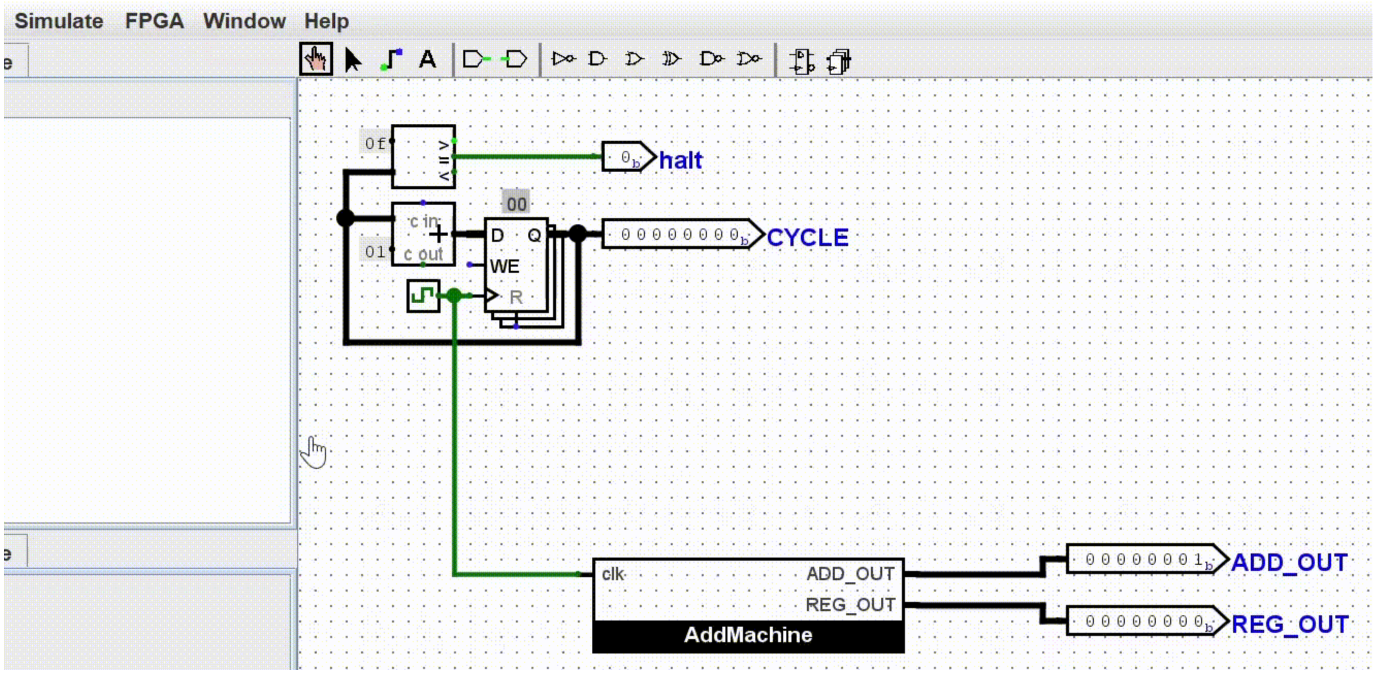


If you `Tick Half Cycle` again, you'll notice that the clock goes back to dark green, but the numbers don't change. This is because the registers, by default, are rising-edge triggered (i.e. triggered when going from a low clock signal to a high signal).

If you're not particularly interested in seeing changes for half cycles, you can also use `Simulate -> Tick Full Cycle` (`F2`).



10. Ticking the circuit manually can be pretty tiring. Good news: Logisim can tick your circuit for you! Go to `Simulate -> Ticks Enabled/Enable Clock Ticks` (`Command/Control + K`). Your circuit should now be counting up, hands-free!



If you want to run your circuit faster, you can change the tick frequency in `Simulate -> Tick Frequency`. To stop, click `Ticks Enabled//Enable Clock Ticks` (`Command/Control + K`) again. If you want to reset the circuit, `Simulate -> Reset Simulation` (`Command/Control + R`).

