# Introducing Computer Theory to Fourth Graders

Jesse Hamner

2020

*Draft Copy: Do not cite.*

**Abstract**

Kids are using technology that is intuitive to control, but entirely opaque to the children's understanding. Why do computers work? How do computers count and add numbers? What makes transistors so important? How can we put transistors together to make a computer?

This workshop aims to produce a short course of study for third to sixth graders, that takes about three afternoons to complete. Children are introduced to math concepts fundamental to the study of computing such as binary counting, exponentiation, basic electronics theory, and computing logic. Several hands-on projects are included in the text and can enhance the students' learning while breaking up the "lecture" format of the workshop.

**Keywords:** STEM; education; computing theory; logic; electronics theory; elementary education; logic gates; science experiments.

All source code for this document are available at the author's github page.

# Contents



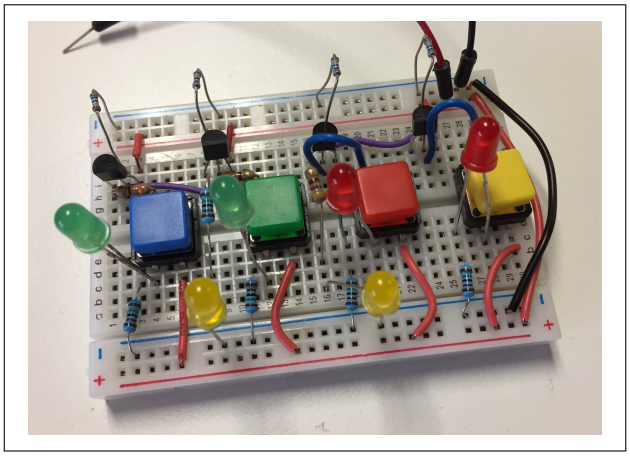


Figure 1: Two logic gates (OR & AND) implemented with N-type transistors on a solderless breadboard. Note pull-down resistors connecting transistor gate pins to ground.

# 1 Introduction For The Participants

This workshop is going to help you learn about computers – but not how to *use* computers. Rather, when we are finished, you will understand a little bit about how and why computers really do work and get answers to math problems. To get to that goal, you'll need to learn some new topics in math, and learn about electricity. You'll also need to learn about basic *electronic components*, too. These are the small pieces that are used together in *circuits* to do useful things with electricity. When assembled, circuits (whether simple or complex) enable electricity to do all the things we can make electricity do, like light our homes, play music on the radio, find directions with a GPS, run a gasoline engine, or show you a web page.

Computers are wonderful, multipurpose machines that take in data, store data, and change data according to instructions that are provided to the computer. But computers can't add or subtract the way you do. They can't read the way you do. They don't think. Really, *they only do math.* Everything computers do is controlled by a series of instructions that *all boil down to math and logic,* even if it looks like magic. The way computers do all the things they do can always be described as a series of instructions. Each of these actions can be performed by some combination of special circuit types, and each circuit is made up of basic electronic components. You can learn enough about each basic component to understand what it does, without needing to do a bunch of math. And, you can understand all the individual instructions, even if you don't want to build your own computer!

Ultimately this workshop will make it possible for you to make, from the most basic electronic components, a simple digital computer that can add two numbers and correctly display the result, or tell you whether or not two numbers are equal. Along the way you'll see how each piece works, and how to put the pieces together. Some workshops will also teach you how to solder components, so you can actually *make* some of the building blocks of computers. There are follow-on discussions about other topics like subtraction and using circuits to store information so a computer can use it later.

This time together should be fun! You can help make it fun for everyone by asking questions and by telling the instructor what you think about the parts of this course. It's not easy to teach this material, so your feedback will help the authors improve this course over time, as the feedback from many people has already improved this workshop.

I'm so glad you're interested in computers. It makes me feel good when people learn things and have fun with this course. Thank you!

Jesse Hamner, PhD
2018–2021

# 2 Introduction for the Adults

A few years ago my friend Adam asked me if I would teach his son about how computers work – not "how do I use Windows" or "tips and tricks for a Raspberry Pi", but rather, how binary works and how logic circuits can add or subtract numbers. He was seeking people to instruct on a very wide variety of topics. In a broader context, he is putting his child on the path to becoming a more human instance of Heinlein's *Competent Man* or, at some level, a polymath. He wanted an afternoon to do it, but this project's requirements mushroomed into a two-day event, or three days if you want to introduce soldering to the participants. I couldn't find any kid friendly workshops like the one I ended up writing, though this project has lots of potential, as does the Crash Course Computer Science series.

My daughter is pretty tech savvy and, like Adam's son, has expressed at least a passing interest in the Raspberry Pi. She has also helped me solder up some boards and fab some antennas. So I was willing to at least attempt the lesson, and I'd throw my kid into the mix too (they're already friends from school). Both kids were fourth graders in 2017, and so I tried to pitch the workshop to them.

## What I Did

Anyone who knows me knows that if you ask me what time it is, I often tell you how to build a clock. This is typically a problem (people don't *want* to hear the story of how the earth cooled, but in order to teach a kid what a logic gate is and how we use them to do math, I was going to have to start pretty far back down the ladder. At least, in order for a kid to really understand how a transistor enables a logic gate, I want them to know a little more than the typical "transistors are just like light switches, okay?"

I did write a little more explanation of basic electronic components than was strictly necessary. And after giving the first bit of the lecture, I learned that I left too much out of the "binary" section, though the kids really enjoyed learning how to count to 31 on one hand especially binary "4", which is just the middle finger!

The basic flow of the lesson is:

- enough math to understand binary
- enough electronics to understand how electricity flows
- enough circuit theory to understand the common logic gates, and
- enough digital logic to see how computers can add or compare two numbers.

That's actually a *lot*. This workshop takes around 5 hours of total time, if you don't do any soldering. It is best split into two sessions, and perhaps three if you teach soldering.

I added "experiments" or "applications" like the classic lemon battery, a saltwater resistor, a Leyden jar (the earliest capacitor), and a cardboard/aluminum foil variable capacitor. There are a handful of thought experiments designed to introduce slightly advanced

topics like integer overflow without making it too hard. It might be useful to make an aluminum foil/corrugated cardboard multi-layer capacitor too, either beforehand or in place of the Leyden jar.

## Next Steps

I've produced some simple logic gate PCBs for discrete components, including OR, AND, NOT, NOR, NAND, and even XOR, but not XNOR gates (although I demonstrate using a NOT gate with an XOR gate in the text) and am including circuit board design files in this repository. Additionally, I have designed some full-adder PCBs using integrated components at the gate level. As each gate is discrete, it is still easy to see the design of the full adder. I may also wire up a 4-bit adder, since stacking all these gates together to make even a 4-bit adder will get ungainly and will be very slow to implement and tough to troubleshoot.

The next hardware widget will be a JK flip-flop gate, to provide some memory to write to and from which to read. The only new component it needed was a three-input NAND gate, and I designed and tested that component, so the work can continue.

As an added bonus, the kids can learn to solder (I've got a separate tutorial in this repository for that, and a slightly more challenging Atari Punk Console soldering project repository as well). Thus, kids can make the same gates used in the tutorial, since I use entirely through-hole components for the discrete gate PCBs.

Participants can solder the gates if they want to, but the workshop should provide enough to make some complete logic circuits without that portion of the workshop, so soldering is optional. A few other PCBs like a 4-switch deck of toggles, a 4-bit XNOR comparator, and small boards of 4 or 8 LEDs (the designs are or will be included in this repository) will be very useful.

# 3 Decimal and Binary Math

The next few sections introduce some math ideas that may or may not be familiar to you. In order to really understand how computers work, you need to understand several math concepts that are not hard, but may be unfamiliar. The first concept is the *exponent*.

## Bases and Exponents

Exponentiation is a mathematical operation, written as $b^n$, involving two numbers, the base, $b$, and the exponent, $n$. Exponentiation means "repeated multiplication of the base."

That is, $b^n$ is the product of multiplying the base number $n$ times:

$$b^n = \underbrace{b \times \cdots \times b}_{n}$$

In this case, $b^n$ is called the "*n-th* power of b", or "b raised to the power n." So 3 to the $4^{\text{th}}$ power looks like this:

$$3^4 \;=\; 3 \times 3 \times 3 \times 3 \;=\; 9 \times 9 \;=\; 81$$

Put another way: the exponent is a number, smaller and on the upper right hand side of a number, that means "multiplying a number times itself zero times, once, or more than once" depending on whether the number is 0, 1, 2, or another number (written $n$ in the description above). It is possible to use fractions as exponents, but we won't talk about that here.

The base can be any number, though most people think most easily in base-10. So in base-10, 10 to the power 2, or 10 to the second power, means $10 \times 10$.

## Order of Magnitude (Columns)

When counting up from zero, by one, in base 10, you eventually get to 9. In order to count any higher, you must "carry the one" over to the next column and reset the counter in the "ones" column (the place that counts by one, from zero to nine). When you move from the "*ones*" column to the "*tens*" column, the next column to the left represents the next *order of magnitude* of the base. "Magnitude" means "size", and in math, it means, specifically, moving from counting by one number (the base) to counting by the base times itself – first twice, then three times, then more. For base-10, you count from 0 to 9 in the right-most column, then from 10 to 90 in the next column to the left, then from 100 to 900 in the next column to the left, then from 1000 to 9000, and so on. Each time the counter gets full (reaches 9), you cannot represent any more of the quantity being counted without moving to the next largest order of magnitude. That is, if your display only shows two digits, once you count past 99, you have no idea is the number is 0, or 100, or 200, or 900, or ten thousand, or 4 billion.

## Squares

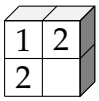
Shapes that are squares have four sides with the same length; that is, the length and width are the same. *Squaring a number* is multiplying the number times itself, just like, in a square, both sides have the same length. The most common way you will see a "squared number" described is with a little 2 up above the number, like $2^2$, or $3^2$, or $10^2$. That smaller '2' is the exponent, that we discussed above.

Squaring the number 10 (that is, multiplying ten times itself) gets: $10 \times 10 = 100$

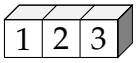

$1 \times 1 =$


$= 1^2$

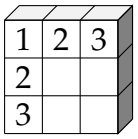

$2 \times 2 =$



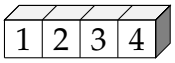

$= 2^2$



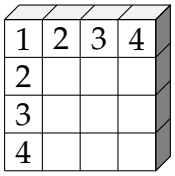

$3 \times 3 =$



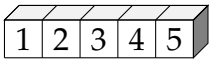

$= 3^2$



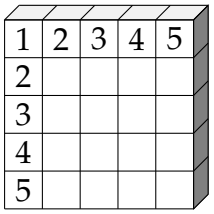

$4 \times 4 =$



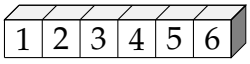

$= 4^2$



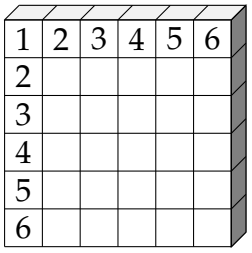

$5 \times 5 =$




$= 5^2$




$6 \times 6 =$




$= 6^2$

## Cubes

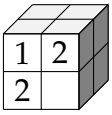
Cubes are the same length on each of *three* sides. When cubing a number, you are multiplying the number times itself, and then multiplying it times itself *again*, because all three sides are the same (number). The most common way you will see a "cubed number" described is with a little 3 up above the number, like $2^3$, or $3^3$, or $10^3$. As above, the exponent tells you how many times you multiply the number times itself (here, that means three times).

Cubing the number 10 gets: $10 \times 10 \times 10 = 1000$

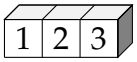

$1 \times 1 \times 1 =$


One times itself is one.

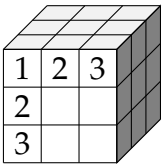

$2 \times 2 \times 2 =$



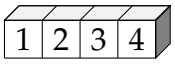

Two sets of four, or
$4 + 4$ (that is, $2^2 + 2^2$)



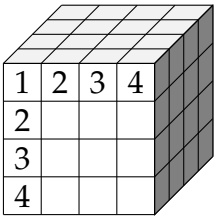

$3 \times 3 \times 3 =$



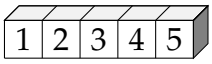

Three sets of nine, or
$9 + 9 + 9$
Put another way:
$9 \times 3 = 27$



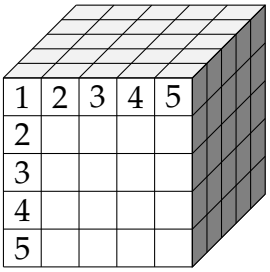

$4 \times 4 \times 4 =$




Cubes get big quickly—
Here's four sets of 16.
$16 \times 4 = 64$




$5 \times 5 \times 5 =$




Five sets of 25.
$25 \times 5 = 125$

## Multipliers and Prefixes

In systems of measurement, when talking about, say, weight, height, or pressure, there are base units and there are ways to refer to these base units in large multiples, or in tiny fractions. You're probably a little taller than one *meter* in height. And it takes one hundred *centimeters* to add up to one meter. The prefix "centi-" means it takes one hundred of *these* to add up to one of the *base unit*, which in this case is a meter.

