CSC258 LAB

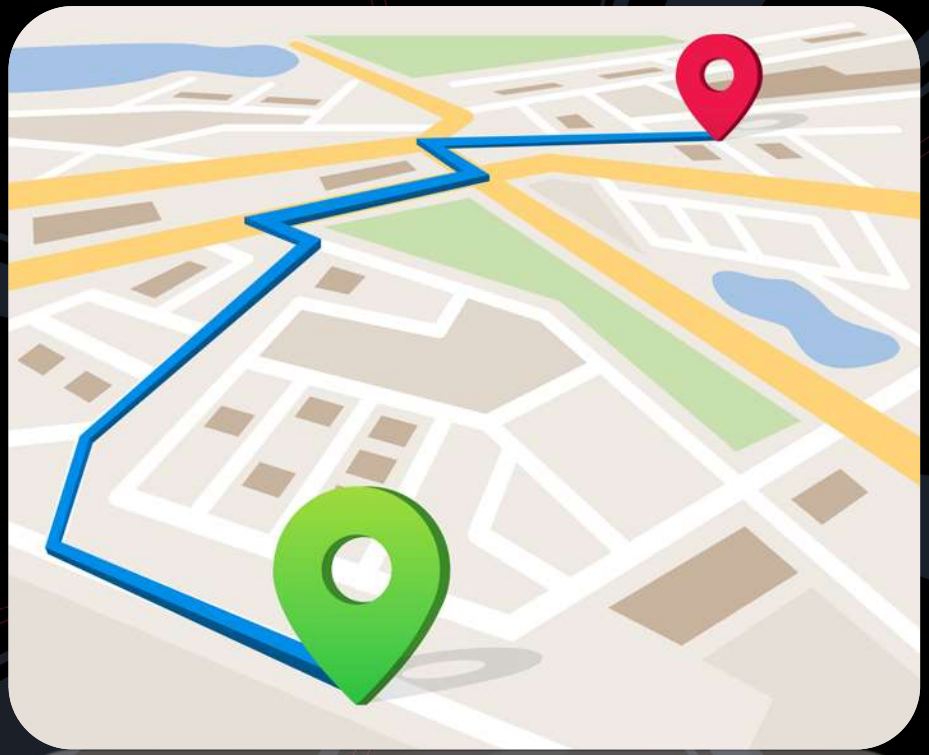
Sanjana Girish

Lab 1 – Preparation

Today's Tutorial

- Learning objectives for Lab 1
- Lab 1 parts
  - Warm-up exercise
  - What to hand in
- Intro to Logisim

Learning objectives

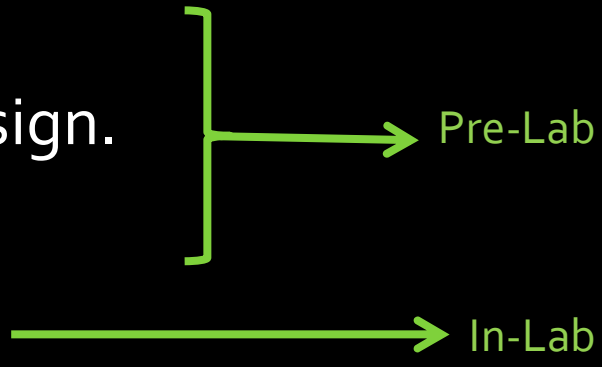


Lab 1 Learning Objectives

- What are the labs about?
  - Creating demo-worthy designs.
- What is Lab 1 about?
  - Learn how to build logic circuits by using logic gates.
  - Produce truth tables for a given design (starting either from a given logic function or from a description of the design's behaviour).
  - Demonstrate familiarity with the graphic tool Logisim.

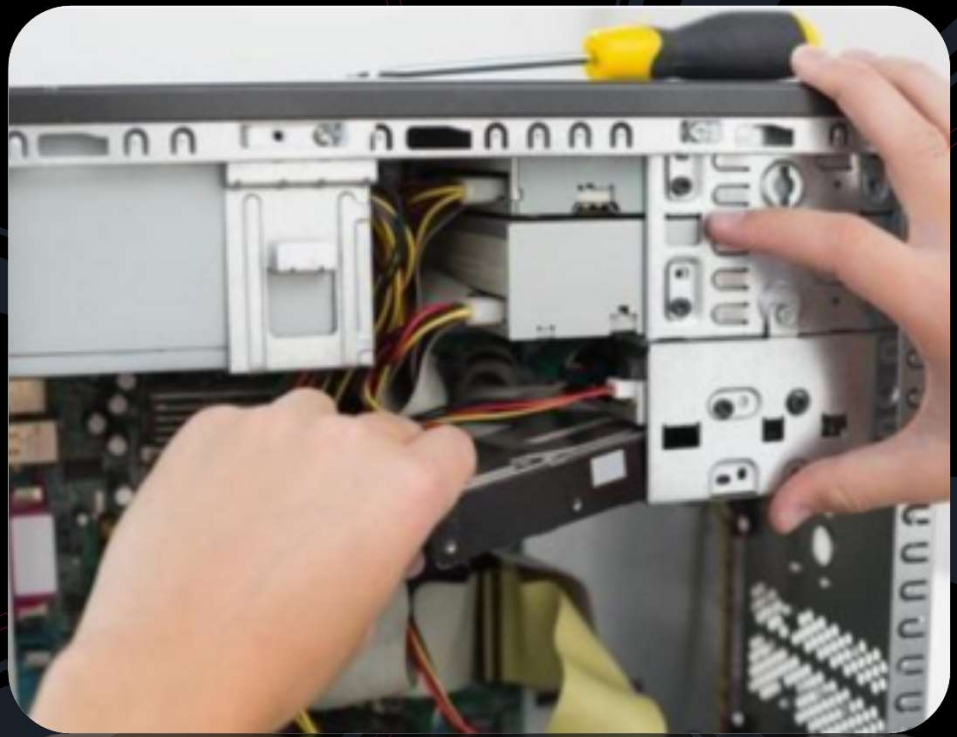
Approach to Lab 1

- Experience is the best teacher.
  - Prepare a design.
  - Implement your design.
  - Debug the circuit.
  - Demo your design



- Try to think of your prelabs as "an assignment due before the beginning of the lab".

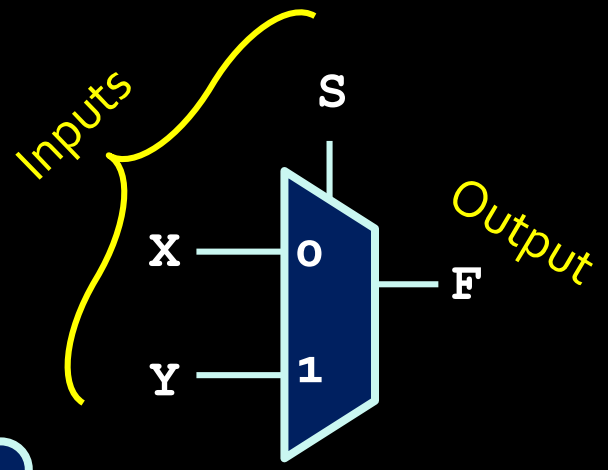
Lab Breakdown



Lab 1 breakdown

- Mark breakdown:
  - Pre-lab: 3 marks
  - Part I: 1 mark
  - Part II: 1 mark
  - Part III: 1 mark

- Part I:
  - Design circuit for multiplexer:



$$F = X\overline{S} + YS$$

Lab 1 breakdown

* = AND (∧)
\+ = OR (∨)

- Part I (cont'd):
  - Note that the following are all different ways of expressing the same thing:
    - $F = X\bar{S} + YS$
    - $F = X*S' + Y*S$
    - F = (X and (not S)) or (Y and S)
  - Need to represent this logical expression in gates.
  - Need to show the truth table for the three inputs X, Y & S and the output F.
- Part I doesn't involve Logisim yet ☺

Lab 1 breakdown

- Part II:
  - Given the function:

```
f = (a+b)' + cb'
```

  - How would you implement this in gates?
  - What is the minimal number of gates you need?
- Part III:
  - Implement these circuits in Logisim.
  - Test your designs (using Poke tool and test files)

Warm Up Example

- Design a circuit that implements the following logic function, using only 2-input AND and 2-input OR gates.

```
f = a*b + (c + b)
```

- Write down the truth table for this design.
  - Note: This expression is common shorthand for:

```
f = a AND b OR (c OR b)
```

Warm Up Example cont'd

- Is there a cheaper implementation (i.e., with fewer gates)?
  - `f = a*b + (c + b)`

What to hand in

Prelab vs Lab Demo

- Prelab exercises are due before 6pm on your lab day.
  - Written/hand-drawn elements in PDF files.
  - Logisim circuits as `*.circ` files.
  - ~~Logisim tests as `*.txt` files.~~
- TAs will ask to look at your Logisim designs, so be ready to share your screen with them.
  - Also be ready to share the hand-written elements in case a question arises about your design process.

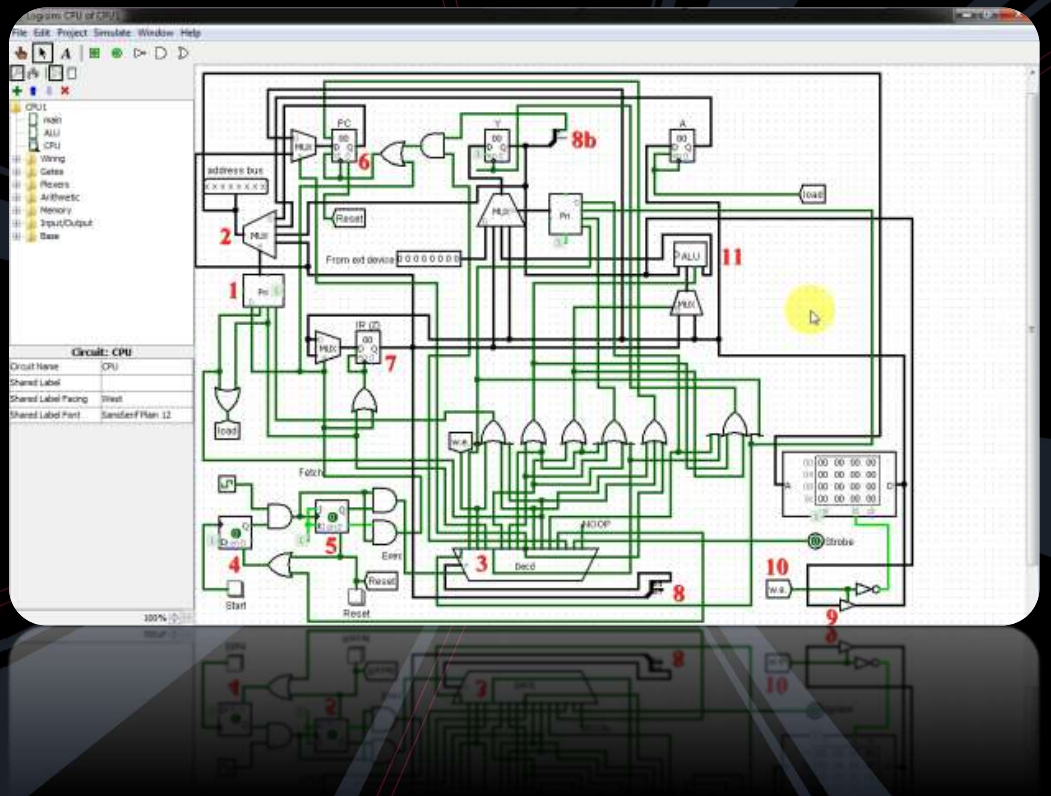
Pre-lab reports

- The hand-written report should include the following:
  - Lab number and title
  - Student info (last name, first name, student #)
  - Exercise parts
    - Each in its own clearly-labeled section.
    - Restate the question (summarized).
    - Provide the calculations (if applicable).
    - Illustrate the solution (including pin labels).
  - PLEASE BE NEAT.
- The Logisim files should be named to reflect the lab number and part number.
  - e.g. `lab1_part2.circ`

Things to note

- This will be the easiest lab you do in the course.
- Whenever possible, use the tools and submit a printed pre-lab report.
- Try to come up with the smallest circuits possible.
  - How do you reduce a complex circuit?
  - For now, think back to boolean algebra axioms!
  - Simple reasoning helps as well ☺

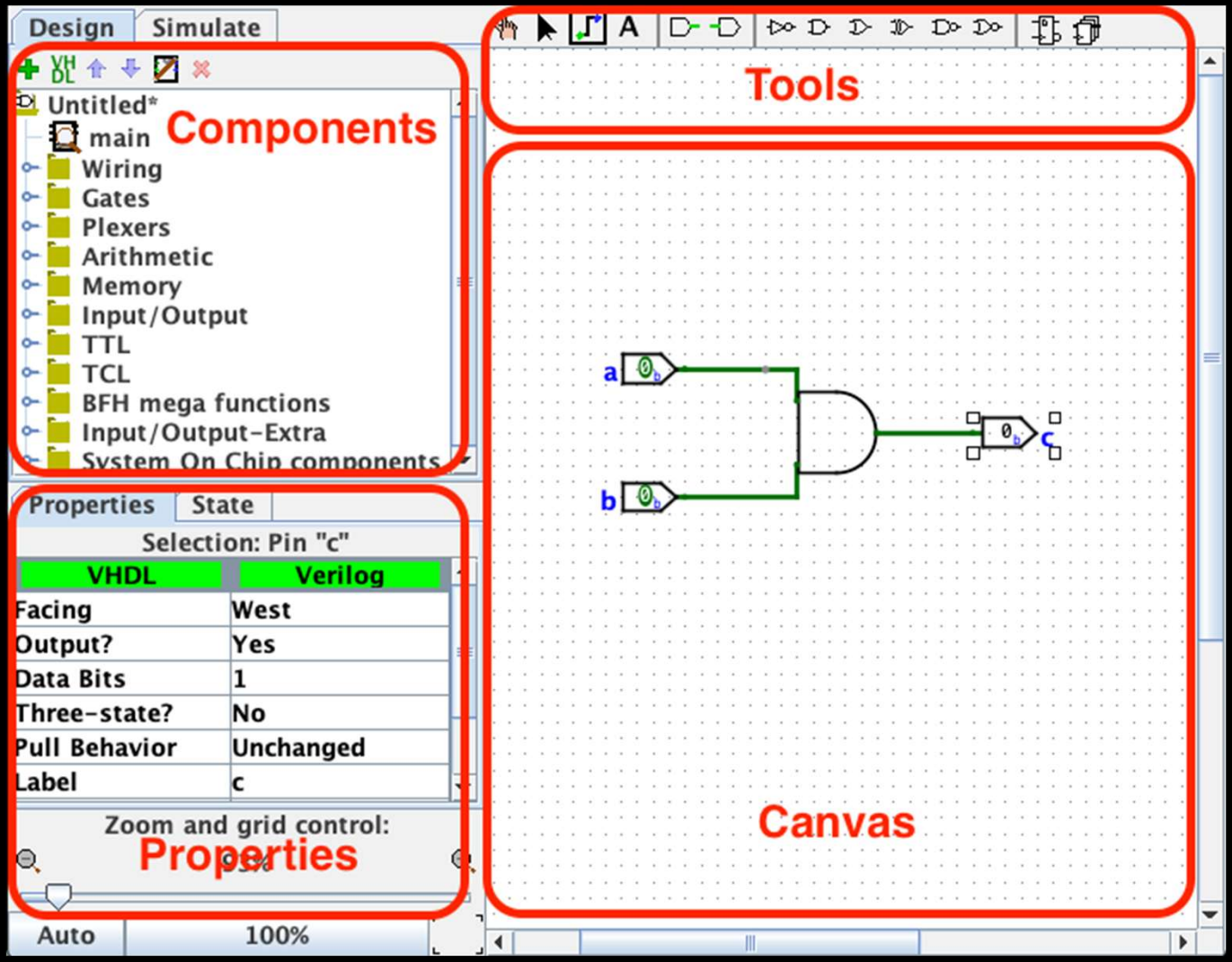
Intro to Logisim



Logisim installation

- Logisim is a powerful logic circuit simulation environment.
  - In this course, we will be using version 3.3.0
    - https://github.com/reds-heig/logisim-evolution/releases/tag/v3.3.0
    - Just the `jar` file is needed.
- Note:
  - Make sure to use Logisim-Evolution downloaded at the above link. Do NOT use the original Logisim or any other variations or versions of it.
  - You will need Java 13 installed to launch this.

Logisim walkthrough



Tools/Views

Poke: Click on wires to inspect their state, click on most components to change their state.

Tools/Views

Select: Selects and moves things in the canvas, and manipulates wires/buses. Click and drag from inputs/outputs to create wires/buses.

Tools/Views

Wire: Creates wires on the canvas.

Tools/Views

Text: Add text on the Canvas.

