Logic Circuits
& Logisim

What is *Logic* in Computer Science?

In CS, we have **Boolean** values and functions.

Values: True (1) or False (0),

represented by the binary digits.

Functions: AND, OR, NOT,...

It must mean something that there are ***three*** of these functions…

Logic gates: *definitions*

input		output
x	y	AND(x,y)
0	0	0
0	1	0
1	0	0
1	1	1

AND outputs 1 only if ALL inputs are 1

AND

input		output
x	y	OR(x,y)
0	0	0
0	1	1
1	0	1
1	1	1

OR outputs 1 if ANY input is 1

OR

input	output
x	NOT(x)
0	1
1	0

NOT reverses its input

NOT

What is *Logisim*?

Logisim is a program that lets us build virtual logic circuits using gates, clocks, and other things!

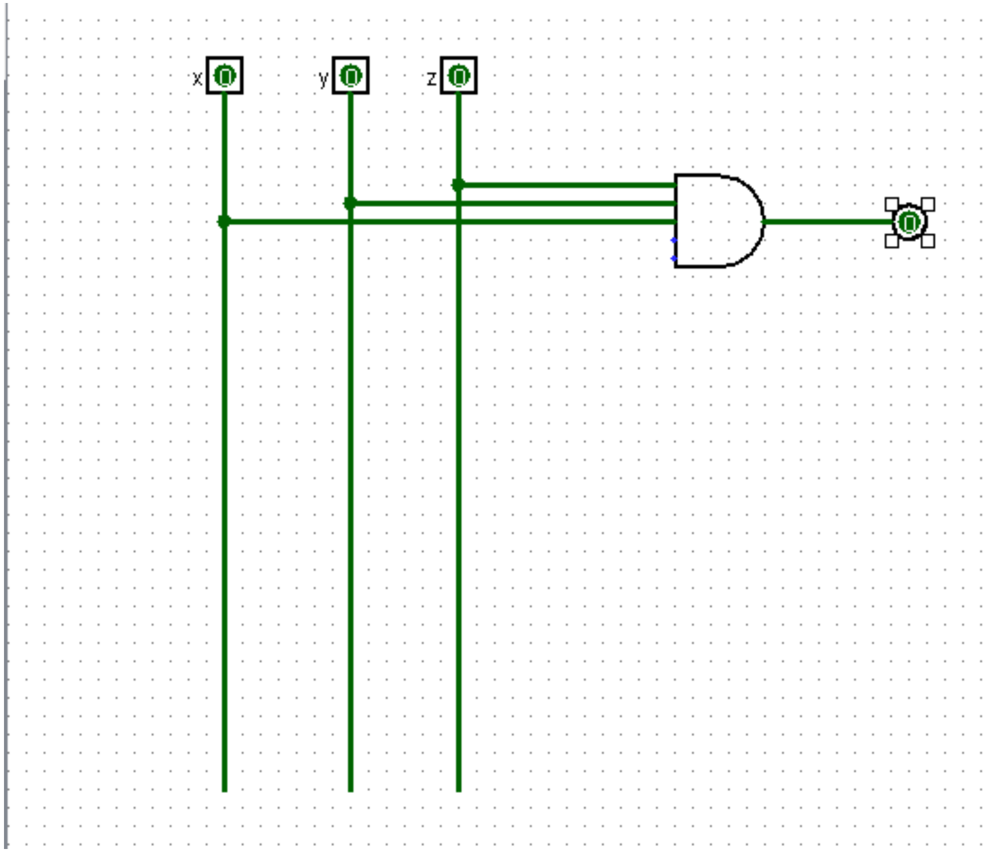
You can download it here:

http://sourceforge.net/projects/circuit/

AND Gates!

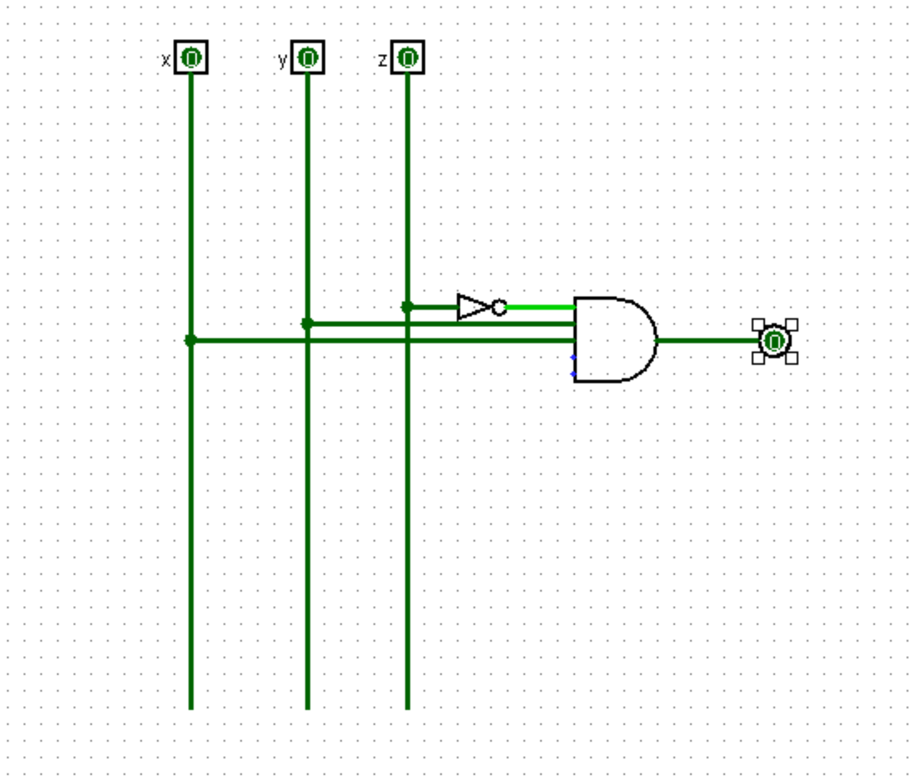
ACTIVITY

What values of x, y, and z would output a 1?



Moar AND Gates!