11. When you're done, try running the provided tests ([using `python3 test.py`]).

Checkpoint

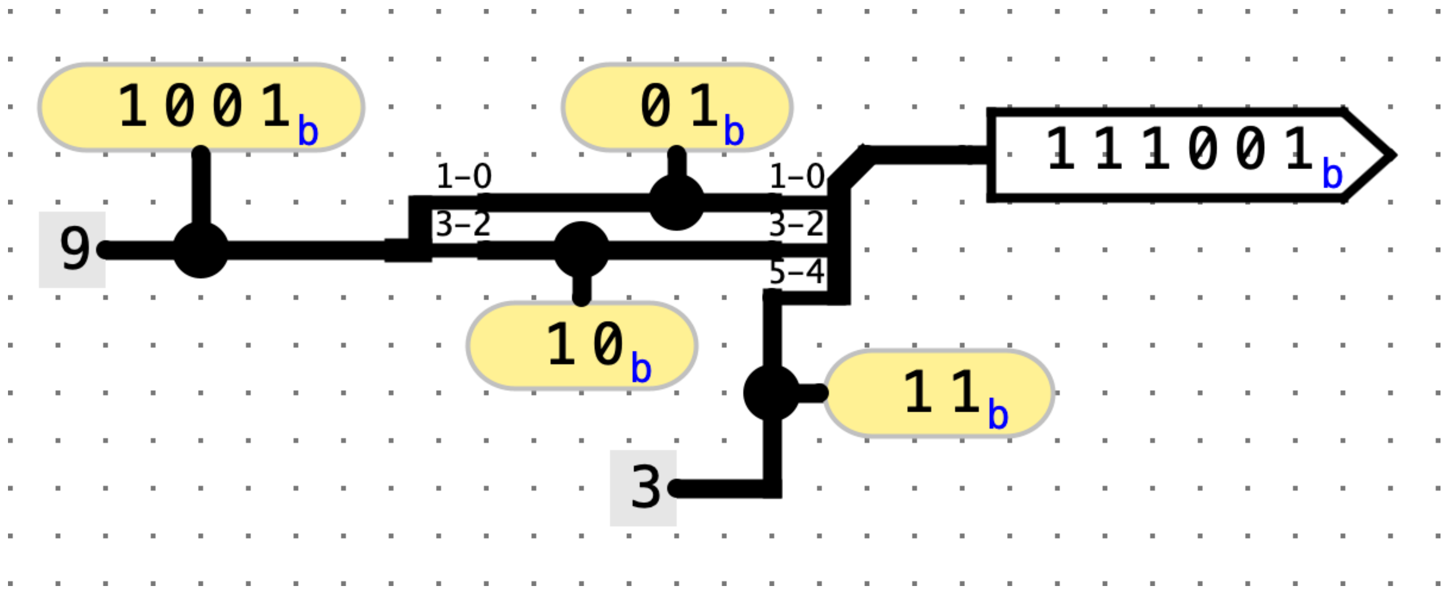
At this point, make sure that you are comfortable with designing and simulating simple digital logic circuits in Logisim environment that use a mix of *combinational logic* and *state elements* (registers).

Advanced Logisim Features

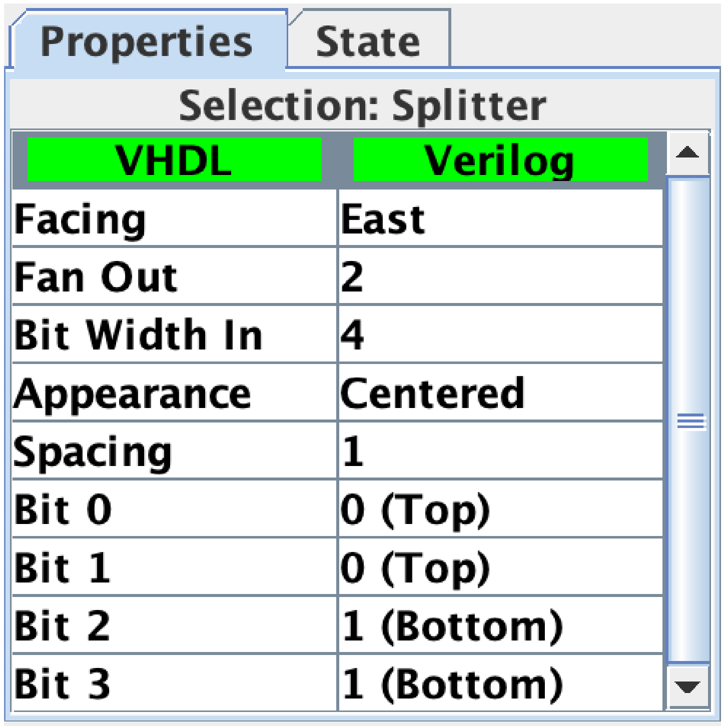
Here are three Logisim features that should both save you a lot of time and make your circuits look much cleaner.

