Circuits and Gates

15-110 – Wednesday 09/16

Learning Goals

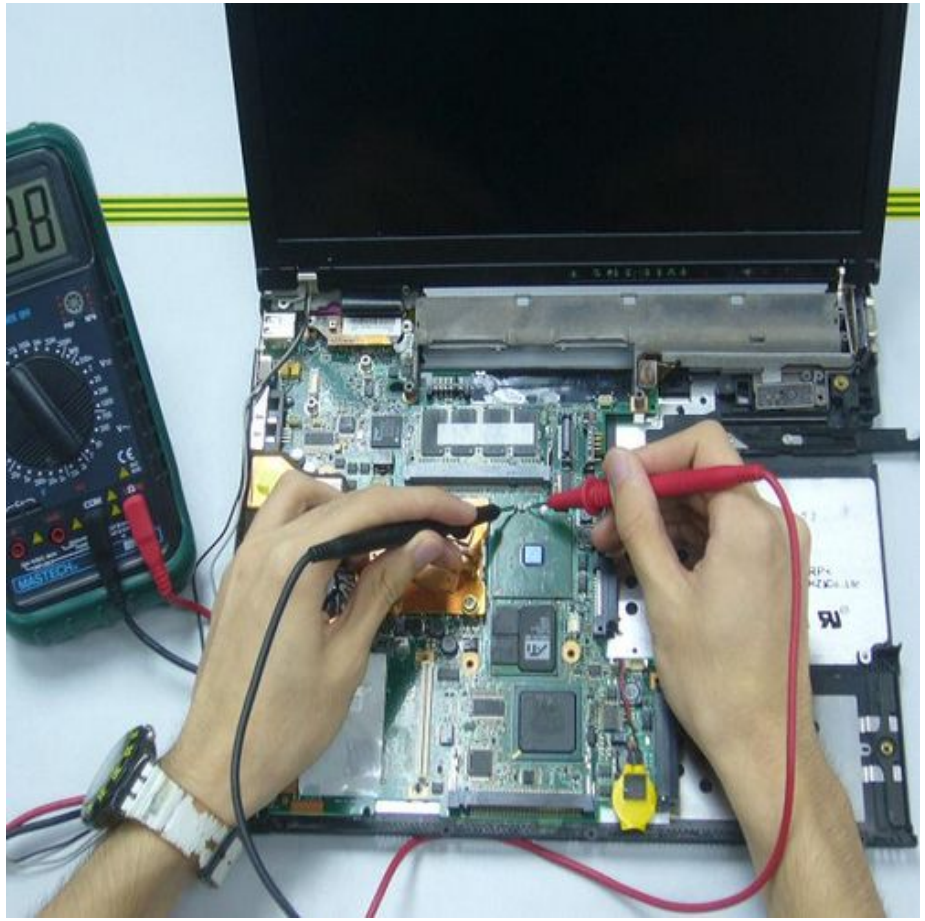
- Translate Boolean expressions to **truth tables** and **circuits**
- Translate **circuits** to **truth tables** and Boolean expressions
- Recognize how addition is done at the circuit level using **algorithms and abstraction**

Computers Run on Hardware

Software: the abstracted concepts of computation- how computers represent data, and how programs can manipulate data.

Hardware: the actual physical components used to implement software, like the laptop components shown to the right.

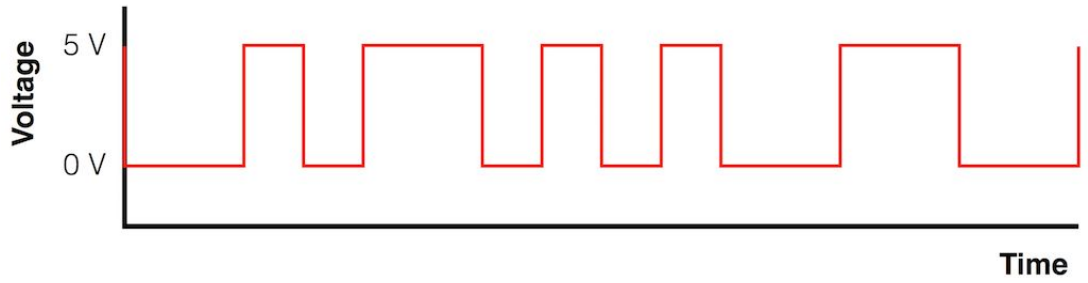
All the operations we perform on a computer correspond to physical actions within the hardware of the machine.
How does this work?



Bits are Electric Voltage

We previously discussed how everything in a computer is represented using bits (0s and 1s).

In hardware, bits are represented as **electrical voltage**. A high level of voltage is considered a 1; a low level of voltage is considered a 0.



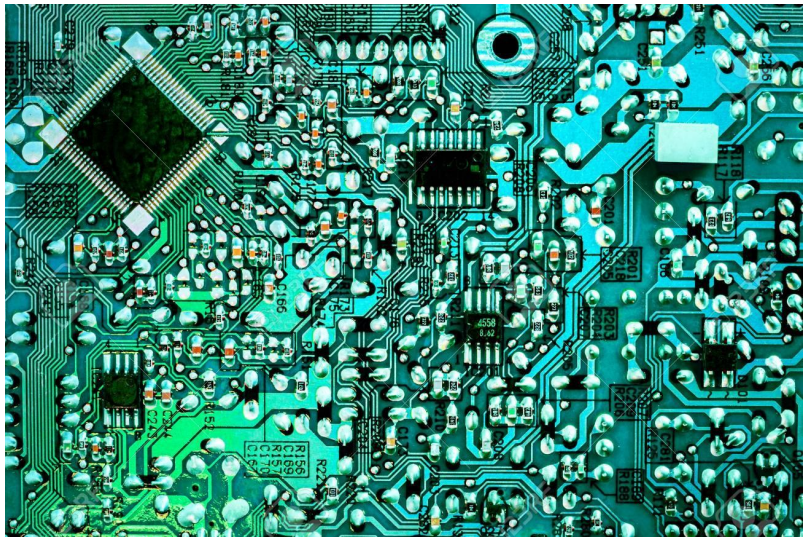
By redirecting electrical flow throughout a system, we can change the values of data in hardware.

