CSCB58 Lab 2
Part A: Multiplexer, Adders, ALU

1 Introduction

In this part, we will practice hierarchical design: how to build complicated components by utilizing multiplecopies of simpler ones. We will also learn about busses – wires that contain multiple bits.

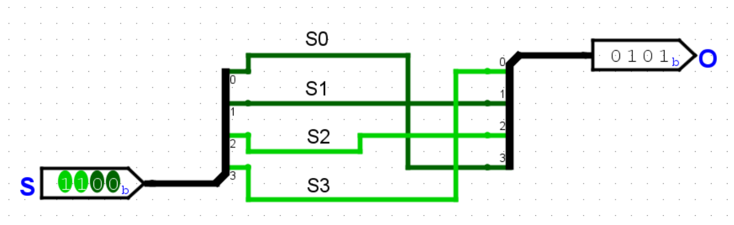
We will implement and use two of the logical devices that we learned in the lecture – multiplexers and adders. In part 3a, you will implement a simple 2-to-1 multiplexer, then and use it to build a 6-to-1 multiplexer. In part 3b, you will implement a full-adder, and use multiple copies to build a 4-bit adder. In part 3c, you will putall of these together to implement a simple *Arithmetic-Logic Unit* (ALU) – the heart of a CPU!

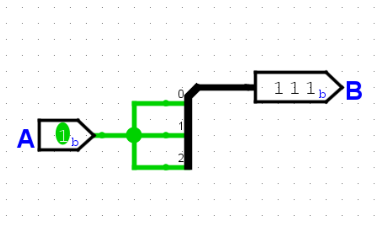
This lab is much longer than the previous two. Make sure to begin early!

2 Buses and Bundles

Consider a circuit that accepts two 8-bit binary number as inputs, X and Y. While each such input signal contains 8 wires (one per bit: X_7 to X_0, B_7 to B_0), they are all "logically" the same signal. It is common to show such signals using a single line called a *bus* or a *bundle*. Such buses are often used in circuits to show binary numbers and other multi-bit signals. We refer to the number of bits (wires) in the bus as the *width* of the wire. An 8-bit bus (or sometimes we'll just say an 8-bit wire) is a bundle of 8 wires.

Note that even though this is shown as a single line, these are still in fact multiple wires. We can therefore easily split the different bits of the multi-bit wire to multiple smaller bus, or combine multiple smaller wire to a single wide bus. In Logisim, both are done using a **Splitter** component, available in the explorer pane under "Wiring". See the documentation in the user guide: Logisim Reference → Additional Features.



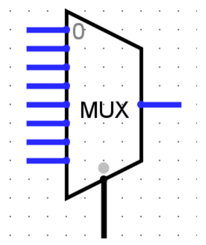


The figure above shows how busses and splitters are used in Logisim. The left side shows a 4-bit wide signal S being split to 4 bits, and then rebuilt in a different order, with bits 0 and bits 3 are swapped: a signal with value `1100` becomes `0101`. Note that this is done without any need for gates or transistors. The figure on the right below shows a 1-bit wire A being replicated 3 times to create a 3-bit bus with all identical bits B.

Many Logisim components such as basic gates, pins, and so on support multi-bit inputs and outputs. Simply set their width ("Data bits") in the Properties window. See the Logisim user guide and the reference document for more information. For example, a 3-bit AND gate with inputs $X = X_2X_1X_0$ and $Y = Y_2Y_1Y_0$ will output the 3-bit output $Z = Z_2Z_1Z_0$ where every bit Z_i is the AND of the corresponding bits A_i and B_i.

You will need to use busses, splitters, and multi-bit gates for this lab.

3a Multiplexers



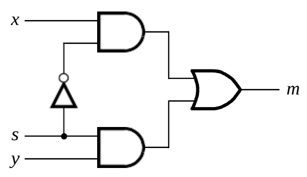
Multiplexers (muxes) are devices which select between data inputs. A mux with an n-bit *select* input can select from up to 2^n data inputs, and the selected data input is connected to the output of the mux. The data inputs are treated like an array, and the *select* input contains the index of the data input that should be forwarded to the output.