Splitters

Splitters allow you to take a multi-bit value and split it up into smaller parts, or (despite the name) combine multiple values that are one or more bits into a single value. Here, we split the 4-bit binary number `1001` into `10` and `01`, then recombine it with `11` into the final 6-bit number `111001`:

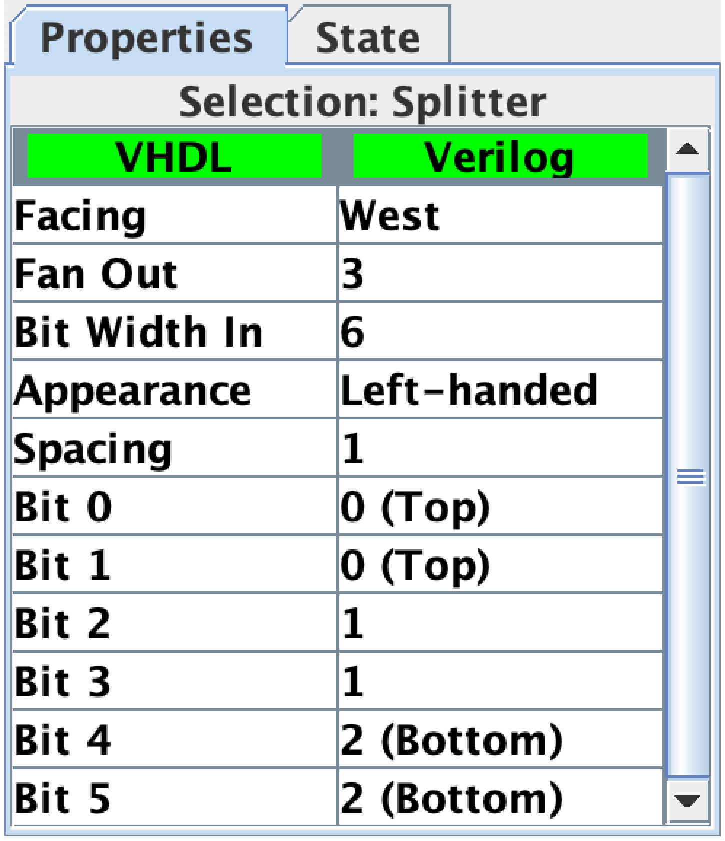


Click on a splitter (using the `Select` tool) to get its attributes in the sidebar (lower left). You can configure attributes like the number of arms on your splitter and the bits present on each arm. For the circuit above, the left and right splitters' attributes looks like this:


Properties | State

Selection: Splitter

VHDL	Verilog
Facing	East
Fan Out	2
Bit Width In	4
Appearance	Centered
Spacing	1
Bit 0	0 (Top)
Bit 1	0 (Top)
Bit 2	1 (Bottom)
Bit 3	1 (Bottom)

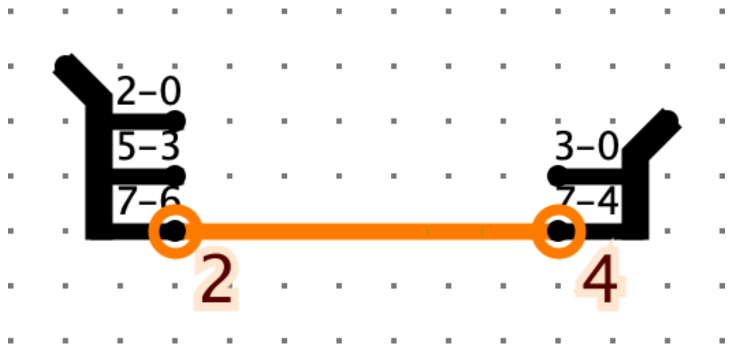

Properties | State

Selection: Splitter

VHDL	Verilog
Facing	West
Fan Out	3
Bit Width In	6
Appearance	Left-handed
Spacing	1
Bit 0	0 (Top)
Bit 1	0 (Top)
Bit 2	1
Bit 3	1
Bit 4	2 (Bottom)
Bit 5	2 (Bottom)

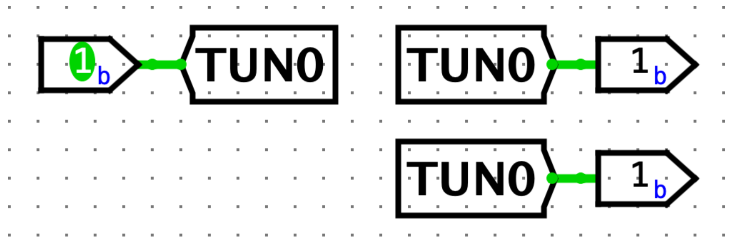
Notice that there's an attribute called `Facing`. You can use this to rotate your splitter. In the above example, the splitter on the right is facing `West` while the splitter on the left is facing `East`.

If you see an error wire that is orange, this means that your bit width in does not match your bit width out. Make sure that if you're connecting two components with a wire, you correctly set the bit width in that component's menu.