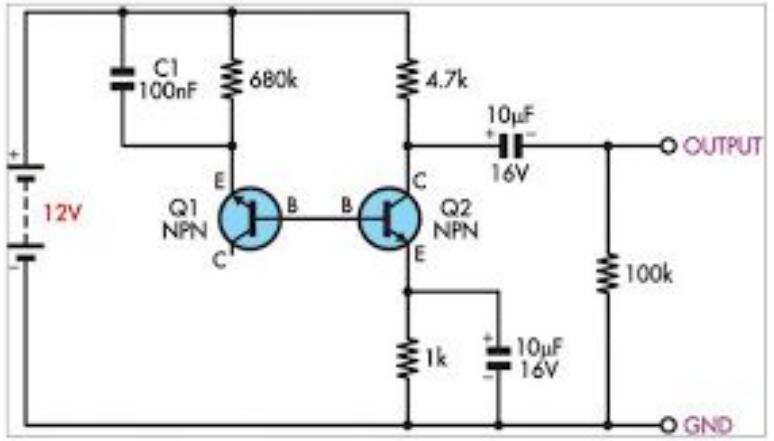
Circuits Manipulate Voltage

The computer uses **circuits** to perform computational actions. Circuits redirect electricity to different parts of hardware.

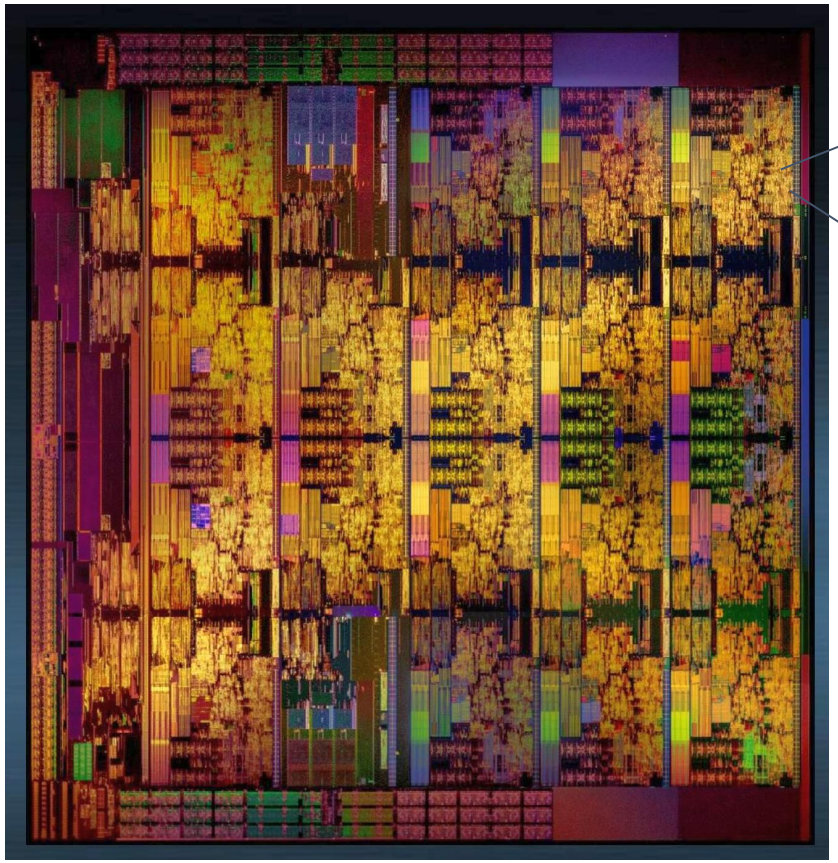
Physical components of circuits (like transistors and capacitors) are out of the scope of this class. If you're interested, take an Intro to Electrical Engineering class!

Instead, we will discuss how to use **gates**, which are abstracted circuit components. Every gate we discuss can be directly translated to a real hardware circuit.

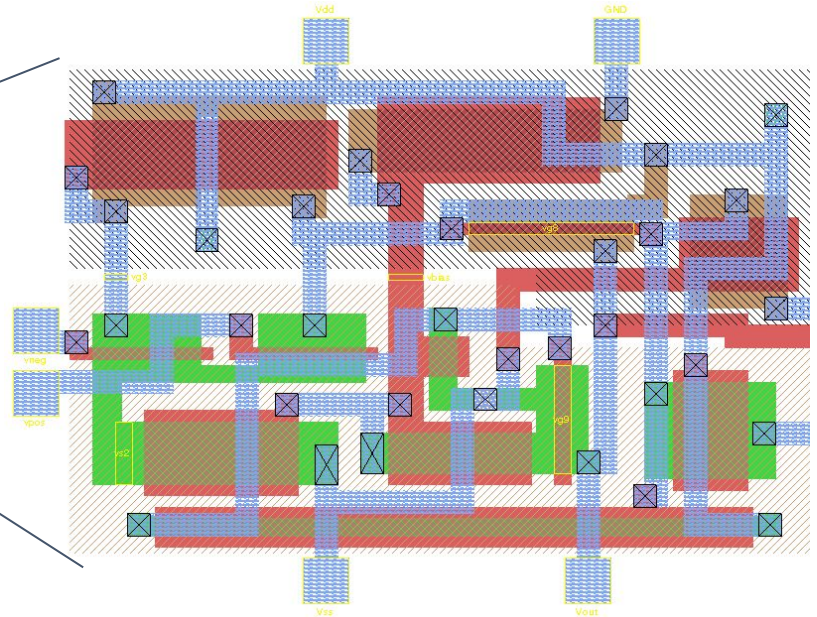




Sidebar: Processor Chips



This Intel core i9 processor chip contains roughly 7 billion transistors.



