Circuits and Gates

15-110 – Wednesday 09/11

Learning Goals

Understand how computers **organize** data at a low level

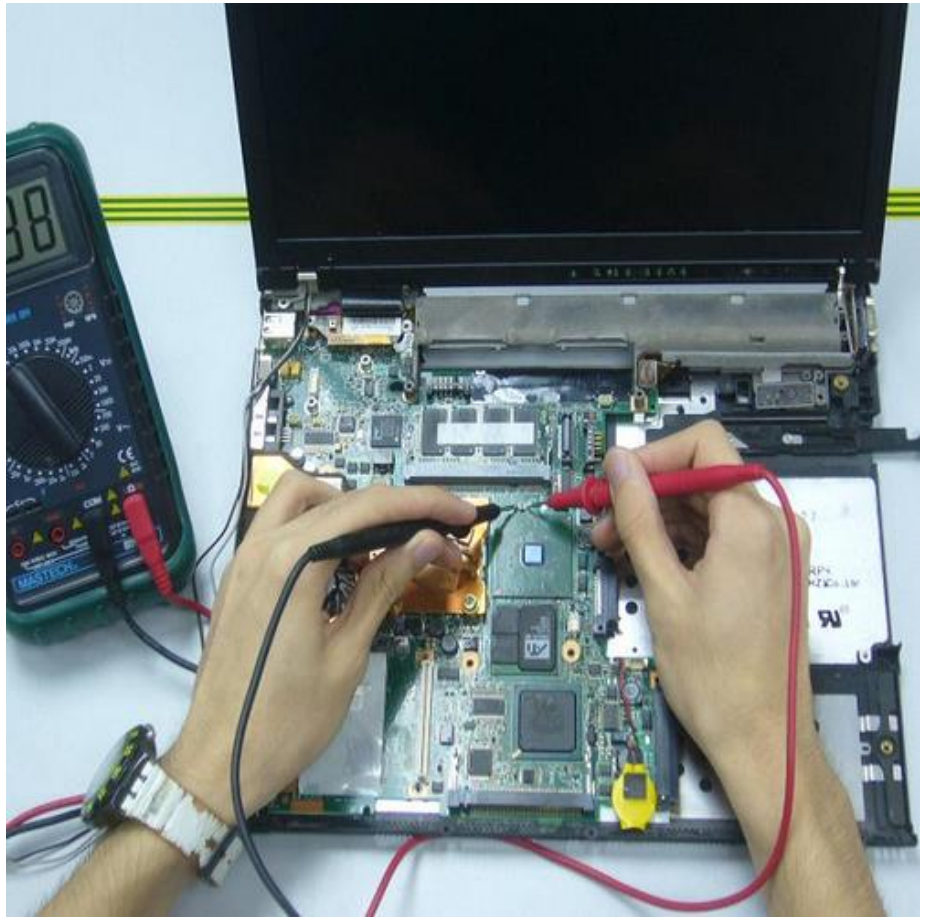
- Computers represent bits as **electricity** at the hardware level
- Computers use **circuits and gates** to manage electrical flow and perform computations
- We can use **algorithms and abstraction** to develop circuits that perform procedures

Computers Run on Hardware

So far, we've discussed computers in terms of **software**- how computers represent data, and how programs can manipulate data.

However, the computers we use are built on **hardware**, like the internal components of the laptop shown to the right.

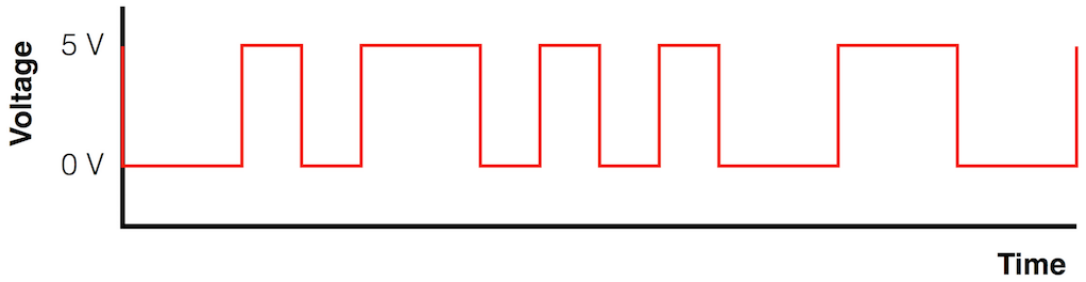
All of the operations we perform on the computer correspond to physical actions within the hardware of the machine. **How does this work?**