The picture on the right depicts a built-in Logisim 8-to-1 mux: it has $n = 3$ select bits that select which of the $2^3 = 8$ input wires will be shown at the output. The select input is shown as a single black 3-bit wide bus at the bottom.

Note that muxes can also select between busses, not just wires. For example, you can have an 10-bit 8-to-1 mux: there are 3 selects bits and 8 inputs, where each input (and the output) is 10-bit wide.

The Basic 2-to-1 Multiplexer

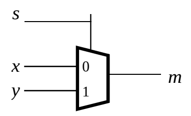
Below is the design of a simple 2-to-1 mux, in which a 1-bit *select* input can select between the two data inputs x and y. Note that the design is the same if x and y have multiple bits!



a) Circuit

s	m
0	x
1	y

b) Truth table



c) Symbol

Figure 1: Design of a 2-to-1 mux

Create a 4-bit 2-to-1 Mux

Create a new Logisim-Evolution project. For later steps, it is important there is no circuit named “main” in your project. **rename the default `main` circuit** to “mux2to1” or something similar (you can do this by modifying “Circuit Name” in the properties pane). Alternatively, create a new circuit “mux2to1” and remove the main circuit.

Implement the above 2-to-1 mux using 4-bit wide signals and gates in the `mux2to1` circuit. Test the circuit to verify that your implementation is working correctly. Make sure that your inputs and outputs are **named** `Pin` components, and not something like an LED or Button, as this will affect using the circuit as a symbol. Note you will need to use a Splitter to replicate the single select bit S to a 4-bit signal that can be used as input to the 4-bit AND gates.

[Optional] Design and Implement an 8-to-1 Mux

You don't need to submit this part and we also don't mark this part. Design a 4-bit wide 8-to-1 mux using only the 2-to-1 mux as building blocks. Create a new circuit called `mux8to1` and implement the 8-to-1 mux there. First create a 3-bit input pin `S` for the select bits. Create 8 input pins, each 4-bit wide, named `Input0`, `Input1`, `...`, `Input7`, and a single 4-bit output called `M`. Now use 2-to-1 muxes to make sure the right Input is selected to the output: for select bits `000` it is `Input0`, for `001` `Input1`, and so on.

Remember that once you created a subcircuit for mux2to1, you can easily place many copies of it. To use the circuits you created previously as a symbol, you simply place it just like you place any other component in the components list: click on it, and then click somewhere in the canvas. See the Logisim-Evolution reference document Section 2.5 if you need help.

Hints:

- You will need to split the 3-bit input `S` to individual select bits.
- Think hierarchically and recusively.
- Start small: can you build a 4-to-1 mux using a small number of 2-to-1 muxes?
- One you figure how to build 4-to-1 muxes, can you build 8-to-1 using 4-to-1 mux and 2-to-1 muxes?
- This is not the place for K-maps, since you are not allowed to use any gates.

3b Addition/Subtraction

In this part you will implement a 4-bit ripple carry adder with optional subtraction using full-adders. You should use the design from the lecture on adders and combinatorial devices.

- Rename the main circuit to `fa` and implement a full-adder using basic gates. You can use the design that includes XOR gates. See the lecture on adders for reference. Name your inputs `X`, `Y`, `Cin`, and your outputs `S` and `Cout`.
- Create a sub-circuit called `adder4`.
- Add inputs and output pins. The circuit will have two 4-bit inputs `X`, `Y` and a one bit input `Sub`. It will output a 4-bit `S` that contains the sum (if `Sub` is 0) or difference (if `sub` is 1) and a 1-bit output `C` for carry-out.
- Use four copies of the `fa` circuit and four XOR gates to implement an adder/subtractor. Again, use the design from the lecture on adders. Note you will need several Splitters for splitting and combining wires.
- Test the adder4 circuit.
- Save the project as `lab3b.circ` in the same directory as `lab3a.circ`.