A tiny chunk of chip circuitry. Every place a red line crosses a green or tan line, a transistor is formed. This chunk contains about a dozen transistors.

We're going to build a little piece of this chip!

Logical Gates

Gates are Hardware's Boolean Operations

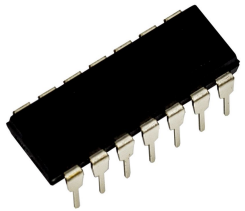
Recall that Booleans have two values (`True` and `False`), just like bits (1/high voltage and 0/low voltage).

We can build a **gate** to have the same effect as a Boolean operation, but with bits as input/output instead of `True`/`False` values.

Let's start with three familiar gates: **and**, **or**, and **not**.

Basic Gates – Actual Hardware

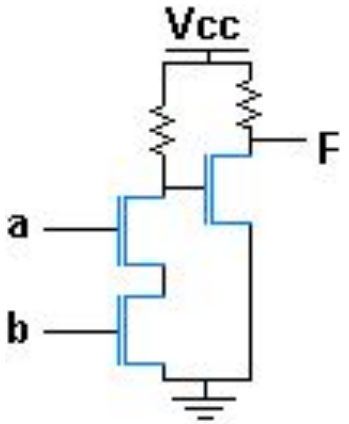
Our three basic gates can be represented in actual hardware



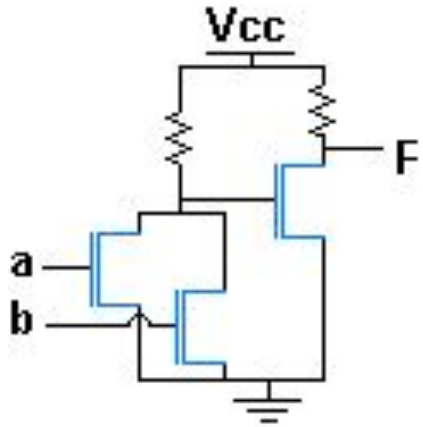
An **and** gate takes two inputs and outputs 1 only if both inputs were 1

An **or** gate takes two inputs and outputs 1 if either input was 1

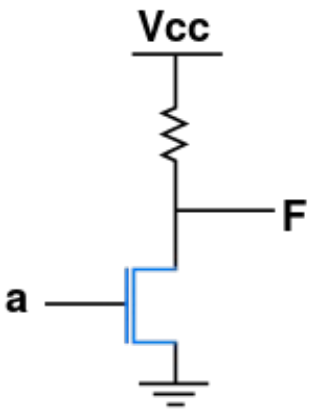
A **not** gate takes one input and outputs the reverse (1 becomes 0, 0 becomes 1)



NMOS
AND gate



NMOS
OR gate

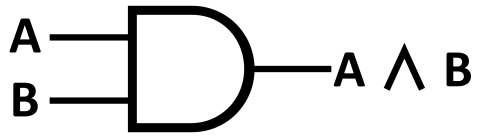


NMOS
NOT gate

Basic Gates – Shorthand

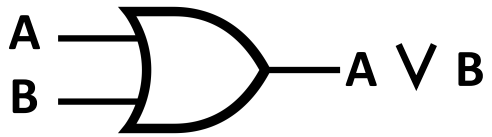
We'll use a shorthand when building circuits with these gates instead

An **and** gate takes two inputs and outputs 1 if both inputs were 1



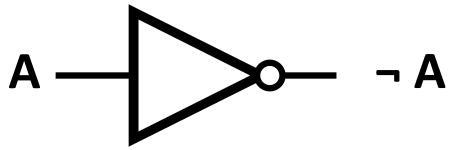
A	B	A ∧ B
1	1	1
1	0	0
0	1	0
0	0	0

An **or** gate takes two inputs and outputs 1 if either input was 1



A	B	A ∨ B
1	1	1
1	0	1
0	1	1
0	0	0

A **not** gate takes one input and outputs the reverse (1 becomes 0, 0 becomes 1)