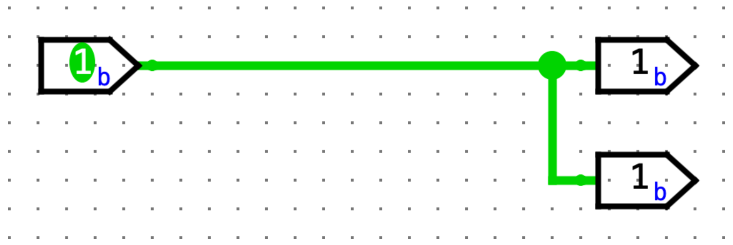


Tunnels

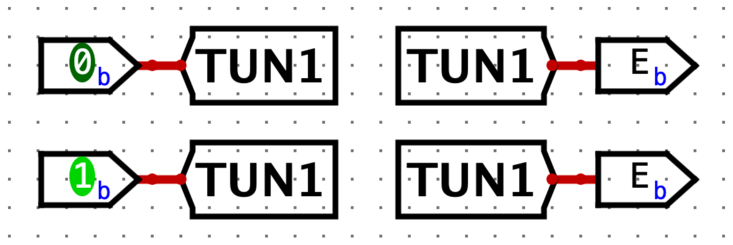
A tunnel allows you draw an "invisible wire" to bind two points together. Tunnels are grouped by case-sensitive labels give to a wire. They are used to connect wires like so:



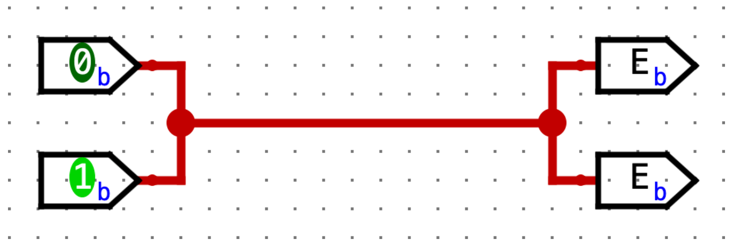
which effectively is:



Some care should be taken as to which wires are connected with tunnels to which other wires, such as in this case:



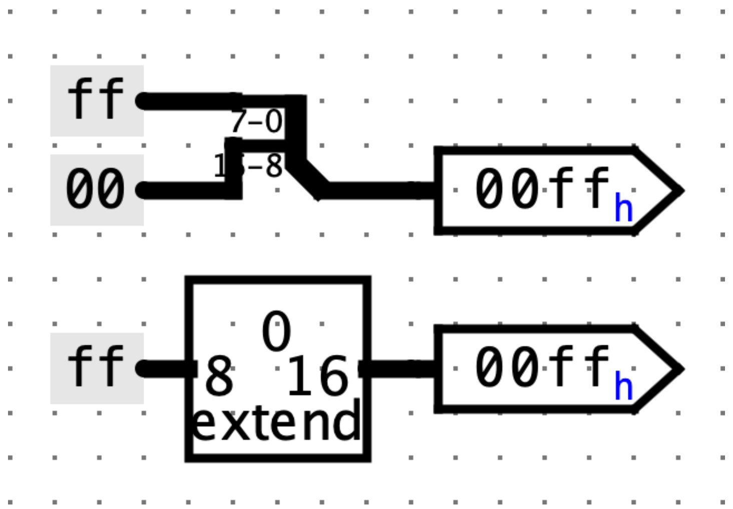
which effectively is:



We *strongly* recommend you use tunnels with Logisim, because they make your circuits look much cleaner (less wiring spaghetti), which means easier debugging.

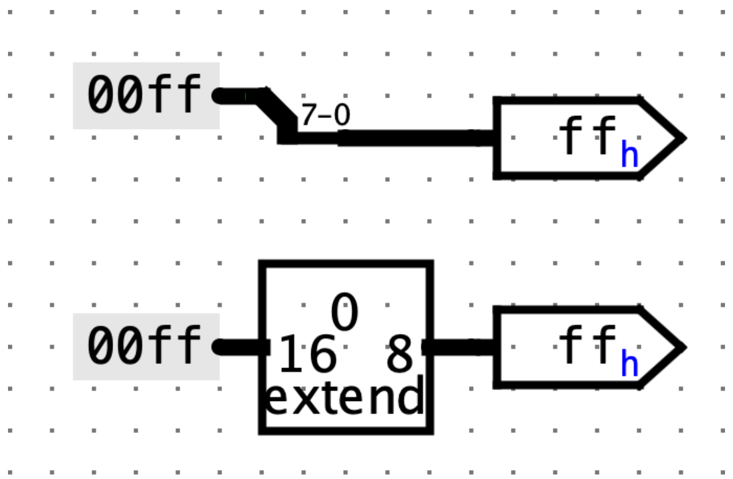
Extenders

When changing the width of a wire, you should use a bit extender for clarity. For example, consider the following implementation of zero-extending an 8-bit wire into a 16-bit wire:



Compared to the splitter, the extender is easier to understand at a glance. This becomes especially helpful when working with complex circuits.