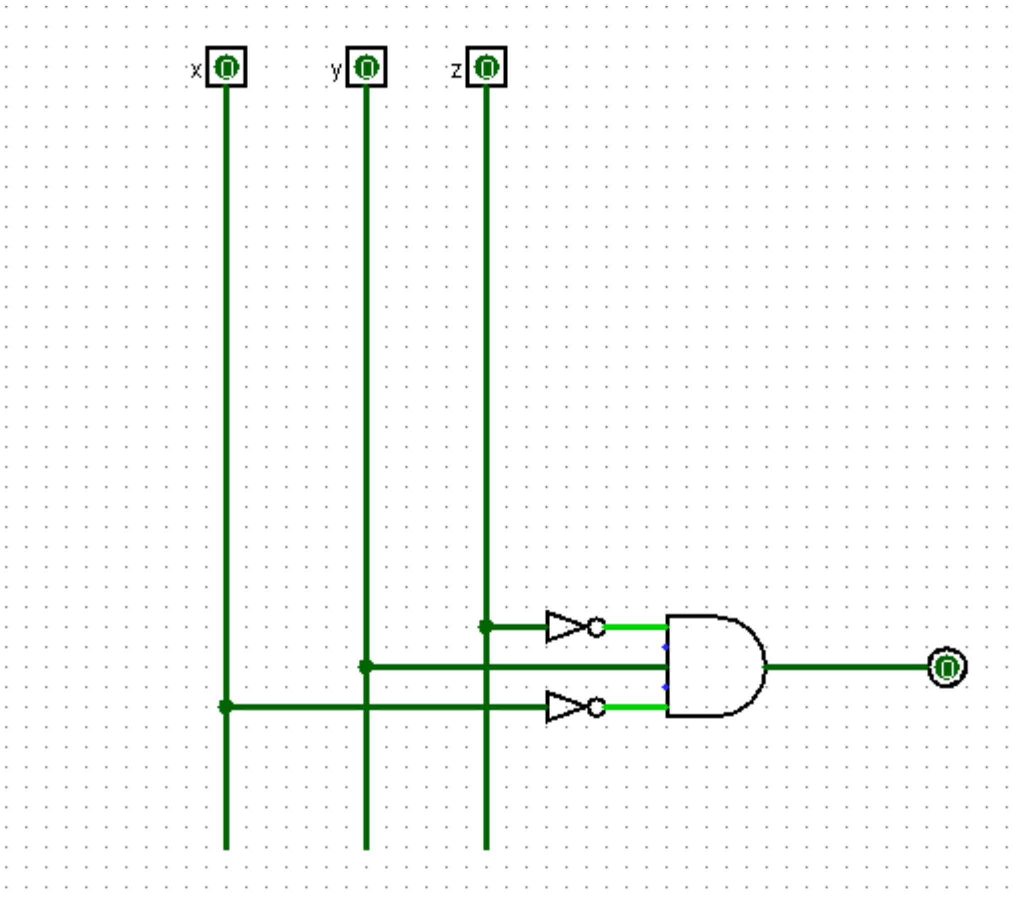
What values of x, y, and z would output a 1?



And moar!

ACTIVITY

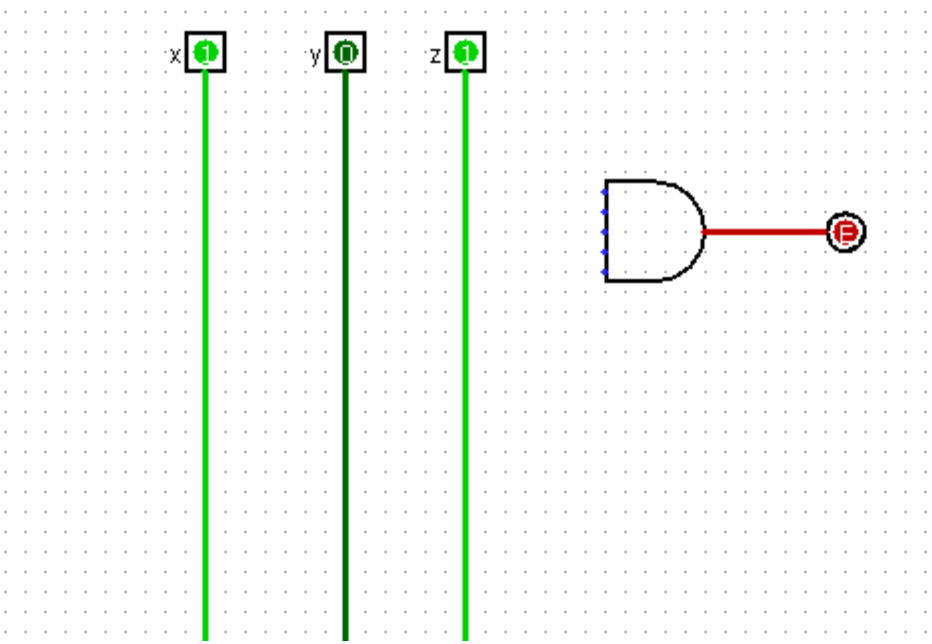
What values of x, y, and z would output a 1?



Wirings

ACTIVITY

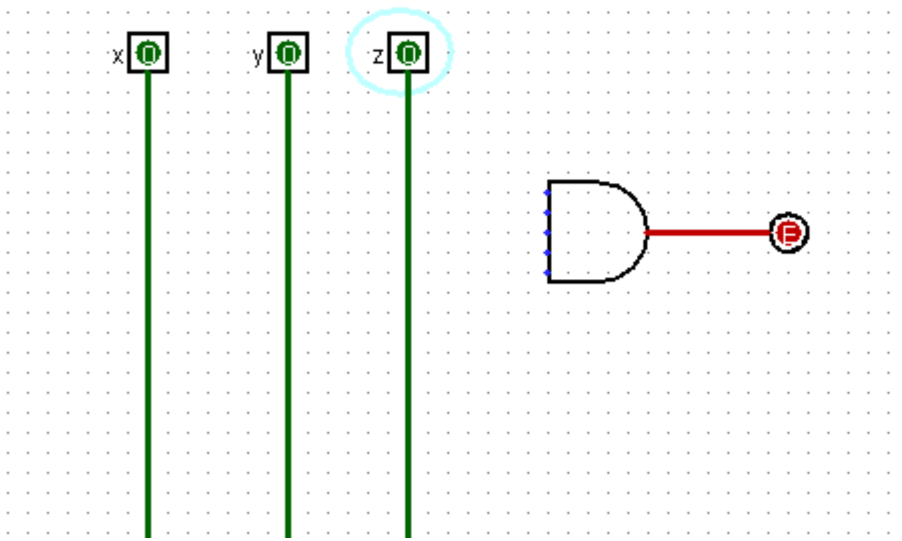
Here x=1, y=0, and z=1. What wirings (connections) should be made such that the circuit outputs a 1?



Moar wirings! Try it yourself

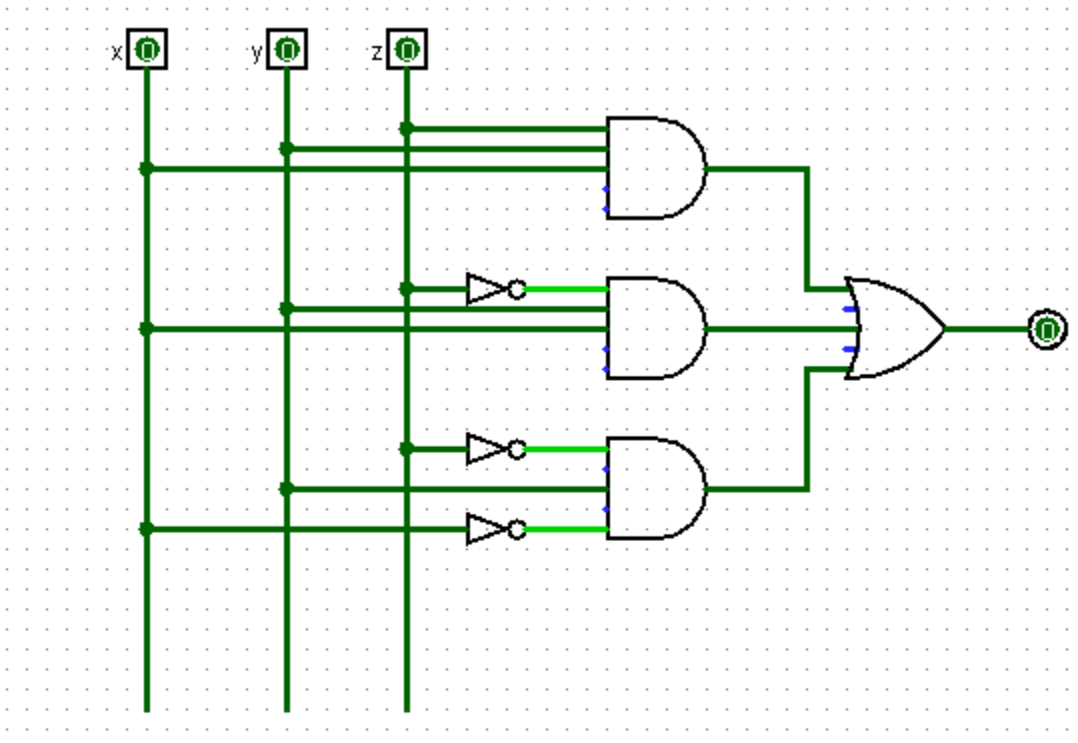
ACTIVITY

Here x=0, y=0, and z=0. What wirings (connections) should be made such that the circuit outputs a 1?