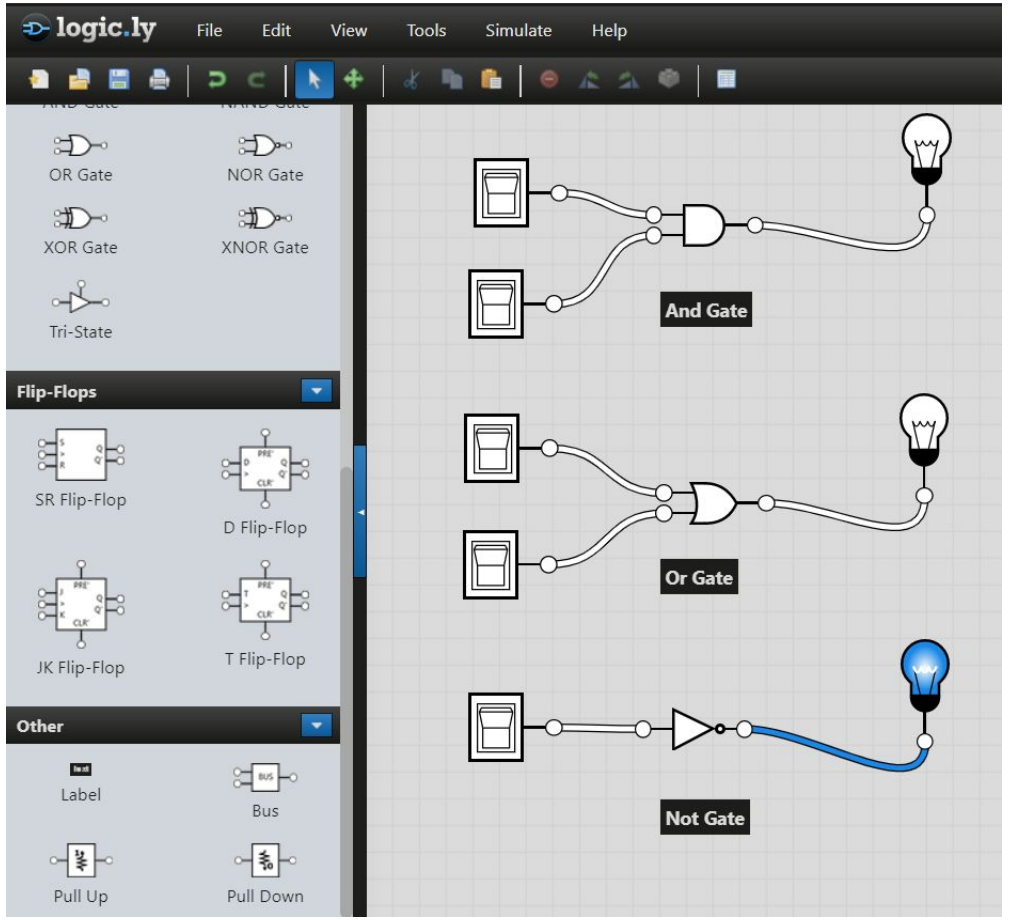


A	¬ A
1	0
0	1

Circuit Simulation

When working with gates, it can help to **simulate** a circuit using the gates to investigate how they work.

There are lots of free online circuit simulators. We'll use this one: https://logic.ly/demo



Algorithms with Gates

Multiple Representations of Gate Algorithms

Just like with Boolean expressions, we can combine gates together in different orders to achieve different results. This lets us build **algorithms** using gates.

When we want to represent an algorithm that uses gates, we can use one of three different representation formats: a Boolean expression, a circuit, or a **truth table**.

Truth Tables Show All Possibilities

So far, we've used truth tables to show all the outcomes of a single gate or operation.

We can also use these tables to show all the possible inputs and outputs of expressions.

For example, the truth table to the right shows all possibilities for the following expression:

X ⋁ ¬Y

As a Boolean expression, this would be:

```
X or (not Y)
```

X	Y	¬Y	X ⋁ ¬Y
1	1	0	1
1	0	1	1
0	1	0	0
0	0	1	1

Three Representations

Boolean Expressions, Circuits, and Truth Tables can all be used to represent **the same algorithm**. Why do we use all three?

- Boolean Expressions are good for quickly representing an algorithm in text
- Circuits are a more visual option, and more interactive
- Truth Tables lay out all inputs and outputs, which helps derive algorithms

Truth Tables Clarify Complex Expressions

Truth tables are especially useful when you need to determine the output of a fairly complex expression, like the rightmost column here. You can break down the expression into **smaller parts** and give each part its own column.

A	B	C	A ⋀ B ⋀ C	A ⋀ ¬B ⋀ ¬C	¬A ⋀ B ⋀ ¬C	¬A ⋀ ¬B ⋀ C	(A⋀B⋀C) ⋁ (A ⋀ ¬B ⋀ ¬C) ⋁ (¬A ⋀ B ⋀ ¬C) ⋁ (¬A ⋀ ¬B ⋀ C)
1	1	1	1	0	0	0	1
1	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	0	0	0	1	0	0	1
0	1	1	0	0	0	0	0
0	1	0	0	0	1	0	1
0	0	1	0	0	0	1	1
0	0	0	0	0	0	0	0

Deriving Algorithms from Truth Tables

If we know a set of inputs and outputs as a truth table, we can derive an equation to represent the inputs and outputs by looking for patterns that match the gates we know how to build.

For example, let's derive an algorithm to produce the truth table shown below.

A	B	C	Output
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	0
0	0	0	0

Deriving Algorithms from Truth Tables

First, note that the Output is only 1 when B is 1. That means that B is **required**, so the algorithm can use B ∧ ??? [`B and ???`] as a first step, to account for 4/5 of the 0s

What should the ??? value be? Note that the only time B is 1 and the Output is 0 is when A and C are both 0. This corresponds to A ∨ C [`A or C`].

Our final equation is B ∧ (A ∨ C) [`B and (A or C)`].

A	B	C	A ∨ C	B ∧ (A ∨ C)	Output
1	1	1	1	1	1
1	1	0	1	1	1
1	0	1	1	0	0
1	0	0	1	0	0
0	1	1	1	1	1
0	1	0	0	0	0
0	0	1	1	0	0
0	0	0	0	0	0

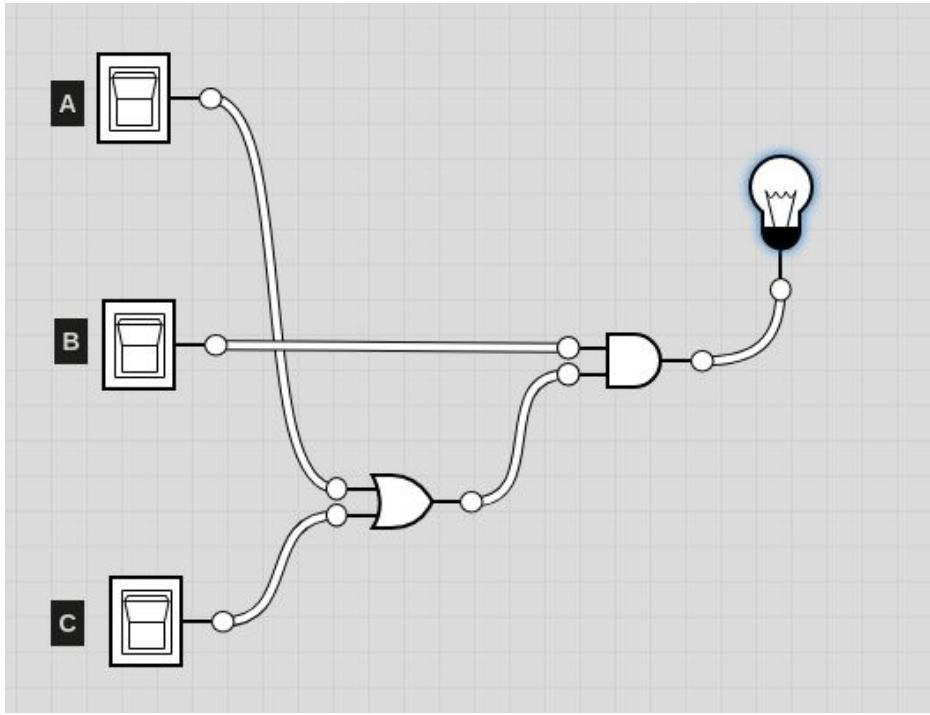
Truth Table to Boolean Expression to Circuit