Additionally, consider the case of throwing out bits. Despite its name, an extender can also perform this operation:



While the splitter is more minimalistic, it's also slightly harder to read.

Exercise 4: FSMs to Digital Logic

Now we're ready to do something really cool: translate an FSM into a digital logic circuit. FSM stands for Finite State Machine. FSMs keep track of inputs given, moves between states based on these inputs, and output something every time they receive an input.

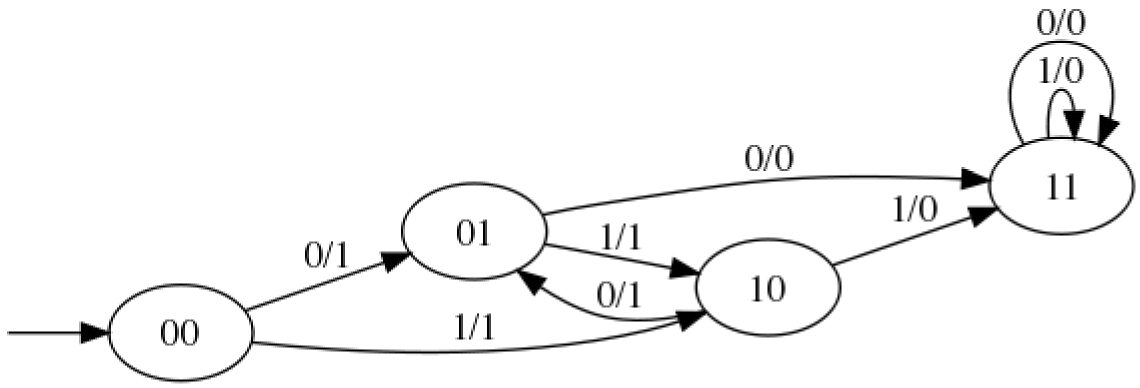
We use a register to store the state of the FSM we're currently in, and combinational logic to map FSM input & current register state to FSM output & next register state.

Action Item

You'll be implementing the following FSM:

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

Note that the FSM is implemented by the following diagram (the four state names `00`, `01`, `10`, `11` are just names written in binary - they have no direct relationship with the actual zeros and ones of the FSM input/output). Take some time to understand how this diagram implements the FSM:



Below is a truth table for the FSM (convince yourself of this). Note that since we're working with a binary number, "bit 0" is the less significant bit (i.e. right digit) and "bit 1" is the more significant bit (i.e. left digit).

CurrentStateBit1	**CurrentStateBit0**	**Input**	**NewStateBit1**	**NewStateBit0**	**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

Follow the steps below. Remember to **save often**, and **avoid moving or editing the provided input/output pins**.

1. Open up `ex4.circ` in Logisim. Note that the top level `FSM` 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.
2. Complete the `StateBit0` and `StateBit1` subcircuits, which produce the 0th and 1st bits of the **next** state using the input bit and current state.
3. When you're done, try running the provided tests.

Checkpoint

At this point, you should have more familiarity with designing and implementing FSMs, and see the close relationship between FSMs and digital logic.

Exercise 5: Practice with Splitters

We're going to construct a circuit that manipulates an 8-bit number.

Action Item

Follow the steps below. Remember to **save often**, and **avoid moving or editing the provided input/output pins**.

1. Open up `ex5.circ` and go to the empty SplitThing circuit.
2. Go to the `Wiring` library and select the `Splitter` circuit. This circuit will take a wire and split it into a set of wires of smaller width. Conversely, it can also take many sets of wires and combine them into one.
3. Change the `Bit Width In` property (bus width) to `8`, and `Fan Out` property (# of branches) to `3`. Now, select which bits to send out to which part of your fan. The least significant bit is bit `0` and the most significant bit is bit `7`. Bit `0` should come out on fan arm `0`, bits `1`, `2`, `3`, `4`, `5` and `6` should come out on fan arm `1`, and bit `7` should come out on fan arm `2`.
    - The `None` option means that the selected bit will not come out on ANY of the fan arms.
    - If you want to change many consecutive bits at once, click the name of the first one (not the arm #), hold `Shift`, and click the name of the last one. All the rows in-between should be highlighted. Now, change the arm # of *one* of the bits, and all the bits should be updated at once.