Altogether now!

Let x=1, y=1, and z=0. What's the output?



Now let's move on to making our *own* circuits!

Minterm Expansion

Truth Table

input		output
x	y	XOR(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

Process

First, look at the rows that output a 1.
Now look at the input values that output each 1. If there's a 0 input, then we NOT that variable and if there's a 1 input, we leave the variable as is, AND those two together.
Do this for each row that outputs a 1 and OR all the rows together.

So let's look at the truth table above. There are two rows that output a 1. Adjacency implies AND, + implies OR, and ! implies NOT.
1st row: !xy
2nd row: x!y
So we OR these two together: !xy + x!y and this is the formula we want to use for our overall circuit.

Now build your own circuit using the set of inputs and outputs below!

ACTIVITY

Can you guess what the function is? :)

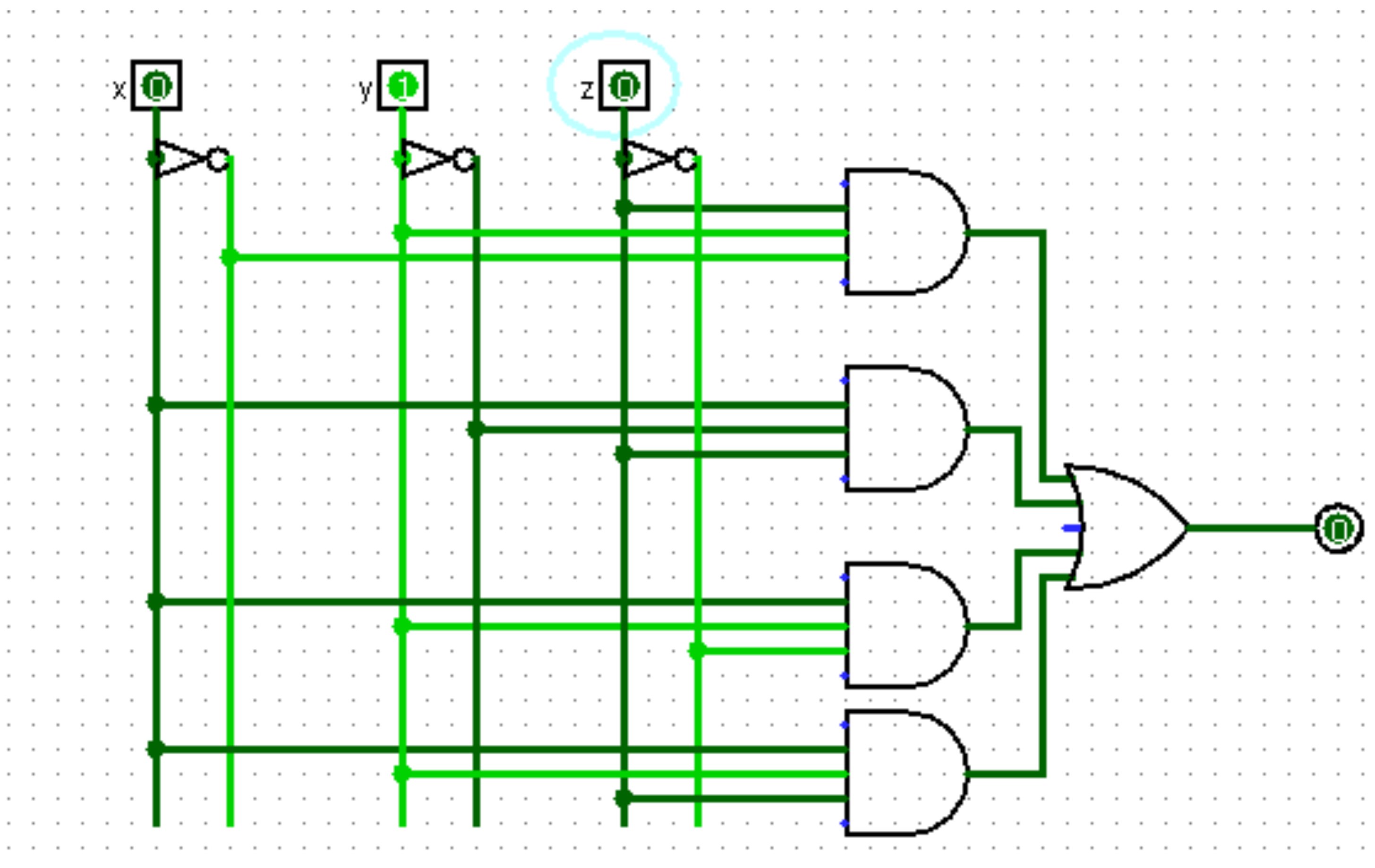
inputs			output
x	y	z	fn(x,y,z)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Truth table

Even more fn !

One possible circuit is this!

ACTIVITY



Now build your own circuit using the set of inputs and outputs below!

ACTIVITY

Can you guess what the function is? :)

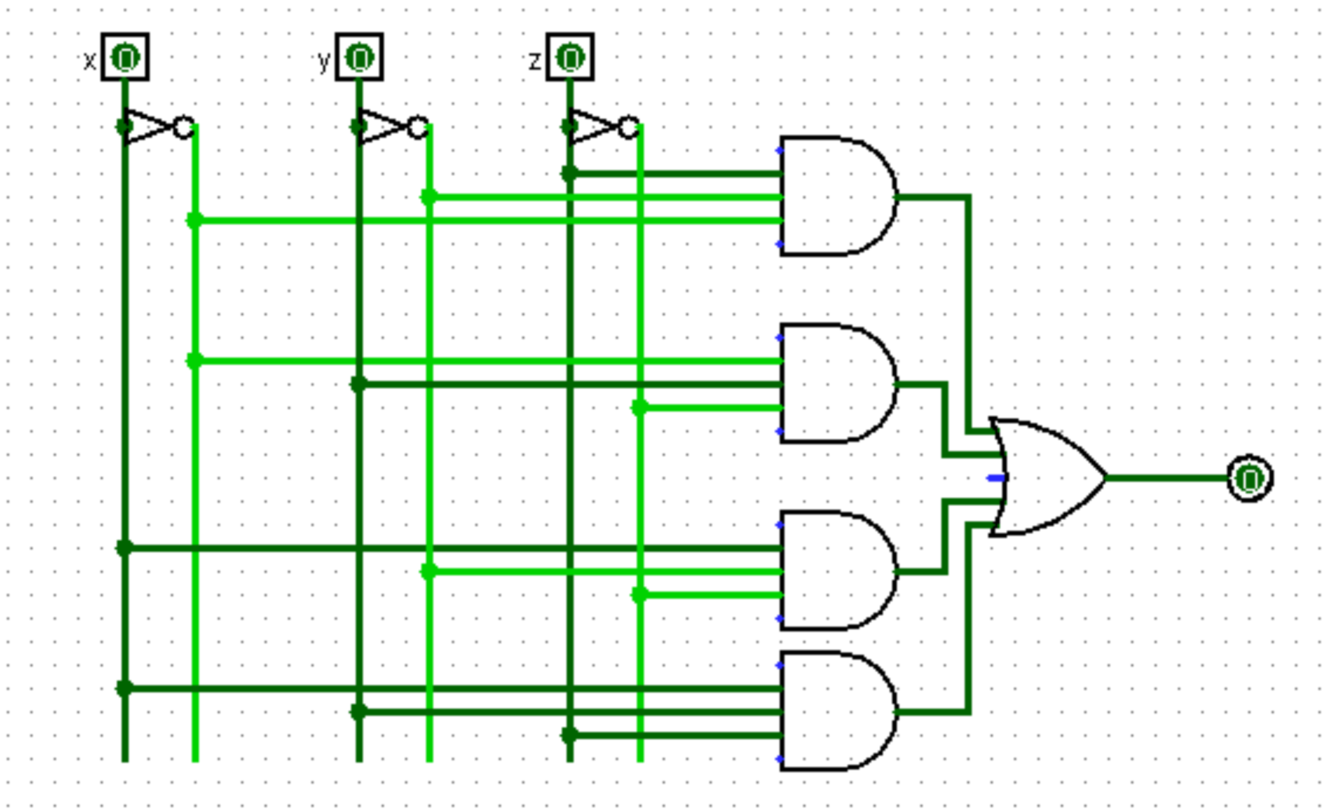
inputs			output
x	y	z	fn(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Truth table