Bits are Electric Voltage

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

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



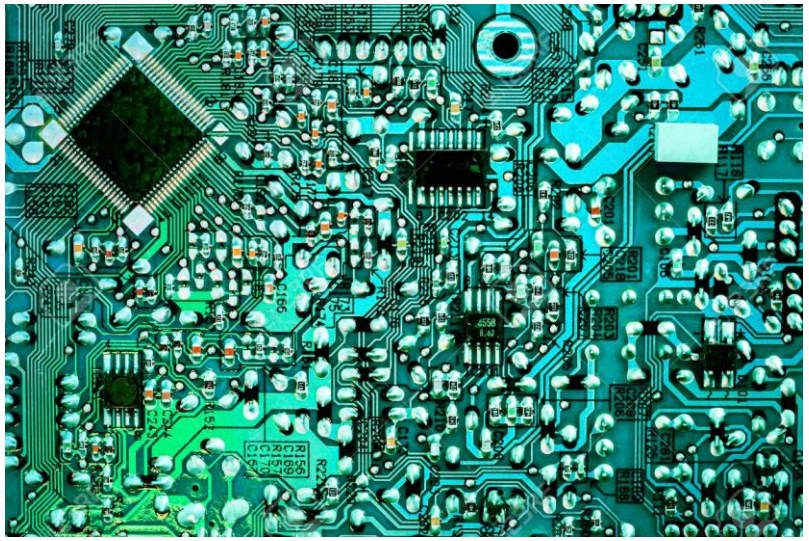
By redirecting electrical flow throughout a system, we can change the values of data in hardware.

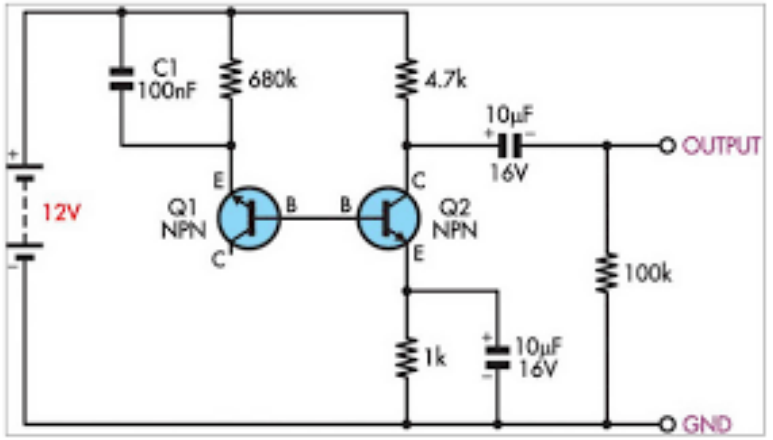
Voltage Passes Through Circuits

To perform an operation (and change data values), the computer uses **circuits** to redirect electrical flow to different parts of the hardware.

We won't discuss how to build the physical components of a circuit (like transistors and capacitors)- you can study electrical engineering for that!

Instead, we will discuss how to put together abstract components of a circuit, called **gates**. Every gate we discuss can be directly translated to a real hardware circuit.





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). Therefore, we can build **gates** to have the same affect as a Boolean operation, but with bits as input/output instead of True/False values.