Tools/Views

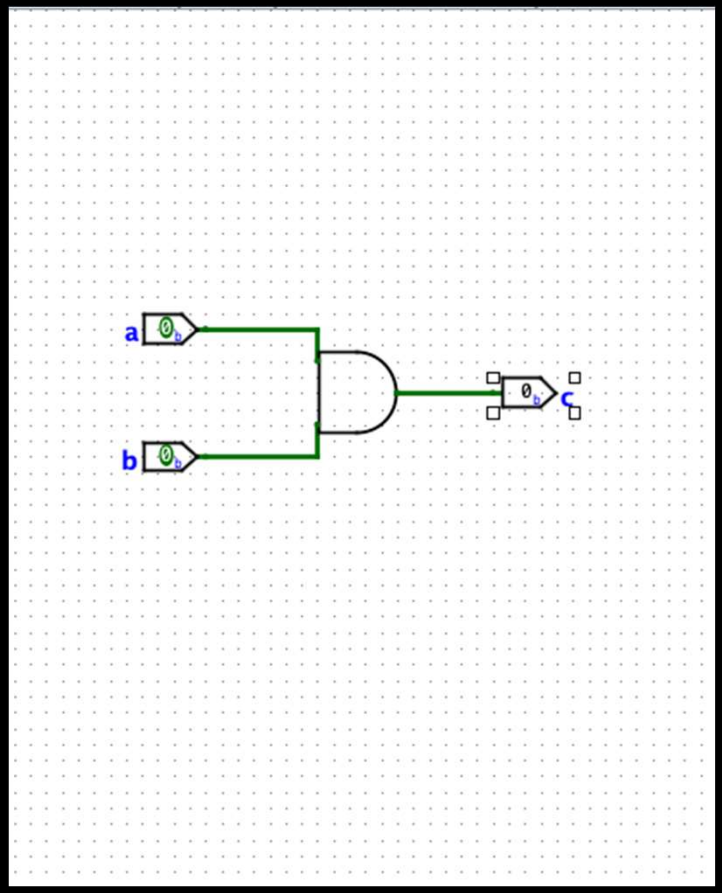
Default Input/Output: default type of input and output for the circuits. You will be using them a lot throughout the course.

Tools/Views

Logic gates: some commonly used logic gates that you can click and drop on the canvas to build your circuits.

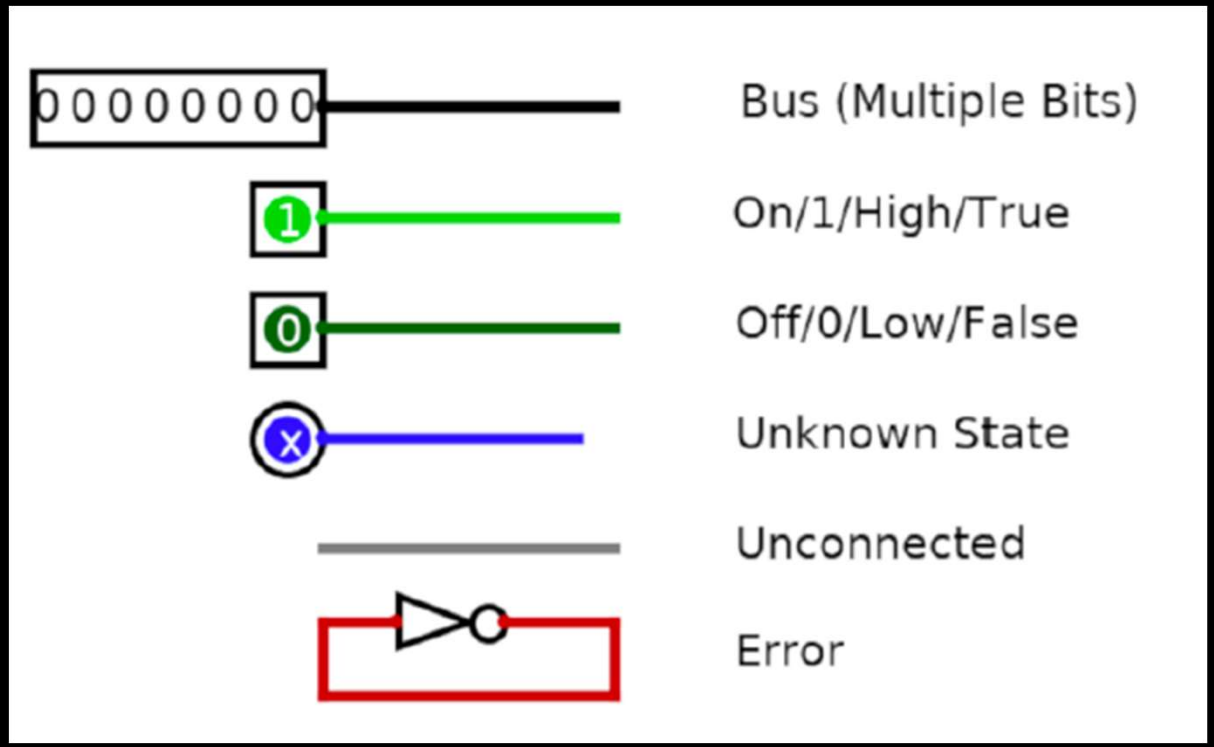
Canvas

- Canvas is where you will be building your circuits by dropping components on the canvas and then connect them with wires.



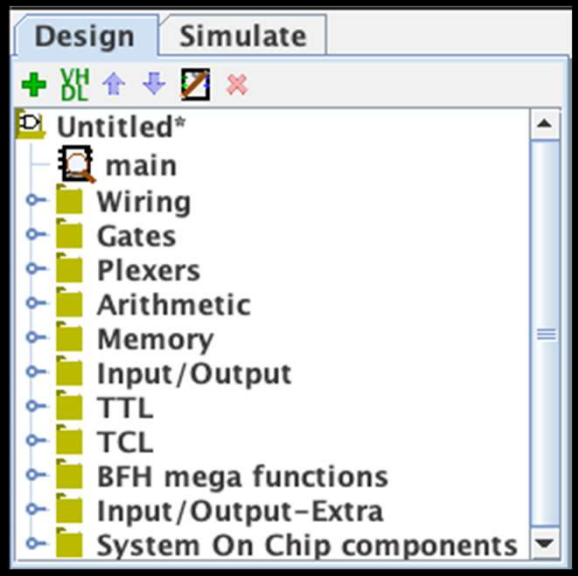
Wires

- Wires and buses can have many states. You can inspect the state of a wire/bus using **poke** in the toolbar.



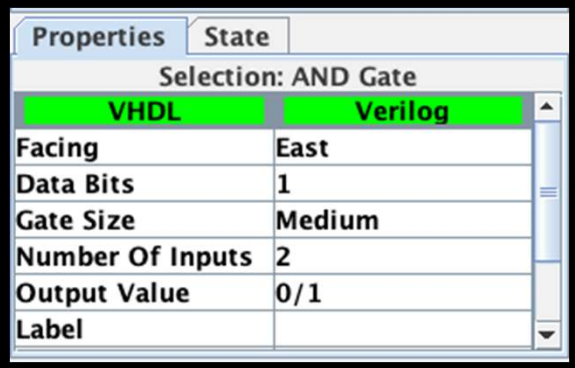
Components

- This contains all the circuits in this le as well as all the built-in components.
- Double-click on each circuit to view it. To place a component from this list, select it, and then click somewhere in the canvas.
- You can add/delete a circuit using the green + sign and the red x sign on the top.



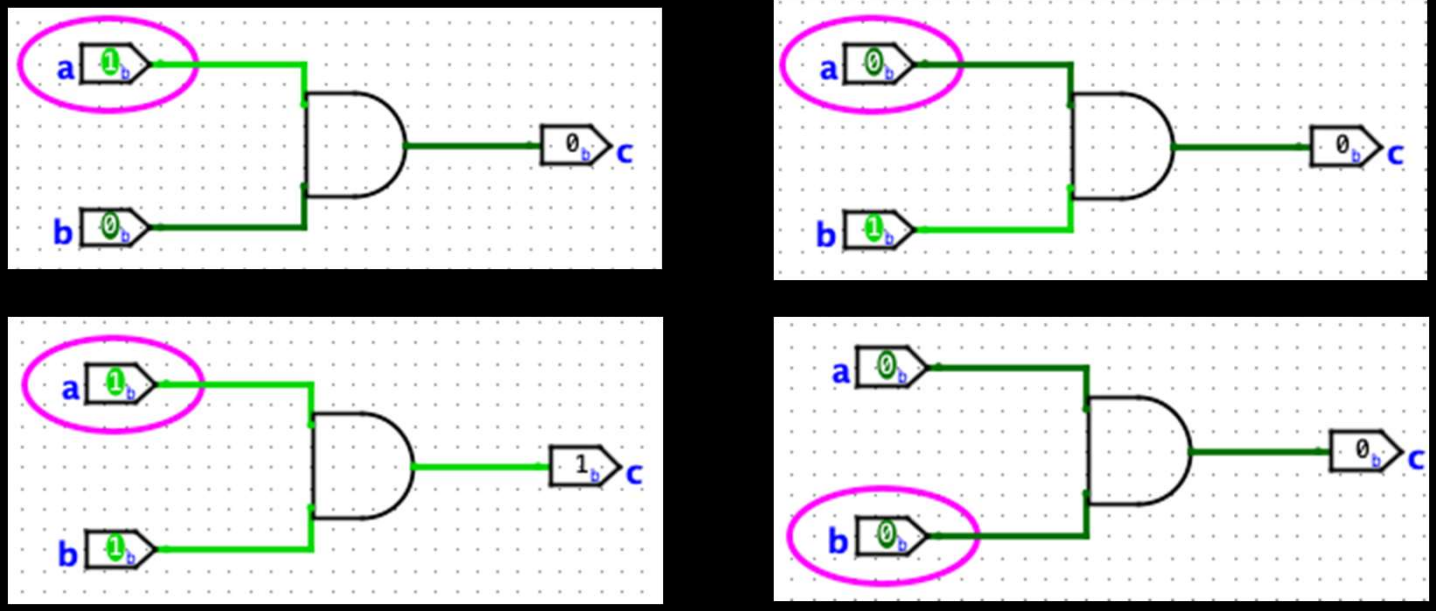
Properties

- If you click on a component on the canvas, you would be able to view and edit its properties.
- For example, this is the properties for an AND gate. You can change the number of input bits and number of inputs here. This will be useful in the future.



Testing in Logisim

- The easiest and most visual way of testing is using the Poke in the tool bar and click on the components to change the state. This will be very useful throughout the course so make sure you try this out.



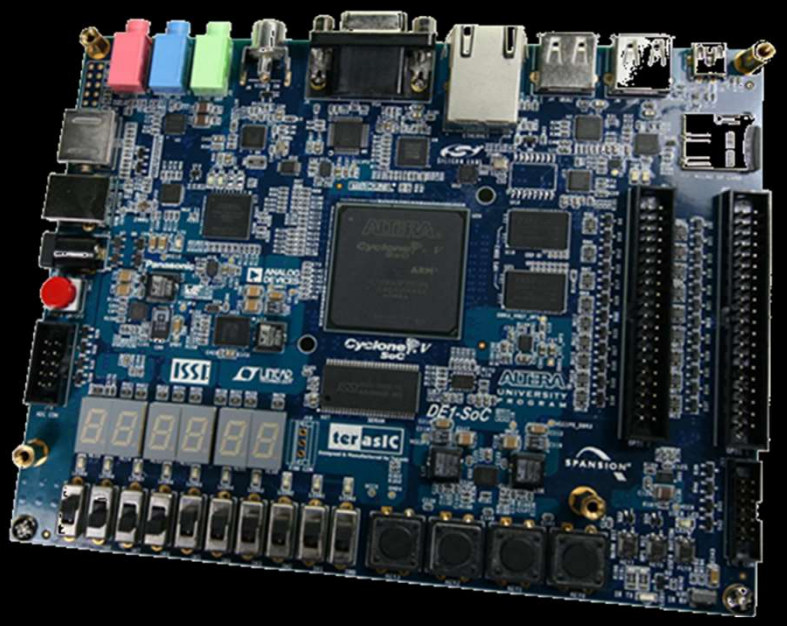
Testing in Logisim

- Another way is through test vector files. (details can be found in the lab handout)
- Steps involved:
  1. list the truth table for your circuit, the values for the inputs and the expected values for the outputs
  2. Logisim will be able to run the tests according to your truth table to test the functionality of your circuit.

Lab 2 Preparation

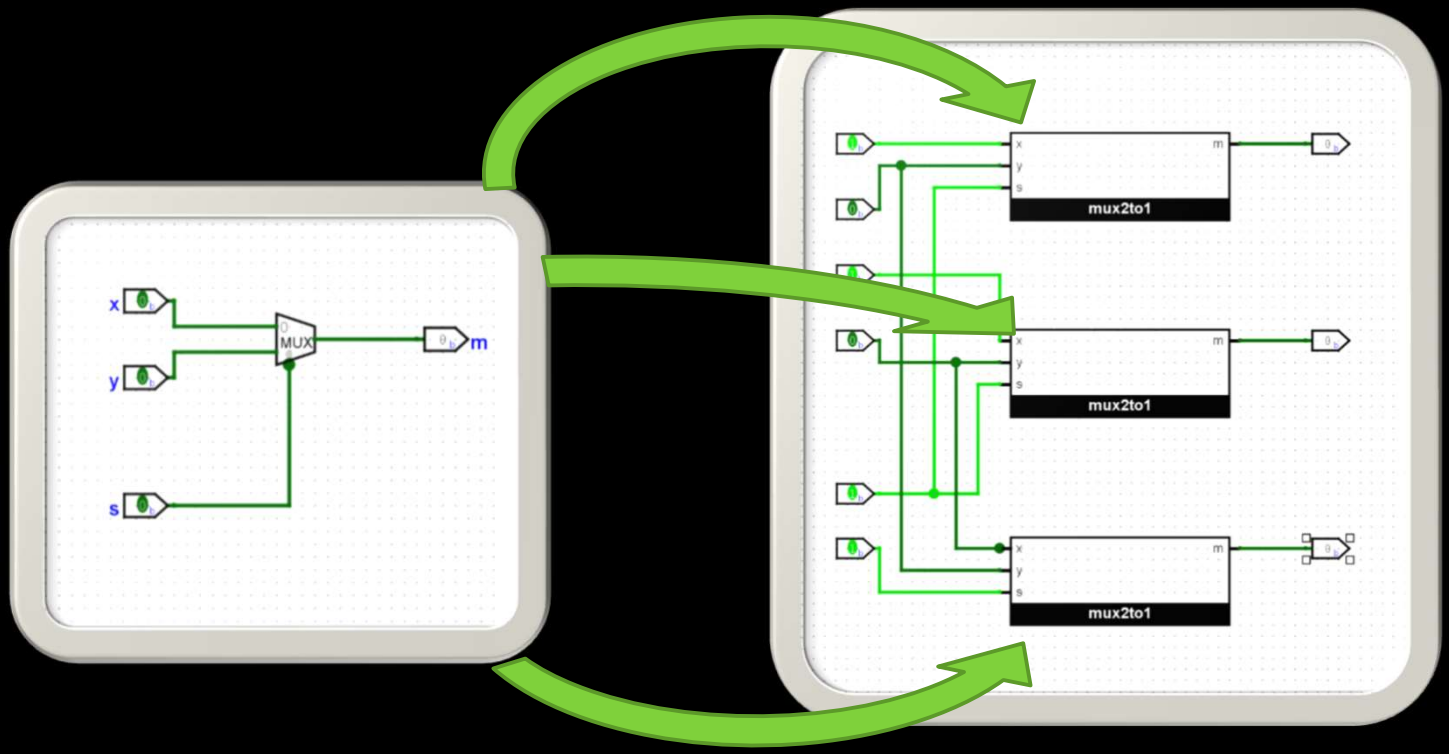
Lab 2

- Lab 2 topics:
  - Multiplexers (cont'd)
  - Design hierarchy
  - Decoders
    - 7-segment displays
- The DE1-SoC
  - Connecting Logisim to DE1-Soc
- Intro to useful components in Logisim.



Tasks for Lab 2

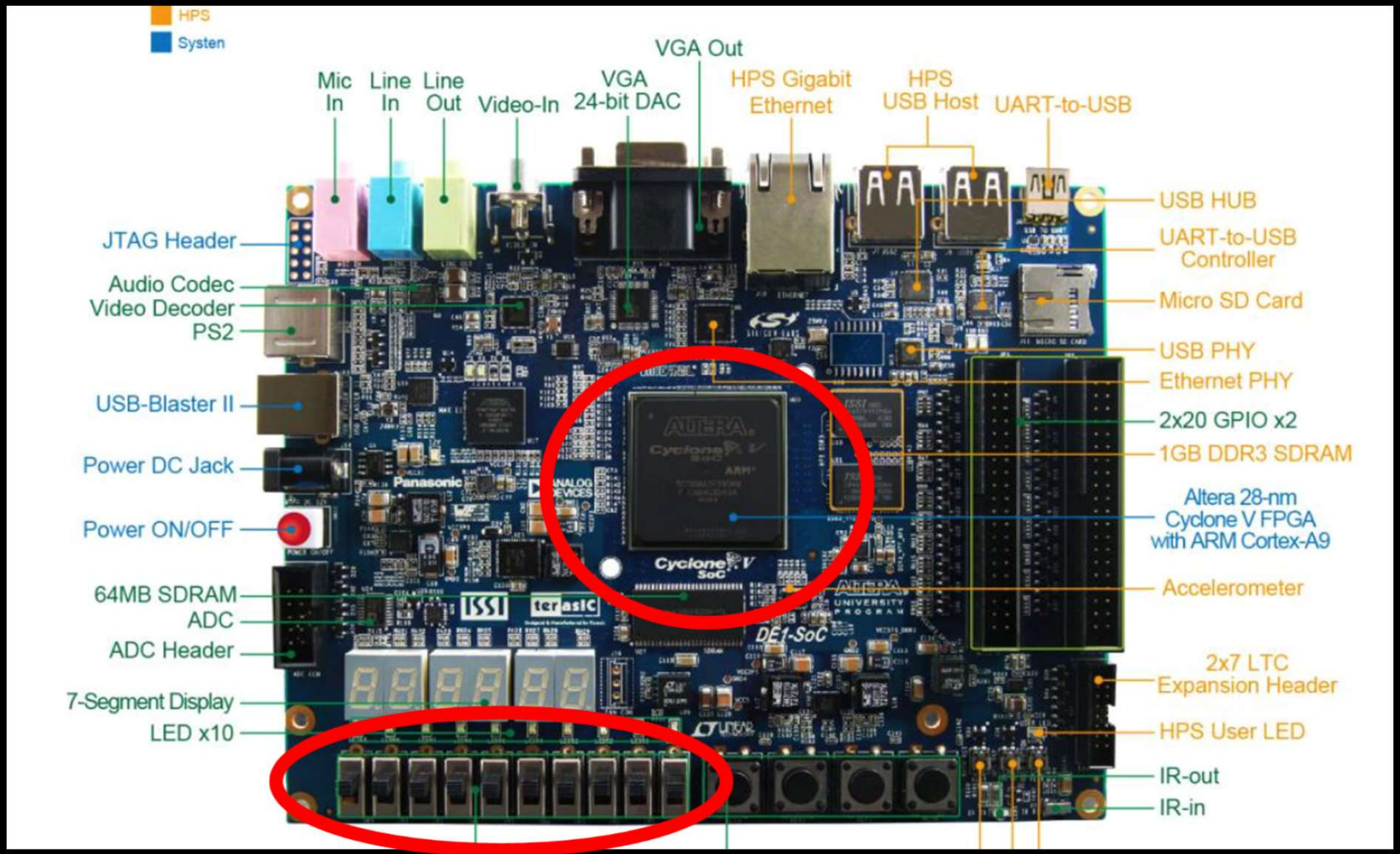
- Part I: Creating modules
  - Once created, a module can be used as a component.