Even more fn !

Your circuit should look similar to this!

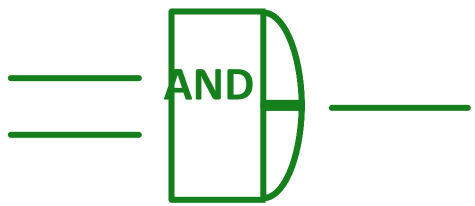
ACTIVITY



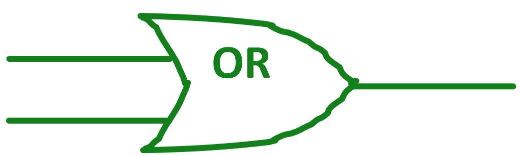
Claim!

We need only three building blocks to compute anything at all

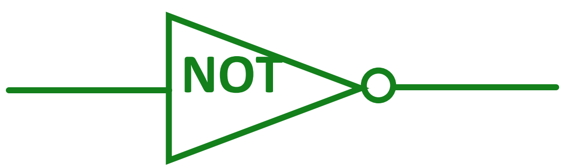
What are these?



AND outputs 1 iff ***ALL*** its inputs are 1



OR outputs 1 iff **ANY** input is 1

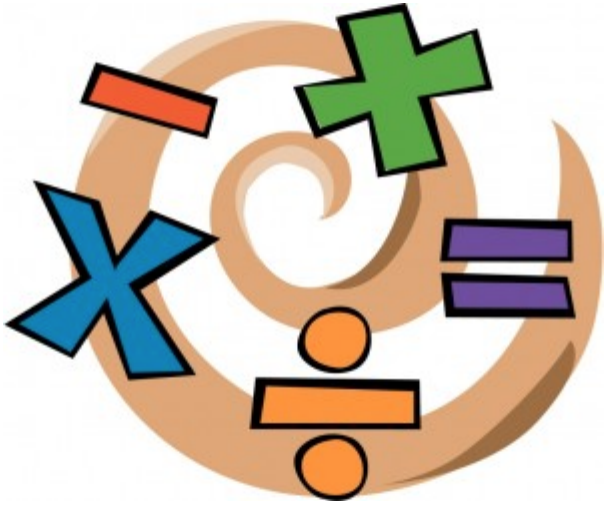


NOT reverses its input

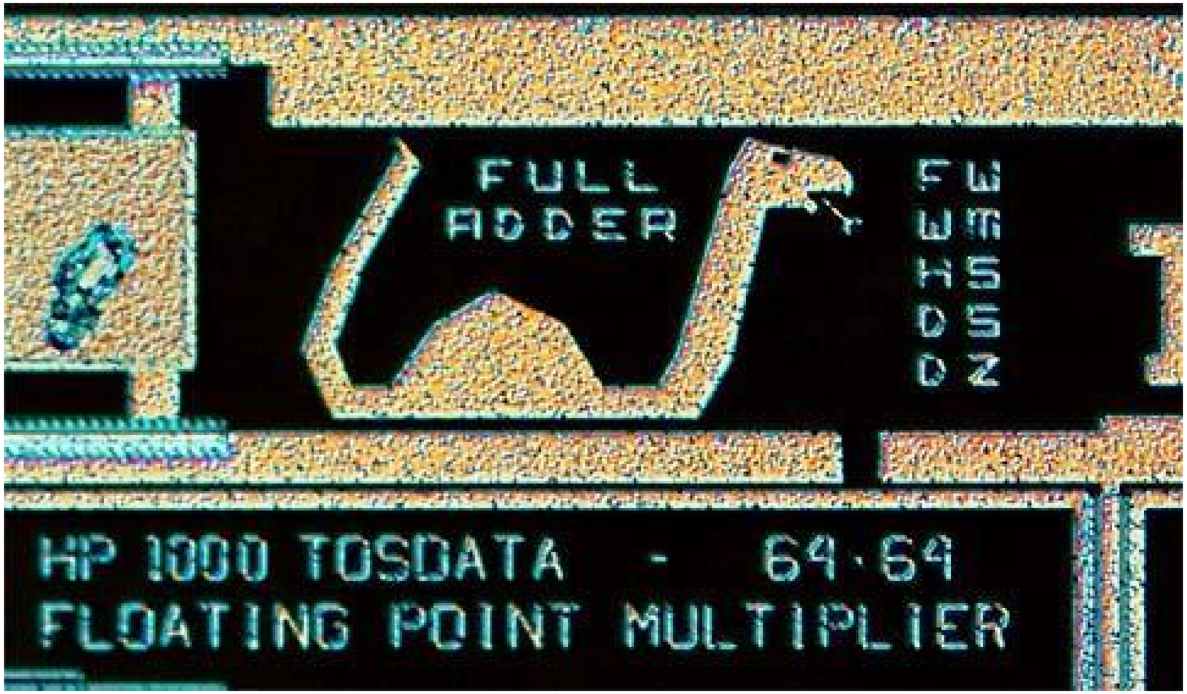
Computing

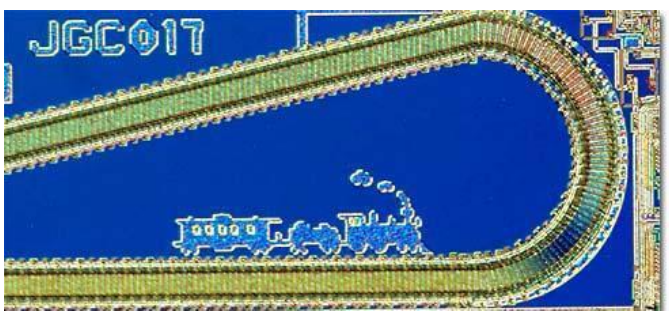
So one of the most basic computations that a computer can do is *addition*!