Let's start by discussing three familiar gates: **and**, **or**, and **not**

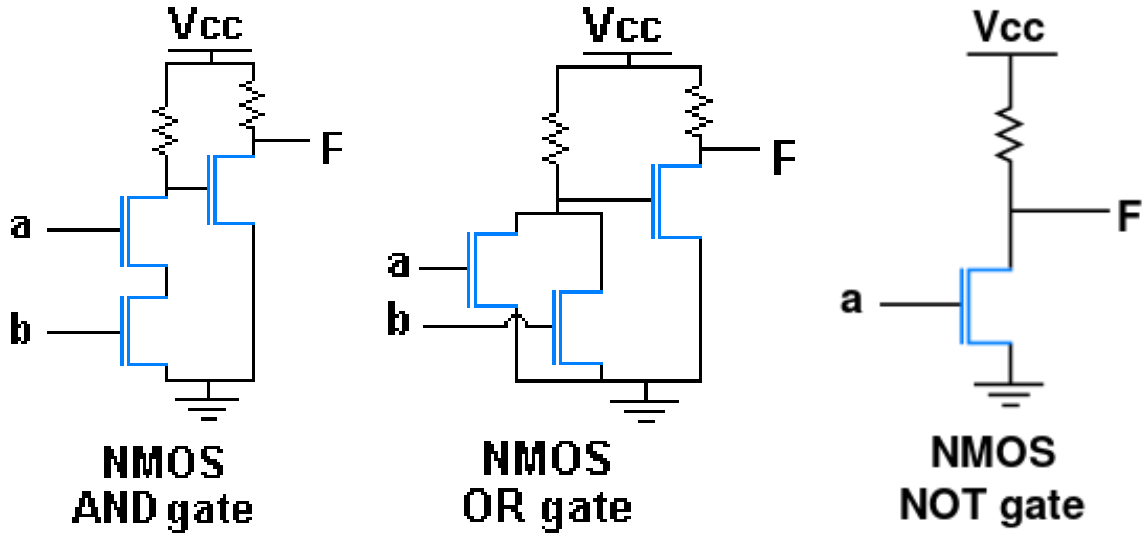
Basic Gates – Actual Hardware

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

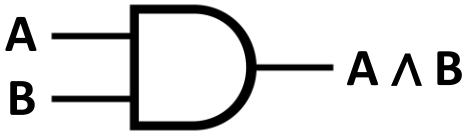
A **not** gate takes one input and outputs the reverse (1 becomes 0, 0 becomes 1)



Basic Gates – Shorthand

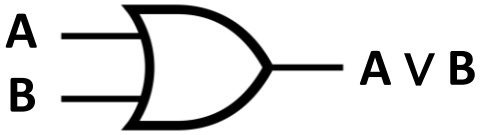
We'll use a shorthand when building circuits with these gates, though

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



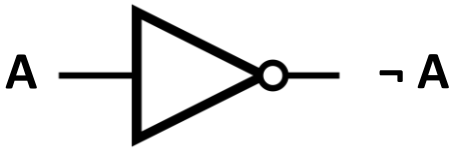
A	B	A ∧ B
1	1	1
1	0	0
0	1	0
0	0	0

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)

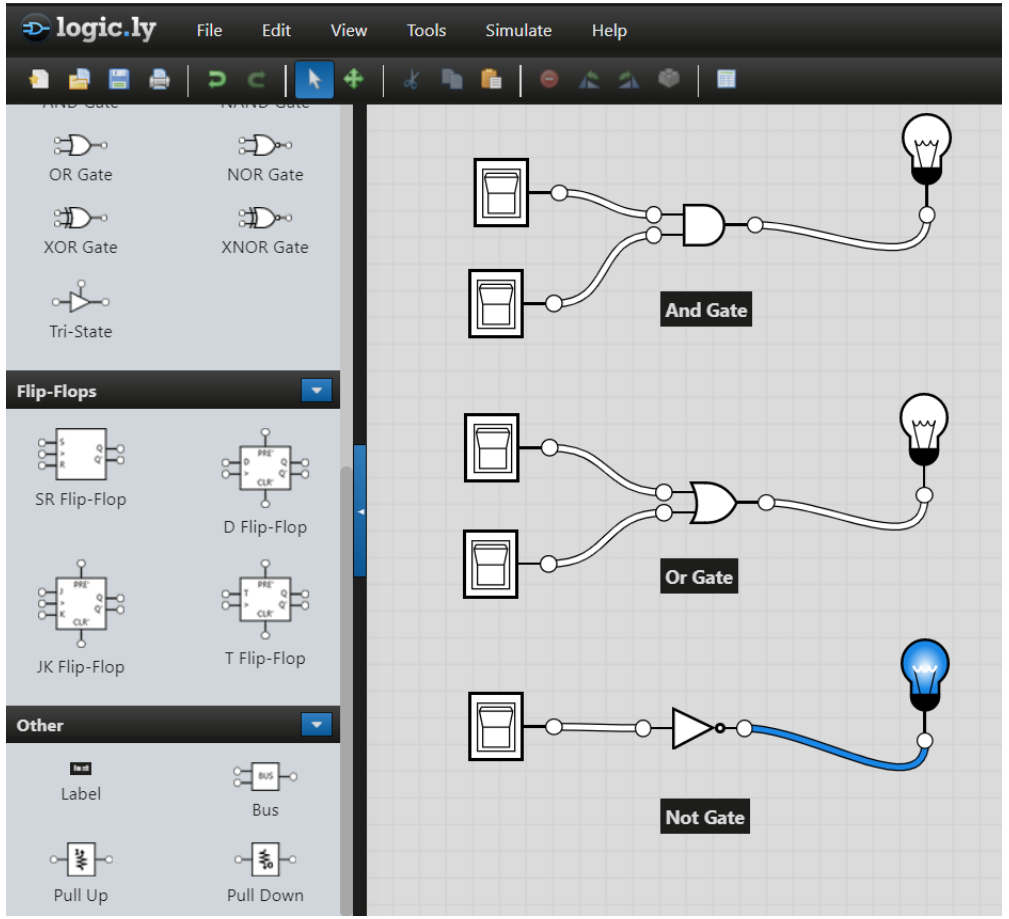


A	¬ A
1	0
0	1

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

Combining Gates

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

Often, when considering algorithms at the gate level, we want to build a **generalizable process** based on a set of inputs and resulting outputs.