3c Arithmetic-Logic Unit (ALU)

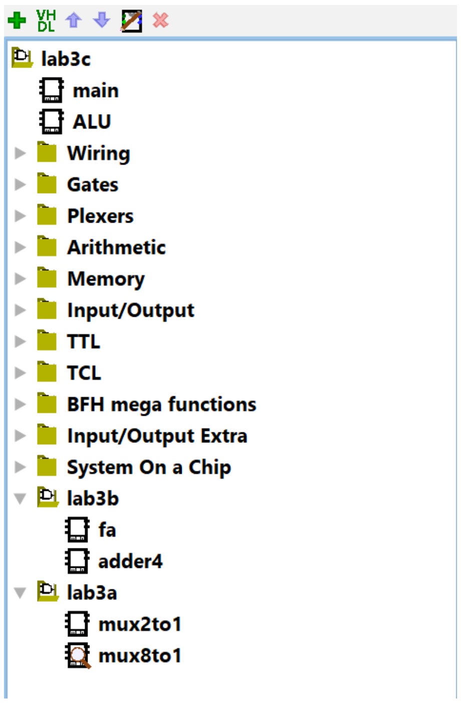
Arithmetic-Logic Units are the heart of CPUs and calculators. By now, you already know enough to build a simple one, which is exactly what we will do.

Given two 4-bit inputs *X* and *Y* and a 2-bit operation code `func`, our 4-bit ALU will output the following:

Operation	**func**	**Output**
Addition	`00`	$X + Y$
Subtraction	`01`	$X - Y$ (assuming X and Y are signed two's complement)
AND	`10`	X AND Y
OR	`11`	X OR Y

Note you already have all the components you need, and there is no need for additional logic: use your adder/subtractor circuit to add and subtract, built in 4-bit AND and OR gates (use Logisim), and a small number of 4-bit 2-to-1 muxes to select the output. To make it more interesting, we will wire up our ALU logic circuit to switches and displays.

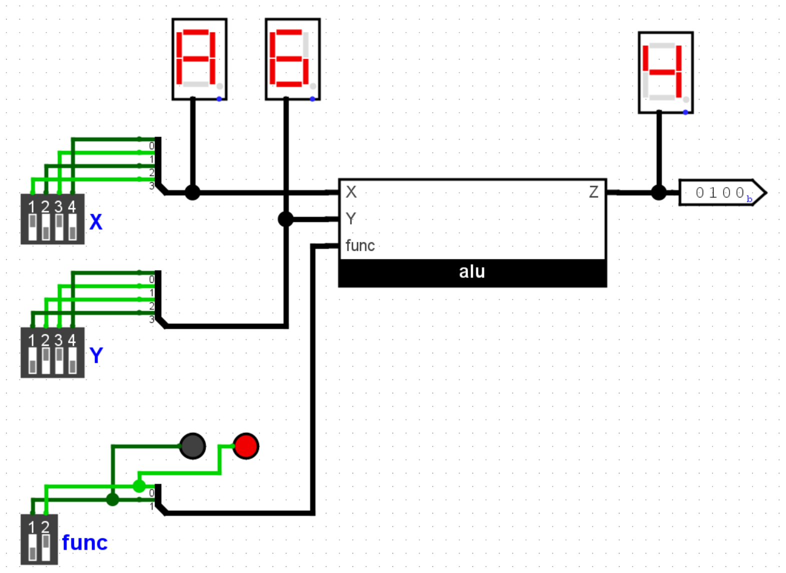
Guidance

1. Create a new empty Logisim project and save it with the name `lab3c.circ` to the same directory as `Lab3a.circ` and `lab3b.circ`.
2. Load your designs from previous parts as libraries:
   click **Project → Load Library→ Logisim-evolution Library...** then select the `lab3b.circ` file. The circuits from that file will appear as components in the explorer pane (see screenshot on the right). Repeat this for `lab3a.circ`.
   