The same goes for weight. You probably weigh between 20 and 40 *kilograms*. The prefix "kilo-" means *one thousand* of whatever the base unit is. Put another way, grams are pretty small amounts of weight, so measuring things like people or cars is impractical if we use grams, because cars weigh millions of grams. Medicines, on the other hand, are usually measured in quantities called *milligrams* – each milligram is only $\frac{1}{1000}$ (one one-thousandth) of a gram. Don't be confused into thinking "milli-" means "million" — "mega" means "million."

When thinking about computers, we hear terms that are often expressed as multiples of things like memory and storage capacity. A kilobyte is 1,000 bytes (a *byte* is 8 individual bits, and each bit is the smallest unit of computing – it can only represent 1 or 0). A megabyte is 1,000 kilobytes (or, 1,000,000 bytes). A gigabyte is one *billion* bytes, and a terabyte is one *trillion* bytes.

## Representing Numbers in Decimal

*Decimal* means "with tens". You've always been taught to count in the decimal system – by ten. When you are counting by ones, once you get past 9, you reset the right-most number to zero and add that ten to the next column, which is the *tens* column (so you only increment that counter by one, since you are adding *one* "ten" to the total). Once you fill up 9 "tens" (90) and count up past 9 in the "ones" column (that is, you add one to 99), you have to set the ones column and the tens column to zero, and add one to the "hundreds" column. You *carry* the ten to the left from the ones, and carry the hundred to the left, to the next-largest set of tens. One hundred is ten tens, one thousand is 100 tens, and so forth.

It looks like this:

| ten thousands | thousands | hundreds | tens | ones | Number |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 2 | 2 |
| 0 | 0 | 0 | 0 | 3 | 3 |
| 0 | 0 | 0 | 0 | 4 | 4 |
| 0 | 0 | 0 | 0 | 5 | 5 |
| 0 | 0 | 0 | 0 | 6 | 6 |
| 0 | 0 | 0 | 0 | 7 | 7 |
| 0 | 0 | 0 | 0 | 8 | 8 |
| 0 | 0 | 0 | 0 | 9 | 9 |
| 0 | 0 | 0 | 1 | 0 | 10 |

When you get to the bottom of the column, you re-set the counter for that column to zero, and add one to the next column over (carry). So you go from 9 to 10, or 19 to 20, or 99 to 100. So any column can only hold between 0 and 9, before you have to carry over to the next *order of magnitude*, which for a decimal system is *ten times the size of one step in the column*. So when you run out of room for the ones column, you go to the *tens* column, which contains ten times as many units as any single step (from 1 to 2, or from 8 to 9) in the column to the right of it. When you run out of room in the column, you have to carry to the next order of magnitude. You go from the ones, to the tens, to the hundreds (100 is $10 \times 10$), to the thousands (1000 is $10 \times 100$), and so on.

To express a number, we count up the amount of each order of magnitude and add them all together. The number 628 is made of $(6 \times 100) + (2 \times 10) + (8 \times 1)$, for instance.

Here are a few examples of numbers expressed in columns. For each one, you count up how many units in the column exist, then count them for each order of magnitude, and add each value together to get the number being described:

| **Text Description** | ten thousands | thousands | hundreds | tens | ones | **Number** |
|---|---|---|---|---|---|---|
| no ten-thousands, no thousands, no hundreds, no tens, and no ones | 0 | 0 | 0 | 0 | 0 | 0 |
| no ten-thousands, no thousands, no hundreds, no tens, and one ones | 0 | 0 | 0 | 0 | 1 | 1 |
| no ten-thousands, no thousands, no hundreds, no tens, and two ones | 0 | 0 | 0 | 0 | 2 | 2 |
| no ten-thousands, no thousands, no hundreds, three tens, and no ones | 0 | 0 | 0 | 3 | 0 | 30 |
| no ten-thousands, no thousands, four hundreds, no tens, and no ones | 0 | 0 | 4 | 0 | 0 | 400 |
| no ten-thousands, no thousands, five hundreds, five tens, and five ones | 0 | 0 | 5 | 5 | 5 | 555 |
| no ten-thousands, six thousands, five hundreds, no tens, and two ones | 0 | 6 | 5 | 0 | 2 | 6502 |
| six ten-thousands, eight thousands, no hundreds, three tens, and no ones | 6 | 8 | 0 | 3 | 0 | 68030 |

*Exercise:* What if, in the above table, you counted past 99,999? What number would you see? That is, what do you see if there is not a "hundred-thousands" column? What happened to the hundred thousand that should be counted? Do you need to know how many hundred thousands are in the number? How would you handle the need for a larger number, if you have it? (Introduces the concept of *overflow*.)

## Representing Numbers in Binary

Computers only understand "1" and "0" – because it can sense the electricity in a wire that is either *on* (having a detectable voltage greater than zero) or *off* (having a reference voltage that is basically at "zero volts", or "ground."). That means that a computer, in order to add, subtract, or store information, has to express *literally anything and everything it can handle* in terms of either ones or zeroes. This system is called *binary*, because computers only understand two "states" – on, or off. Instead of "base-10" counting (where moving to the next column happens when you pass 9), binary is "base-2" counting; the numbers move to the next column when the number passes 1—because there's no such number as "two" if you're a computer!

In order to add numbers in binary, we can't count to ten. We must count to one, and then, if the resulting number is greater than one, carry the digit to the next column. Counting in binary is interesting because it is different, and because it introduces us to a new way of doing math.

In order to add two numbers, computers have to do the following:

1. store each number in a binary format;
2. compare each column (ones, twos, fours, eights, etc.) of each number and see if the numbers in that column add up to more than one;
3. carry the "one" to the next column (the next *order of magnitude*, which is two times the previous column). That is, if the number goes from one to two, the "one" will go to zero and the "two" will be moved – and added – to the next column over to the left;
4. keep on adding and carrying digits until the addition is complete;
5. count up the values of each order of magnitude (either one, or none) and add them together; and
6. report the new number as a [binary] result.

**Binary and decimal representation for each number from 0 to 15:**

| Description | eights | fours | twos | ones | base 10 |
|---|---|---|---|---|---|
| $0+0+0+0$ | 0 | 0 | 0 | 0 | 0 |
| $0+0+0+1$ | 0 | 0 | 0 | 1 | 1 |
| $0+0+2+0$ | 0 | 0 | 1 | 0 | 2 |
| $0+0+2+1$ | 0 | 0 | 1 | 1 | 3 |
| $0+4+0+0$ | 0 | 1 | 0 | 0 | 4 |
| $0+4+0+1$ | 0 | 1 | 0 | 1 | 5 |
| $0+4+2+0$ | 0 | 1 | 1 | 0 | 6 |
| $0+4+2+1$ | 0 | 1 | 1 | 1 | 7 |
| $8+0+0+0$ | 1 | 0 | 0 | 0 | 8 |
| $8+0+0+1$ | 1 | 0 | 0 | 1 | 9 |
| $8+0+2+0$ | 1 | 0 | 1 | 0 | 10 |
| $8+0+2+1$ | 1 | 0 | 1 | 1 | 11 |
| $8+4+0+0$ | 1 | 1 | 0 | 0 | 12 |
| $8+4+0+1$ | 1 | 1 | 0 | 1 | 13 |
| $8+4+2+0$ | 1 | 1 | 1 | 0 | 14 |
| $8+4+2+1$ | 1 | 1 | 1 | 1 | 15 |

*Problem 1:* what happens in the above table if you add 1 to 15? What number would the computer report if it only has four columns to use? (As above with counting past 99,999, this problem refers to *overflow*).

*Problem 2:* Let's say the machine runs by itself at some regular speed, and adds one to the number each second. Let's also say you are using this counter as a way to control a single blinking light, since "1" means "there is electricity available to that wire" and so a "1" would turn the light on. Look at each column of numbers (that is, each *order of magnitude!*). Remembering that "a line that has a voltage" is a 1 and "no voltage" is a zero, which line (column) would make the light blink fastest? Which line would blink the slowest?

**How Computers Count to 2,020**

| $2^{15}$ | $2^{14}$ | $2^{13}$ | $2^{12}$ | $2^{11}$ | $2^{10}$ | $2^{9}$ | $2^{8}$ | $2^{7}$ | $2^{6}$ | $2^{5}$ | $2^{4}$ | $2^{3}$ | $2^{2}$ | $2^{1}$ | $2^{0}$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | zero |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 2019 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 2020 |

**How Computers Count to 65,535**

| $2^{15}$ | $2^{14}$ | $2^{13}$ | $2^{12}$ | $2^{11}$ | $2^{10}$ | $2^{9}$ | $2^{8}$ | $2^{7}$ | $2^{6}$ | $2^{5}$ | $2^{4}$ | $2^{3}$ | $2^{2}$ | $2^{1}$ | $2^{0}$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | zero |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 32,767 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 32,768 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 65,535 |

If the computer counts up from zero, then once it gets past the $2^{14}$ column, it jumps to the $2^{15}$ (32,768) column.

*Useless skill:* Did you know you can count to 31 on one hand? You have five fingers, and each finger can be open or closed, and $2^5$ is 32. Use your thumb for the "0 or 1" ($2^0$, "two to the zeroth power") column; the digit to its left represents two to the first power (the "twos digit"); the next digit to the left represents two to the second power (the "fours digit"); and so on. Watch out for "4" though!

| 128 | 64 | 32 | sixteen | eight | four | two | one | |
|---|---|---|---|---|---|---|---|---|
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | **Result** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | **2** $(2+0)$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | **3** $(2+1)$ |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | **4** $(4+0+0)$ |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | **7** $(4+2+1)$ |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | **8** $(8+0+0+0)$ |
| | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | _____ |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | _____ |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | _____ |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | _____ |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | _____ |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | _____ |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | _____ |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | _____ |

*Tip:* if you have a sequence of all ones, like 7 or 15 or 31, rather than adding up each binary column, you can just subtract one from the next largest column base number. So `0111` equals `1000` minus one.

**Joke:**

*Remember:* There are only 10 types of people in the world; those who understand binary, and those who don't.

(If the joke needs explanation, write 10 like `0010` and then figure up the value in binary)

# 4 Computer Math

In this section, I introduce a few vocabulary terms that help you understand how computers do certain things, like add two (binary) numbers. In most respects, adding these numbers is no different from how you already add numbers. When the numbers in any column is too big to be held in that column, you must "carry" the extra number to the next larger column. The difference, and it is slightly awkward at first, is remembering that any column can only hold 0 or 1 – if you reach 2, you have to carry the "two" to the next column.

## Addition

Adding two values is straightforward. Simply align the values on the least significant bit and add each column, moving any "carry" to the bit one position left, just like you already do with "regular" (base-10) arithmetic.

```
  Binary:     Decimal:

  0001 0110   (16 + 0 + 4 + 2 + 0) =   22
+ 0000 0011    (0 + 0 + 0 + 1 + 2) =    3
===========                          ====
  0001 1001   (16 + 8 + 0 + 0 + 1) =   25
```

Your turn! Add these numbers in binary, column by column, carrying each "2" as needed. After you add them, convert each number to decimal to check your work.

| | |
|---|---|
| Binary: Decimal:<br><br>0000 0000<br>+ 0000 0011<br>==========  ====<br><br>----------  ---- | Binary: Decimal:<br><br>0000 0010<br>+ 0000 0010<br>==========  ====<br><br>----------  ---- |
| Binary: Decimal:<br><br>0000 0110<br>+ 0000 0011<br>==========  ====<br><br>----------  ---- | Binary: Decimal:<br><br>0000 1010<br>+ 0001 1011<br>==========  ====<br><br>----------  ---- |

There is more information about binary arithmetic in the Appendix – you can see how to represent negative numbers, and how to subtract numbers, in binary, just the way a computer does. But for now, you know enough to move on.