They add two sequences of bits using what's called an **adder circuit**!



Adders!



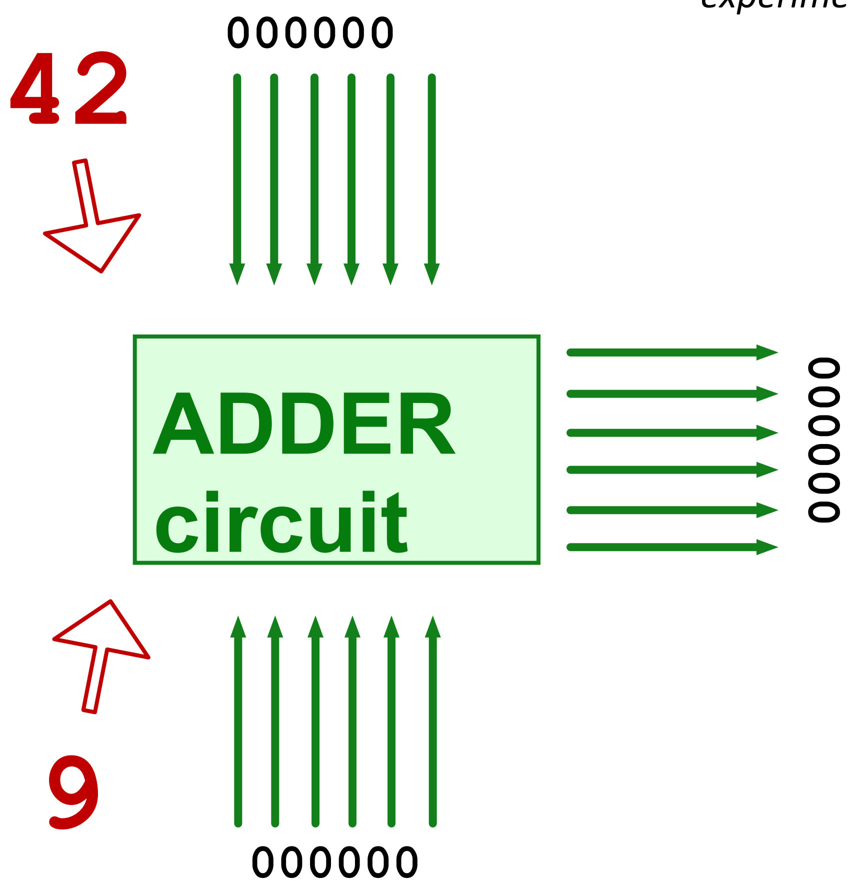


chugging right along…

In a computer, each bit is represented as a voltage (**1** is +5v and **0** is 0v)

Computation is simply the deliberate combination of those voltages!

Feynman: *Computation is just a physics experiment that always works!*

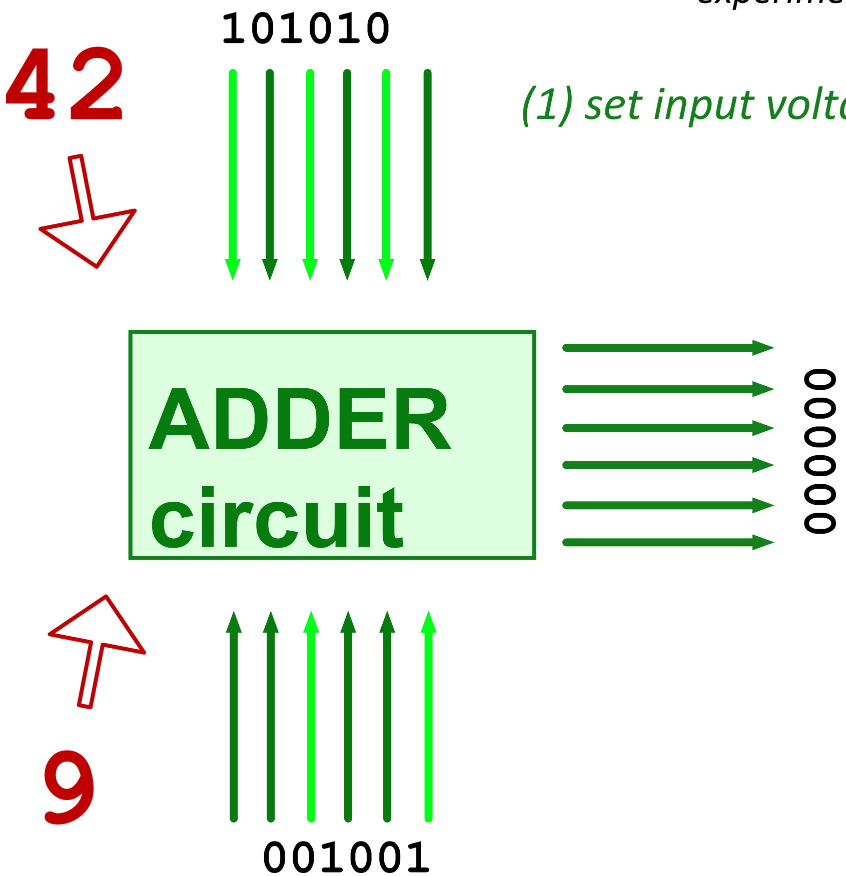


In a computer, each bit is represented as a voltage (**1** is +5v and **0** is 0v)

Computation is simply the deliberate combination of those voltages!