Once we've used a truth table to figure out a logical expression, we can use it to create a corresponding circuit.

Just combine the appropriate gates in the order specified by the parentheses.

The circuit to the right has the exact same behavior as the truth table we made before, as it combines an Or gate and an And gate in the same order.

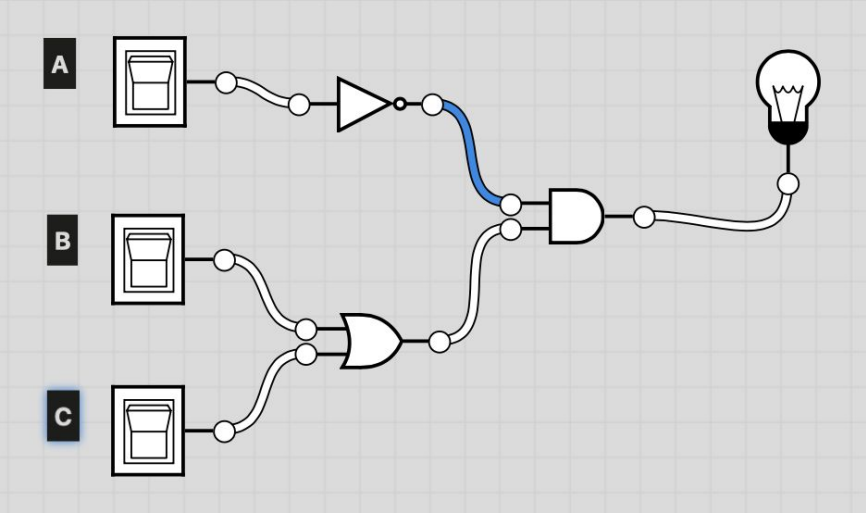
$B \wedge (A \vee C)$



Circuit to Boolean Expression to Truth Table

Likewise, given a circuit, we can construct its truth table.

Given the circuit shown below, we can construct a truth table either by logically determining the result, or by simulating all possible input combinations.

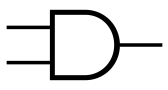


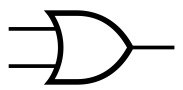
A	B	C	Output
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	1
0	0	1	1
0	0	0	0

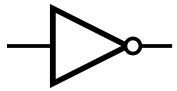
Activity: Find the positive inputs!

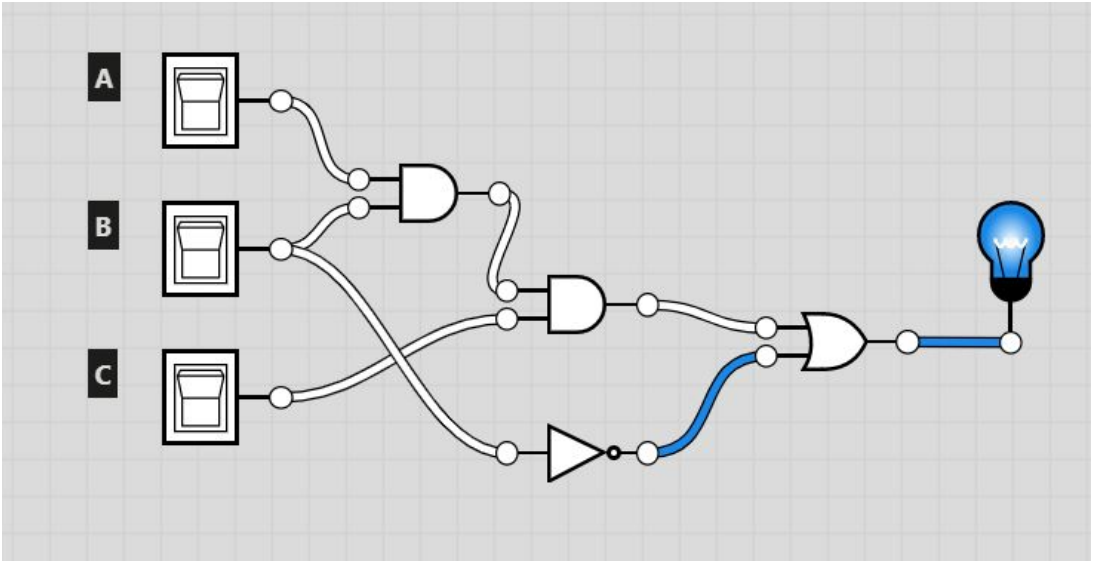
Convert the following circuit to the equivalent Boolean Expression, then write the equivalent truth table.

Which input combinations will result in the circuit outputting 1 (the light bulb lighting up)?