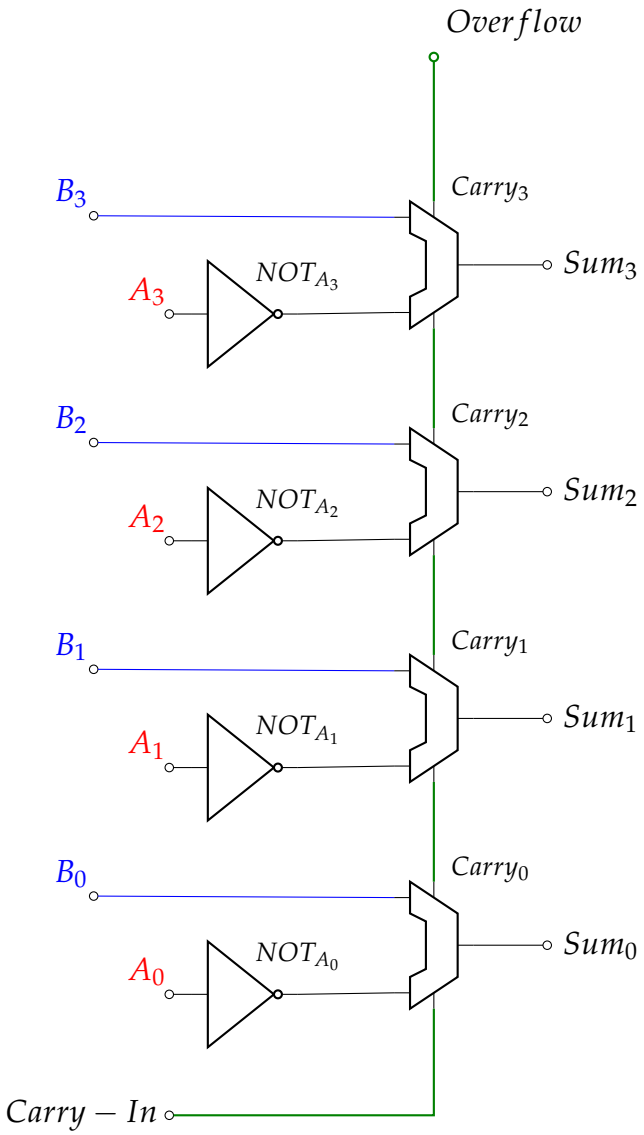



Figure 2: How subtraction gets implemented with logic gates to subtract $A$, a four digit binary number (something like `0010`, or "2"), from $B$ (perhaps `1000`, or "8"). Each pair of bits goes into a full adder (the trapezoid-shaped symbol). Each of these pieces is explained in this workshop – you *really can* understand this by the end of the day!

# 5 Electricity

Electricity is the presence and flow of some amount of electric charge. It is the flow of *electrons* through conductors such as copper wires.

Electricity is a form of energy that comes in positive and negative forms, that occur naturally (as in lightning), or is produced by people (as happens in a generator). It is a form of energy we use to power machines and electrical devices. When the charges are not moving, electricity is called *static electricity* (does that sound familiar? Where else have you heard that term?). Static electricity means a charge is present, but has nowhere to go. When the charges are moving they are an *electric current*, sometimes (but rarely) called 'dynamic electricity'. Lightning is a highly dangerous kind of electricity in nature. Static electricity can cause things to stick together. Electricity can be dangerous, especially around water. Note that *you* are made up of a lot of water.

The experiments we will perform don't use dangerous levels of electricity, so they are safe to handle. You can lick a 9V battery, which feels weird but is not dangerous.

> Experiment: make a lemon battery to power an LED with large-gauge copper wire, and large galvanized screws. At least four, and more practically, six, lemon batteries will be required to power even a modest LED. Wire the batteries in series with alligator clips.

## Charge

Electric charge is a basic property of electrons and protons. Electrons are negatively charged and protons are positively charged. Things that are negatively charged and things that are positively charged pull on (attract) each other. Things that have the same charge push each other away (they repel each other). This behavior is called the *Law of Charges*. Things that have equal numbers of electrons and protons are neutral. Things that have more electrons than protons are negatively charged, while things with *fewer* electrons than protons are positively charged.

## Voltage

Voltage is a force that makes electricity move through a wire. It is measured in volts. Voltage is also called *electric tension* or *electromotive force* (EMF). It was named after Alessandro Volta.

Though it's not the most helpful definition, the best definition of voltage is *the difference in electric potential between two points*. Voltage is always measured between two points, for example between the positive and negative ends of a battery, or between a wire and ground.

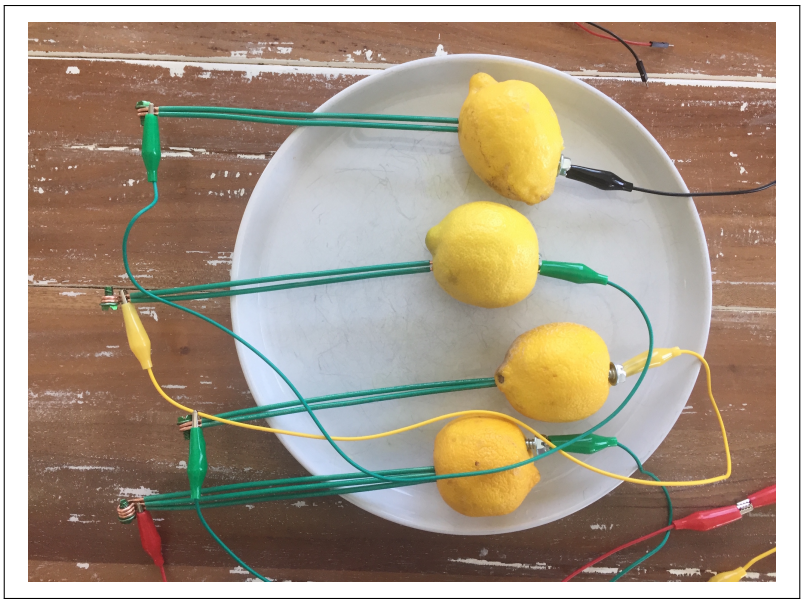


Figure 3: This four-lemon battery was constructed using large zinc-coated sheet metal screws and copper grounding stakes (the plastic coating does not extend all the way down the stake). Together, the lemons made about 4 volts (open circuit / no load) and was just barely able to light up an LED.

The most common analogy for electricity is water in pipes. If we think of water in pipes, voltage is the *water pressure* pushing on the water to get it to move through the pipes. Voltage is proportional, meaning that 3 Volts pushes twice as hard on electrons (to get them to move) as 1.5 Volts pushes.

Static shocks, like the kind you get on metal surfaces in the winter when wearing a wool sweater, can be up to 30,000 volts – but it's not dangerous to you because the amount of electrical charge is so small. The voltage has a very high *potential*, but cannot do much work.

*Experiment:* Push 9V battery terminals into your arm. Then lick the 9V battery terminals. Why did you feel the electricity on your tongue and not your arm?

## Current

An electric current is a flow of electric charge. Using our water analogy, current represents *how much water is moving* through the pipe (liters per second, or cubic meters per second, or whatever). The SI unit of electric current is the *ampere* (A), almost always shortened to "amp". This is equal to one coulomb of charge in one second. A coulomb is an exact count of electron charges (6,241,509,629,152,650,000 elementary charges – that's over six quintillion electrons!). Electrons are very small, and contain only a very, very tiny amount of charge. An amp of current can be a lot or a little, depending on how much voltage is

pushing the current. 5 amps at 2,000 volts would kill you. 1 amp at 5 volts can't even shock you. Smaller currents than one amp are usually measured in *milliamps*, or one-thousandths of an amp. So 500 milliamps is half an amp. 100 milliamps is a tenth of an amp. Milliamps are usually abbreviated "mA".

Electricity can be dangerous. Never work with electricity unless an adult has shown you how to be safe, and told you the work you are doing is safe.

A little saying to help remember how electricity can be dangerous is:

**"volts jolt; mils [milliamps] kill."**

That is, high voltages like static electricity shocks are not necessarily dangerous, but even low currents can be deadly. Either way, adult supervision is important!

## Resistance

Resistance is how much a material or component prevents the flow of electricity. Continuing the water analogy, resistance can be thought of as an obstruction or partial blockage in a pipe—regardless of how hard you push on the water, the resistance prevents as much water from flowing as there would be without the blockage. Substances with very, very high resistances are *insulators* – they prevent the flow entirely under most conditions. Substances that pass electricity at least some are called *conductors*. Copper, silver, and gold are excellent conductors. Aluminum, zinc, and iron are good conductors too, but not all metals are good conductors. Titanium is a poor conductor, but is not an insulator.

Many resistors are made with some amount of carbon inside them – carbon conducts electricity but not very well. The more carbon between the wires, the higher the resistance. Modern resistors don't necessarily use carbon, but it is still used. While a resistive material does pass electricity, it also slows down the flow, and prevents some electricity from passing. This electricity that gets "used up" by the resistor is turned into heat.

*Experiment:* A salt-water resistor. Attach wires to an LED (without the resistor this time, which is normally a bad idea). Hook the positive end to a 3 to 5 volt supply, leaving the negative terminal open. Drop one wire from the negative terminal of the LED into a glass jar half-full of (distilled, if you have it) water. Tape the wire to the lip of the jar, and make sure the wire is submerged. Put the wire from the negative terminal in the jar, under water, but keep the wires pretty far apart. Does the LED come on? Would you expect it to? Now add some salt to the water. What happened? (May need to add more salt).

## Ohm's Law

Electricity and electronics can be understood using many equations, but we'll only look at one of them. *Ohm's Law* says that the voltage, current, and resistance in a circuit are all related, in a way that can be calculated because the values are related in a knowable proportion. Ohm's Law states that the current through a conductor (wire) between two points is directly proportional to the voltage across the two points.

The law was named after a German scientist, Georg Ohm, who wrote an early book about electricity in 1827. He described measurements of voltage and current through simple electrical circuits containing various lengths of wire.

If we abbreviate "voltage" to V, "resistance" to R, and use I for "current" (the scientist who first understood current pretty well was French, and he referred to current as *intensité de courant*, meaning "current intensity", so we use "I" for current).

$$V = I \times R \tag{1}$$

That is, voltage is equal to however much current is flowing, times how much total resistance is in the circuit. Because the individual components are proportional, the law can be written other ways:

$$I = \frac{V}{R} \tag{2}$$

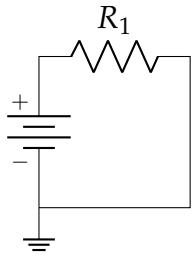
One very practical application of Ohm's law is that it allows an engineer to limit how much current can flow in a circuit by adding a certain level of resistance. Often it is safer and more practical to prevent a lot of current from flowing, or only allow a certain amount through. Even a little extra resistance can make the circuit (and the components in the circuit) safer and more durable. Resistors consume some energy, and turn electricity into heat. Electric stove burners are really just very big, heavy resistors; they generate heat by only allowing a certain amount of current through the big metal coil.

# 6 Basic Electronic Components

The following sections briefly introduce electronic components. Each of these pieces can be used to make complex, powerful circuits.*

## Resistors

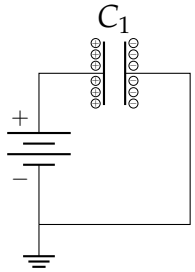
A resistor limits the electrical current that flows through a circuit. Resistance is the restriction of current. In a resistor, the energy of the electrons that pass through the resistor are changed to heat and/or light. For example, in a light bulb there is a resistor made of tungsten which converts the electrons into light and a great deal of heat.




A very simple circuit. Electricity flows through the resistor (maybe it's a light bulb). The resistor controls how much current flows in the circuit, and heats up.

## Capacitors

A capacitor (also called a *condenser*, which is the older term) is an electronic device that stores electric energy (a "charge"). It is similar to a battery, but is smaller, lightweight and charges up much quicker. Typically, capacitors hold much less energy than a battery, but can provide almost their entire charge to the circuit quickly. Capacitors are used in many electronic devices today, and can be made out of many different types of material. A capacitor-like effect can also result just from two conductors being close to each other, whether you want it to exist or not– if you've ever been shocked by a door knob or metal refrigerator in the winter, you have been one plate of a capacitor, because you've been storing a charge!



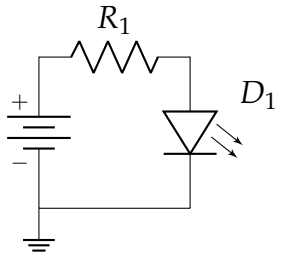

*Most of these definitions come from Wikipedia, though they have been edited and augmented by the author in several places.