We can find the logical operations that will turn the inputs into the needed outputs by constructing a **truth table** out of this information and filling it in with operations.

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 outcomes of more complex expressions.

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

X ∨ ¬Y [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

Truth Table Help Clarify Complex Expressions

Truth tables are mainly useful when you need to determine the output of a fairly complex expression, like the leftmost column here.

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	0	0	0	1	1
1	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	0	0	1	0	0	0	1
0	1	1	0	0	0	0	0
0	1	0	0	1	0	0	1
0	0	1	0	0	1	0	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 always 0 when B is 0. 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)].

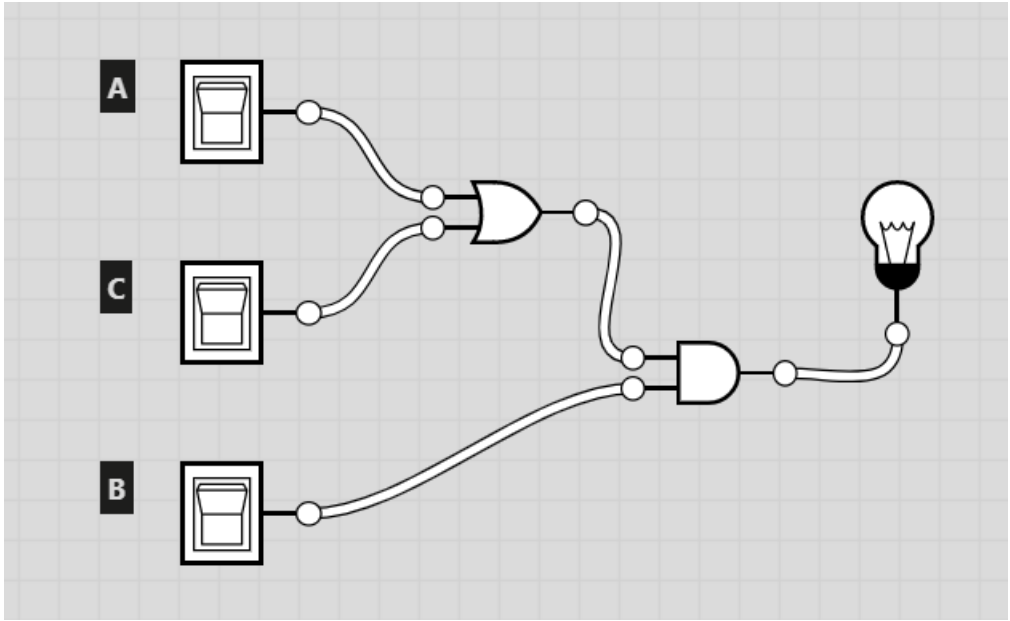
A	B	C	A ∨ C	B ∧ (A ∨ C)	Output
1	1	1	1	1	1
1	1	0	1	1	1
1	0	1	1	0	0
1	0	0	1	0	0
0	1	1	1	1	1
0	1	0	0	0	0
0	0	1	1	0	0
0	0	0	0	0	0

Truth Table to 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.

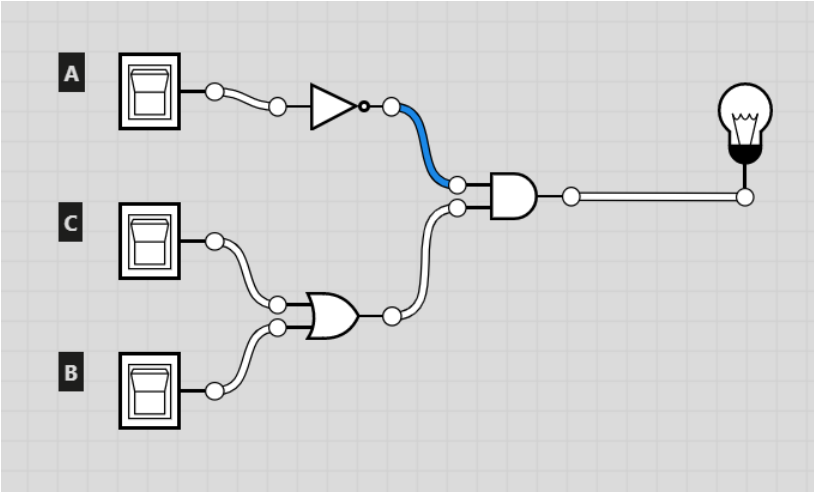
The circuit to the left has the exact same behavior as the truth table we made before, as it combines an Or gate and And gate in the same order.