4. Route `INPUT0` to the splitter. Attach a 2-input `AND` gate to fan arms `0` and `2` and route the output of the `AND` gate to `OUTPUT0`.
    - If your splitter is facing the wrong direction, you can change the `Facing` property to the opposite cardinal direction.
5. Now, interpreting the input as a "sign and magnitude" number, place logic gates and other circuits so that `OUTPUT1` receives the negative "sign and magnitude" value of the input. Sign and magnitude is an alternate way of representing signed values - like 2's complement, but simpler! The combinational logic requires very few gates.
6. We will need another splitter to recombine the fans into a single 8-bit bus. Place another splitter with the proper properties (Bit Width In: 8, Fan Out: 3, correct fan widths). Play with the `Facing` and `Appearance` properties to make your final circuit as clean-looking as possible. At this point, `OUTPUT1` should be the negation of the input (when interpreting the `INPUT0` and `OUTPUT1` as "sign and magnitude" values).
7. When you're done, try running the provided tests

Checkpoint

If we decide to take the input and interpret it as a 2's Complement number, what inputs would produce `OUTPUT0 = 1`? *Hint: What do the first and last bits of a 2's complement number being 1 tell you about the number?*

Exercise 6: Rotate Right

With your knowledge of splitters and your knowledge and experience with multiplexers, you are ready to implement a non-trivial combinational logic block: `RotRight`, which stands for "Rotate Right". The idea is that `RotRight` will "rotate" the bit pattern of input `INPUT0` to the right by `AMOUNT` bits. So, if `INPUT0` were `0b1000000000000101` and `AMOUNT` were `0b0011` (`3` in decimal), the output of the block would be `0b1011000000000000`. Notice that the rightmost `3` bits were rotated off the right end of the value and back onto the left end. This can be represented as a bitwise operation by `R = A >> B | A << (16 - B)`.

Action Item

Implement a subcircuit named `RotRight` with the following inputs:

- `INPUT0` (16-bit), the 16-bit input to be rotated
- `AMOUNT` (4-bit), the rotation amount (why 4 bits?)

You can find the starter subcircuit in `ex6.circ`. Remember to **save often**, and **avoid moving or editing the provided input/output pins**.

The output should be `INPUT0` rotated right by `AMOUNT` bit positions, as outlined above. You are **NOT** allowed to use Logisim shifters in your solution, though all other combinational logic (MUXes, constants, gates, adders, etc.) is allowed. Logisim's built-in MUXes (find them under the `Plexers` menu) might be especially helpful. Your solution shouldn't involve a clock or any clocked elements, like registers.

- Before you start wiring, you should think very carefully about how you might decompose this problem into smaller ones and join them together. You should feel very free to **use subcircuits** when implementing `RotRight` (we've provided `rot1`, `rot2`, `rot4`, and `rot8`, why?). If you don't, things might get complicated.
- Just because we gave you a bitwise operation doesn't mean it's the best way to look at this problem. Think about the input bits of `B` and think about how to effectively use splitters! Can you do something with the binary form? Remember why binary is good for use in computers: a `1` is easy to represent as an `ON` signal, and a `0` is easy to represent as an `OFF` signal. Let's say we want to rotate `9` times. `9` is `1001` in binary, or `1*8 + 0*4 + 0*2 + 1*1`. Can you use this to make a cleaner circuit? Making use of the rot* circuits we have provided is a good idea that will keep things clean!

When you're done, try running the provided tests.

Exercise 7: Buggy Traffic Lights

An FSM is a highly versatile and simple tool to understand and develop systems for all kinds of applications. In this exercise, we model real-world traffic lights as an FSM.