Open capacitors in a steady state. No current flows, but there is a charge.

## Diodes

A diode is an electronic component with two electrodes (connectors). It allows electricity to go through it only in one direction.

Diodes can be used to convert alternating current to direct current (this circuit is called a diode bridge). They are often used in power supplies and can be used to decode AM ("amplitude modulation") radio signals (like in a crystal radio). Light-emitting diodes (LEDs) are a type of diode that produce light.



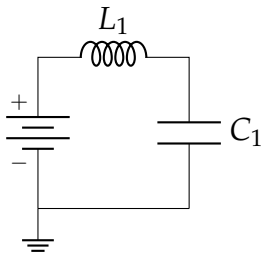

A very simple circuit. Electricity flows through the resistor, that controls how much current flows into the LED. If the battery was put in backwards, no electricity could flow in the opposite direction, so the LED would not light up.

## Inductors (Coils)

An inductor, also called a coil or (rarely) a reactor, is a two-terminal electrical component that stores electrical energy in a magnetic field, when electric current is flowing through it. An inductor typically consists of an electric conductor, such as a wire, that is wound into a *coil*. Sometimes inductors include a magnetic bar or ring, around which the wire is wound. Other times the wire is wound with only air in the middle (which still works, because the Earth has a magnetic field).

They are often used as electronic filters, to separate signals of different frequencies, and in combination with capacitors to make tuned circuits, used to tune radio and TV receivers.

The symbol for an inductor is a coil icon. Inductors are usually abbreviated with an L:



A simple circuit with one coil and one capacitor. This sort of circuit is one way to make an *oscillator*. This kind of circuit is found in radios.

## Transistors

A transistor is a *semiconductor* (see below) device used to amplify or switch electronic signals and electrical power. It is composed of semiconductor material usually with at least three terminals for connection to an external circuit. A voltage or current applied to one pair of the transistor's terminals controls the current through another pair of terminals. Because the controlled (output) power can be higher than the controlling (input) power, a transistor can amplify a signal. Today, some transistors are packaged individually, but many more are found embedded in integrated circuits.

The transistor is the fundamental building block of modern electronic devices. It might be the most important – or at least the most influential – engineering discovery, ever.

Why are transistors important? They enable a lot of powerful new devices, automation, information analysis, and even artificial intelligence. They have enabled huge leaps in our ability to do new things, since 1947.

- They can act like switches – with no moving parts, and that can be controlled with a tiny electrical signal
- They can take a small signal and turn it into a powerful signal; they can make sound louder, broadcast radio signals farther, or detect very tiny changes in other materials (thus, you can have electronic thermometers, or radio telescopes)
- When combined into logic gates (more on that below), they enable super-fast computations that have also changed how we do work, science, medicine, and almost every form of research

## A Little History

Many years ago, a "computer" meant "the person who did the math problems." A few people invented adding machines like an abacus, and *slide rules* (see Figure 4) made even difficult calculations very possible, but they still required a person to do the math. In the same way, people had learned about electricity, but couldn't do much with it except power light bulbs, street cars, and telegraphs. Talking to someone many miles away meant either using Morse code, or writing a letter.

The first discovery that led to the transistor was the *vacuum tube diode*, which uses a hot wire in a glass enclosure with no air in it to push electrons to another conductor, just like any other diode. Unfortunately, vacuum tube diodes are big, fragile, hard to make, and produce a huge amount of heat. The *vacuum tube triode*, which does everything a transistor does, came next. Since electrons are negatively charged, a slight negative charge

Figure 4: A pocket-sized slide rule from the late 1960s. This example is one of many that were given as promotional gifts to people who worked on NASA's lunar missions.

on the third wire (between, but not touching, the hot-wire emitter and the collector) permitted people to control the flow of electrons from one side to the other – it acted like a switch.

Vacuum tube triodes have all the same shortcomings as the vacuum diode, and have even more parts to fail. Despite their shortcomings, vacuum tubes changed the world in a *huge* way: they enabled radios, televisions, radar, and the first digital computers. But these computers were the size of a small house, weighed 30 tons, and required powerful cooling.

In the 1940s, three scientists discovered that some substances could either behave like an insulator *or* a conductor, depending on how much electricity or heat was applied to it. Being able to change a slab of glass or silicon from an insulator into a conductor meant they could do everything a vacuum tube could do, except much smaller, cheaper, and cooler. About nine years after they announced the discovery and proved that it works, the three men won the Nobel Prize. They had discovered a *semiconductor*.

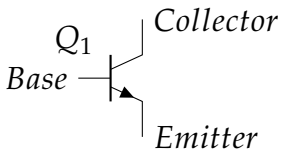
There's a longer documentary about the transistor on YouTube here.

The world changed amazingly quickly when we started using transistors. Today, there are many different kinds of transistors. The computer in your house has many *billions* of transistors inside it. So does your phone, your car, your microwave, your television...wow!

Some transistors are made especially for certain applications, like making audio louder, or pushing radio signals, or just for doing computations. These are all transistors, but they are different.

## Electronics and Transistors

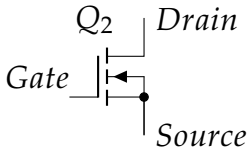
Transistors are usually abbreviated with $Q$. There are several kinds of transistors, but they work mostly the same.




Most transistors have three pins. For this kind of transistor (called an "NPN"):

- Electricity comes in the *collector*
- Electricity applied to the *base* opens up the flow of electricity to the *emitter*
- Electricity, when able, flows out the emitter and towards a ground.
- Therefore, the base is like a switch. It turns the transistor from an insulator into a conductive wire.

Most computers use a transistor type called a "FET" (*Field Effect Transistor*), that is very efficient and does not require very much electricity to switch from "on" to "off". They are faster and run cooler than the older types of transistors. Somewhat confusingly, these transistors have different names for their pins, compared to the older (but still preferable for some applications) "BJT" type transistors, seen above.




For questions of computers, we will treat transistors like a switch and nothing more. That's easy, right?

But how do transistors do all the cool stuff they do? Why are pocket calculators or iPhones possible? How do collections of electronic components enable us to do math, send text messages, or simulate a nuclear explosion without anything going kaboom?

At the absolute base of the entire electronics world, just above transistors, are *logic gates*. They are the simplest building blocks of digital computers.

A good additional introduction video is here (`ed.ted.com`).

# 7 Logic Gates

The basic building blocks of computers are called *gates*. Their only function is to report, with some yes-or-no electrical output, the result of their circuit following some input from somewhere else in the circuit. The way to understand how gates make decisions is called "Boolean Algebra", but really it's a fairly simple set of decisions that get strung together in useful ways.

## Boolean Logic

Boolean logic is the sort of decision making and answers that computers use. You can use it too, of course, and it will help you learn to program computers. The answer to every question in Boolean logic is either "yes" or "no" – "true" or "false". In computers, that's a one or a zero, of course. Every operation (decision) is composed of a set of operations: *and*, *or*, and *not*. A British man named George Boole introduced these ideas to the world in 1847[†]. He did not realize he was laying the foundations for computing, but 50 years later another very important mathematician, Claude Shannon, understood that Boole's system for understanding decisions could be used to design electronic circuits[‡].

It may be helpful to see some of the logic and operations (decisions) made in a graphical form, looking at the overlap between circles, or between a circle and its surrounding area.

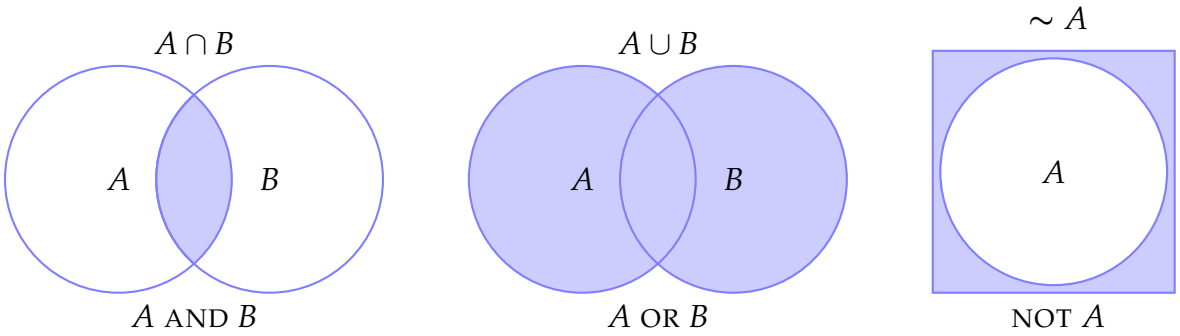



Figure 5: A graphic representing AND, OR, and NOT with overlapping circles.

Also note: more complex decisions can be made by combining these operations: NOT + AND, for instance).

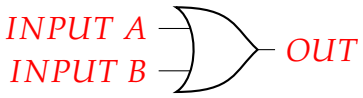
[†]*The Mathematical Analysis of Logic* (1847), and *An Investigation of the Laws of Thought* (1854). For more, see his Wikipedia entry.

[‡]Shannon, Claude E. 1938. "A Symbolic Analysis of Relay and Switching Circuits". *Trans. AIEE.* 57 (12): 713–723.

## The OR Gate

OR gates take two inputs and provide a "yes" if line 1 OR line 2 are equal to 1 – that is, if either line has a voltage level above zero.

The symbol for an OR gate is:




The "Truth Table" (how they behave) is:

**OR Gate Truth Table**

| A | B | OUT |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

OR gates are very simple. They are so simple that transistors are not even needed, though they certainly can be used.

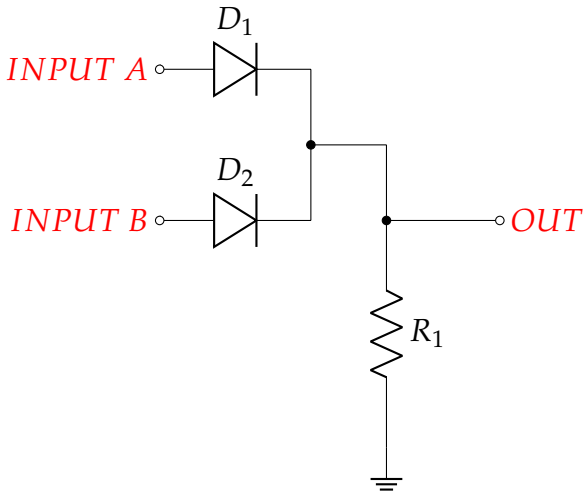



Figure 6: A very simple OR gate schematic. Diodes conduct electricity only in the "forward" direction. So Input A cannot affect Input B. But either way, if there is a signal on Input A or on Input B, OUT will carry a voltage (and therefore, a "1"). Diode *propagation delay* (how much time the signal takes to cross through the diode) is slower than transistors, meaning transistorized circuits run faster.

*Experiment:* Wire up an OR gate on a breadboard, like the one seen in the cover image. Use LEDs as indicators for Input A, Input B, and OUT. Feel free to use diodes, but using transistors is pretty easy. It's best to use N-type transistors as switches for each LED and for the OR-output LED.

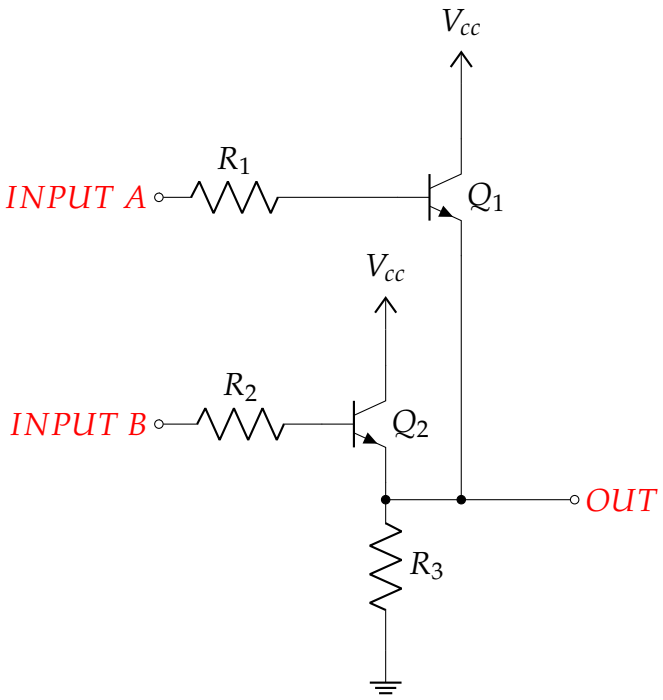



Figure 7: A simple transistor-based OR gate. Transistors are better than diodes in this application because transistors amplify (or at least do not degrade) signals, while diodes lose half a volt or more. In this schematic, each transistor acts as an independent switch for the output – either switch pushes OUT to high.