Circuit 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!

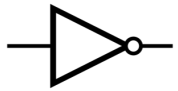
Given the following circuit, which input combinations will result in the circuit outputting 1 (the light bulb lighting up)?

If you aren't sure, try writing out a truth table!

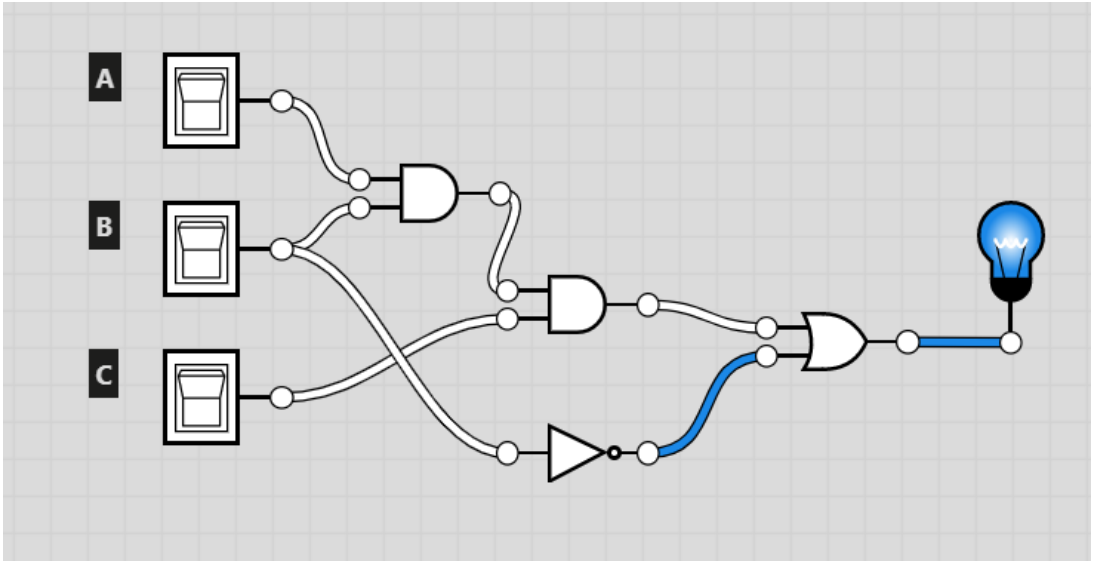
Submit your answer on the live Piazza poll. Make sure to select ALL answers that work!

and

or



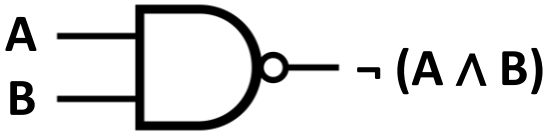
not



A Few More Gates

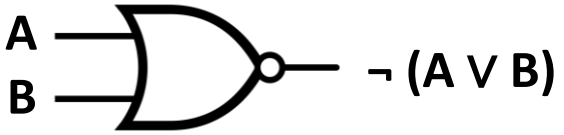
Let's add a few more gates to simplify our circuits.

A **nand** gate is ¬ (A ∧ B)



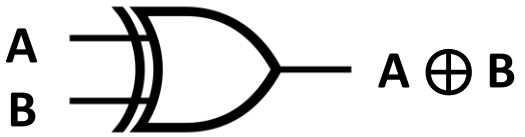
A	B	¬ (A ∧ B)
1	1	0
1	0	1
0	1	1
0	0	1

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

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 two core actions: **addition** and **saving state**.