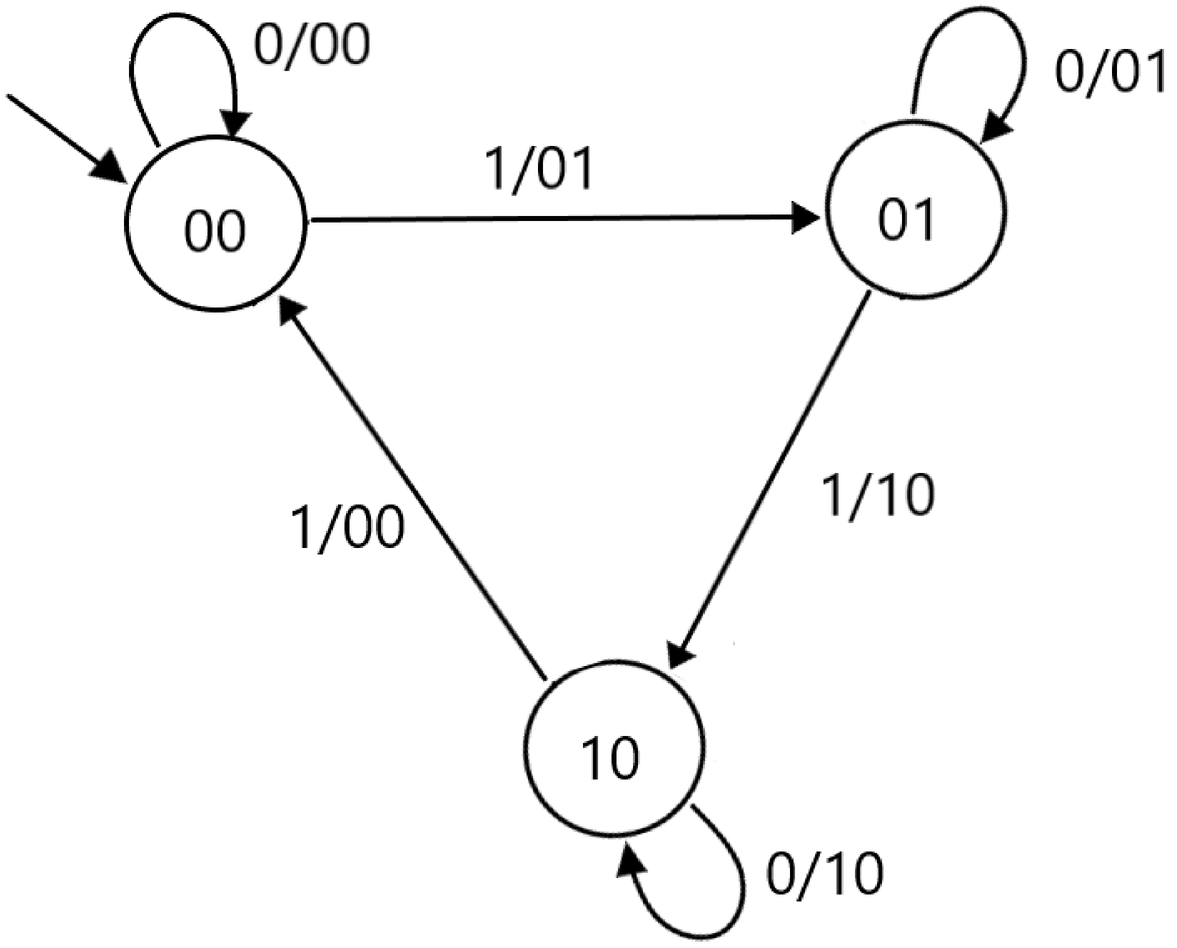
Like Exercise 4, we use a register to store the state of the FSM we're currently in, and combinational logic to map FSM input & current register state to FSM output & next register state. However, the interesting bit about this exercise is that we've provided you with a buggy implementation of the traffic light FSM, and you must debug it to make it work correctly.

- The FSM has 3 states:
  1. `RED` encoded as `00`
  2. `GREEN` encoded as `01`
  3. `YELLOW` encoded as `10`
- The input is a 'trigger signal'. When high (`1`), the FSM moves to the next corresponding state, i.e., our traffic light changes color. When low (`0`), the FSM loops in the current state, i.e., our traffic

light maintains the same color. Such a trigger signal could be periodic, controlled manually, or controlled by another system (such as a traffic monitoring system).

- The output of this FSM is same as as its next state (see diagram below).
- The traffic lights switch as follows: `RED` -> `GREEN` -> `YELLOW` -> `RED` and so on.

The FSM is implemented by the following diagram. Take some time to understand how this diagram implements the FSM:



Below is a truth table for the FSM (convince yourself of this). Note that since we're working with a binary number, "bit 0" is the less significant bit (i.e. right digit) and "bit 1" is the more significant bit (i.e. left digit).

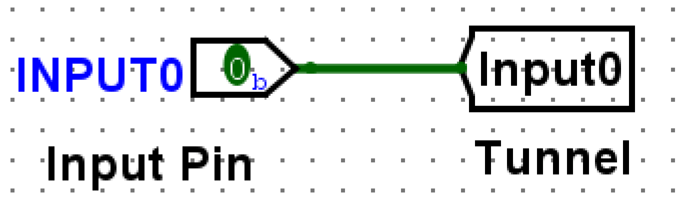
CurrentStateBit1	**CurrentStateBit0**	**Input**	**NewStateBit1**	**NewStateBit0**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1

CurrentStateBit1	CurrentStateBit0	Input	NewStateBit1	NewStateBit0
0	1	1	1	0
1	0	0	1	0
1	0	1	0	0

Action Item

You can find the starter circuit in `ex7.circ`. Remember to **save often**, and **avoid moving or editing the provided input/output pins**.