Tasks for Lab 2

- Part I: Creating modules
  - Create a simple module that makes a wrapper for the mux circuit we provide.
  - Set inputs to buttons (labeled `SW0`, `SW1`, `SW2`) and output to LED (labeled `LEDR`)
    - Labels correspond to DE1-SOC inputs and outputs

Meet the DE1-SoC board!



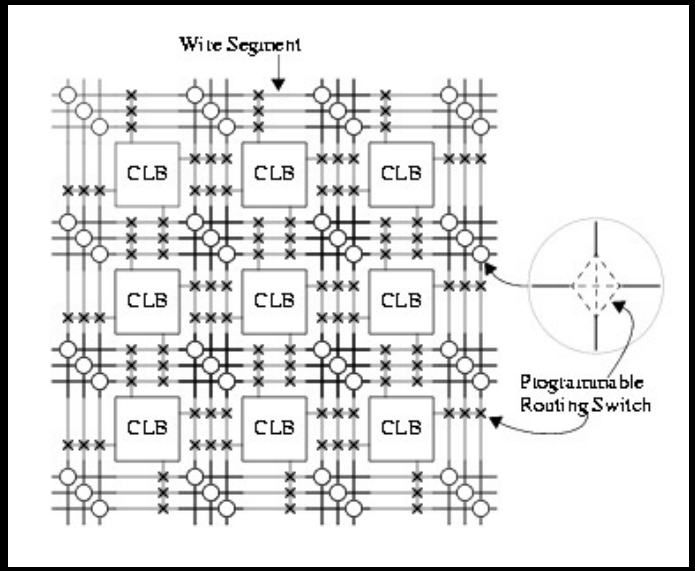
Meet the DE1-SoC board!

- In a non-COVID semester, we would be using Logisim to upload your designs onto the DE1-SOC.
  - Maybe in 2021 ☹
- What is the DE1-SOC?
  - It's a System On a Chip (SoC) with:
    - Altera's Cyclone® V 5CSEMA5F31 FPGA, and
    - a Dual-core ARM Cortex-A9 hard processor (HPS)

    - 64 MB SDRAM on FPGA device
    - Six 7-segment displays
    - 10 toggle switches
    - 10 LEDs
    - 9 green LEDs
    - Four pushbutton switches

What does that mean?

- Key term: FPGA.
  - Stands for Field Programmable Gate Array.
  - A regular network of logic that can be programmed and reprogrammed to implement any circuit.
  - Circuits aren't generally built by hand; they're programmed using languages like Verilog, VHDL or Logisim.



Connecting Logisim to DE1-SoC

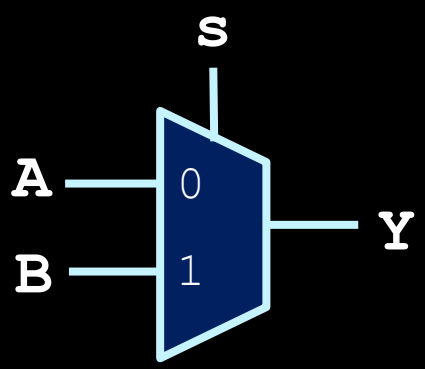
- The circuits we build in Logisim can be downloaded to the DE1-SoC board.
- You can map your inputs/outputs in Logisim to the switches or LEDs on the DE1-SoC board and then test your circuit on the board.
- Even though we currently do not access to the boards in the lab, it is still good to know the process and learn about the board.

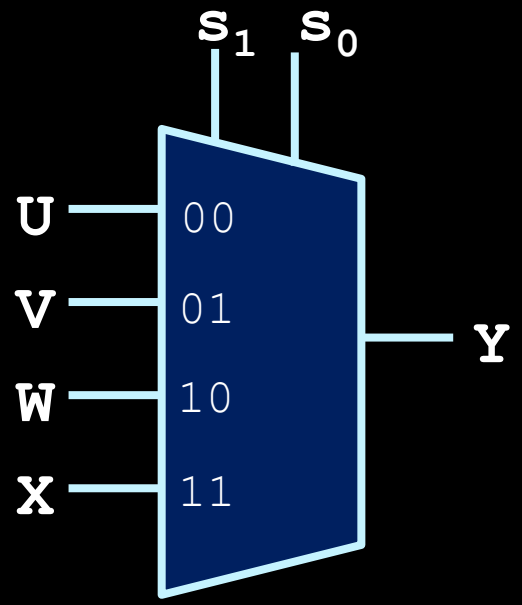
Connecting Logisim to DE1-SoC

- The details about the process can be found in lab2 handout.
- If you want to upload your design:
  - Make sure that you follow the steps carefully and understand how the circuit would have been if it can be downloaded on the board.
  - Note: you will need a tool called Quartus installed to compile in Logisim. Therefore we encourage you to test out the compilation of your design on the teach.cs machines. Quartus has been installed on them.

Tasks for Lab 2

- Part II: Designing with modules.
  - Make a 4-to-1 mux out of 2-to-1 muxes.



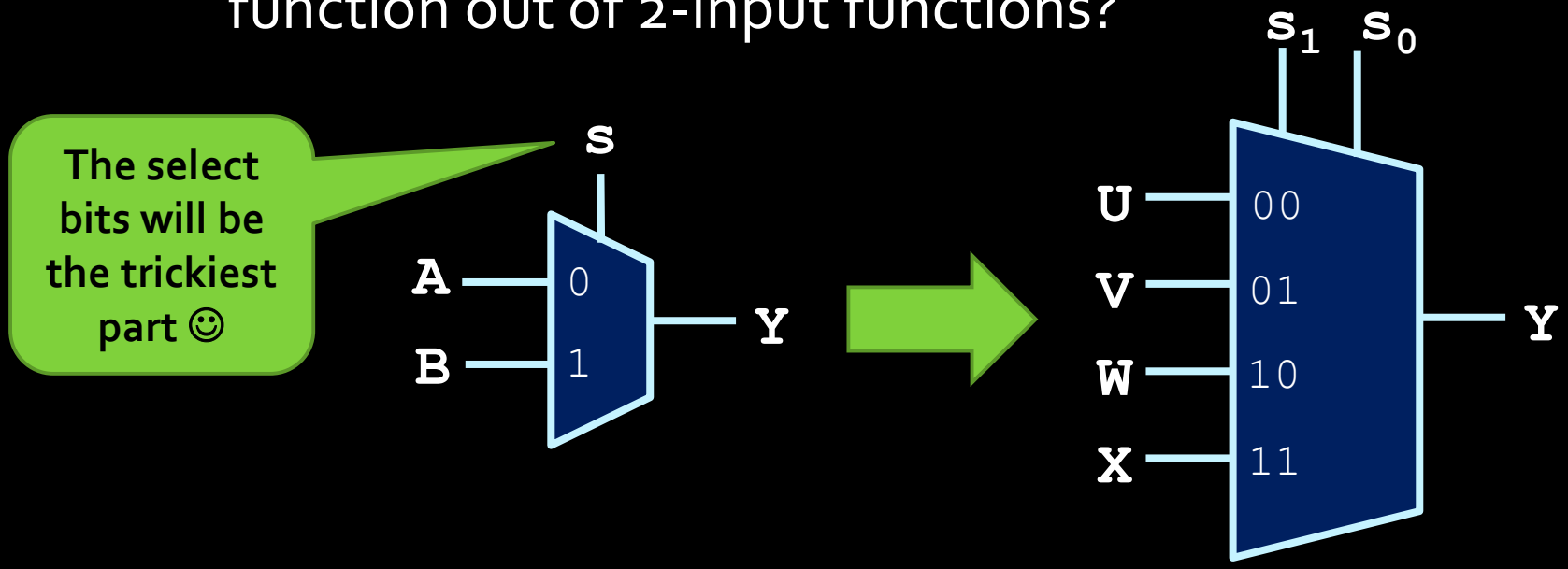


s_0	Y
0	A
1	B

s_1	s_0	Y
0	0	U
0	1	V
1	0	W
1	1	X

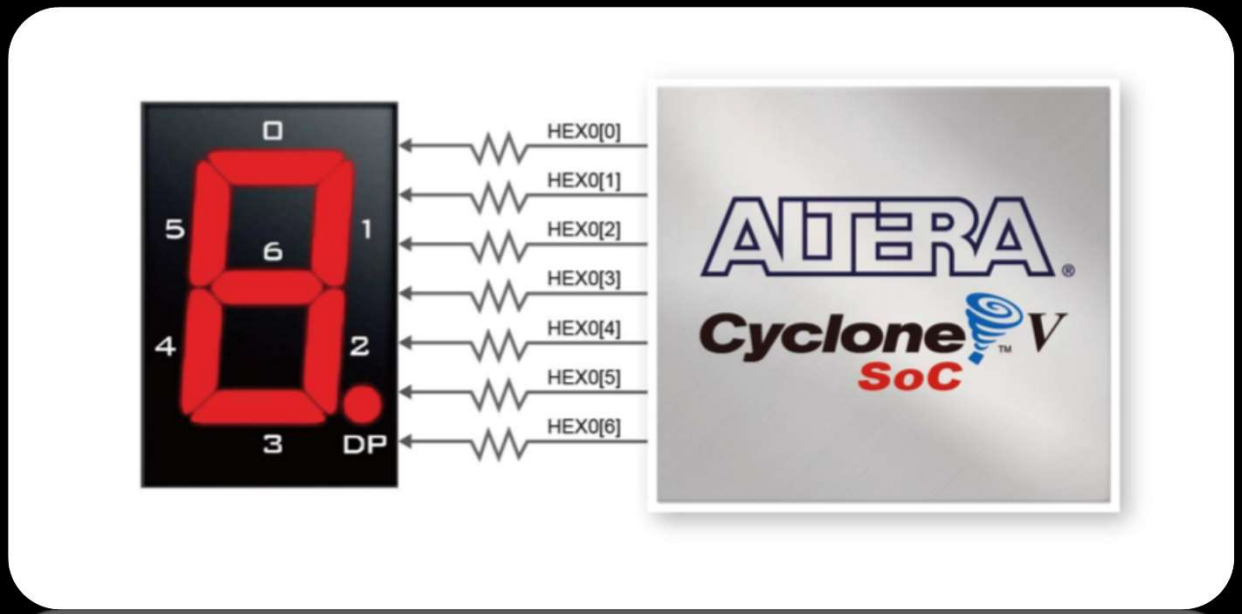
Tasks for Lab 2

- Part II: Designing with modules.
  - If each 2-to-1 mux can handle 2 inputs, how to build something that handles 4?
  - How would you make a 4-input function out of 2-input functions?



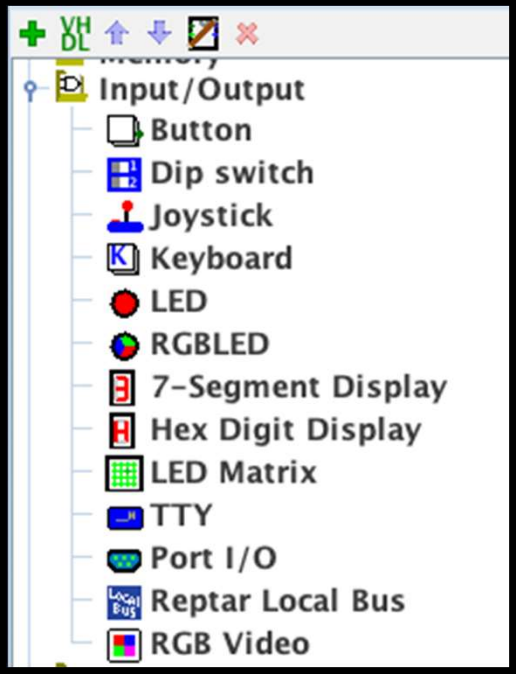
Tasks for Lab 2

- Part III: The 7-segment decoder.
  - This is one of the components in the Logisim toolkit.
  - Also one of the components on the DE1-SOC board!



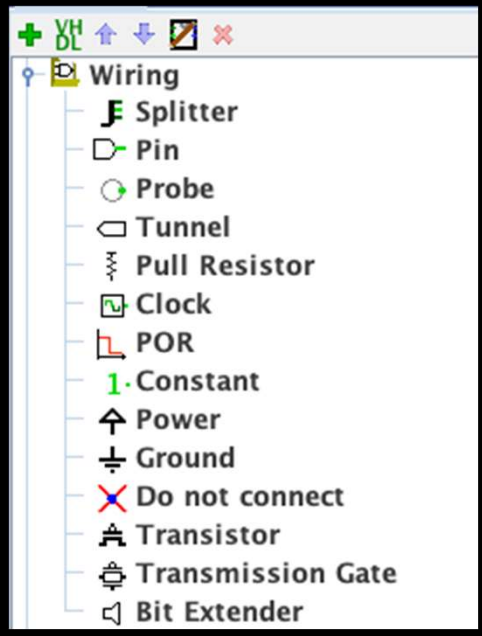
Exploring Logisim Components

- This components and others are listed under Input/Output, for future reference:
  - Button: Can be mapped to the switches and buttons on the DE1 board. Only outputs a 1 when held down with Poke.
  - 7-Segment Display: Can be mapped to the 7-segment display on the DE1 board.
  - LED: Can be mapped to the outputs on the DE1 board.
- Note: always start with the default input/output type from the tool bar and only switch to the above if necessary



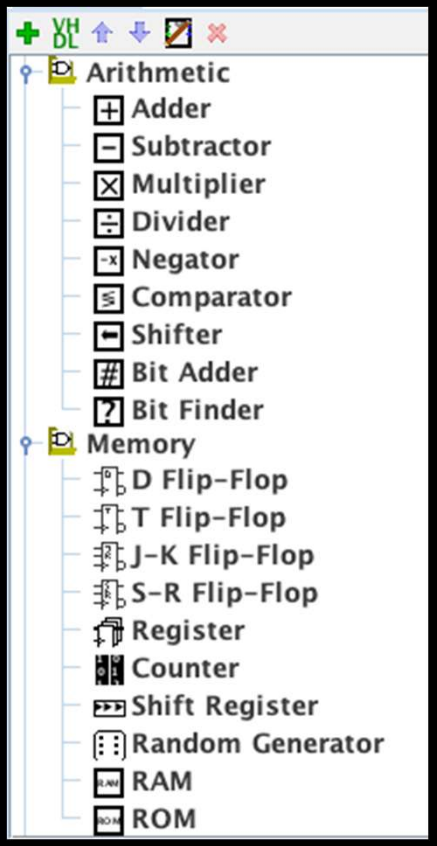
Useful Components in Logisim

- Wiring:
  - Splitter: Splits buses into individual wires or smaller buses. Works both ways.
  - Clock
  - Constant: Outputs a constant value (can be multiple bits on a bus).
  - Bit Extender: Pads or sign extends bits on a bus.
- You can even make a transistor circuit with the components at the bottom ☺

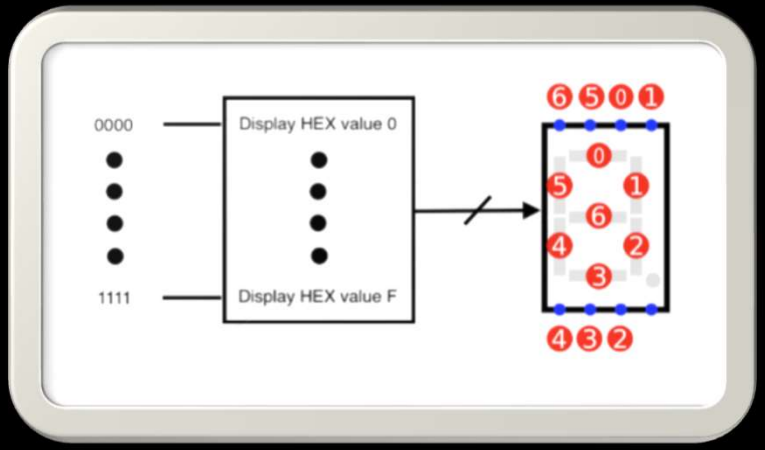


Useful Components in Logisim

- Arithmetic and memory:
  - Some of the arithmetic components will be useful in later labs. Details about each one can be found i:n http://www.cburch.com/logisim/docs/2.3.0/libs/arith/index.html
    - This doc is for an earlier version, some components may look different now.
  - We will be exploring the memory components later in the course.