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

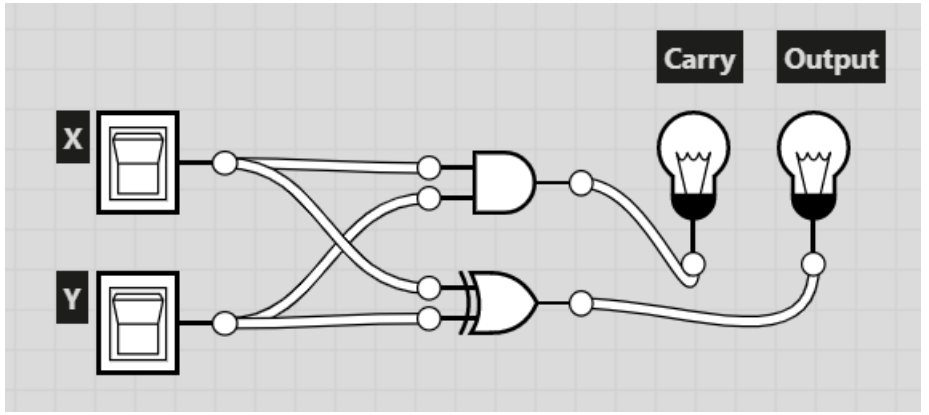
Addition with Gates – Half-Adder

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

How can we compute Output and Carry logically? Note that Output 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	Carry	Output	X ∧ Y	X ⊕ Y
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

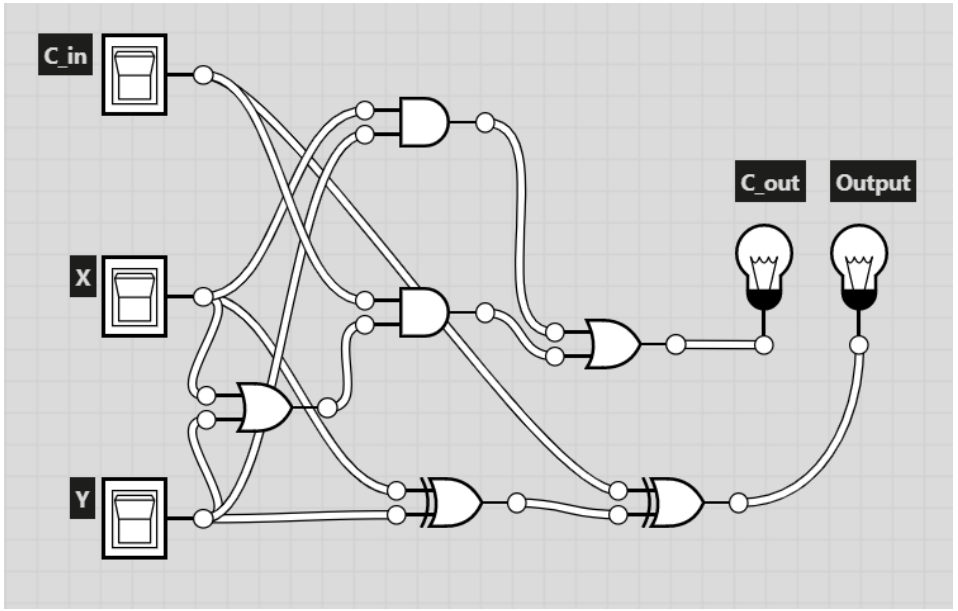


Addition with Gates – Full Adder

Now let's take it a step further. When we're adding a multi-digit number, we also might have to add in a number that was **carried** from the previous digit's addition. We'll call this C_{in}, and change Carry to C_{out}. We should add C_{in} as a third input, and update C_{out} and Output.

Now, to calculate C_{out}, note that it's equivalent to X ∨ Y [X or Y] when C_{in} is 1, and equivalent to X ∧ Y [X and Y] when C_{in} is 0. On the other hand, Output is now the result of Xor-ing C_{in} and (X ⊕ Y).

C_{in}	X	Y	C_{in} + X + Y	C_{out}	Output	((X ∨ Y) ∧ C_{in}) ∨ (X ∧ Y)	(X ⊕ Y) ⊕ C_{in}
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