3. Create a sub-circuit called `ALU` and implement the ALU above. It will have 2 4-bit inputs `X` and `Y`, a 2-bit input `func`, and a 4-bit output `Z`. You are only allowed to use one `adder4`, one 4-bit AND gate (configured "Data Bits"), one 4-bit OR gate, and a small number of `mux2to1`.
4. We are going to add some bells and whistles. Real hardware boards we can often have switches, buttons, 7-segment displays and LED lights to use for inputs and outputs. Let's do the same for our simulated board.

- In the top-level main circuit, add a *dip switch* (you will find this component in the explorer pane under **Input/Output → Dip switch**) and set the number of switches to 4 (use the properties pane). We will use *this* switch as a 4-bit input for `X`.
- Use a Splitter to bundle the output of the switches into a 4-bit bus. Pay attention to connect the right bit to the right switch! We want the left-most switch to be bit 3, and the right-most switch to be bit 0.
- Since `X` is a 4-bit number (a hexadecimal (base 16) digit) we can also display it using a 7-segment display: add a **Input/Output → Hex Digit Display**), and connect the 4-bit wire for `X` to the display.
- Repeat the above steps for the second input `Y`.
- Add a third dip switch with 2-bits for `func`, and again use a Splitter. To show the value of `func`, we will use LED lights (**Input/Output→LED**): add one LED light for bit 0 of `func`, and another LED light for bit 1 of `func`.
- Add an `alu` to the circuit and wire up all the inputs correctly.
- Add another Hex Digit Display and a 4-bit output pin for the output `Z`

At the end of this step, your circuit should look like this:



5. Test your circuit, and once satisfied save it to `lab2a3c.circ`.

4 Summary of tasks

Below is a short summary of the steps to be completed for this lab:

1. Read through the entire handout.
2. Implement the 2-to-1 mux in part 3a.
3. Implement the adder/subtractor in part 3b.
4. Implement the ALU in part C using the circuits you built in parts 3a and 3b.
5. Submit a zip file, named **lab2_YourName.zip,** to Quercus with all 3 parts: `lab3a.circ`, `lab3b.circ`, and `lab3c.circ`. (**You also need to submit your Part B work in the same zip file**)
6. Demonstrate your solution to the TA and answer their questions.

Part B: Sequential Logic

1 Introduction

In this part, we'll examine the behavior of latches and flip-flops – circuits which maintain internal state. Designs that utilize these circuits are called *sequential*, since the sequence of inputs matters.

2 SR Latches

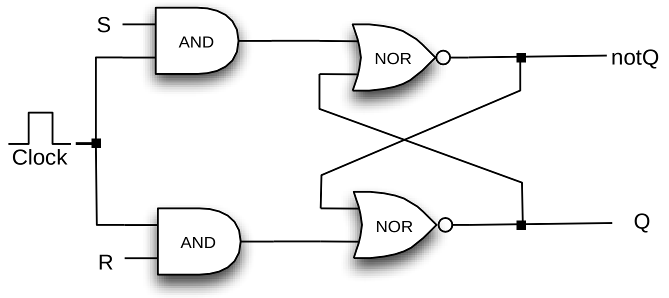


Figure 1: A gated (or clocked) SR latch. This implementation is a little different from what we saw in class, but this is indeed an SR latch.

[Optional] Truth Table

You don't need to submit this part and we also don't mark this part. What would the truth table for a gated SR latch like the one above looks like? You may use the following values as entries in the cells of the truth table: `1`, `0`, `prev Q`, `prev notQ`, `X` and `?`.

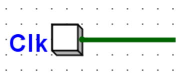
`prev Q` means that this output will have the same value as `Q` used to have, `prev notQ` means that the output will have the same value as `notQ` used to have, `X` means we don't care what the output is, `?` means the output can be 0 or 1, but we cannot be sure.

Think:

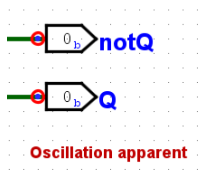
- How many columns of input does your table have? Clearly, it should have columns for `S` and `R`.
- What about the `Clock`?
- How would you represent the internal state (the initial values of `Q` and `notQ`)?
- How many rows does the table have or needs to have?
- How many circuit state and input combinations do you need to test to fully verify the circuit?