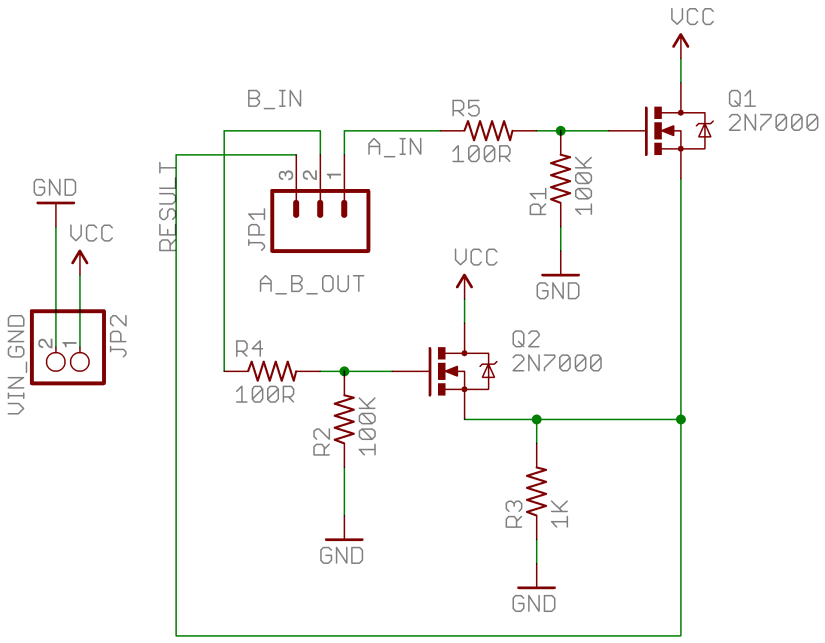



Figure 8: A schematic from a real circuit design program showing a practical, real implementation of the idealized schematic in Figure 7. The wires connecting components are the green lines. Mostly there are more resistors to help keep the transistors behaving well. For instance, the 100K resistors make sure the transistors are always at a low logic level (zero volts) when the inputs are 'off'.

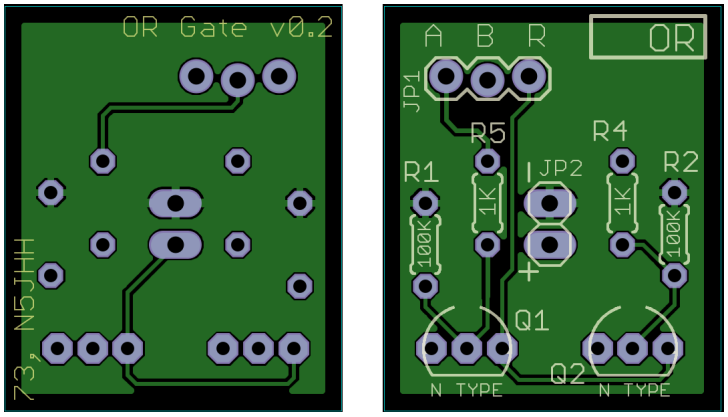
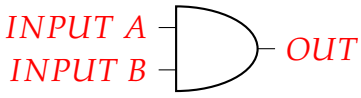



Figure 9: A circuit board designed using the above OR gate schematic. You can solder these without much trouble. You can see the top and bottom of the board here, but all components go on the top. However, both top and bottom have *traces*, that carry electricity instead of using loose/floppy wires. The larger resistors are the lowest value (they're just there to make the transistors behave a little better): 100R means "100 Ohms". The smaller resistors are either 100,000 Ohms (that is, 100K Ohms) or 1,000 Ohms (1K Ohms). The OR gate takes two *inputs*, A and B, and produces one *result*, R.

## The AND Gate

AND gates take two inputs and provide a "yes" (a one, or a "logic high" signal) only if both line 1 AND line 2 are equal to 1.

The symbol for an AND gate is:




The "Truth Table" (how they behave) is:

**AND Gate Truth Table**

| A | B | OUT |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

There are much more complicated AND gates, but see below for an easy-to-understand example (and it totally works). Remember that transistors act like switches: the *gate* acts like the switch that opens up the flow of electricity from the *source* to the *drain*. Look at the schematic in Figure 10 – note that the drain of the top transistor ($Q_1$) flows into the source of the other transistor ($Q_2$), meaning that even if $Q_2$ is turned on, it may not necessarily start carrying electricity (because $Q_1$ may not allow the electricity from $V_{cc}$ to pass through). Thus, both gate 1 AND gate 2 have to be turned on in order to see voltage at the output.

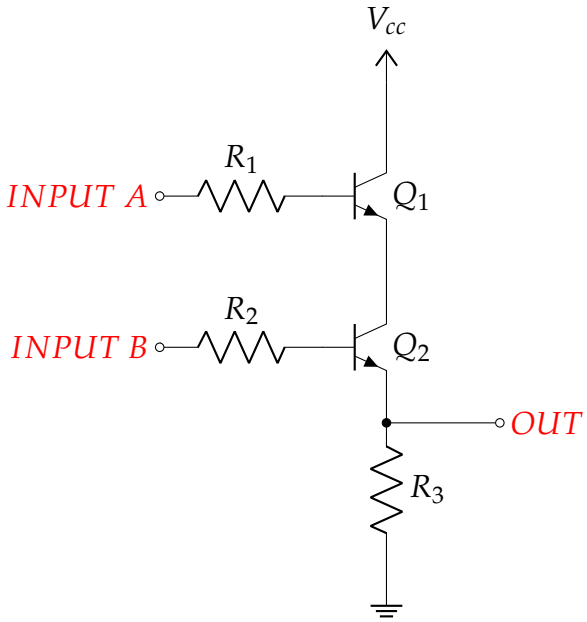
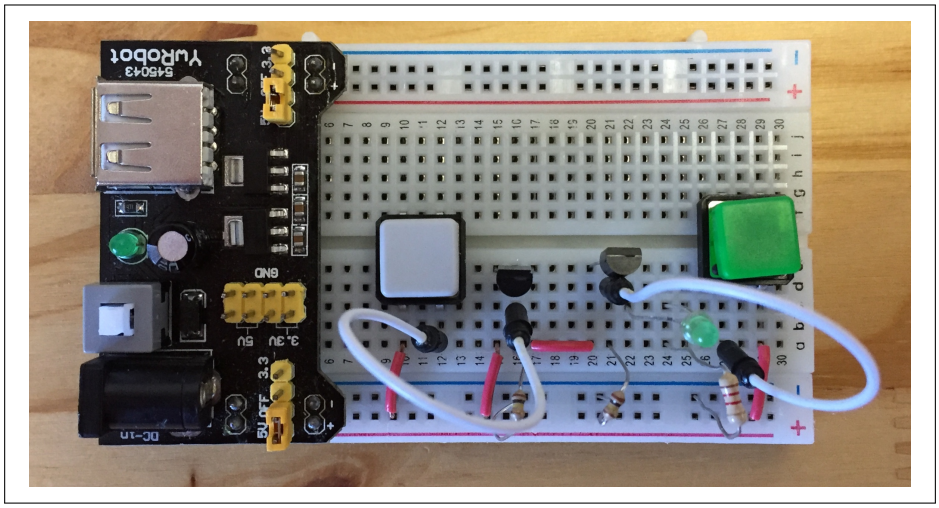



Figure 10: A simple AND gate schematic. When Input A is a zero, no current flows into the "collector" (voltage input wire) of Transistor 2. When Input B is zero, no current can flow to OUT, regardless of whether Input A is 0 or 1. Only when Input A is 1 (current can flow past Q1) and when Input B is 1 (current is available to Q2, and Q2 is turned "on", allowing current to pass) does OUT go "high". $R_3$ prevents too much current from flowing to ground, and is not strictly necessary in all gates, but is included here because it is common to see them. $R_1$ and $R_2$ should be about 100 ohms. $R_3$ should be about 100K ohms. This design is just for demonstration.



A single AND gate on a solderless breadboard. The transistor on the left, controlled by the gray button, makes electricity available to the input of the second transistor, controlled by the green button. Only if both transistors are "on" (that is, A *and* B are passing electricity) will the LED light up. Using this picture, wire up an AND gate on a breadboard with N-type transistors. Use LEDs as indicators for Input A, Input B, and the result.

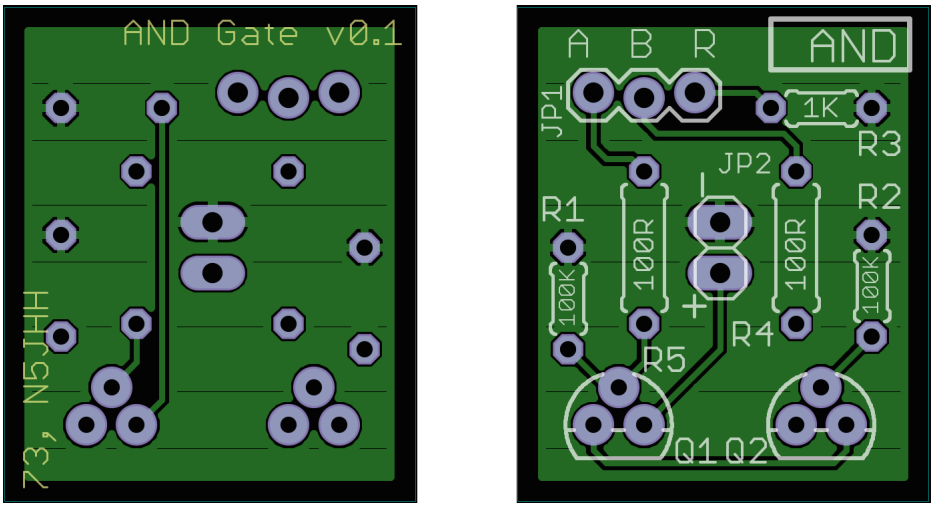
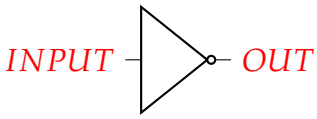



Figure 11: A circuit board designed using the above AND gate schematic, though with two 100K pull-down resistors also implemented (these guarantee the transistor's gate are reliably held at 0 volts instead of allowed to *float*, until either of the inputs goes high (that is, pushes some voltage into the transistor gate). The pull-down resistors are important practically, but they do not appear in this circuit diagram for simplicity. This board looks almost exactly like the OR gate, but the wiring is different! Also, you can see both the front and back sides of the circuit board in this image. The AND gate takes two *inputs*, A and B, and produces one *result*, R.

## The NOT Gate

NOT gates, sometimes called *inverters*, take one input and reverse the value of that input. So a 1 becomes a 0, or vice versa. Put another way, this gate creates the *complement* of the input value.

The symbol for a NOT gate is:




The "Truth Table" (how they behave) is:

**NOT Gate Truth Table**

| IN | OUT |
|---|---|
| 0 | 1 |
| 1 | 0 |

A simple NOT circuit, using a very old resistor-transistor logic (RTL) design. It is easy to understand, though this example is not how NOT gates are actually implemented any more, because of the poor efficiency—wasted energy turns into heat, which then has to be cooled somehow. It is better to not make the heat in the first place.

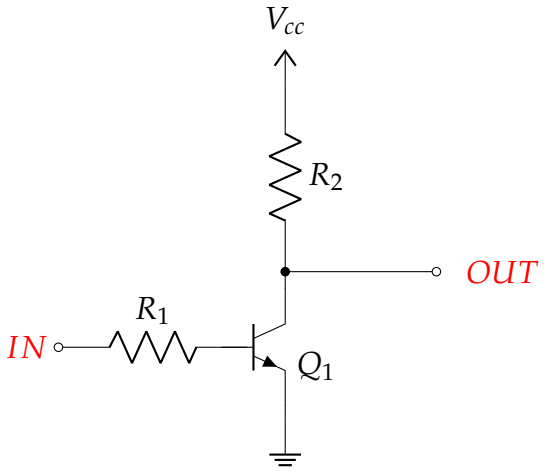



Figure 12: A very simple NOT gate schematic. When IN is logic low (off), electricity cannot pass through to the ground, and so it is available at the OUT terminal—OUT is a 1. When IN is carrying current, the transistor opens up and electricity starts flowing to ground – the "path of least resistance". As electricity flows to ground instead of OUT, OUT has a very low voltage available – making it a zero.