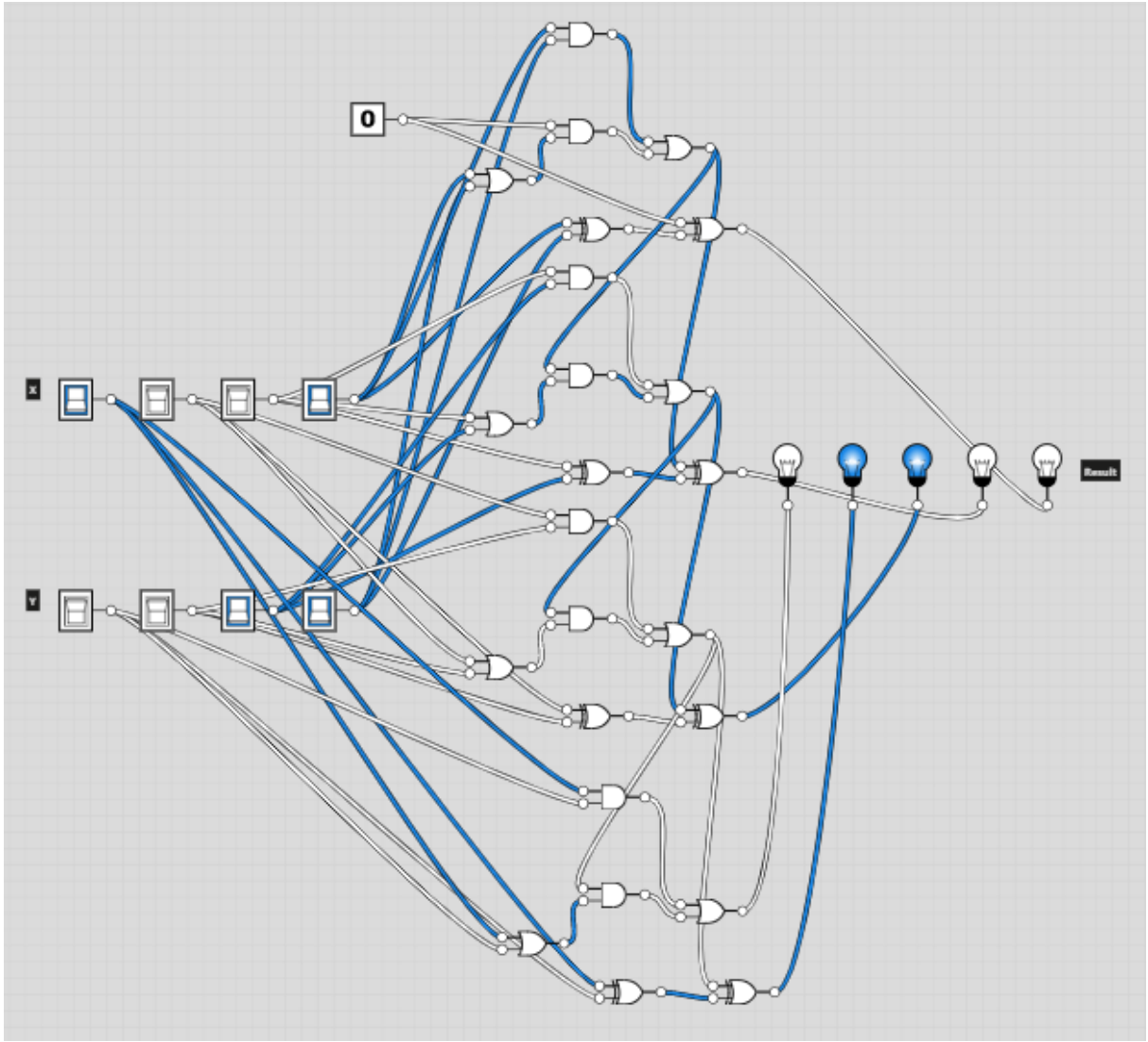


Addition with Gates – N-bit Adder

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

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.

But the result is really confusing to look at...