Feynman: *Computation is just a physics experiment that always works!*

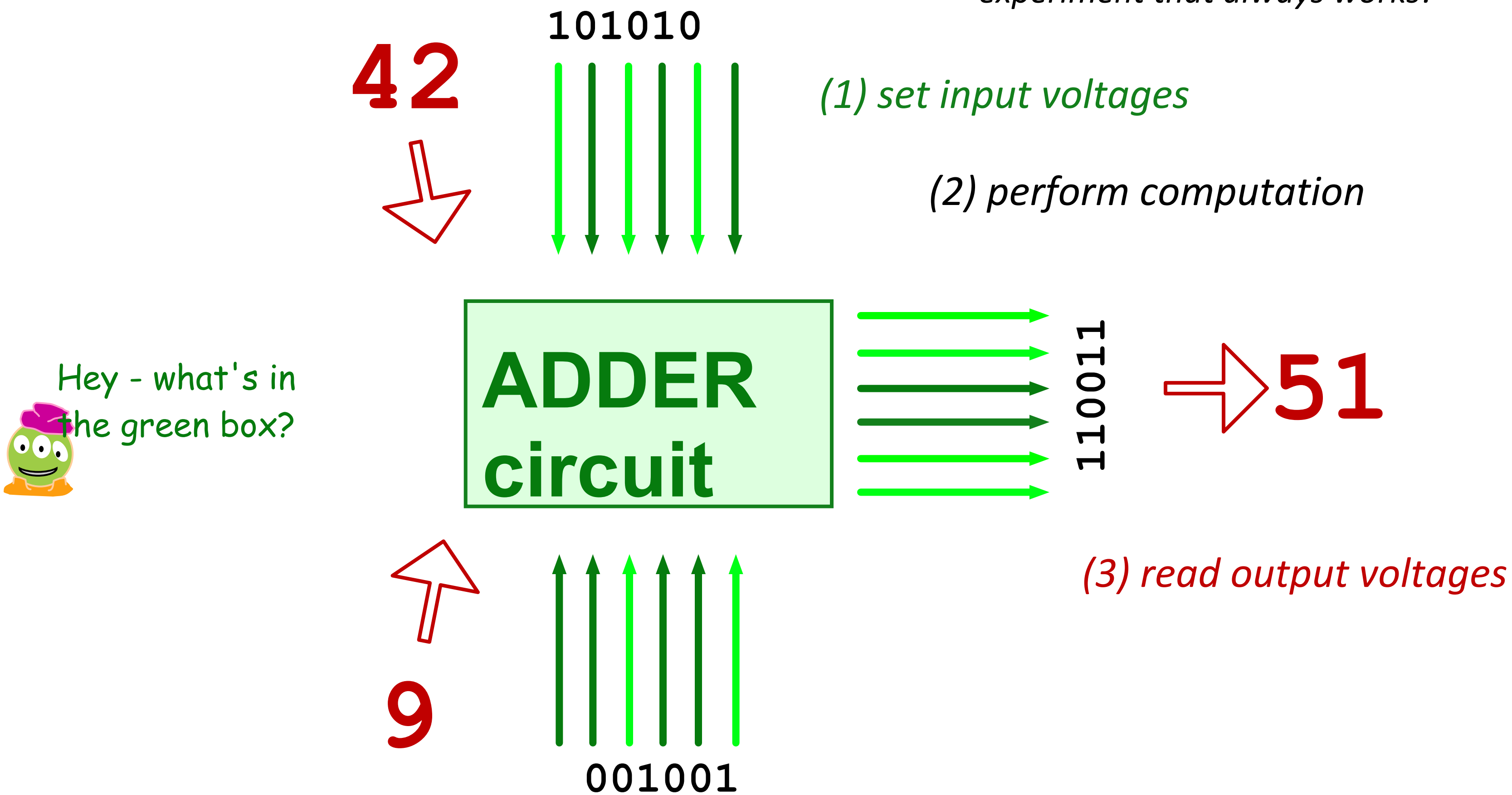
(1) set input voltages



In a computer, each bit is represented as a <u>voltage</u> (**1** is +5v and **0** is 0v)

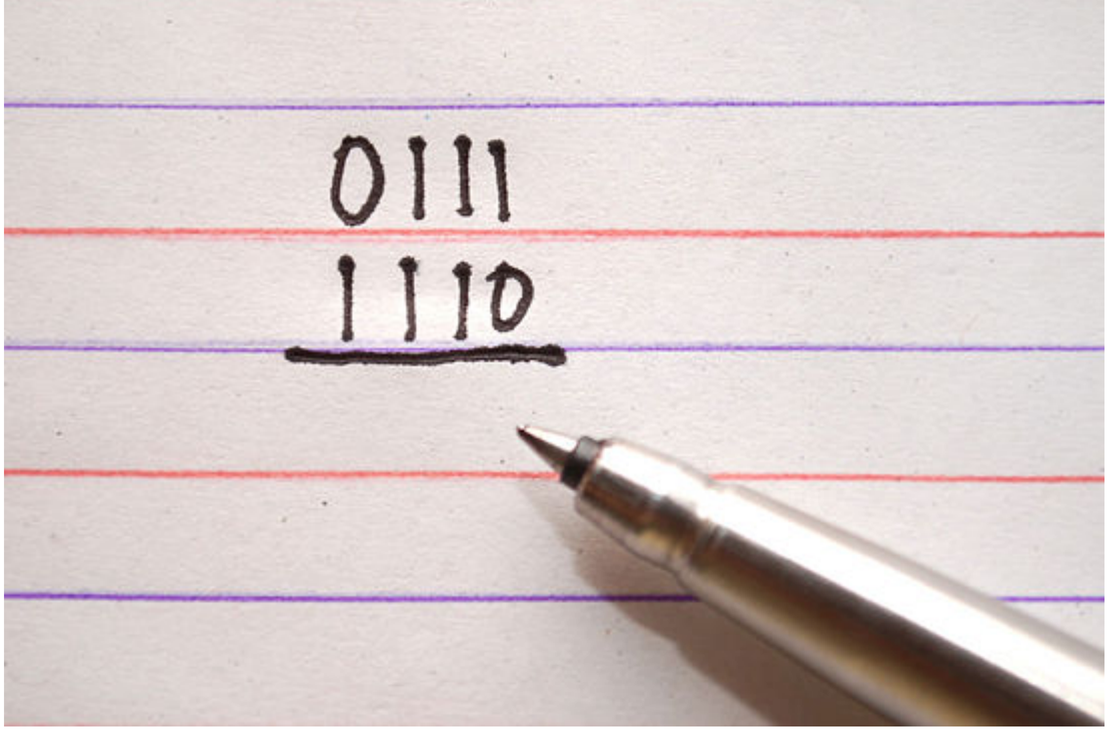
Computation is simply the deliberate combination of those voltages!

Feynman: *Computation is just a physics experiment that always works!*



Adding in Binary!

To make an adder circuit, let's first try adding in binary by hand!



How do we do this?

Adding in binary is almost exactly like adding in decimal!

We start from right to left. 1+0=1, 0+0=1.

However, what does 1+1=?

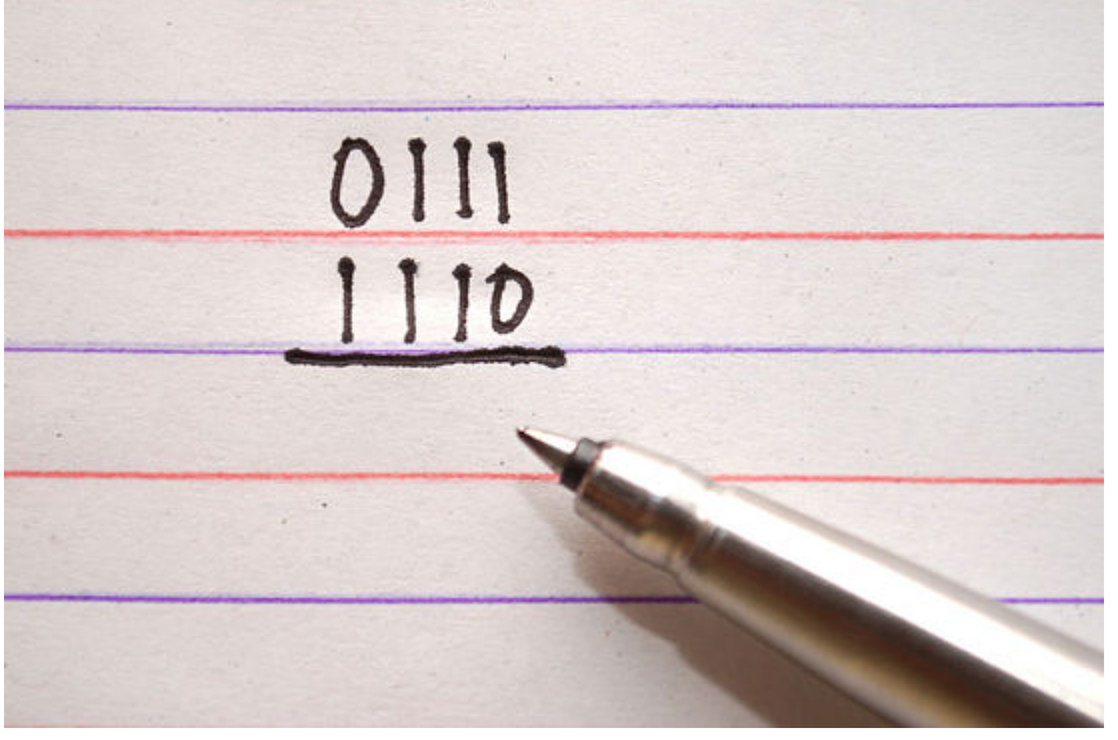
Hint: Just add normally and represent the sum in binary!