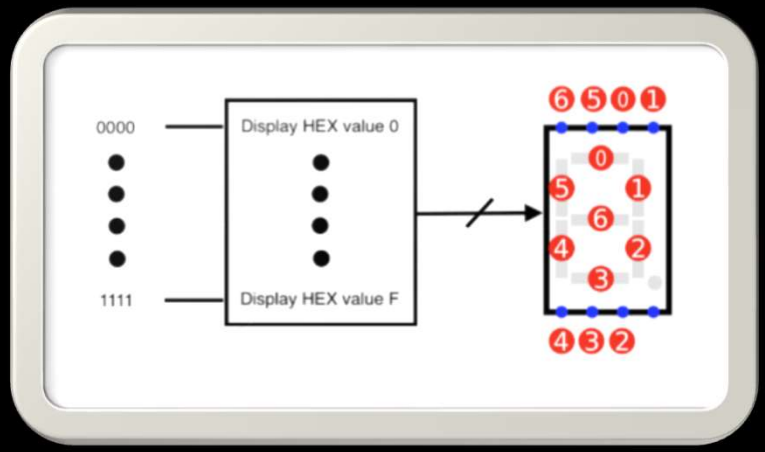
Tasks for Lab 2

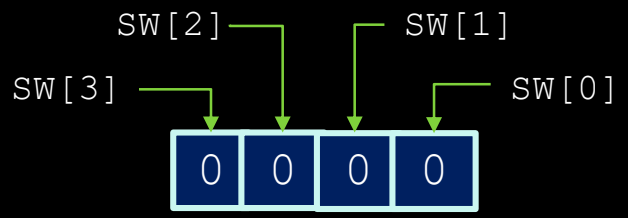


- The diagram on the right illustrates how to use the inputs to the 7-segment decoder (or the HEX decoder) to activate the segments.
- Each segment is active-high, meaning that if you set an input to `1`, the corresponding segment will turn on.
  - The DE1-SOC board is the opposite (i.e. active-low)

Tasks for Lab 2

- Ultimate goal:
  1. Take a 4-bit input coming from the switches on the and interpret those as binary number:

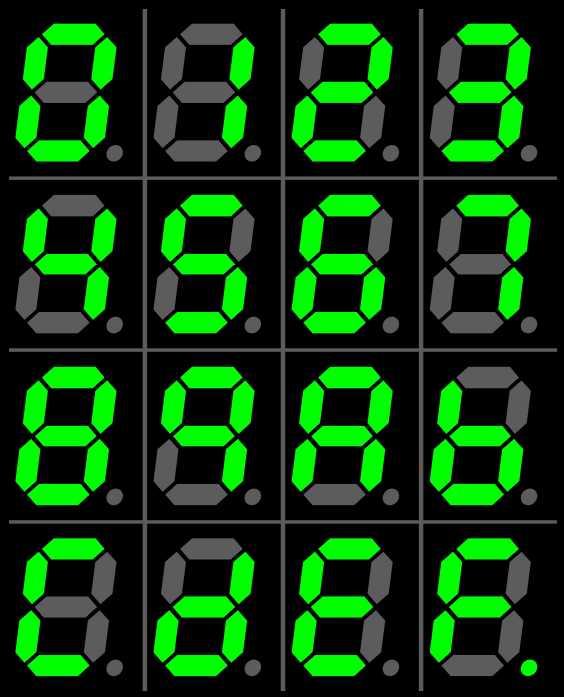




  2. Create seven circuits, one for each segment on the right to activate each segment based on the 4-bit input values.
     - For example: If input is `0000`, display "`0`" on the segments. If input is `1111`, display "`F`" on the segments.

Activating 7-seg displays

- The diagram on the right illustrates the 16 digits we want to show on the 7-segment display.



- How do we make this happen?
  - Consider segment 0 (the top segment in each digit).
  - Need to set it high in the following input cases:

Input		Display
0000	--	"0"
0010	--	"2"
0011	--	"3"
0101	--	"5"
0110	--	"6"
0111	--	"7"
1000	--	"8"
1001	--	"9"
1010	--	"A"
1100	--	"C"
1110	--	"E"
1111	--	"F"

- How do we express this?

Activating 7-seg displays

- Answer: Karnaugh Maps!

	$\overline{SW1}*\overline{SW0}$	$\overline{SW1}*SW0$	$SW1*SW0$	$SW1*\overline{SW0}$
$\overline{SW3}*\overline{SW2}$				
$\overline{SW3}*SW2$				
$SW3*SW2$				
$SW3*\overline{SW2}$				

- If we can fill in these table values, we can figure out the circuit's behaviour.

Segment 0 truth table

SW3	SW2	SW1	SW0	HEX[0]
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Segment 0 Karnaugh Map

- Now to fill in the table below….

	$\overline{SW1}*\overline{SW0}$	$\overline{SW1}*SW0$	$SW1*SW0$	$SW1*\overline{SW0}$
$\overline{SW3}*\overline{SW2}$	1	0	1	1
$\overline{SW3}*SW2$	0	1	1	1
$SW3*SW2$	1	0	1	1
$SW3*\overline{SW2}$	1	1	0	1

- What are the groupings that you see here?
  - Yes, overlapping is allowed ☺

Segment 0 Karnaugh Map

- What are the equations for these groups?

	$\overline{SW1}*\overline{SW0}$	$\overline{SW1}*SW0$	$SW1*SW0$	$SW1*\overline{SW0}$
$\overline{SW3}*\overline{SW2}$	1	0	1	1
$\overline{SW3}*SW2$	0	1	1	1
$SW3*SW2$	1	0	1	1
$SW3*\overline{SW2}$	1	1	0	1

SW3*SW2*SW1

SW2*SW1

SW3*SW2*SW0

SW3*SW1

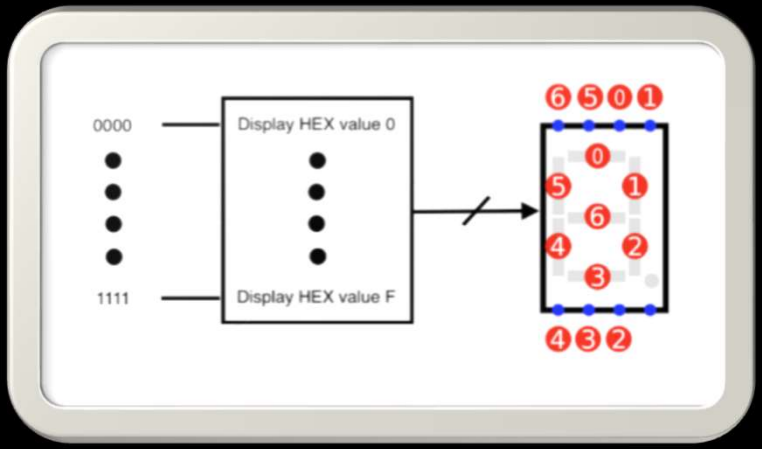
SW3*SW2*SW0

SW3*SW0

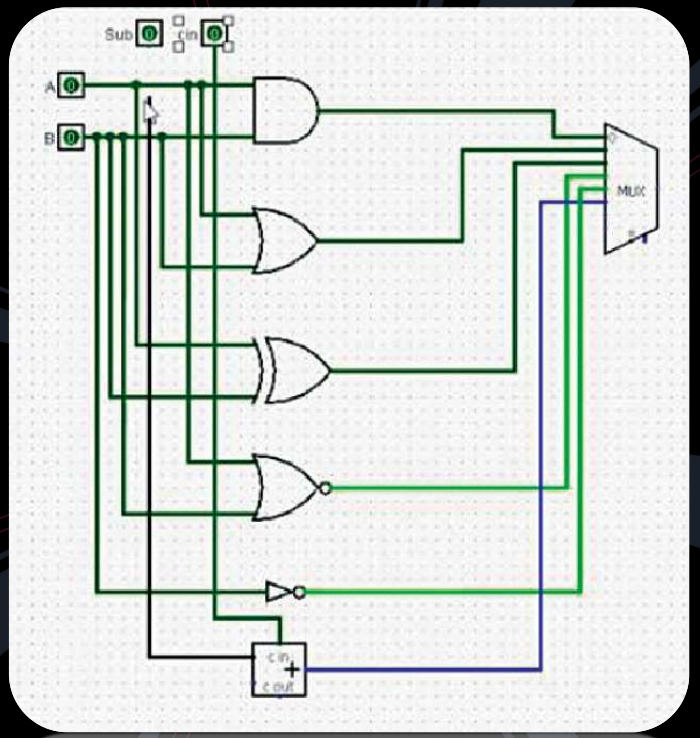
Can you figure out which terms are inverted to make these groups work?

Tasks for Lab 2

- Repeat this process seven times to implement the behaviour for each of the seven segments in the HEX display.
  - Try to get the minimal circuit for each!
- Once you're done, your seven circuits go into the decoder module in the middle of the diagram above.
  - Make sure to test each segment as you go!

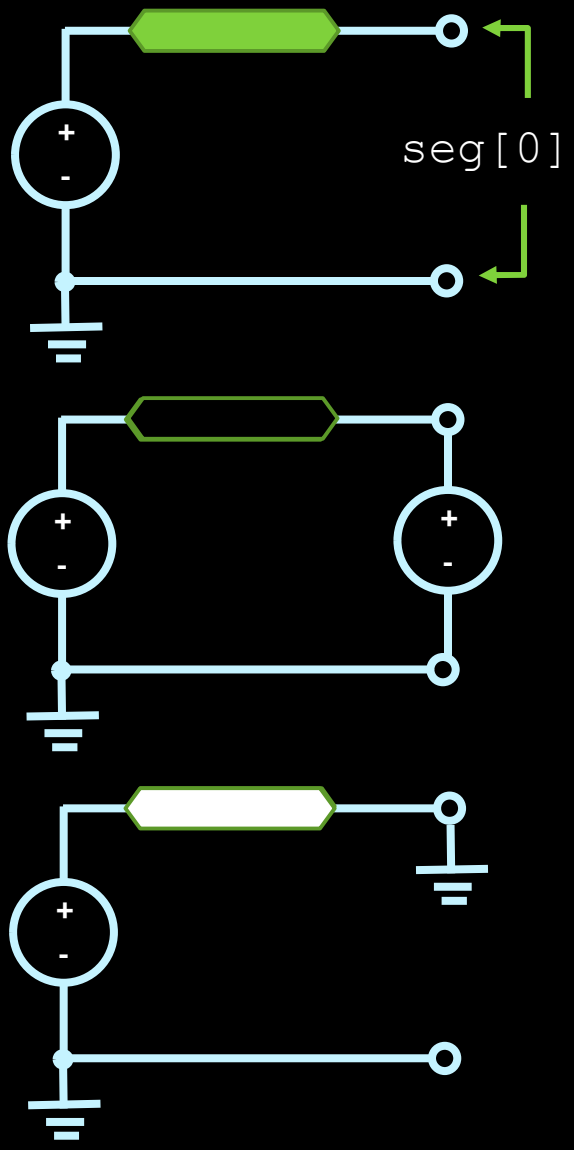


Using Logisim with DE1-SOC



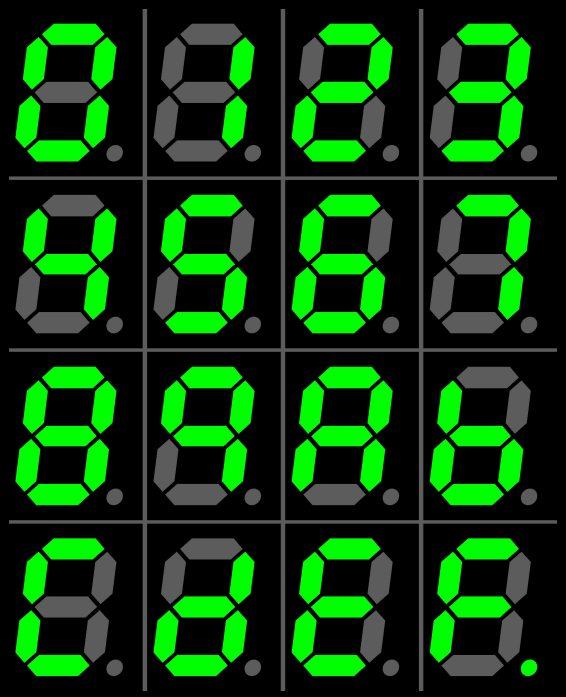
How 7-segments work on DE1

- The default for 7-segments on the DE1 boards are "active low", meaning that they turn on when their input signal is `0`, not `1`.
  - If you set segment `0` high, there is no voltage drop across the segment, so it doesn't turn on.
  - If you set segment `0` low, the voltage drop across the segment makes current flow, causing it to turn on.
- The default for 7-segments in Logisim are "active high" therefore you need to change this in properties.



Activating 7-seg displays

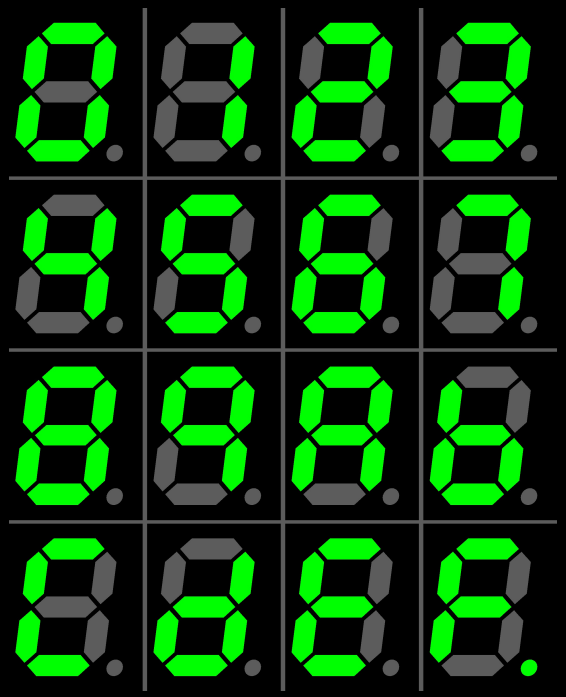
- Need to set segment `0` (top segment) **low** in these input cases instead:
  - `0000 -- "0"`
  - `0010 -- "2"`
  - `0011 -- "3"`
  - `0101 -- "5"`
  - `0110 -- "6"`
  - `0111 -- "7"`
  - `1000 -- "8"`
  - `1001 -- "9"`
  - `1010 -- "A"`
  - `1100 -- "C"`
  - `1110 -- "E"`
  - `1111 -- "F"`



- How do we express this?

Activating HEX displays

- Could also set segment 0 (top segment) **high** in the other input cases:
  - 0001 -- "1"
  - 0100 -- "4"
  - 1011 -- "B"
  - 1101 -- "D"



- Can be expressed as a four-part Boolean expression:

```
HEX[0] = ~SW[3] & ~SW[2] & ~SW[1] & SW[0] |
         ~SW[3] & SW[2] & ~SW[1] & ~SW[0] |
         SW[3] & ~SW[2] & SW[1] & SW[0] |
         SW[3] & SW[2] & ~SW[1] & SW[0];
```

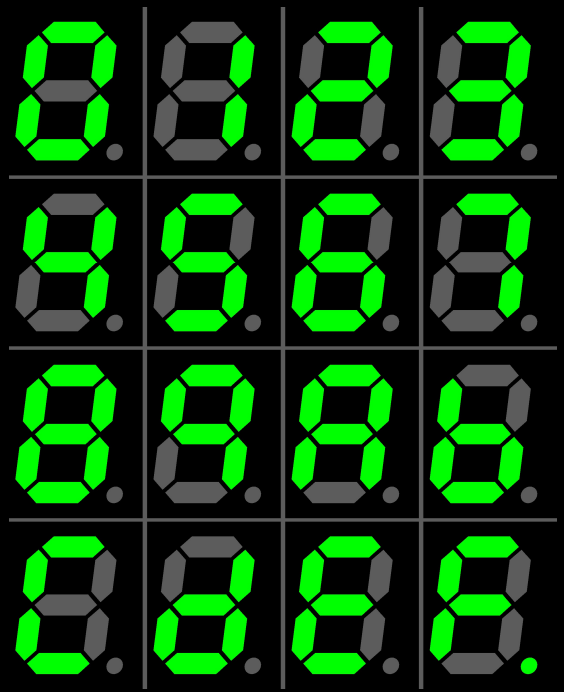
Activating HEX displays

```
HEX[0] = ~SW[3] & ~SW[2] & ~SW[1] & SW[0] |
         ~SW[3] & SW[2] & ~SW[1] & ~SW[0] |
         SW[3] & ~SW[2] & SW[1] & SW[0] |
         SW[3] & SW[2] & ~SW[1] & SW[0];
```

- Can this be reduced any further?
  - …sadly, no ☹
- How do we know?
  - Karnaugh maps!

Activating HEX displays

- Can you write the expressions for HEX[1] to HEX[6]?
- Can you reduce these expressions to the simplest gate form?



Lab 3 Preparation

What this lab requires

- More practice with Logisim, and explore some new components.
  - Arithmetic operators!
- Implementing some logical devices.
  - Full adder circuit, ALU.
- Learning some hierarchical design.
  - Mux + adders = ALU.