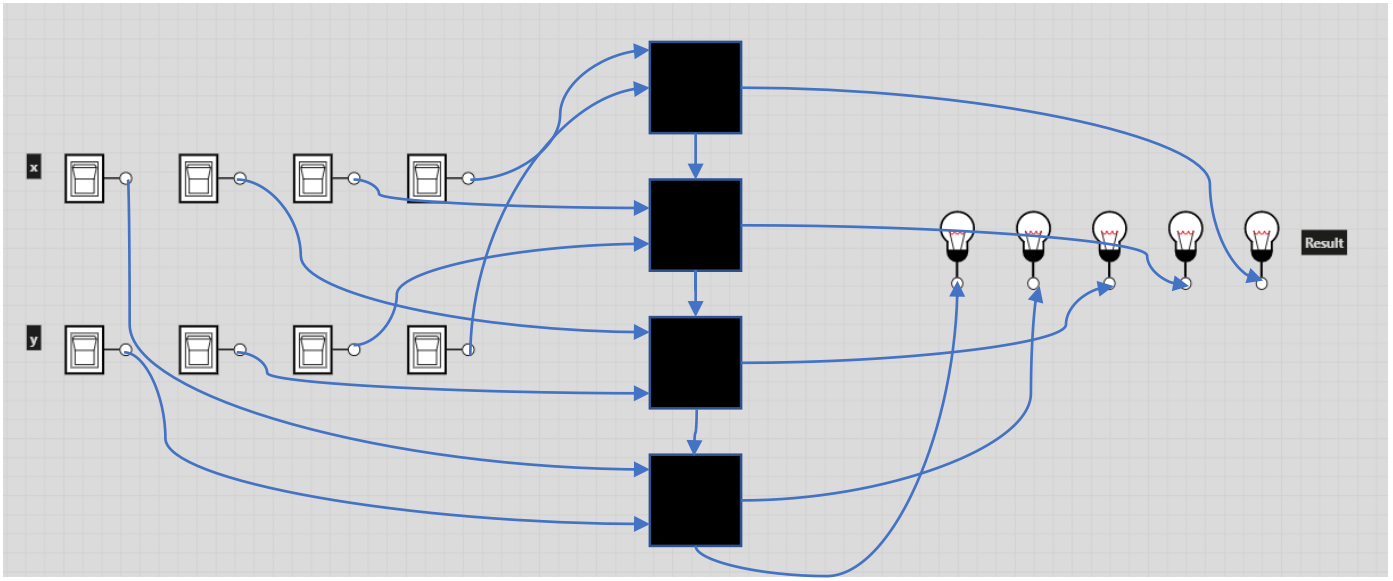
Addition with Gates – N-bit Adder

To make this easier to process, 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
- The output is 1100=8+4
- That's 12! It works!



Saving Bit State

We can now compute addition, but the result disappears if we change the inputs. How can we **save data state** so that we don't need to keep providing the same inputs?

Think of saving state as having two inputs: one that tells you when to **set** the state, another that tells you when to **reset** the state back to 0. If the Reset input is 1, the Output should always be 0. Otherwise, if the Set input is 1, the Output should be 1.

How do we make the Output stay as 1 if we turn the Set Input on and then off? We need to make the output state **independent of the input**. To do that, we'll try a neat trick: provide the previous output of the circuit **as input to the next output**. As electricity cycles, it will keep sending a high voltage signal back through the circuit, so the output value will stay the same until it's told to turn off by Reset.

Saving Bit State – Truth Table

To make a truth table for saving state, start with the easy rows- any row where Reset is 1 (Output 0), and any row where Set is 1 (Output 1).

We're left with two rows- one where Prev. Input is 1 (and the Output should be 1), and one where all values are 0 (and Output should be 0).

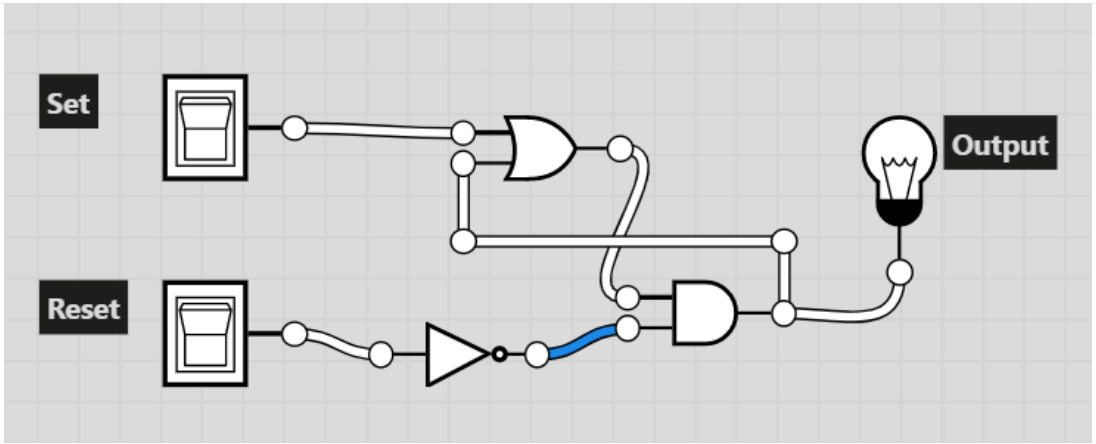
Set Input	Prev. Input	Reset Input	Output
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	0
0	**1**	**0**	**1**
0	0	1	0
0	0	0	0

Flip-Flop Circuits

To build a circuit out of the truth table, we'll need to make the Output 0 if Reset is 1 (use And and Not), and otherwise 1 if Input or Prev. Input is 1 (use Or). The resulting circuit is shown to the right.

This circuit is called a **flip-flop**, and can be used to store state in a computer.

Note that a flip-flop will only work if electricity is flowing through the computer. We need to use other approaches to save state when the power runs out!



Learning Goals

Understand how computers **organize** data at a low level

- Computers represent bits as **electricity** at the hardware level
- Computers use **circuits and gates** to manage electrical flow and perform computations
- We can use **algorithms and abstraction** to develop circuits that perform procedures