Rules for Adding in Binary

1. Start from *right* to *left*.
2. 1+0=1, 0+0=1, 1+1=10.
3. For 1+1, we bring down a 0 and carry a 1 to the next column.
4. Like for regular addition in decimal, remember to add in any carry numbers!

Example

Try out the one below! What are some smaller operations that were needed to do this?



Logisim: Adder Circuit

From our example, there were times where we had to add ***three*** bits together, instead of just ***two***! Numbers can "carry" from one column to the next.

To build an adder circuit that adds numbers together, we need to create **3-bit full adders**!

More ***output*** bits?

3 bits of input			2 bits of output	
x	y	c_{in}	c_{out}	sum
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

the **output** is the sum of the three input bits, IN BINARY !

A *full adder* sums three input bits to two output bits, ***a binary number***

(A 2-bit adder is a *half adder*)

Circuit-design solution: ***share the inputs***, but **design separate circuits** for each output bit...

Building a Full Adder

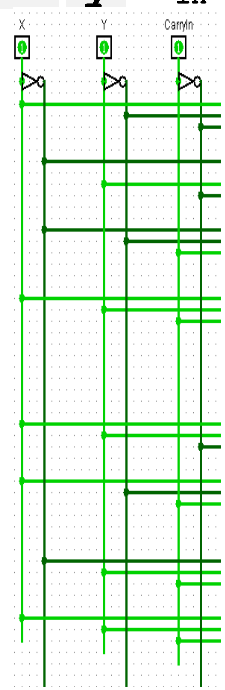
ACTIVITY

Create a separate circuit **for each output bit** !

input			output	
x	y	c_{in}	c_{out}	sum
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

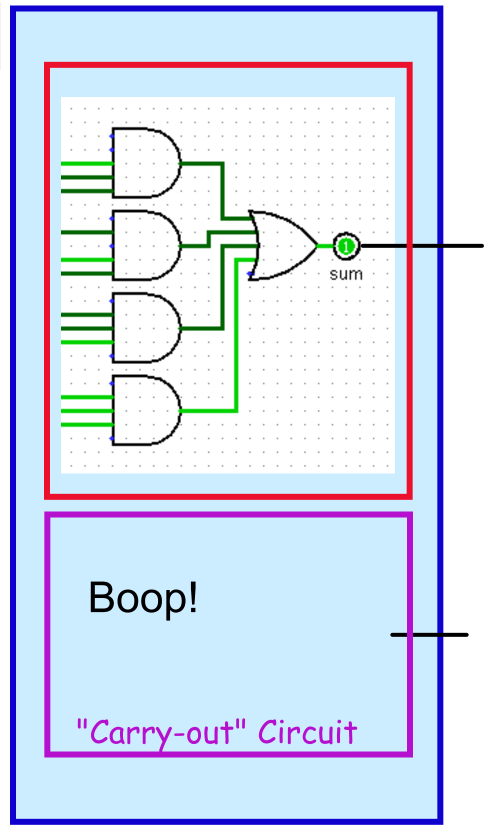
3 input bits

x y c_{in}



"Rails" of each bit and its inverse

Full Adder



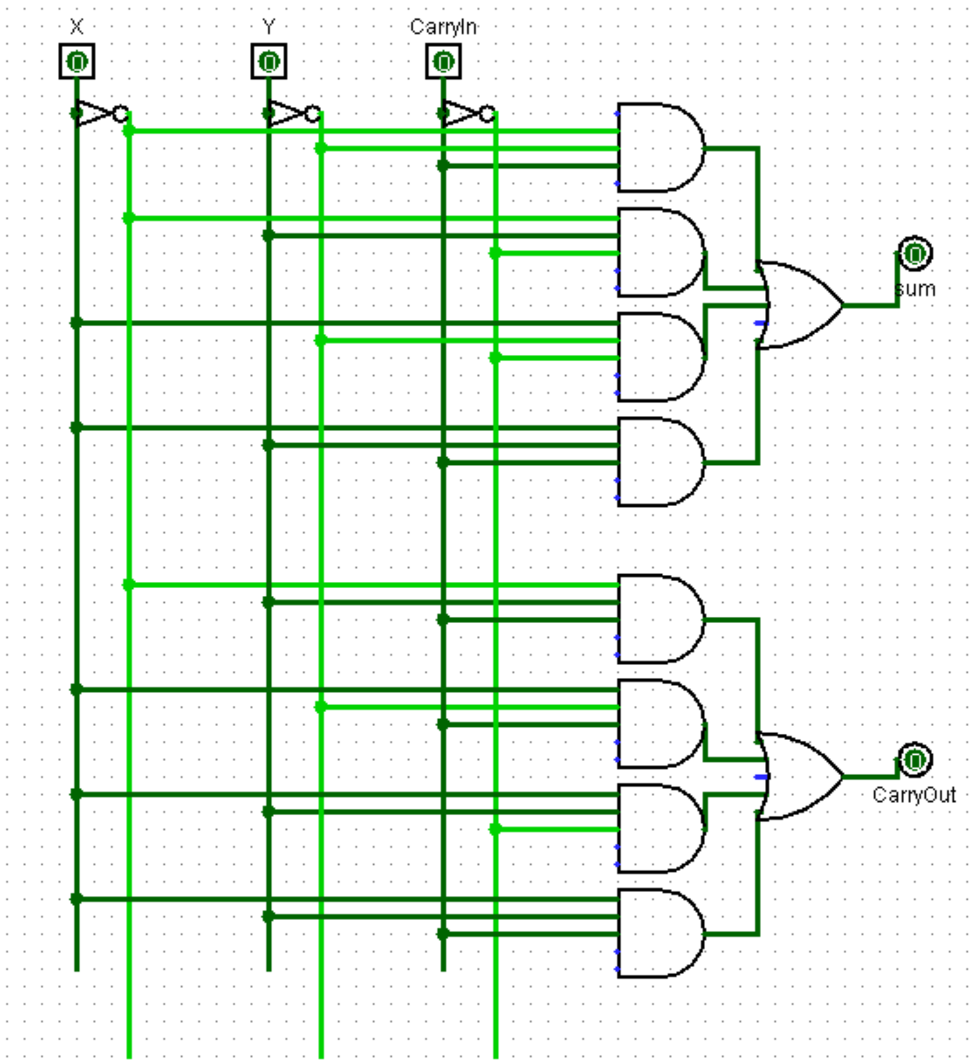
sum output bit

2 output bits

c_{out} output bit

Remember: this is addition ***in silicon*** !

A 3-bit full adder!



Putting it all together!