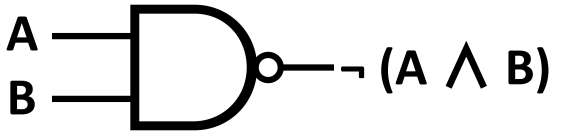




A Few More Gates

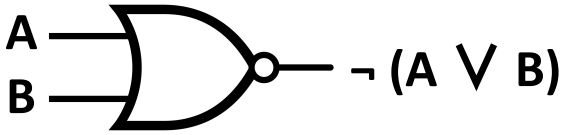
Let's add a few more gates to simplify our circuits.

A **nand** gate is ¬ (A ∧ B)



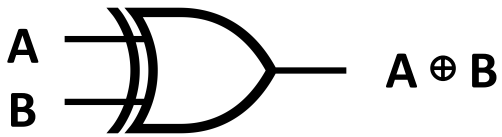
A	B	¬ (A ∧ B)
1	1	0
1	0	1
0	1	1
0	0	1

A **nor** gate is ¬ (A ∨ B)



A	B	¬ (A ∨ B)
1	1	0
1	0	0
0	1	0
0	0	1

An **xor** gate is 1 if **exactly one** of A and B are 1 (and the other is 0). It is the same as (A ∧ ¬B) ∨ (¬A ∧ B).



A	B	A ⊕ B
1	1	0
1	0	1
0	1	1
0	0	0

Sidebar: All You Need is Nand

You can actually emulate **any gate** using just **nand** gates. This means you can build any circuit using nothing but nand.

For example, `not x` is equivalent to `x nand x`.

Optional take-home activity: see if you can figure out how to represent `x and y` & `x or y` using just nand!

Abstraction with Gates

Writing Real Algorithms with Circuits

Now that we know the basics of interacting with gates and circuits, we can start building circuits that do real things.

We'll focus on a basic action that computers do all the time:
integer addition.

Addition with Gates

Let's say that we want to build a circuit that takes two numbers (represented in binary), adds them together, and outputs the result. How do we do this?

First, **simplify**. Let's solve a sub problem. How do we add two one-bit numbers, X and Y? What are all the possible inputs and outputs?

Note that 1 + 1 = 10, because we're working in binary

X	Y	X + Y
1	1	10
1	0	01
0	1	01
0	0	00

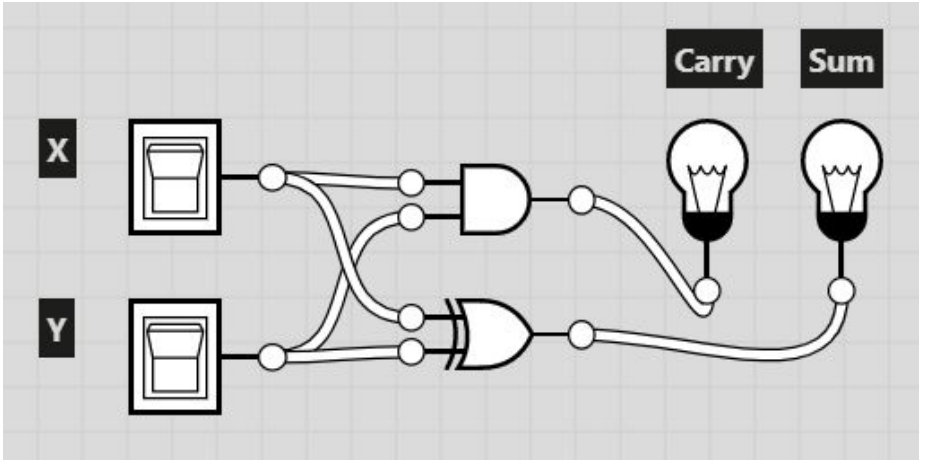
Addition with Gates – Half-Adder

Because we need two digits to hold the result, we need two result values: Sum (the 1s digit) and Carry (the 2s digit).

How can we compute Sum and Carry logically? Examine the truth table: Sum is just an Xor function, and Carry is just an And function!

We can make a circuit to do one-bit addition, as is shown on the right. This is called a **Half-Adder**.

X	Y	X + Y	X ∧ Y	X ⊕ Y	Carry	Sum
1	1	10	1	0	1	0
1	0	01	0	1	0	1
0	1	01	0	1	0	1
0	0	00	0	0	0	0



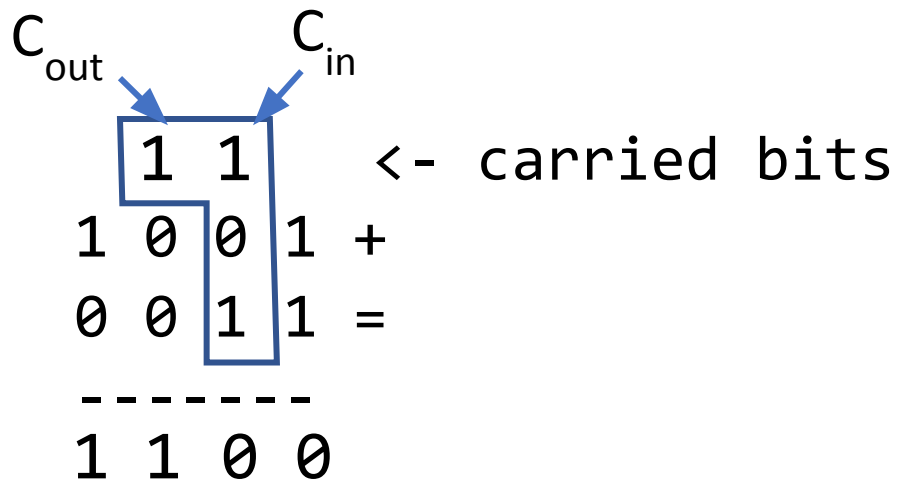
Addition with Gates Over Multiple Digits

Now expand the circuit to handle numbers with multiple bits (e.g. 4-bit numbers). What needs to change?

When adding two numbers, we might need to **carry** an output over to the next column of the addition.

For the two's column, call the carried-in bit C_{in}, and next carry C_{out}.