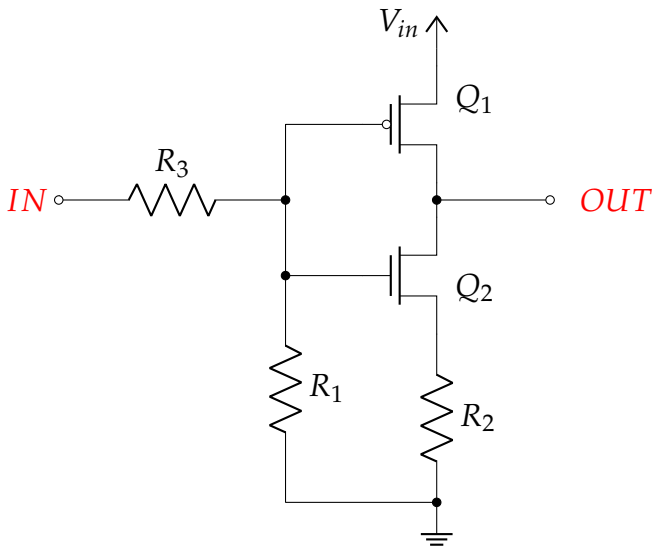



Figure 13: A practical inverter (NOT) circuit. While more complicated than a simple NOT circuit, it is more efficient, wasting much less energy. It uses one P-type and one N-type *field effect transistor* (FET). P-type transistors clamp off electricity when their gate *has* voltage on it. N-type transistors clamp off when their gate *does not have* voltage on it. So, here, when the input is high, the top transistor clamps off voltage input, and the bottom transistor opens up, allowing any current to flow to ground–but there's no current available! The situation reverses when the input is low: the top transistor opens up, allowing current to flow, and the bottom transistor clamps off, preventing current from going anywhere – but putting a high signal at the output.

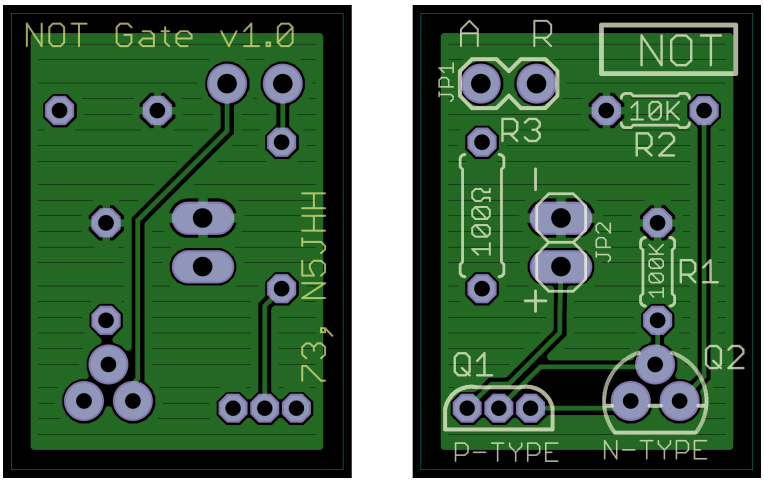



Figure 14: A circuit board designed using the more elegant NOT gate schematic seen in Figure 13. The NOT gate takes exactly one *input*, A, and produces a *result*, R.

| Part | Value | Device | Package | Description |
|---|---|---|---|---|
| JP1 | Input/Output | pin header | 1x2 header | Standard 2-pin 0.1″ header |
| JP2 | $V_{in}$ & $GND$ | pin header | 1x2 header | Standard 2-pin 0.1″ header |
| Q1 | ZVP3306A | transistor | TO-92-3 | P-type MOSFET transistor |
| Q2 | BS270 | transistor | TO-92-3 | N-type MOSFET transistor |
| R1 | 100K | small resistor | 0204/5 | Pull-*down* resistor |
| R2 | 10K (10,000Ω) | Small resistor | 0204/5 | Current-limiting resistor |
| R3 | 100R (100Ω) | $\frac{1}{4}$-watt resistor | 0207/7 | Current-limiting resistor |

Table 1: The list of components needed to make the NOT gate seen in Figures 13 and 14.

# 8 Complex Logic Gates

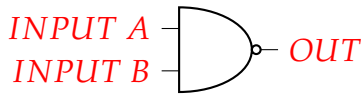
Combining gates is possible to produce more complex circuits that answer different questions, like outputs that tell us if two inputs are "NOT AND" or "NOT OR" or "Exclusively OR". Each of these gates allows new comparisons between two inputs, but each is composed of the fundamental three gates, seen above. There is even an "Exclusive NOT OR" gate, which is how computers determine if two numbers are equal to each other.

## The NAND Gate

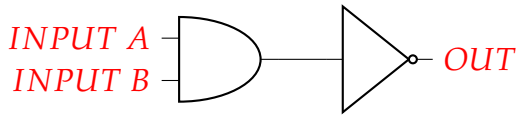
A "NOT-AND", or "NAND", gate just reverses the value obtained from an AND gate – so it has the exact opposite truth table. Since a NAND gate just reverses the output of AND, one way to create a NAND gate is to put a NOT gate on the result (output) line of an AND gate and make a NAND gate, just like you were using Legos. The AND gate goes through the NOT gate, and becomes NOT-AND, or "NAND".

---

The NAND gate symbol is:




You can also think of a NAND gate as two separate gates:




See the circle on the output? That means "*not*".

The "Truth Table" (how they behave) is:

**NAND Gate Truth Table**

| A | B | OUT |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

---

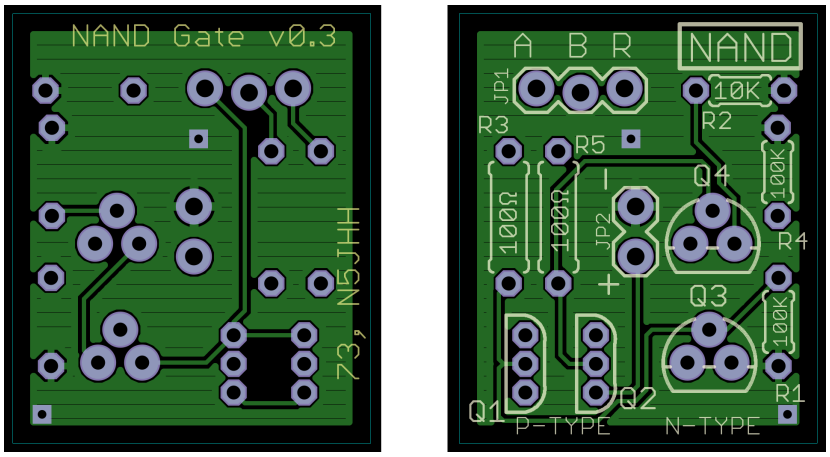


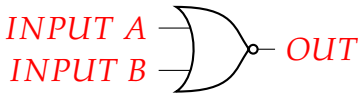

Figure 15: A circuit board designed to make a NAND gate. Note that it uses four transistors.

## The NOR Gate

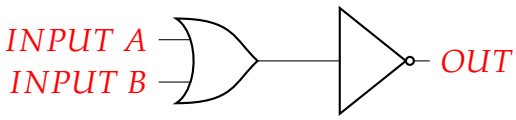
Not surprisingly, there is also a NOR gate, that has the reverse truth table to an OR gate. NOR gates are easy to implement, though as we have said elsewhere, more complicated NOR gates save power.

---

The symbol for a NOR gate is:




You can also think of a NOR gate as two separate gates:




See the circle on the output? That means "*not*".

The "Truth Table" (how they behave) is:

**NOR Gate Truth Table**

| A | B | OUT |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

---

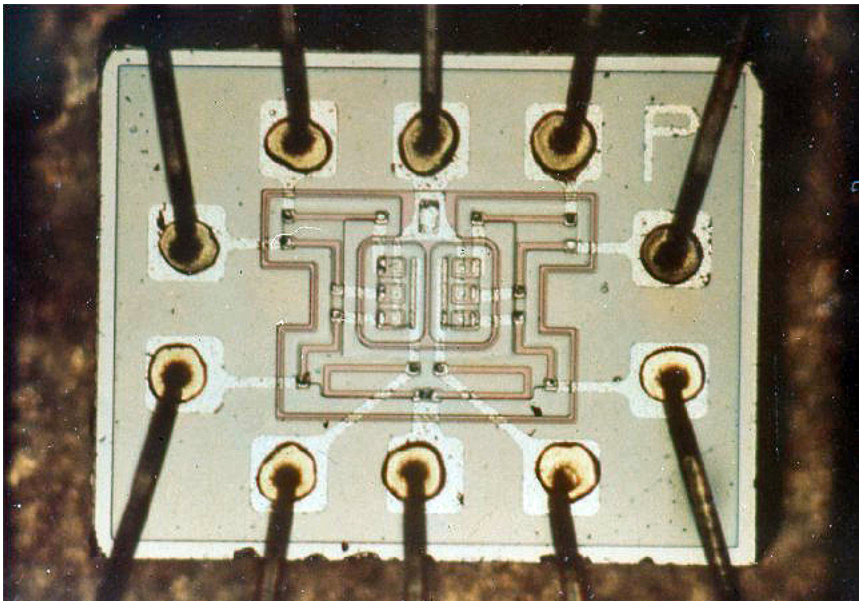
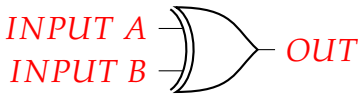


Figure 16: An early integrated circuit NOR gate. It has six transistors, eight resistors, and three pairs of A/B inputs (thus, it has three separate yes/no outputs). This chip was part of the guidance computer that took Americans to the moon in 1969. *Photo credit: Wikimedia Foundation*.

## The Exclusive-OR Gate

There is also the "eXclusive-OR", or "XOR", gate. The XOR gate output (result) is 1 when *the inputs are not the same*. It returns a 1 if, and *only* if, both inputs are different. That is, it provides a one "exclusively if" there is a single "one" (sometimes called "logic high", for "voltage above zero") in the two inputs. XOR gates take a lot of transistors to implement. The most common version requires 12, though it is possible to make an XOR gate with 8 transistors. See Figures 17, 18, and 19.

The symbol for an XOR gate is:




(the curved line behind the OR gate means that it is "exclusive")

The "Truth Table" (how they behave) is:

**XOR Gate Truth Table**

| A | B | OUT |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

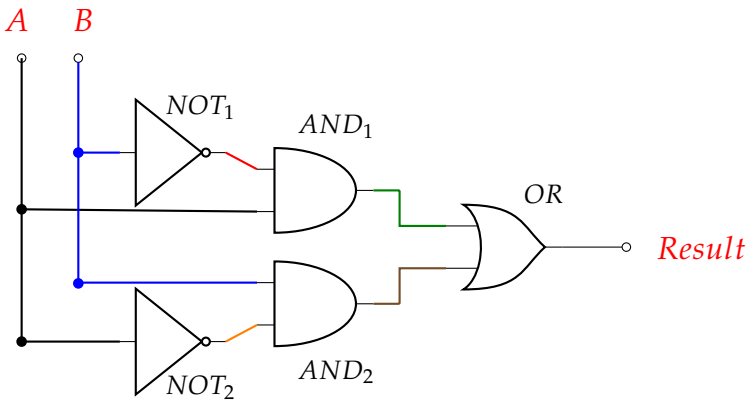



Figure 17: XOR gate, as a composite of the simple gates.

XOR gates return "true" if—and *only* if—the inputs are different. Put another way, the inputs "must sum to 1", or the inputs "must not be equal". OR gates return "true" if the inputs are different, but *also* if both inputs are 1. So, it isn't the same. There are two cases where XOR can be "true"–where A is 1 and B is 0, or where A is 0 and B is 1. The logic of the XOR circuit must handle either case, and list the result of 1 ("true") for only those two cases.

Take a look at Figure 17: You can see that there are two AND gates. Recall that AND gates only return "true" if both inputs are 1. For XOR, we require that both inputs are different. So to correctly identify "A is 1 and B is 0", we use an inverter (a NOT gate) on the B input

– so if the B is a zero, the NOT gate converts the B to a 1, and then the AND gate gets both inputs as 1. The lower AND gate ($AND_2$) captures the opposite case: if A is 0 and B is 1, then the A input gets inverted by the NOT gate into a 1, and for that case, also, the AND gate correctly reports that the inputs are "true". Since the OR gate will return "true" if either input is 1, then both opposite cases where XOR is true can be passed along to the result.