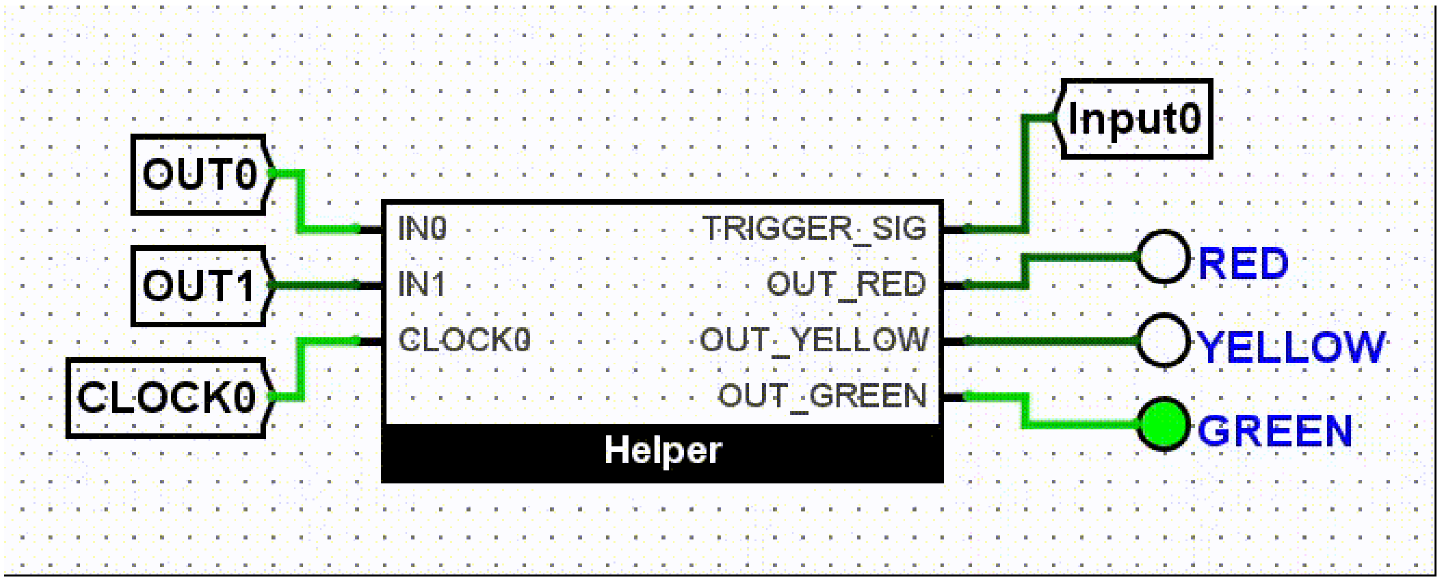
Keep in mind that tunnels are different from input/output pins. A tunnel has a label inside it whereas an input/output pin has a value inside it and a label beside it.



Three bugs have been planted into the `ex7.circ` file, in the following subcircuits:

1. `StateBit0`: Observe the wire colors and see if something is amiss here.
2. `StateBit1`: Again, look at the wire colors. Does a connection seem off? Think about the truth table to help guide you to the right answer.
3. `FSM`: The `clk` input pin carries the clock signal from the test harness. We can manually simulate this signal by using the `Poke` tool and toggling the value of the `clk` input pin. Poke around other wires and pins to observe their values for each (manual) clock tick. This should help you see why the FSM deviates from expected behavior.

Your job is to fix `ex7.circ` so it behaves correctly, as shown below:



Please note that there are **no errors in `FSMLogic` or `Helper`**. (The `Helper` subcircuit is used to map the FSM states to the LEDs and to provide a trigger signal. You don't need to look at it for this exercise, but feel free to poke around :).

To undo all of your changes to the file, from inside the `lab05` directory, run

```
$ git checkout starter/main -- ex7.circ
```

When you're done, try running the provided tests.

Checkpoint

At this point, you should have more familiarity with debugging Logisim circuits and programming FSMs.

Exercise 8

We are working to improve the labs for next semester, so please fill out this survey to tell us about your experience with the past few labs. The survey will be collecting your email to verify that you have submitted it, but your responses will be anonymized before the data is analyzed. Thank you!

Submission

Save, commit, and push your work, then submit to the **Lab 5** assignment on Gradescope.

Checkoff

Dark Mode