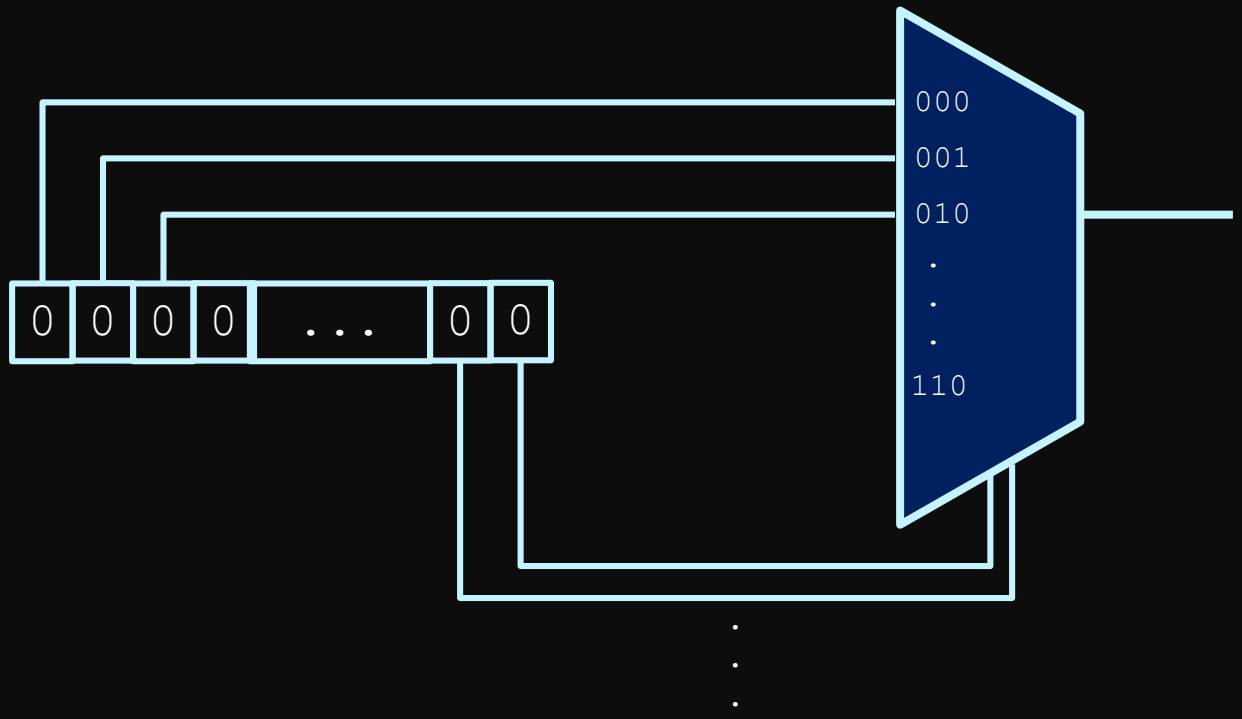
Part 1: Mux + Splitter

- In components, you can find mux under the category Plexer. Just drag and drop on your canvas.

- Key points:
  - In Properties of the multiplexer:
    - Adjust the number of select bits to change the size of the multiplexer.
    - Adjust the number of data bits to match the size of the input data.

Part 1: Mux + Splitter

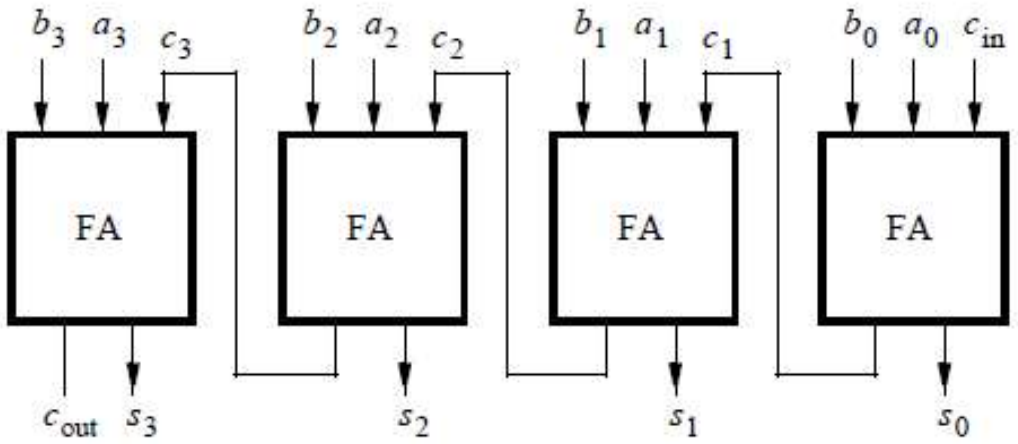
- What you're supposed to do:
  - Multi-bit input into 7-to-1 mux.



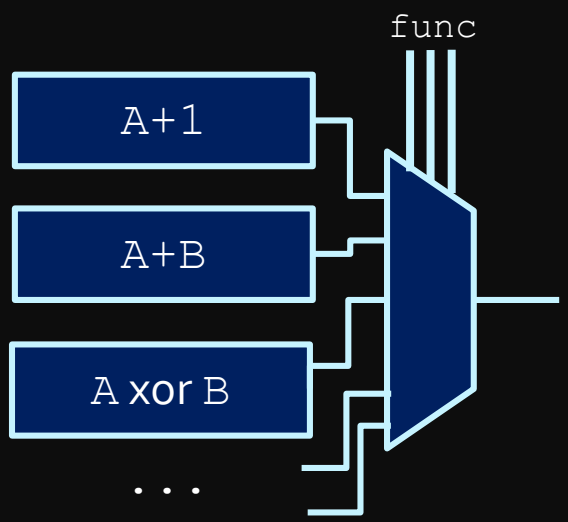
Part 2: Ripple Carry Adders

- Implement a Ripple Carry Adder by connecting (chaining) four full-adders together.
  - Must use hierarchical design!

b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



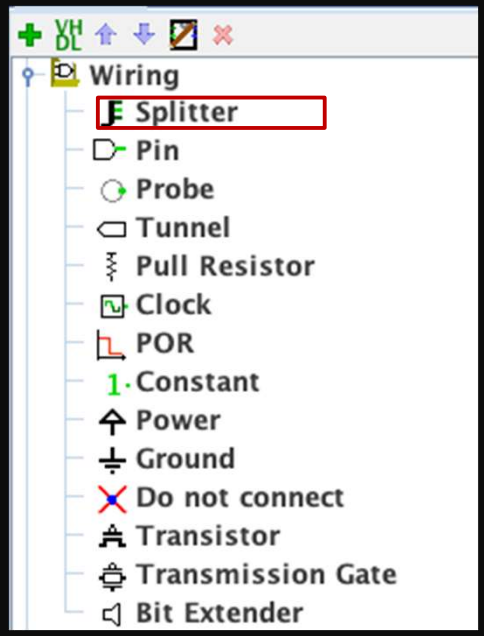
Part 3: ALU



- The ALU (Arithmetic Logic Unit) uses a mux to choose a single output value from a series of modules.
  - Each of these multiplexer inputs is connected to a module that performs a single arithmetic or logical operation.
- Once each operation module has been implemented with general inputs `A` and `B`, connect these ALU inputs to the switches and the ALU outputs to both the LEDs and 7-segment display (in hexadecimal).
  - Try to you reuse the modules you made in Lab 2.
  - The handout includes syntax for some new (and potentially useful) operators

Splitter in Logisim

- Splitter can be very useful in Logisim. You can find them under Wiring in components.
- They can be used in two ways:
  - Separating a multi-bit signal,
  - Concatenating signals together.



Bitwise Operations

- Bitwise Operations
  - If you use a bitwise operation with two n-bit operands, the result is also an n-bit vector.
  - For example:

    0101 & 0011 → 0001

    - More general mathematical notation:
    - ($X_{n-1}X_{n-2}..X_1X_0$ & $Y_{n-1}Y_{n-2}..Y_1Y_0$) results in $W_{n-1}W_{n-2}..W_1W_0$ where W_i is (X_i & Y_i) for every i in $[0, n-1]$.
- In Logisim, this can be achieved using Splitter.
  - For a bitwise AND:
    - Split the signal into separate bits,
    - Use AND gates on each pair of bits,
    - Concatenate the result bits together

Reduction Operators

- Reduction Operations have the same approach as bitwise operations, but
  - They take a single multi-bit operand, and
  - The result is a single-bit output.

- In Logisim, we achieve this using splitters as well.
  - For reductive AND:
    - Split the signal into individual bits
    - Connect these bits to the inputs of a single AND gate

Replication and Concatenation

- The binary value `011` (3 in decimal) is the same as `0011` or `0000000011`.
  - Adding zeros in the most significant bits of a positive or an unsigned number does NOT change the number being represented!
  - Example:
    - If the output of a module is 3-bits and you want to feed it to a 5-bit input of another module, you'd need to use both replication & concatenation!
- In Logisim, this is achieved using Bit Extender under Wiring in components.

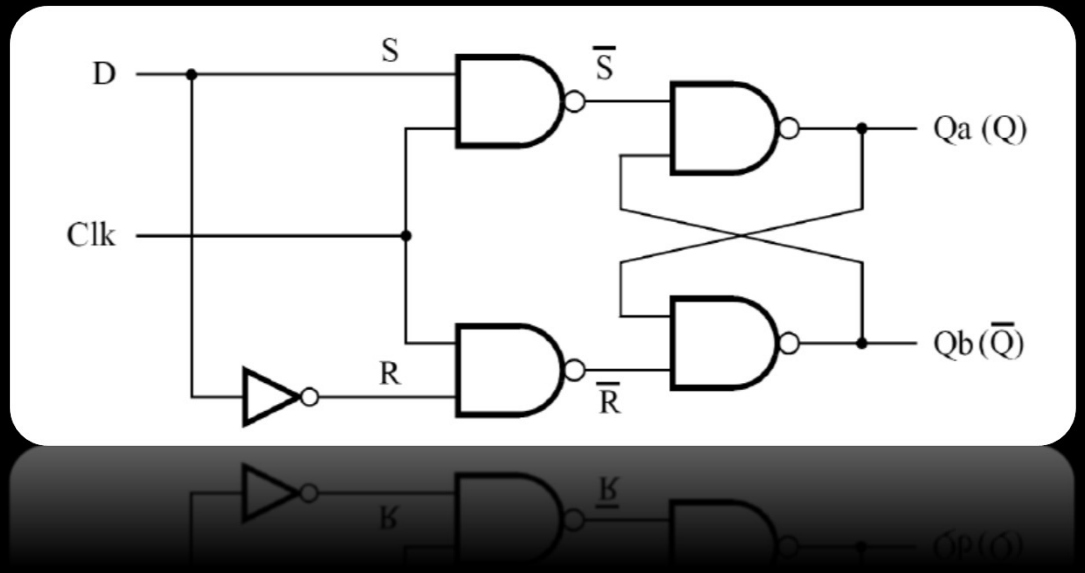
Lab 4 Preparation

What Lab 4 is about

- Seeing how latches and flip-flops work
- Using a register to store & provide values
  - Store output of ALU
  - Provide input to ALU
- Implement a shift register.

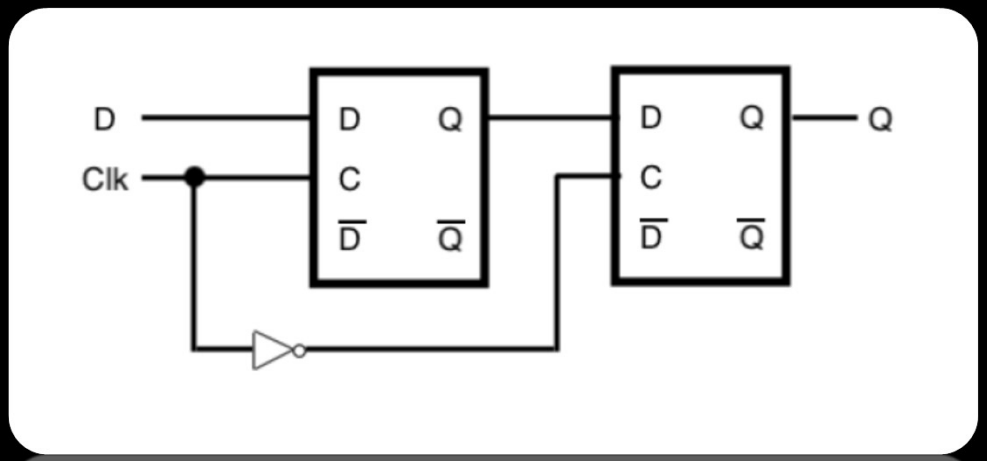
Lab 4 – Part I

- Create a D latch out of NAND gates!



Lab 4 – Part I

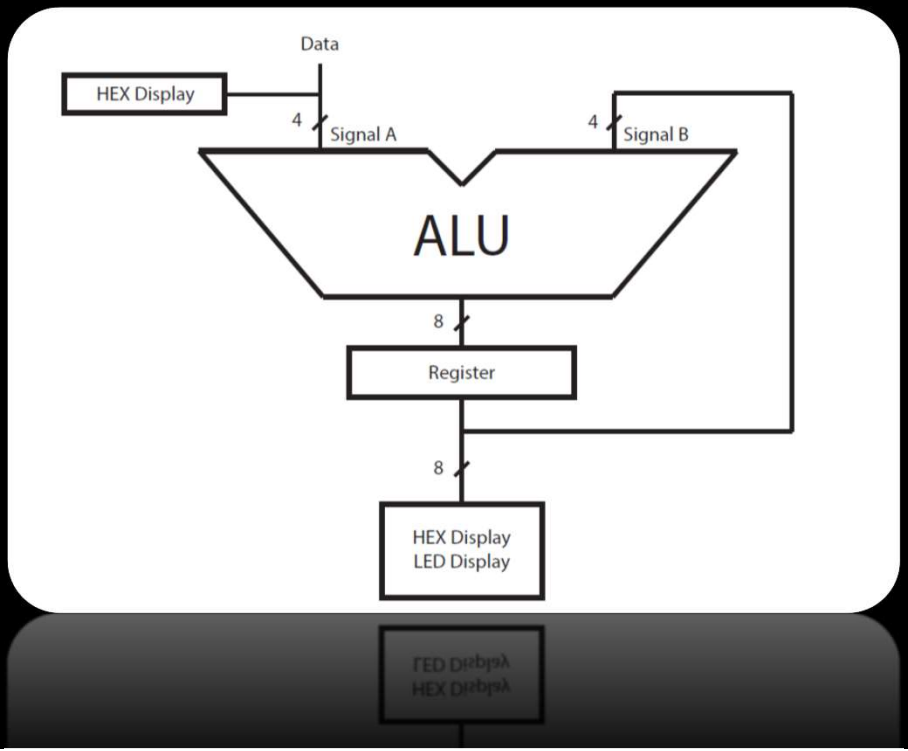
- …then, create a D flip-flop out of D latches ☺



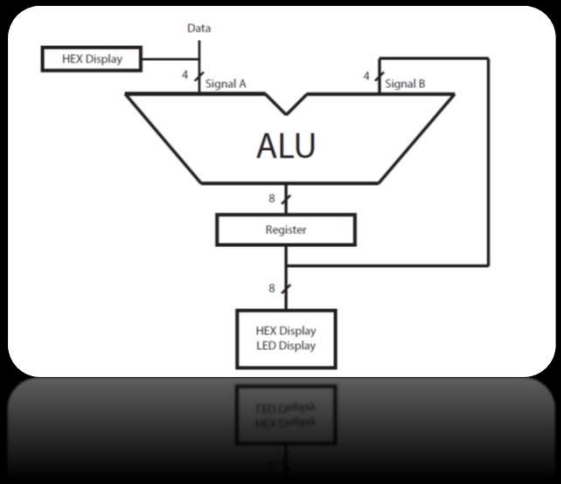
- How are you going to test this?
  - Try using test vectors, and see what happens.

Lab 4 – Part II

- Enhance the ALU from last week:
  - More operations (e.g. multiplication, shifting)
  - Store the result in a register.



Lab 4 – Part II



- Old operations:
  - `0:` `A+1` (using your adder)
  - `1:` `A+B` (using your adder)
  - `2:` `A+B` (using the Logisim adder)
  - `3:` `A xor B` in lower four bits, `A or B` in upper four bits
  - `4:` Reduction `OR` operation on `A` & `B`
- New operations:
  - `5:` Left shift `B` by `A` bits
  - `6:` Logical right shift `B` by `A` bits
  - `7:` `AxB` (multiplication)

All of these are supplied through the collection of components in Logisim.

What does it mean to shift?

- Suppose that B = `00010110` and
  A = `00000011`
- "Left shift B by A bits" = shift every bit in B three bits to the left (since A = 3).
  - One bit to the left: B → `00101100`
  - Two bits to the left: B → `01011000`
  - Three bits to the left: B → `10110000`
- One bit gets shifted off the left side each time, and a zero is shifted into the right.

Logic vs Arithmetic Shift

- Function 6 of the ALU is a logical right shift.
  - This is the same as the left shift, but to the right.
- What does the word "logical" signify?
  - `B` is storing binary values, but not a number.
  - When shifting right, shift in zeroes on the left side.
- What if `B` is storing a binary number?
  - Then you perform an arithmetic right shift.

Logic vs Arithmetic Shift

- Arithmetic right shifts replicate the sign bit instead of using zero to fill in the most-significant bit(s).
  - Used when shifting signed numbers.
  - For unsigned numbers, use logical shift (like in Part 2).