We need to modify our half-adder to have a third input C_{in} and update the computations for C_{out} and Sum.

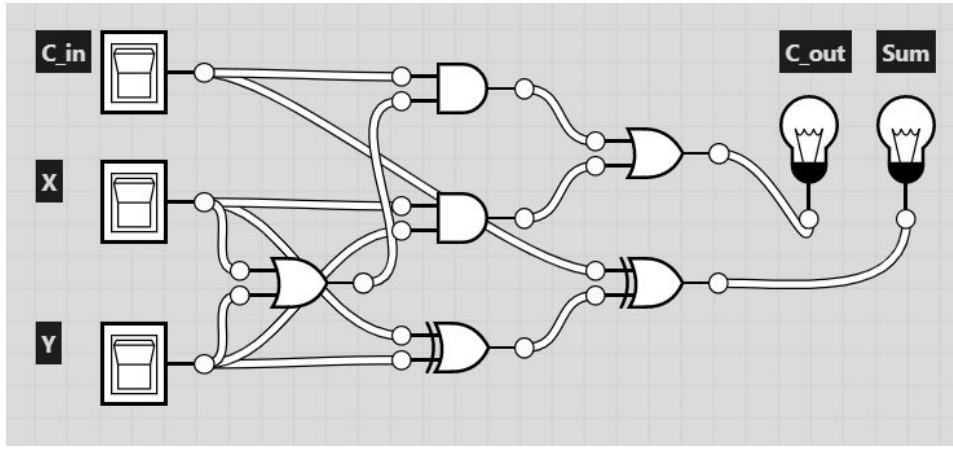


Addition with Gates – Full Adder

To calculate C_{out}, note that it is equivalent to X $\vee$ Y [X or Y] when C_{in} is 1, and equivalent to X $\wedge$ Y [X and Y] when C_{in} is 0.

Sum is now the result of Xor-ing C_{in} and (X $\oplus$ Y).

C_{in}	X	Y	C_{in} + X + Y	((X $\vee$ Y) $\wedge$ C_{in}) $\vee$ (X $\wedge$ Y)	(X $\oplus$ Y) $\oplus$ C_{in}	C_{out}	Sum
1	1	1	11	1	1	1	1
1	1	0	10	1	0	1	0
1	0	1	10	1	0	1	0
1	0	0	01	0	1	0	1
0	1	1	10	1	0	1	0
0	1	0	01	0	1	0	1
0	0	1	01	0	1	0	1
0	0	0	00	0	0	0	0

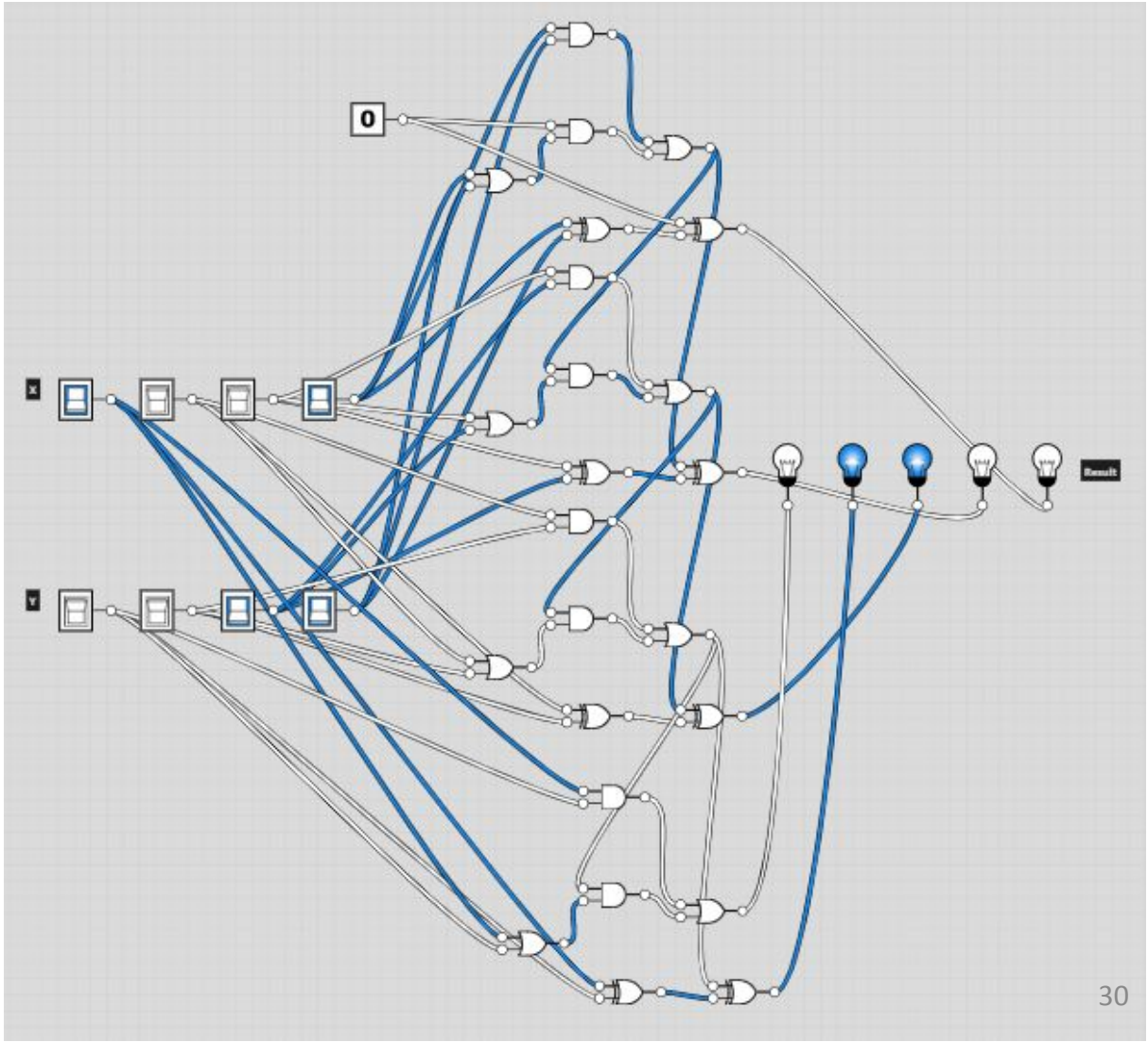


Addition with Gates – N-bit Adder

Finally, we want to add two 4-bit numbers together. If we want to add two four-bit numbers together, we can just **chain together** the Full Adder we've created four times.