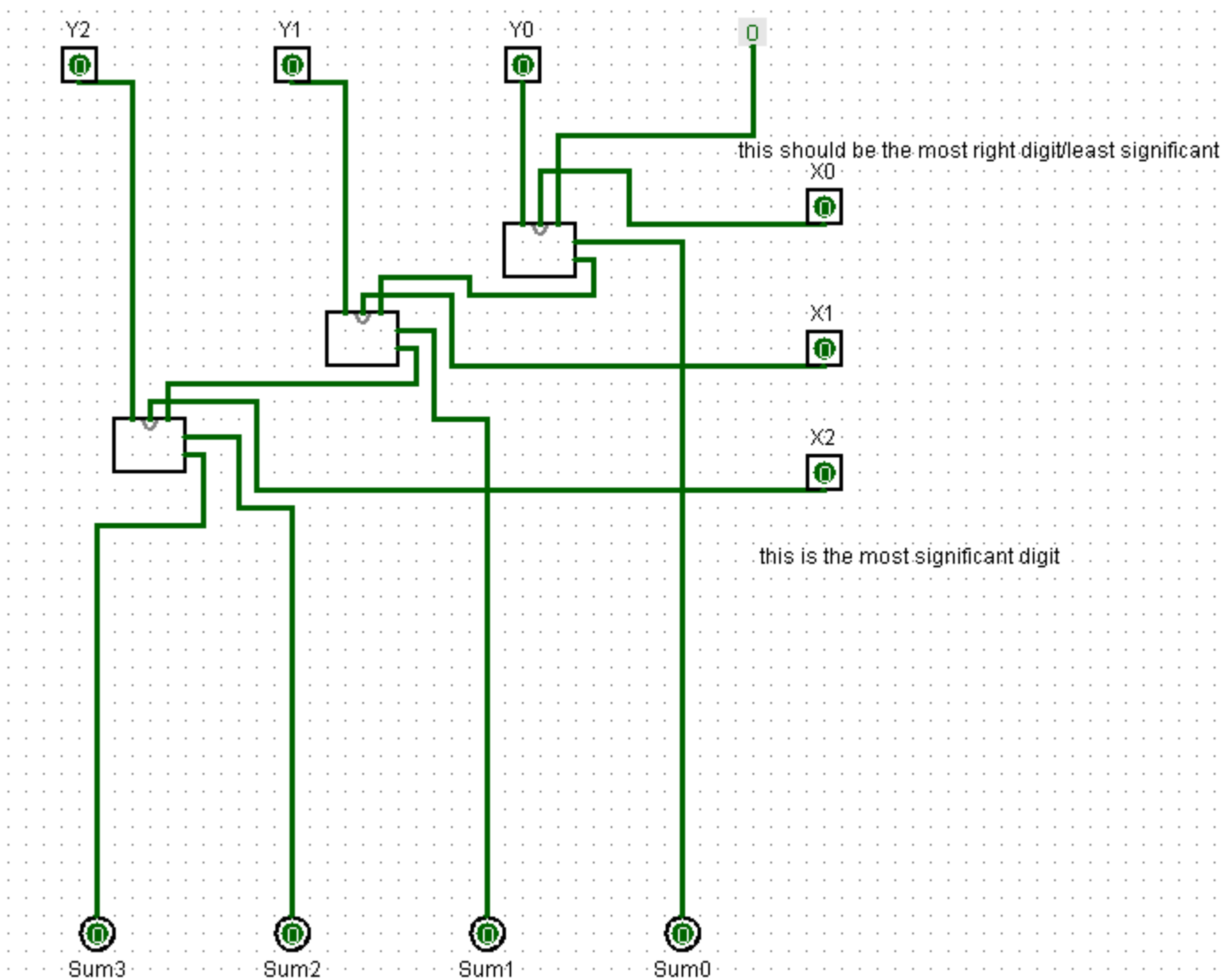
ACTIVITY

Can you see how we would use the full-adders to add *n*-bit numbers together?

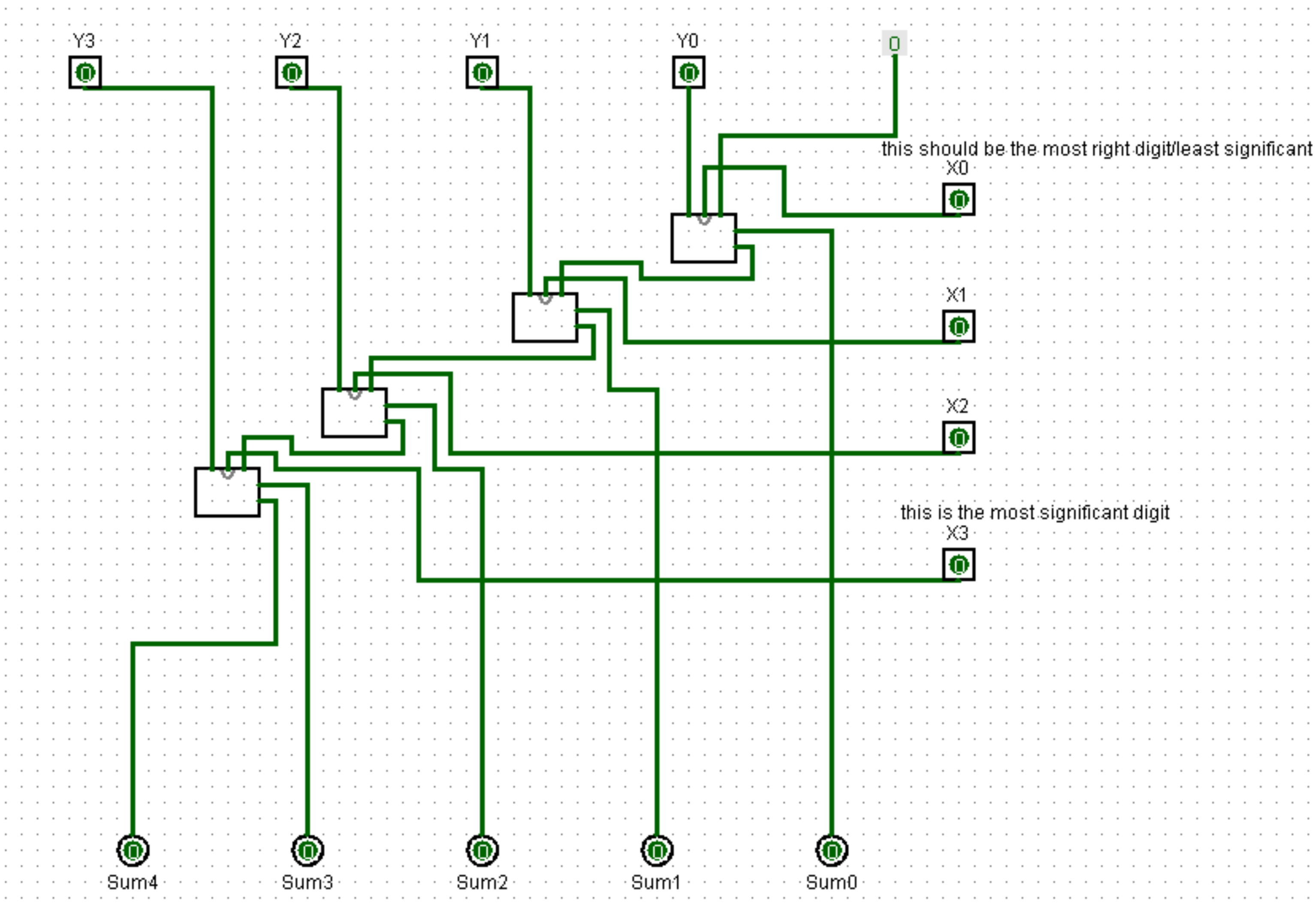
These types of adders are called ***ripple-carry adders***. It's this method that simulates us adding binary numbers by hand!

If you still have extra time left, try to see if you can build a ***3-bit ripple-carry adder*** in Logisim.

A 3-bit Ripple-Carry Adder!



A 4-bit Ripple-Carry Adder!



As you can see, it's not difficult to make an *n*-bit ripple-carry adder!