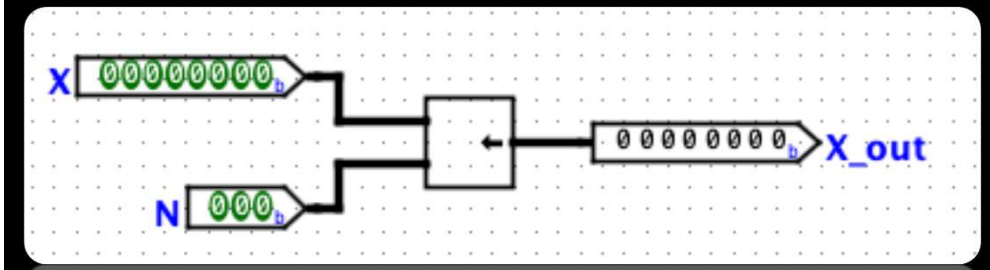
- Example: Shift `10010000` right by 3 bits
  - Arithmetic → **`111`**`10010`
  - Logical → **`000`**`10010`

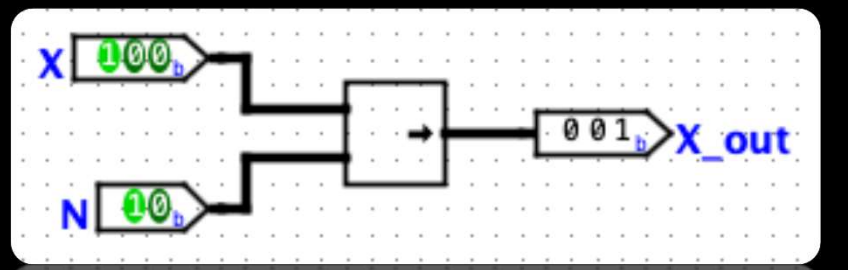
Logic vs Arithmetic Shift

- Why do we shift in the sign bit?
- Shifting B left A bits multiplies B by 2^A.
  - Again, assuming that B = 00010110 (22_{10})
  - If A=1, B→ 00101100 (44_{10})
  - If A=2, B→ 01011000 (88_{10})
- Shifting B right A bits divides B by 2^A, but only if you preserve the sign bit!
  - Suppose that B = 11110110 (-10_{10})
  - Arithmetic shift right: B→ 11111011 (-5_{10})
  - Logical shift right: B→ 01111011 (123_{10})

Logisim Shifter Components

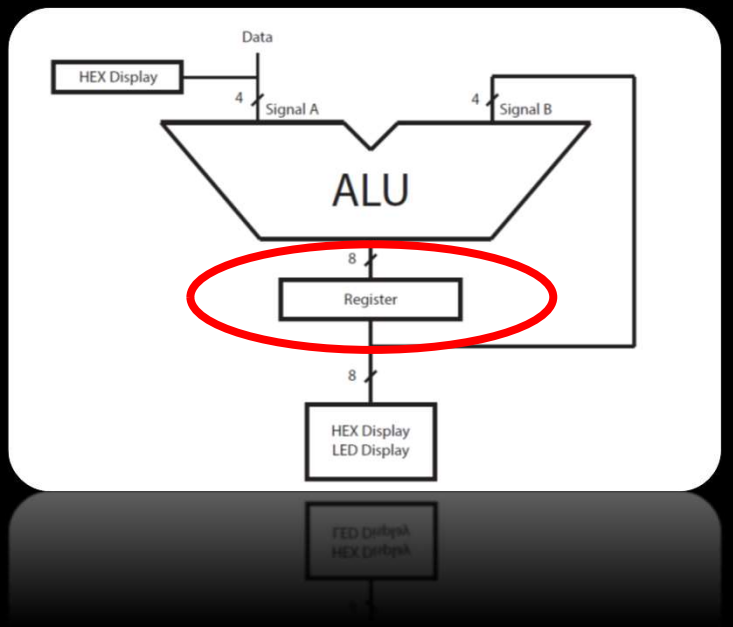
- Getting back to Lab 4, part II…
- Shifter components can be found in Logisim under Arithmetic > Shifter.
  - You can change the shift type in Properties.
  - More details can be found at:
    - http://www.cburch.com/logisim/docs/2.3.0/libs/arith/shifter.html
- Illustrated below: Logical Shift (left and right)





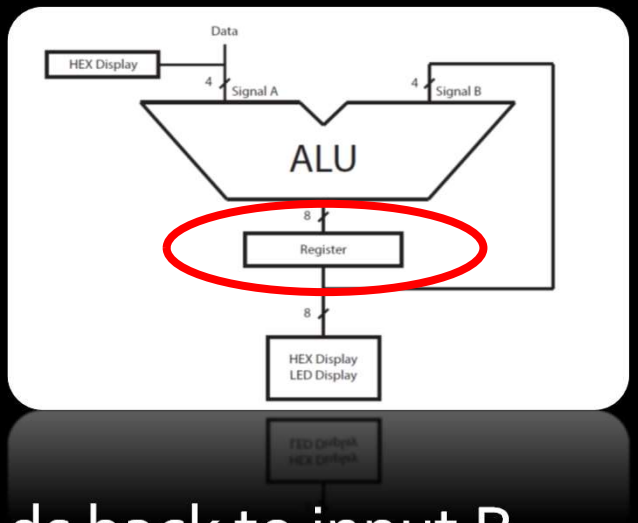
Lab 4 – Part II

- The other major addition to the ALU is the register that stores the output value.
  - 8 bits (flip-flops) long.
  - Compoment found in Memory > Register.
  - You can change the number of flip-flops in Properties > Data bits.



Why is the register here?

- The ALU is a transparent device, meaning that input values are passed straight through to the output (with some small delay).



  - Since the output of the ALU feeds back to input B, without the register the ALU output would change constantly.
  - The register has a clock signal that only updates its contents when the clock changes from `0` to `1`.

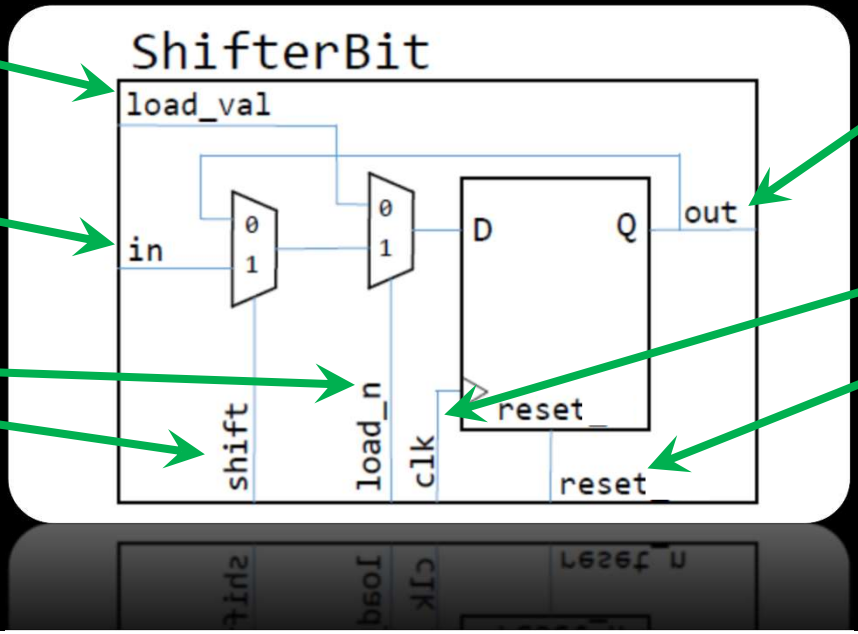
Final notes about Part II

- Make sure that for addition operations, the output preserves the carry bit.
  - For example, if `A=1101` and `B=1011`, the ALU output should be `00011000`.
  - If you didn't do this for Lab 3, make sure to do it for Lab 4 ☺
- The input B to the ALU is the least significant bits of the ALU output.
- Remember to display the inputs and outputs on seven-segment displays (and outputs on LEDR)

Lab 4 – Part III

- Make an 8-bit shift register out of 8 individual shifter units (each unit stores a single bit)
  - Similar to how the ripple carry adder was created.
- Below: the diagram for a single shifter unit.

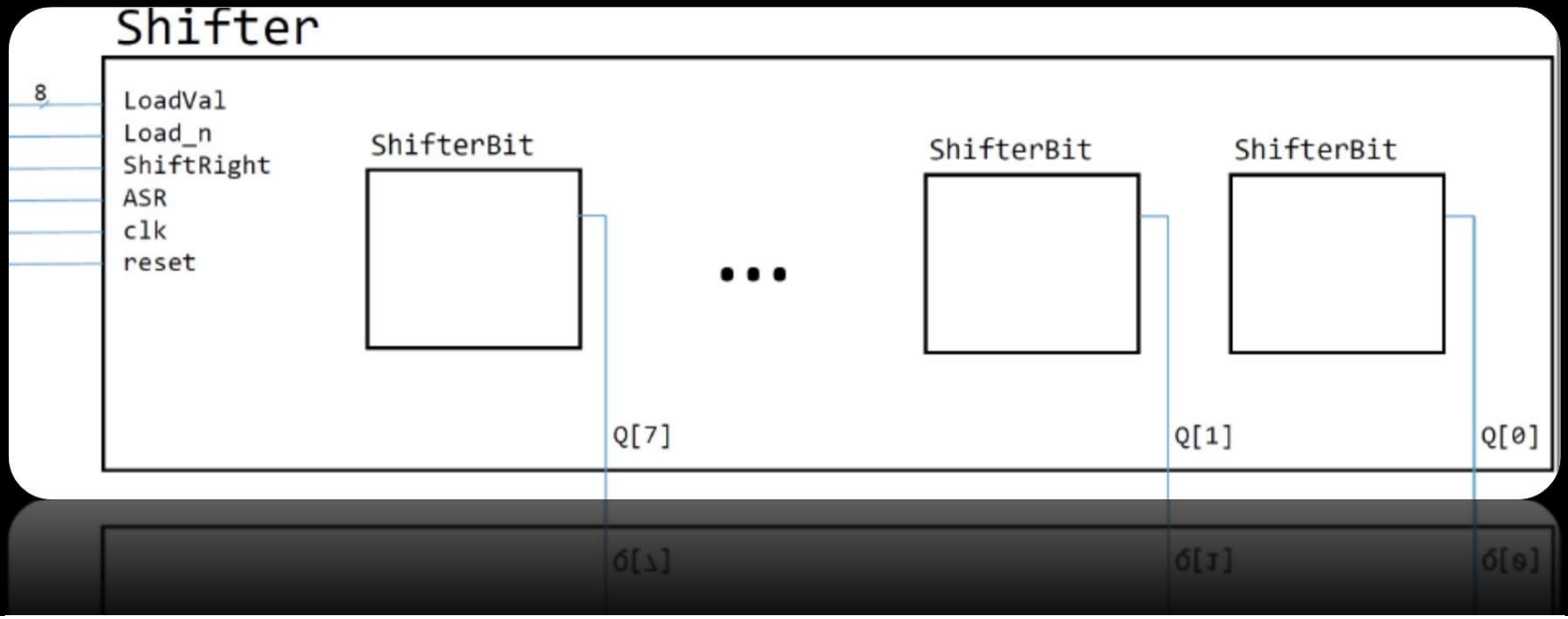
load_val: external value to load into register.
in: input value coming from next shifter unit
shift, **load_n**: are we doing a shift or load operation?



out: outputs the contents of this shifter unit.
clk: external clock signal for register.
reset: clear unit contents to zero.

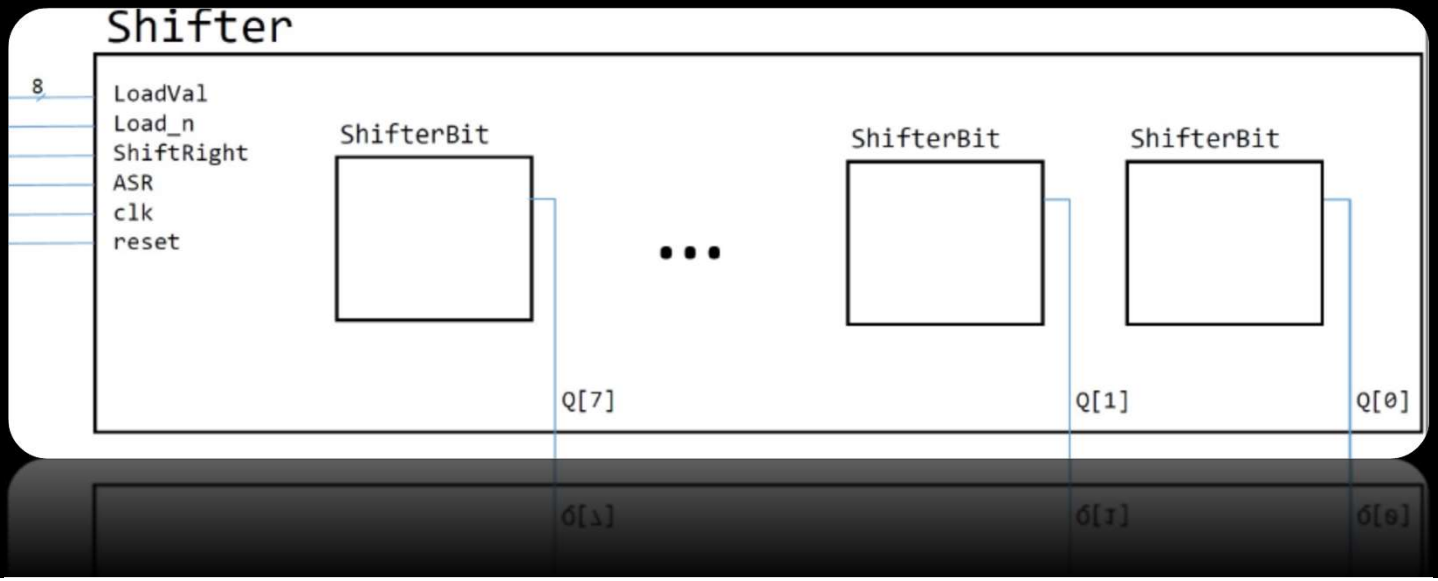
Lab 4 – Part III

- Once you've made a module for the shifter unit, connect 8 shifter units together to make the shift register.
- Note:
  - This shift register only shifts to the right.
  - The shift register can load a new value from the 8 input bits in `LoadVal` (only happens when `Load_n` is ***low***)



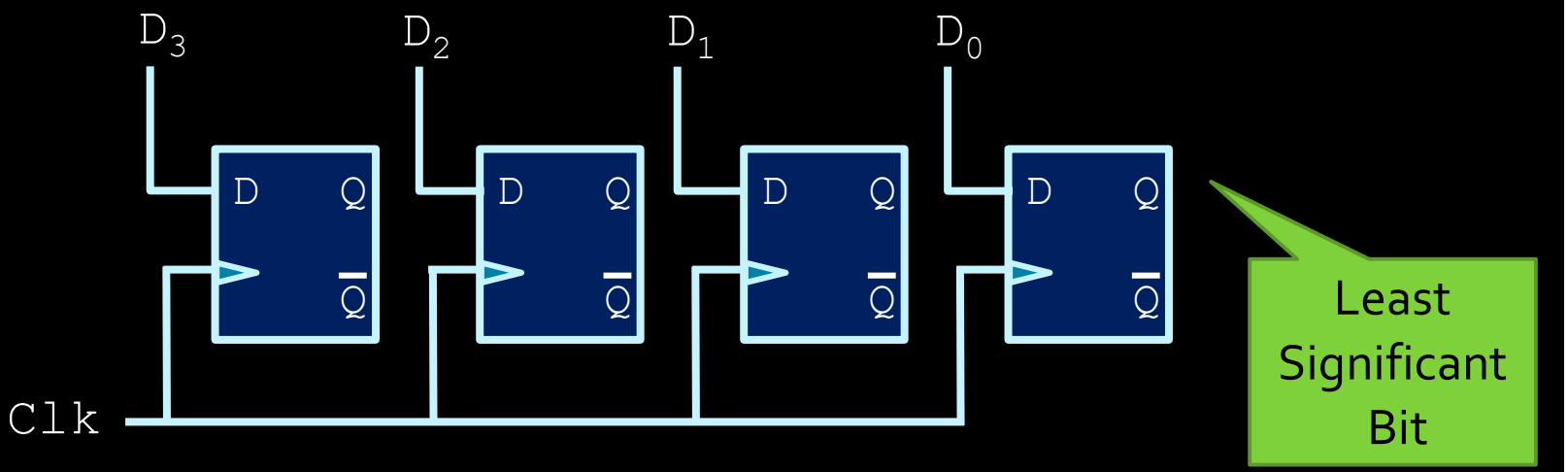
Lab 4 – Part III

- Shifter signals:
  - `Load_n` signals the ShifterBit units to load a new value from the `LoadVal` input (that you provide)
  - `clk` (the clock signal) signals when the ShifterBit contents should change (both shifting and loading operations).
  - `reset` is asynchronous (independent of the `clk` signal).
  - `ShiftRight` tells you to shift, `ASR` tells you what kind of shift to do (`0`=logical, `1`=arithmetic).



Load register

- N-bit number = N D-flipflops synchronized to the same clock signal
- You can load a register's value (all bits at once), by feeding signals into each flip-flop:
  - In this example: a 4-bit load register.