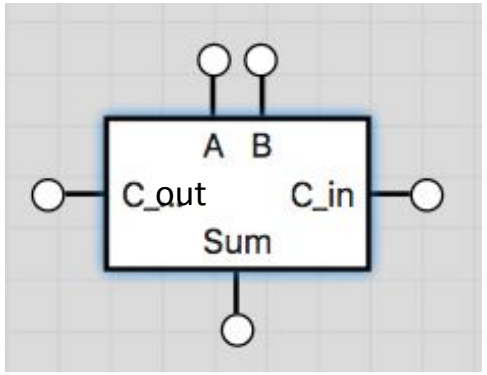
Instead of inputting C_{in}, we pass in the C_{out} from the prior computation (and pass in 0 for the 1s digit). This process repeats the concept of the Full Adder multiple times, in order to make a more complex circuit.

The result is really confusing to look at...



Addition with Gates – N-bit Adder

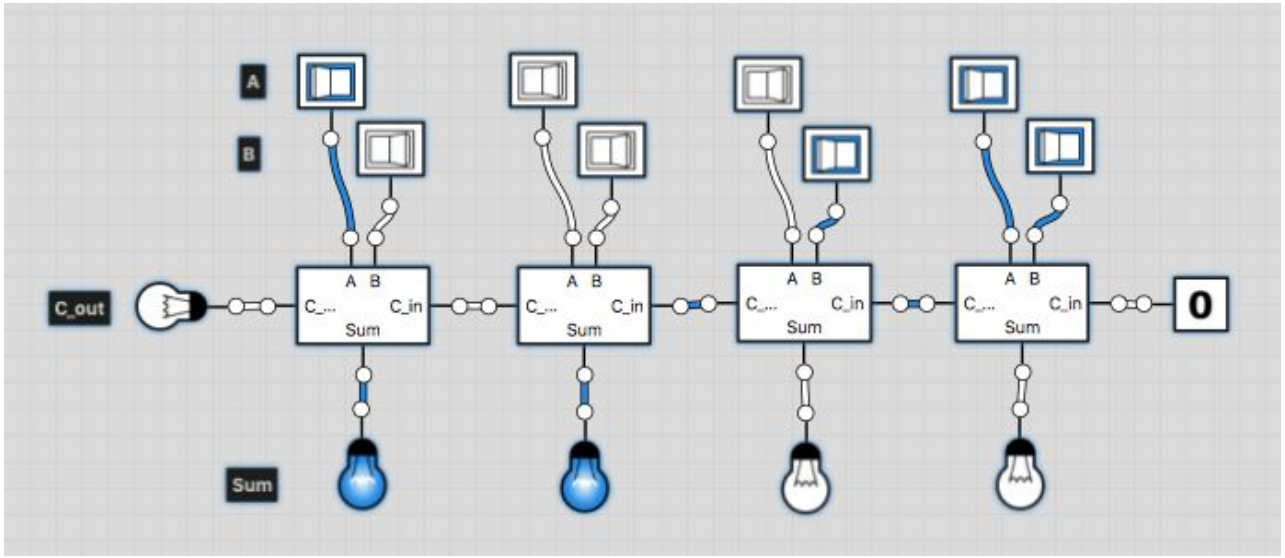
To make this easier to understand, use **abstraction** to replace each Full Adder with a box. That box holds the Full Adder circuit within it, but it doesn't need to bother with all the internal components.



Now we can do proper addition!

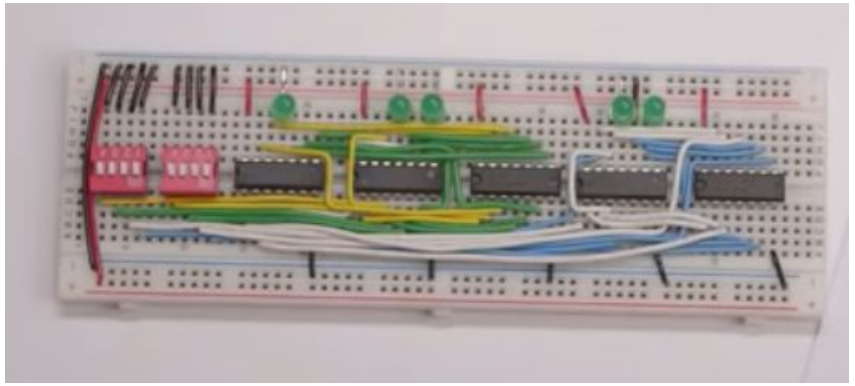
Let's try it out. What's 9 + 3?

- 9 is 8+1=1001, 3 is 2+1=0011
- Walk through the full adders...
- The output is 1100=8+4
- That's 12! It works!



See a 4-bit Adder in Hardware

You can use the abstract circuit we've designed to build an **actual hardware circuit** that does 4-bit addition (or more!).



See a demo of what that looks like here: https://youtu.be/wvJc9CZcvBc?t=742

Learning Goals

- Translate Boolean expressions to **truth tables** and **circuits**
- Translate **circuits** to **truth tables** and Boolean expressions
- Recognize how addition is done at the circuit level using **algorithms and abstraction**
- **Feedback: https://bit.ly/110-feedback**