Task: Implementation

Implement the gated SR latch above as a sub-circuit in Logisim-evolution. Include three input pins as input to the circuit instead of connecting the clock input to a clock component (so that later you can reuse this circuit as a symbol). During testing, we will want to control when clock pulses arrive; using an input pin for the clock input gives us the flexibility to connect it to an input device like a *push button* (**Input/Output →Button**, see image on the right) or a dip switch (see previous part).



Simulator tip: Recall that SR latches have a forbidden input state: a specific sequence of inputs can put the circuit into oscillations (see lecture for details). When Logisim-evolution's detects this oscillation, it will stop the simulation entirely and will stop responding to changes in inputs. See the picture on the right. When that happens, you should reset the simulator: first, choose **Simulate → Reset Simulator** from the menu, and then **Simulate → Run Simulator** to make sure it is running.



Using your truth table, test your circuit. Does the output of the circuit match your truth table?

Save the project and submit it as part of the lab. **Demonstrate your circuit to the TA:** you must be able to show the TA an example where it's clear the gated SR latch is NOT edge-triggered.

3 D Flip-flop

The output of a gated latch changes immediately whenever the inputs change if the clock is high – during the positive "pulse" of the clock (we could implement latches that change during the "negative pulse" when the clock input is low). In contrast, the output of a flip-flop only changes when the clock input is changing. Typically, we prefer positive edge flip-flops: flip-flops that change when the clock input changes from low to high. We can also use *negative edge-triggered flip-flop* or *falling edge-triggered flip-flop*: flip-flops that change when clock goes from high to low.

Briefly, a D latch is like a gated SR latch whose S and R signals are always negations of each other; a master-slave D flip-flop is a series of two D latches (or a D latch followed by SR latch) which have opposite clock signals. Refer to the lectures for more details.

- Create a sub-circuit for a gated D latch and implement it using the SR latch you implemented previously. Your design should take two inputs, `D` and `C`, and two outputs: `Q` and `notQ`
- Create another circuit and implement a **positive edge-triggered D flip-flop** using two copies of the D latch circuit. Your design should take two inputs, `D` and `Clock`, and two outputs: `Q` and `notQ`
- Test your circuit to verify that your design works as expected.
- Demonstrate the circuit to the TA. Convince your TA that your positive edge-triggered D flip-flop works correctly.

4 Summary of tasks

Below is a short summary of the steps to be completed for this lab:

1. Before the lab, read through the lab handout.
2. Implement the gated SR latch.
3. Use your gated SR latch circuit to build a D latch circuit.
4. Use your D latch circuit to build a positive edge-triggered D flip-flop circuit.
5. Submit your to quercus: SR latch circuit, D latch circuit, and positive edge-triggered D flip-flop circuit in the same zip file as the Part A.
6. Demonstrate your implementation of the gated SR latch, D latch, and positive edge-triggered D flip-flop to the TA.

Overall Evaluation (Part A & B)

As always, marks are based not only on submitted work but also on oral examination by TA. You also need to be able to make reasonable explanations about any details to the TA.

Solution	Submitting everything on time	1 mark
	Part A:	
	Mux	1 marks
	Adder/subtractor	1 mark
	ALU	2 mark
	Part B:	
	SR and D-latch	1 mark
	Positive edge-triggered D flip-flop	2 marks
Understanding	TA oral score	1 to 4

Final lab marks (up to 8) are determined by multiplying your solution subtotal by the oral score (1–4) and dividing by 4:

$$total = solution\ marks\ \times \frac{oral\ score}{4}$$

<u>Note:</u> The interview with the TAs for this lab will occur during week 6 (Feb. 13 – Feb. 17) in your assigned practical time slot in-person. Failure to show up will result in receiving a ZERO on the oral score, and by extension, the lab.