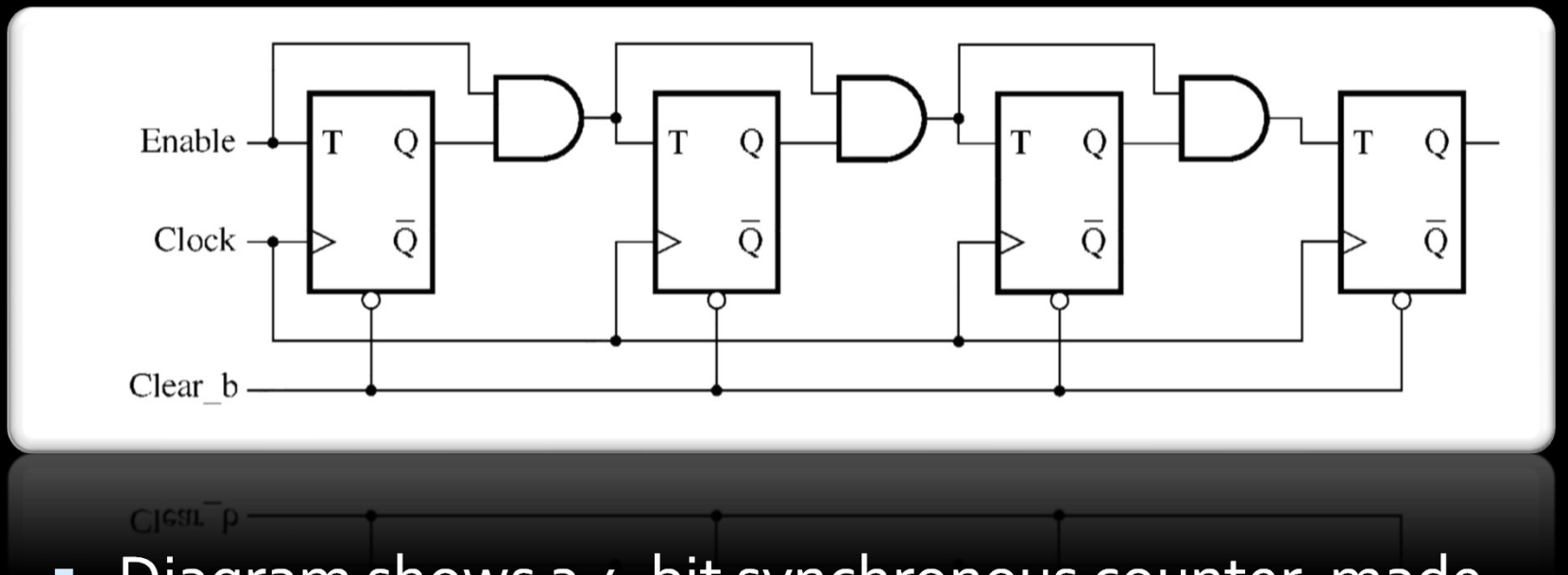


Lab 5 Preparation

Lab 5 Components

- Part I: Create a counter
  - Use the synchronized counter circuit described in lectures.
- Part II: Slow down the clock
  - Use the counter and the on-board clock to create a slower clock.
- Part III: Morse code decoder
  - Decode incoming Morse Code signals!
  - Uses code from Part II ☺

Part I – 4-bit Counter



- Diagram shows a 4-bit synchronous counter, made with T-flip-flops
  - The T flip-flops here have an active-low asynchronous reset (`Clear_b`).
- Need to use hierarchical design to make an 8-bit counter.

Part I (continued)

- Prelab parts:
  - Draw and label 8-bit counter schematic.
  - Build your circuits for flip-flop and counter
  - Simulate your counter to confirm correctness.
  - Mapping your input/output

Part II: More Counters

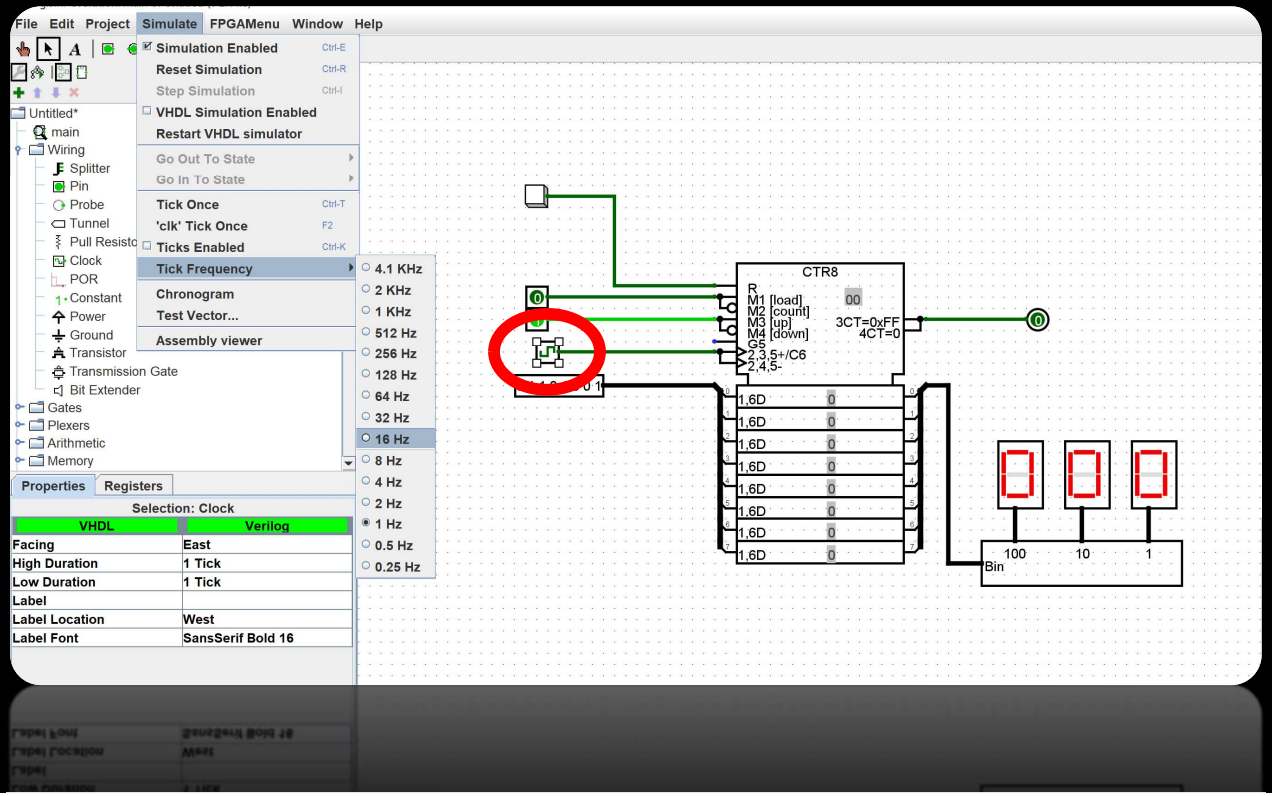
- Main goal:
  - Display incrementing digits on a hex display at different speeds (e.g., once per second, twice per second, etc)
- How do we do this?
  1. Need access to a clock signal.
  2. Need to adjust this clock signal to the right speed.
  3. Need to increment a counter at this adjusted clock speed.

Part II (continued)

- Step #1: Finding a clock signal
  - The DE1-SoC has a 50MHz clock
    - For Logisim, we are using the 16 Hz clock signal.
  - Hertz (Hz) => number of cycles per second
    - 50MHz => How many clock cycles per second?
    - Would you be able to see that?
  - In Logisim, Clocks are under Wiring in components. Note that you should only use this clock for the top level of your design and for the modules being extended it needs to be the default input type.

The clock signal

- Clock signals (from the Wiring menu).
  - Adjust the clock frequency and enable the clock ticking from the Simulate menu.



Part II (continued)

- Step #2: Slowing down the clock.
  - Load a counter with a countdown value (through a parallel load) and produce an enable signal when the countdown reaches zero.

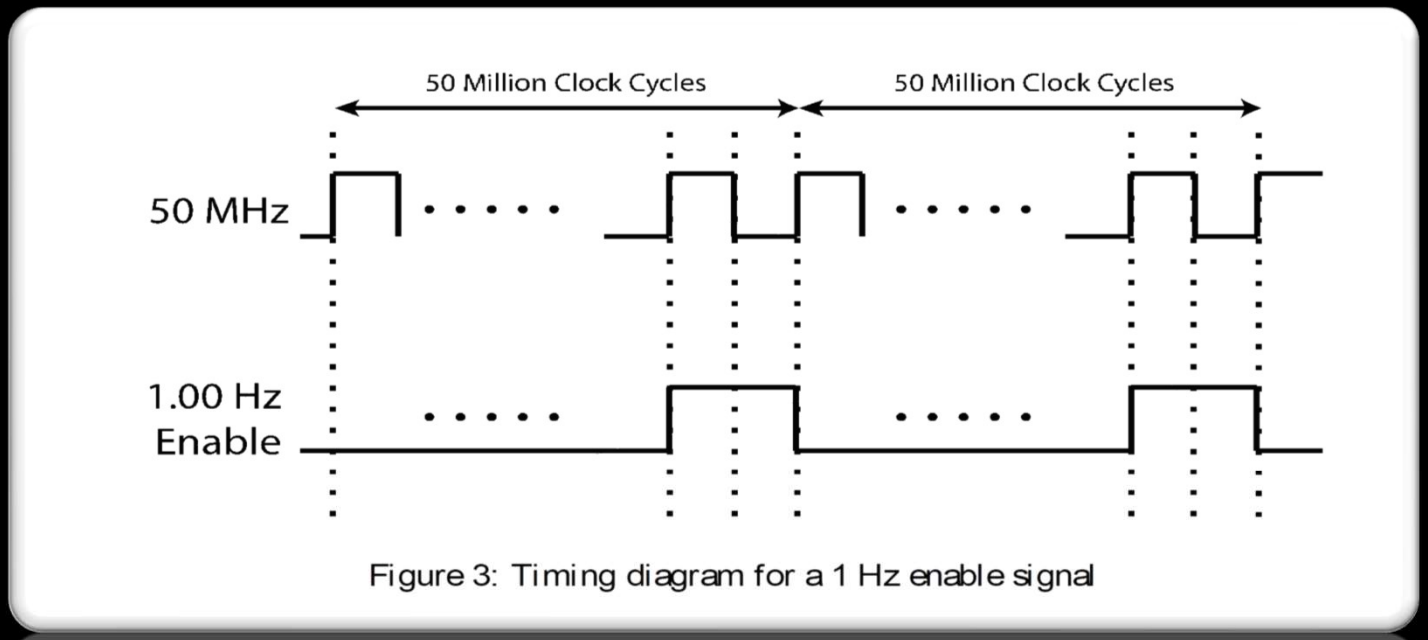
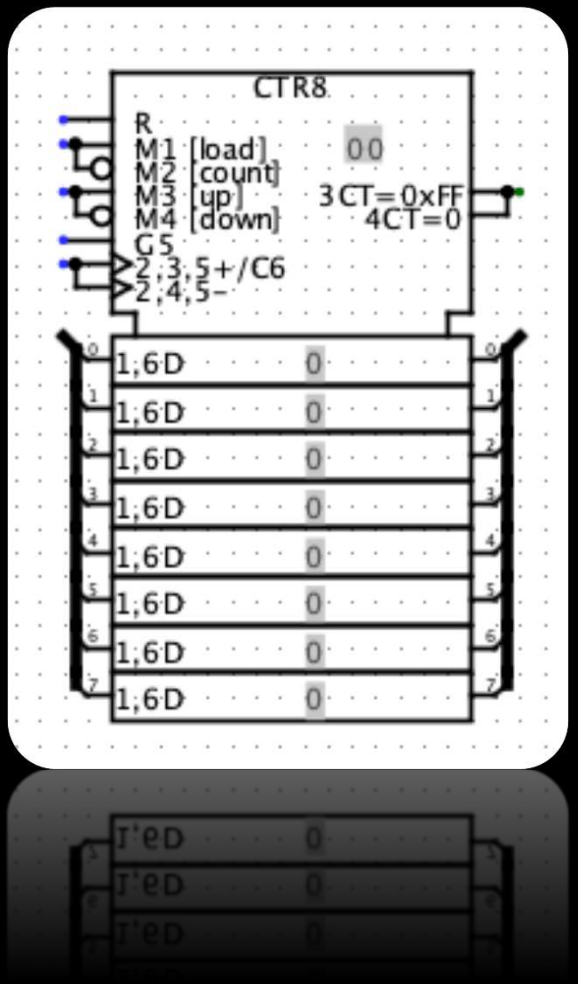


Figure 3: Timing diagram for a 1 Hz enable signal

- Another module will use this enable signal to determine whether it will change on the next clock pulse or not.

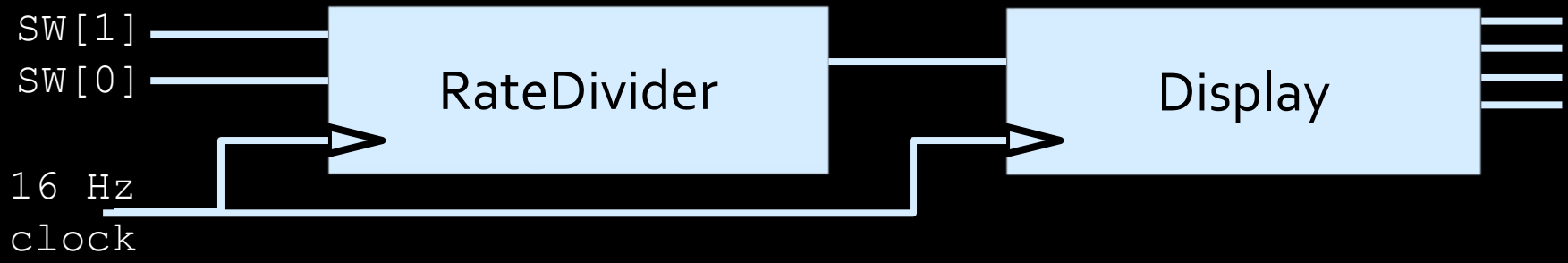
Part II: An Aside

- Counters components can be found under Memory in the component library.
- Details about their operation are in the Lab5 handout and other documentation.
  - Make sure to read carefully and play with it as you read!



Part II (cont'd)

- Step #3: Updating the display counter.
  - You will need 2 counters for this:
    - A `RateDivider` counter (from previous step)
    - A `Display` counter (that feeds to the 7-segment decoder)
    - Both will be synchronized to the same 50MHz clock.



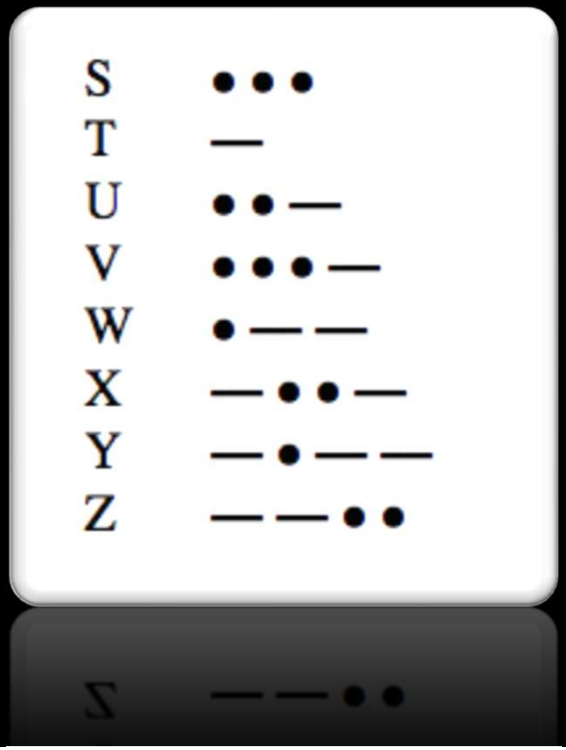
  - Recall the purpose of an `Enable` signal in a counter.
    - How often do you want the `Display` counter to increment?
    - `SW[1]` and `SW[0]` will control that rate.

Part II – Final Thoughts

- The lab handout presents hints on how to implement each of these counters, including a diagram of how to use the Logisim counter to make a new clock signal.
- As long as you use a counter as your base, you have the freedom to implement the design however you like.

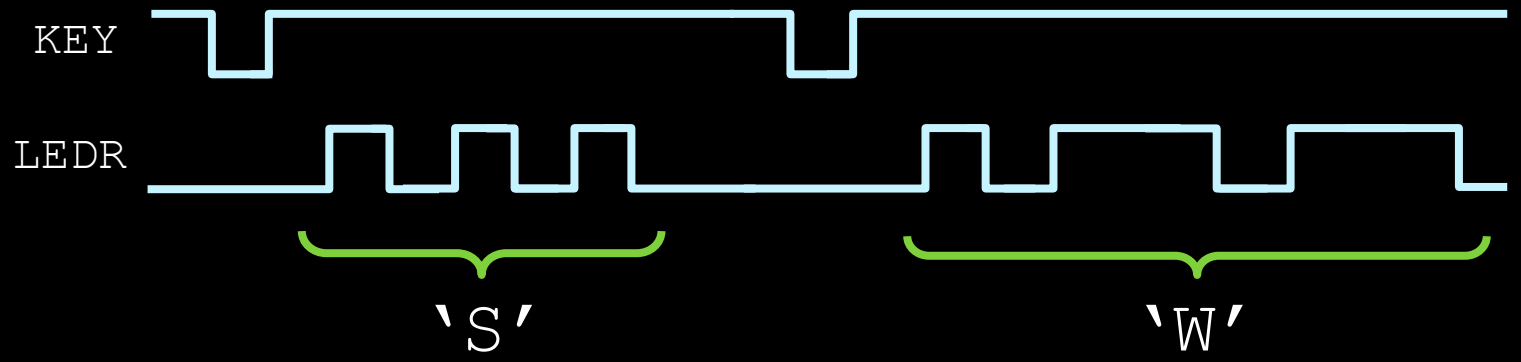
Part III – Morse Code

- Morse Code: "A method of transmitting text information as a series of on-off tones, lights, or clicks that can be directly understood by a skilled listener or observer without special equipment. "



Part III (continued)

- You will be transmitting individual letters using a single red LED
  - Dot => 0.5 seconds LED on
  - Dash => 3 * 0.5 seconds LED on
  - Pause (between symbols or in the end of transmission) => 0.5 seconds LED off



Part III (continued)

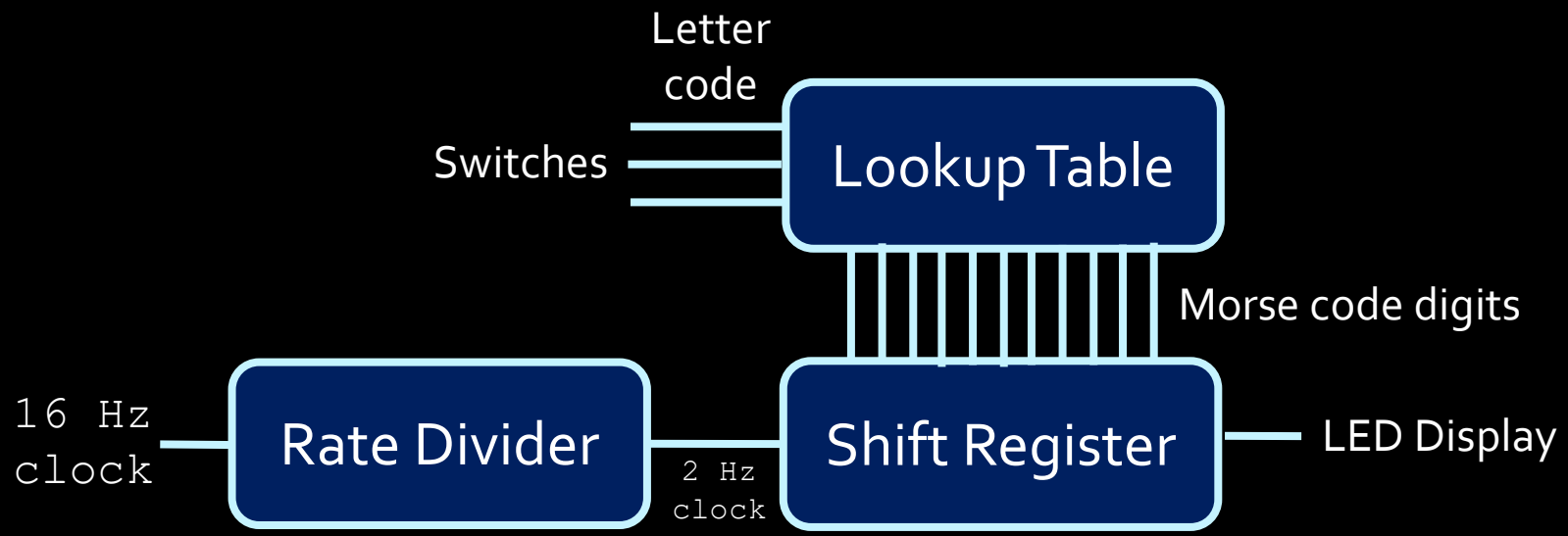
- How do you do this?
  - Step #1: Create a Lookup Table (LUT)
    - Switch values as input, binary representation of the corresponding Morse code letter to transmit as output.
  - Step #2: Create a shift register
    - When a selected input sends a signal, load a shift register with the current value from the lookup table.
    - Shift out a bit on each clock cycle and send it to the LED.