It's also worth knowing that XOR gates can be used like a NOT gate, by holding one of the two inputs at 1. Then the other input will always be negated. If one input of an XOR gate is always held at zero, then the XOR gate always passes the same value that comes in on the other input (a gate that passes on its input without changing it is called a "buffer").

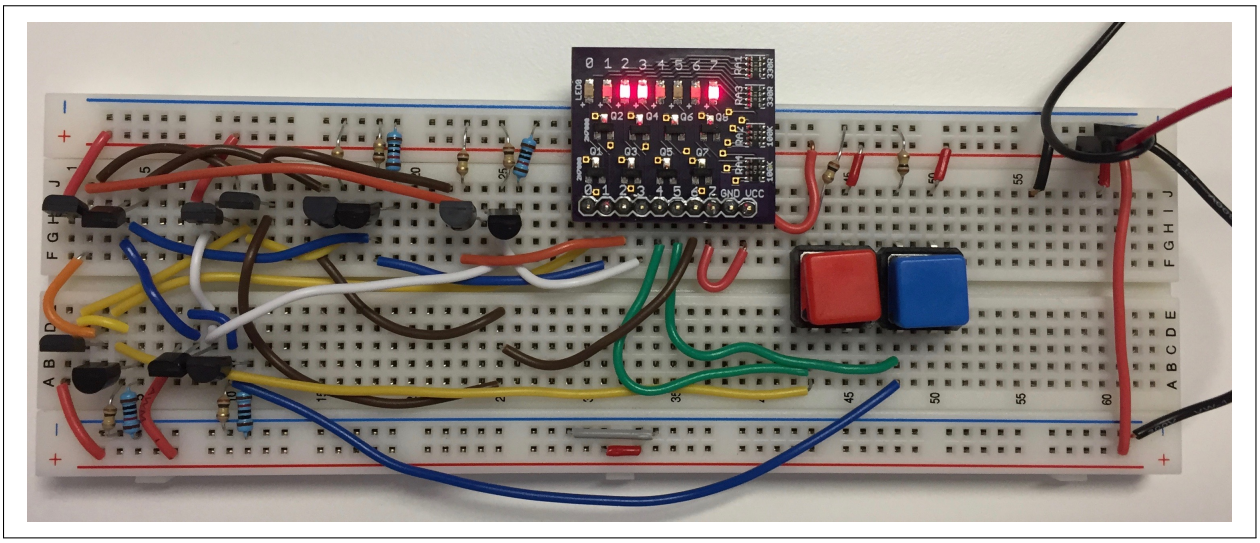


Figure 18: An XOR gate wired up on a breadboard with discrete transistors and resistors. This is the same schematic (and the same color-coded signal wires) seen in Figure 19. The red switch is signal A and the blue switch is signal B. On the LED line, 0 and 1 are A and B, and 2 and 3 (lit up) are NOT A and NOT B.

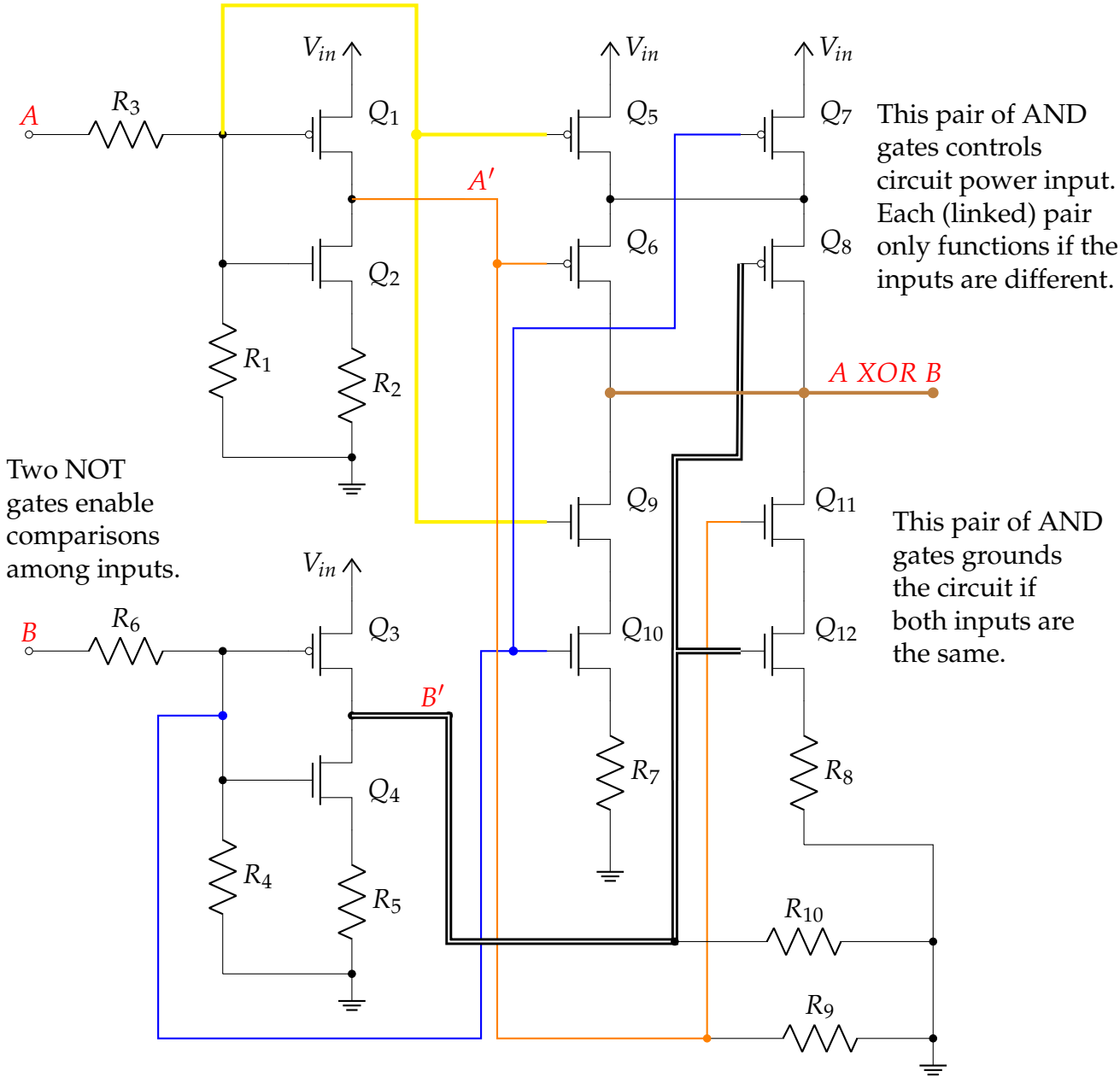



Figure 19: An exclusive-or (XOR) gate, composed of 12 transistors and some supporting resistors. You don't need to understand how it works, but it's not that hard if you break it down into the colored signals, two NOT gates, two AND gates with N-type FETs, and two linked AND gates with P-type FETs. A "0" signal opens a P-type FET, and a "1" signal opens an N-type FET.

# 9 Optional: Solder Some Gates

If you have the time and equipment, it can be fun and useful to solder some of the components together to make some of the basic logic gates. Several completed PCBs are already available, so this exercise is optional.

> *Instructor note:* Students should solder the NOT gate. This board has the fewest components and it will slow the students down quite a bit to even solder a simple gate. Keeping track of the components, orientation, and values of each, will not feel quite as fun as melting solder to bond a couple of wires or a couple of resistors. As a result, the more advanced gates and whole circuits will already be put on PCBs to make later experiments less tedious.

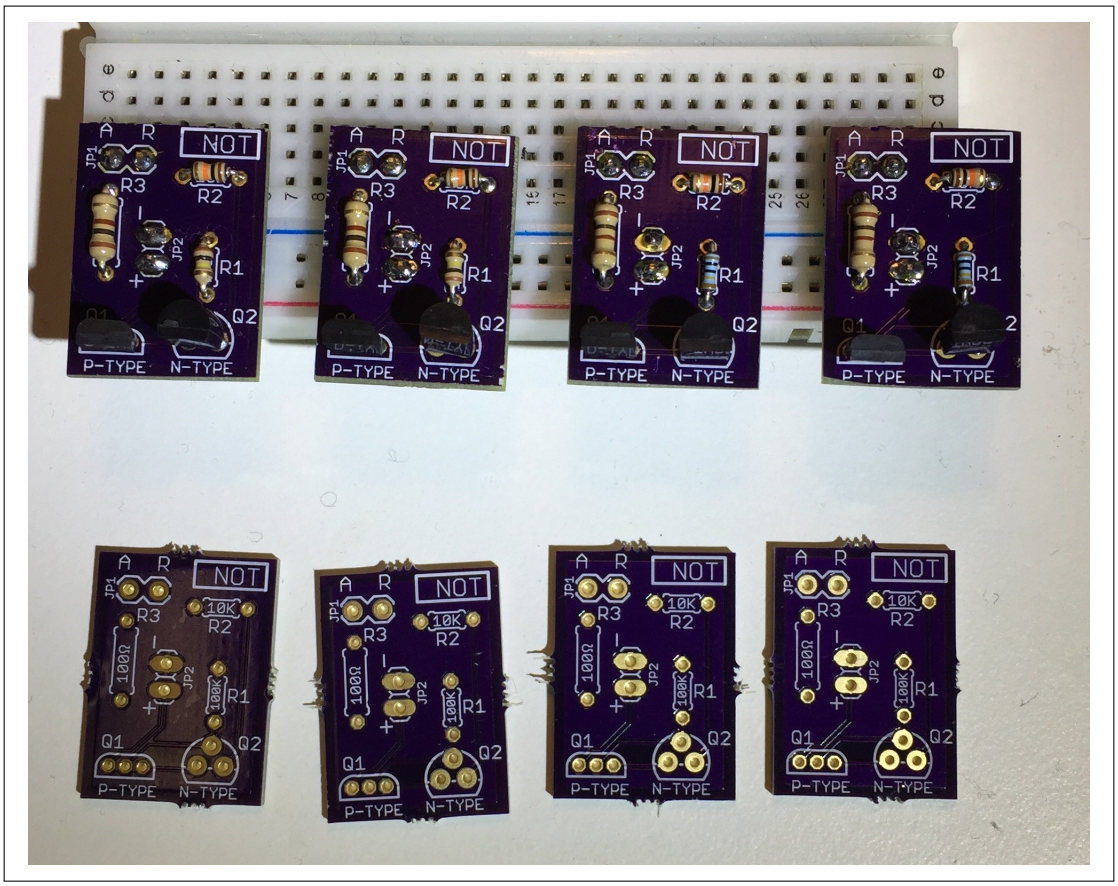



Figure 20: A photo of some assembled NOT gates, and some bare PCBs you might use when creating the gates. You solder a set of specific components in the correct places, and then you end up with the building blocks of computers.

## Solder A Simple Gate

- Get component list

- Identify each resistor by color bands (see Appendix, Figure 36)
- Read part numbers on the transistors to make sure (need magnifying glass)
- The transistors are not necessarily the same, and transistors *can be put in the wrong way*, so be certain about the orientation of the components before soldering them
- Similarly, the values of the resistors matter. Ensure that each resistor matches the value labeled on the circuit board. Some of the resistors have been chosen because they are of different sizes and colors, but the bands on the resistor are the best way to check the component
- Bending the resistor leads tightly is a challenge for small and inexperienced hands: the best way is to pinch the resistor body close to the lead, and place the index finger of the other hand at the joint where the lead meets the resistor body. Then, "roll" the finger, pressing tightly the whole time, such that the lead stays close to the body but ends up perpendicular to the body.
- If resistor leads aren't quite tightly bent, use pliers on the underside of the PCB to pull the resistor down into place.
- Resistors should be flat against the circuit board!
- Solder resistors first
- Then solder the transistors
- Lastly, solder the headers (note: it may be helpful to use headers with long pins on both sides, to enable both breadboard use and discrete-wiring use, but otherwise solder the headers with the long pins facing *down*)
- Soldering the headers may be easier if the headers are put in place on a solderless breadboard, with the gate PCB laid on top of them, and then soldered in place. *Caution:* if you take too long to solder the headers this way, the heat can begin to melt the plastic body of the solderless breadboard.
- Always test each gate to ensure that it works. Occasional errors (even in the PCB manufacturing process!) can be frustrating, and it's easier to catch these errors while the iron is literally hot

This project includes a separate PDF project that teaches basic through-hole soldering. *Because soldering irons get very hot, adult supervision and safety glasses are required.*

Soldering Tutorial

Jesse Hamner

**Abstract**

Learning to solder opens up a broad and deep section of STEM and STEAM education. Children, when equipped with tools that allow them greater access to electronic devices, tools, and programming, will get an early understanding of how electricity works, and how electronics are made. This tutorial, originally written for undergraduate-level novices, is a 2-hour workshop taking children from the most basic ideas of soldering, through a completed, if simple, soldering project.