Part III (continued)

- How to decide on the binary representation in the LUT
  - Each bit corresponds to 0.5 seconds of light (`1`) or no light (`0`).
  - Example: • – ("dot dash")
    - Multiple ways this could be represented:
      - `101110`
      - `10001110`
      - `10111000`
      - `100111000000000`
  - All of these look like "dot dash" in the end, so it's up to you (or up to the longest letter you encode)
- How do you make the shift register move bits out every half a second?
  - Rate divider from Part II ☺

Part III (continued)

- How it all looks:



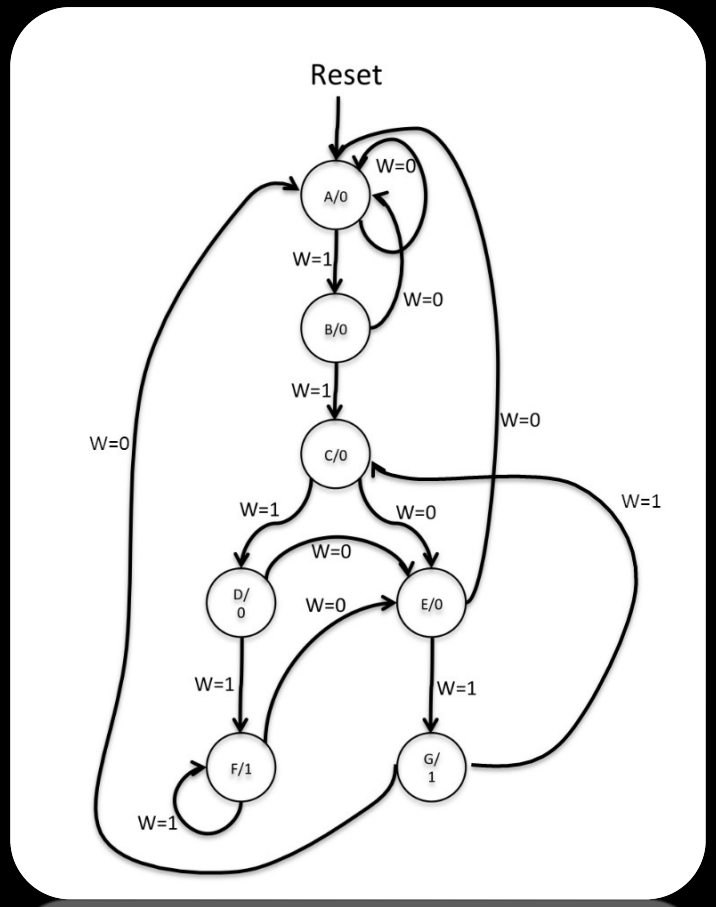
Lab 6 Preparation

Lab 6 Components

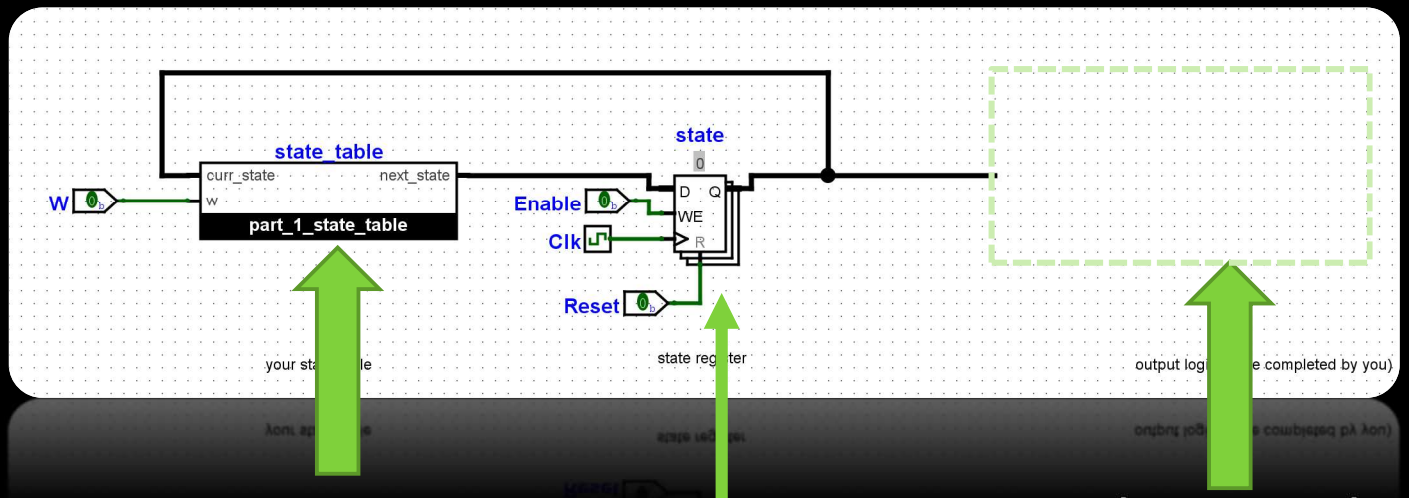
- Part I: Create a Finite State Machine
  - Make a clocked sequence recognizer.
- Part II: Control a datapath
  - Combine datapath + FSM to perform ALU functions.
- Part III: Divider (bonus)
  - Dividing number using a simple adder/subtractor
  - This is a bonus part, for those who are looking for a little extra challenge ☺
    - You can still get full marks for leaving this out, and 10 marks out of 8 for completing it perfectly.

Part I: Finite State Machine

- Sequence recognizer:
  - Make output high if the sequence `1111` or `1101` was seen on the input.
- Starter circuit provided!
  - Assign flip-flop values to each state in diagram
  - Create state logic to assign new flip-flop values based on previous values.
  - Many ways to do this!

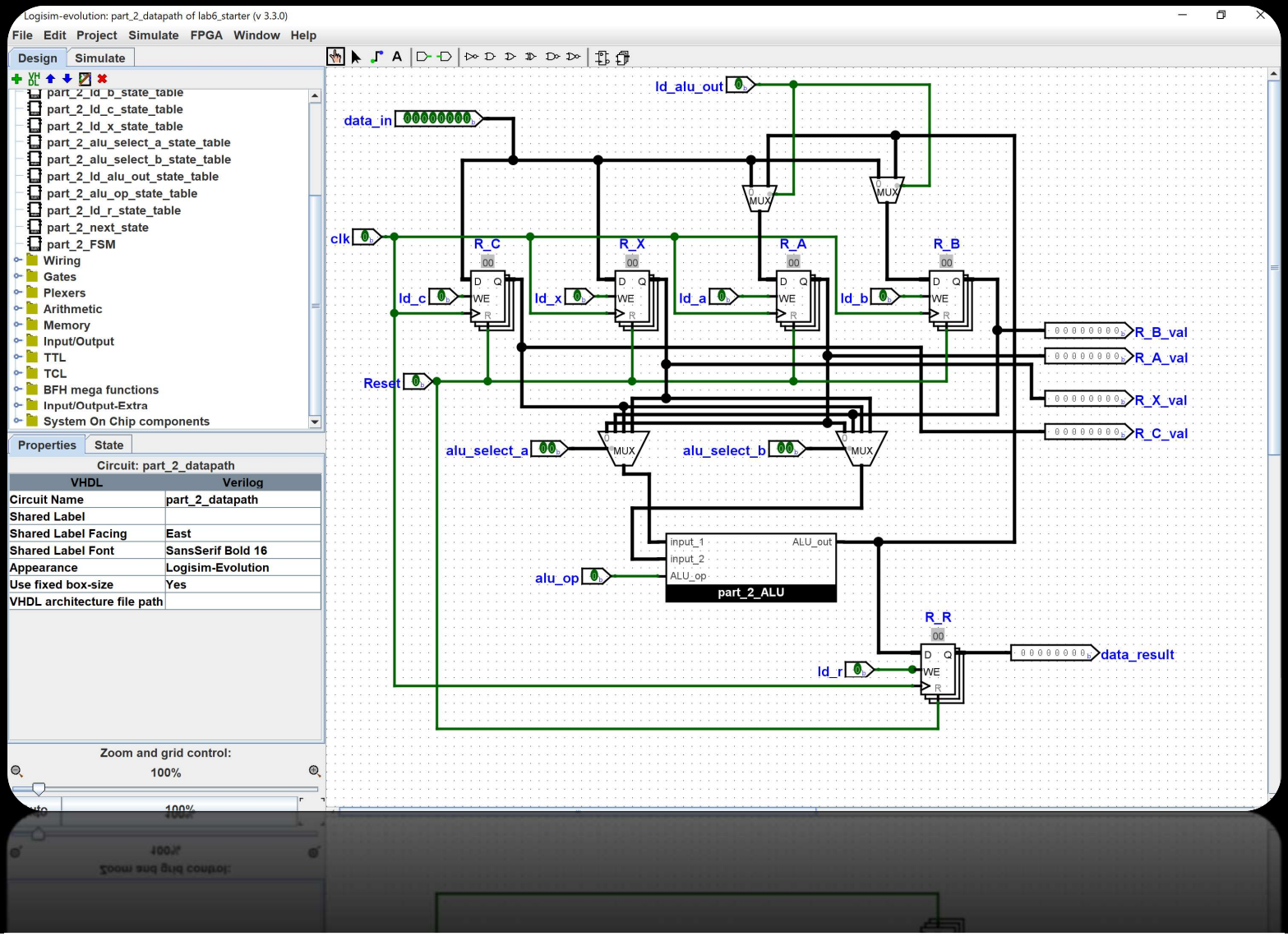


What you need to make



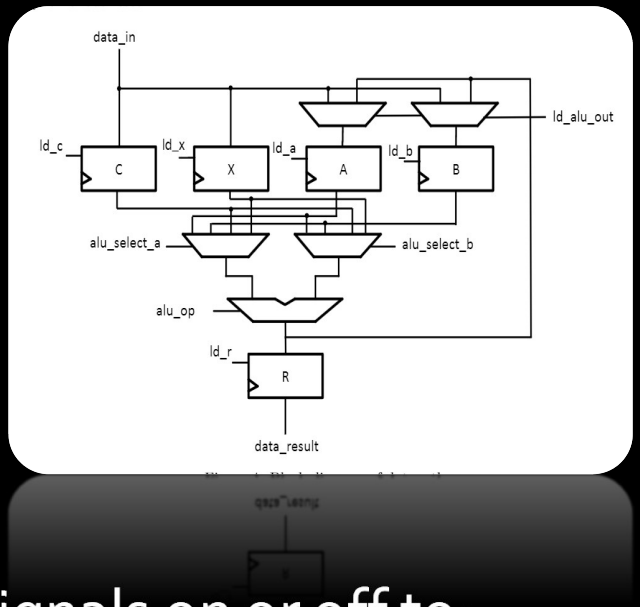
- Complete `part_1_state_table` with a combinational circuit that updates the state value stored in the 3-bit state register.
- Calculate the new state value based on the previous state and the current input.
- Create a combinational circuit that decides if the past four bits have been `1111` or `1101`, based on the current value in the state register.

Part II: Datapath Control



Part II: Datapath Control

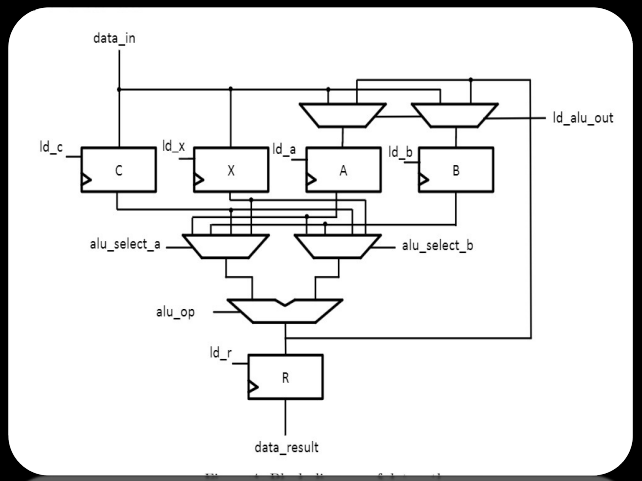
- Recall the ALU datapath example we did in class.
  - This is the same thing ☺
- We provide the datapath circuit, you provide the FSM for the controller.



  - Controller turns the datapath signals on or off to:
    - Move data from registers to the ALU
    - Perform an ALU operation
    - Store the result back into a destination register.

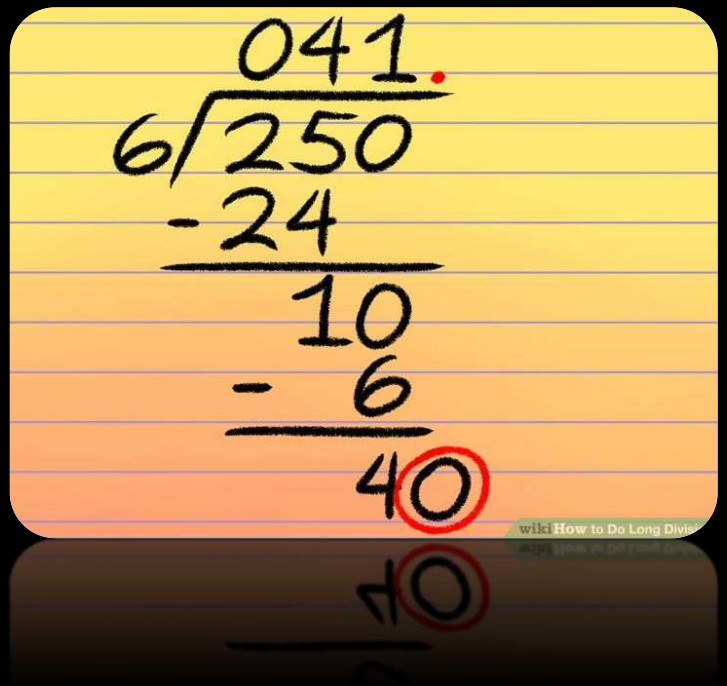
Part II: Datapath Control

- Steps to follow:
  1. Determine how to load registers with starting values. Then figure out how to move the data values around to accumulate the final result.
  2. For each of the data movements above, determine which control signals need to be turned on or off to make that movement happen.
  3. Make a finite state machine that implements the state sequence from the preceding step, where each state emits the control signals to the datapath.



Part III: Divider Circuit

- Note: This part is optional, but can be done for bonus marks in the course.
- Basic idea from decimal long division:
  - From left to right, find where the divisor can be subtracted from the dividend.
  - Doing this in binary is simpler, except that we keep the divisor static, and move everything else!



Thoughts for Lab 6

- You're all grown up now.
- The only restriction for Lab 6 is that you must use the datapath that we provide and not change it.
- Beyond that, there are no limitations to the implementation approach that you use.
  - You can use the files that we provide, or not.
  - You can implement your FSM the way we did in class, or not.

Thoughts for Lab 6

- The one thing we will check this week is the readibilty and modularity of your design.
  - If your `*.circ` file is getting full and/or complicated, break it up into smaller modules (like helper functions).
    - The collection of modules we provide for each datapath signal is a guide to this (but not a required approach).
  - Your modules should contain circuits that are simple and compact, either through module creation or an elegant design approach.
    - Consider using the Tunnel (under Wiring) to make connections across long distances:
      - http://www.cburch.com/logisim/docs/2.6.0/en/libs/base/tunnel.html
- **Only submit the modules that you use**.