**Keywords:** STEM; education; electronics; elementary education; soldering.

The author's daughter, repairing a DC motor for her elementary school.



Figure 21: The soldering tutorial PDF front page.

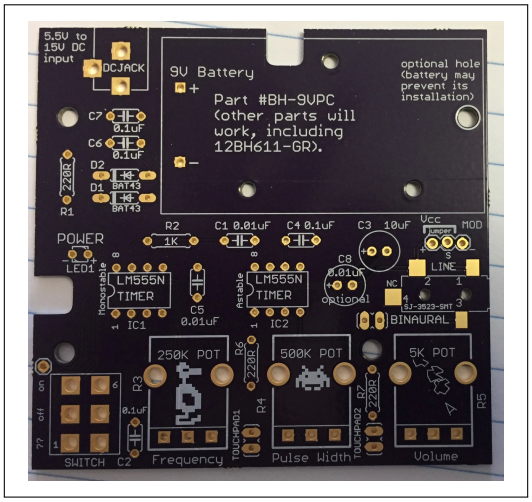
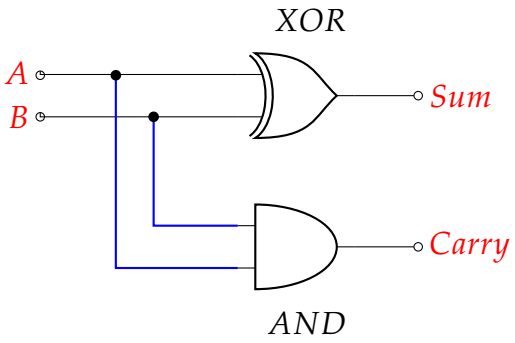



Figure 22: For even more soldering (but still made for beginners), check out the Atari Punk Console project.

# 10 Logic Circuits

## A Half Adder

A *half adder* demonstrates the way in which computer logic gates can correctly add (binary) numbers, though here it can add only two *binary digits* ("bits"). Thus, each half adder can only count to *two*, but that is enough! Long cascades of half adders and *full adders* enable computers to count large numbers. Below, see two examples of half adders. The second example replaces the XOR gate with a series of simpler gates, but the results are the same. Each half adder takes two bits and adds them, passing on the bits for further work or as an answer itself.




**Truth Table**

| A | B | Sum | Carry | Decimal |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 2 |

Figure 23: A half-adder circuit. It adds two bits together. If the result is larger than 1, the "carry" signal is high.

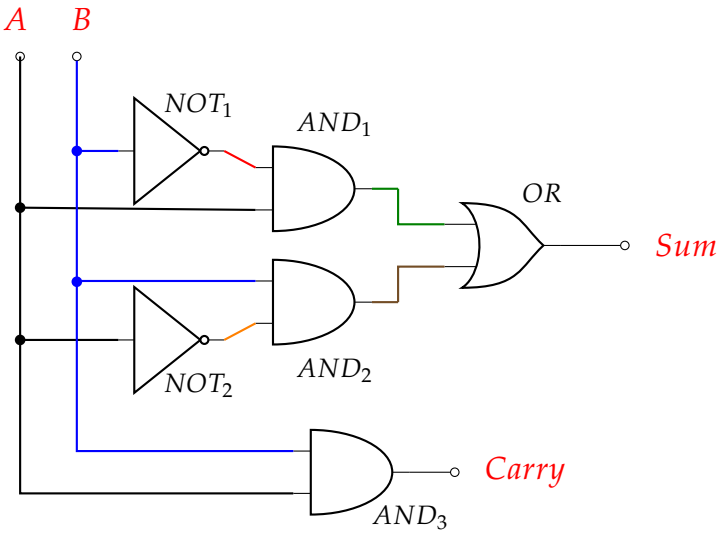



Figure 24: Another half-adder circuit. The exclusive-OR gate has been replaced by an equivalent circuit made up of five simpler gates. You can see why XOR requires so many transistors!

## Full Adder

This type of adder is a little more difficult to implement than a half-adder. The main difference between a half-adder and a full-adder is that the full-adder has *three* inputs and two outputs. The first two inputs are A and B, just like with every half-adder, and the third input is an input carry, designated as $C_{in}$. Enabling the circuit to handle a carry *in* from another adder makes it possible to add many binary orders of magnitude, since many carry-outs and carry-ins will be needed to add numbers larger than 1. Once we implement a full adder, we can string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next.

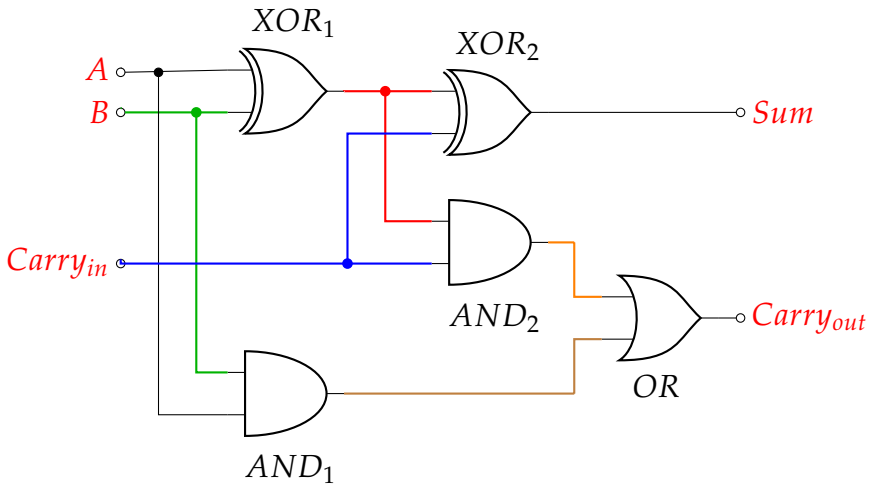



Figure 25: A full-adder circuit, made up of two half-adder circuits. A full-adder enables carry-in as well as carry-out. This means that the full adder can add *three* inputs together, and keep track of all of them. If the sum is two or three, the carry-out signal goes high. These units can be linked together to make binary addition work even for very large numbers.

Since full adders can accept three inputs, the truth table is pretty big.

**Full Adder Truth Table**

| A | B | Carry In | Sum | Carry Out | Decimal |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 2 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 2 |
| 0 | 1 | 1 | 0 | 1 | 2 |
| 1 | 1 | 1 | 1 | 1 | 3 |

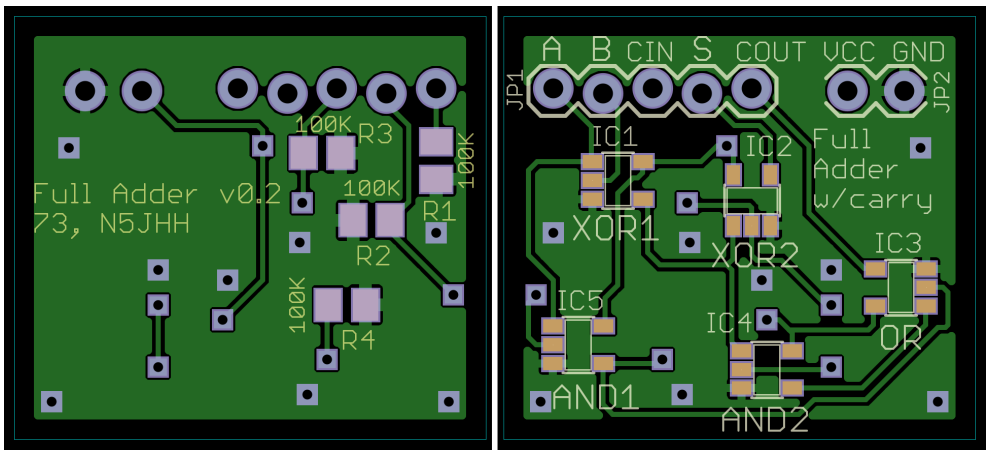
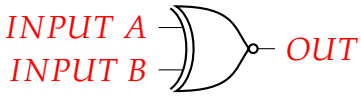



Figure 26: A single full-adder circuit for adding two bits plus a carry-in digit, implemented on a small circuit board using individual integrated circuit gates instead of discrete transistors. You can see the gates easily, but the board is much simpler to work with, compared to forming up a full adder from five of the single-gate boards.

## Equality

Comparing two one-bit numbers to see if they are identical is done with XNOR gates. These gates are the same as an XOR gate passing output into a NOT gate.

The symbol for an XNOR gate is:




The "Truth Table" (how they behave) is:

**XNOR Gate Truth Table**

| A | B | OUT |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

To compare large numbers, the circuit requires an XNOR gate for each binary digit (bit) in the numbers being compared. If any XNOR gate returns zero, the numbers are not equivalent. To build all of the XNOR gates into a single yes or no answer, each XNOR output could go into a series of AND gates – that is, since AND returns a "yes" only if both inputs are one, a single "no" prevents the AND gate from returning "yes".

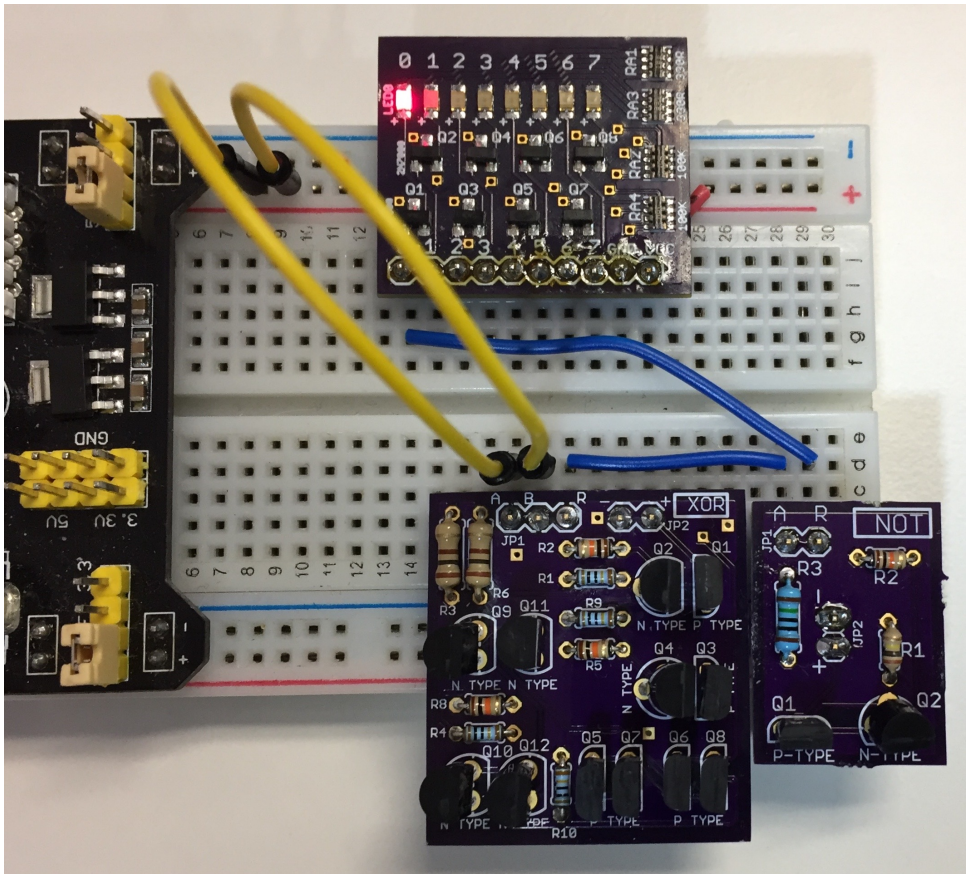



Figure 27: A discrete-transistor XNOR gate made up of one XOR and one NOT gate. Tying the output of the XOR gate to the input of the NOT gate makes the result of the NOT gate an XNOR of the original two XOR gate inputs—answering the question "are the two inputs equal?"

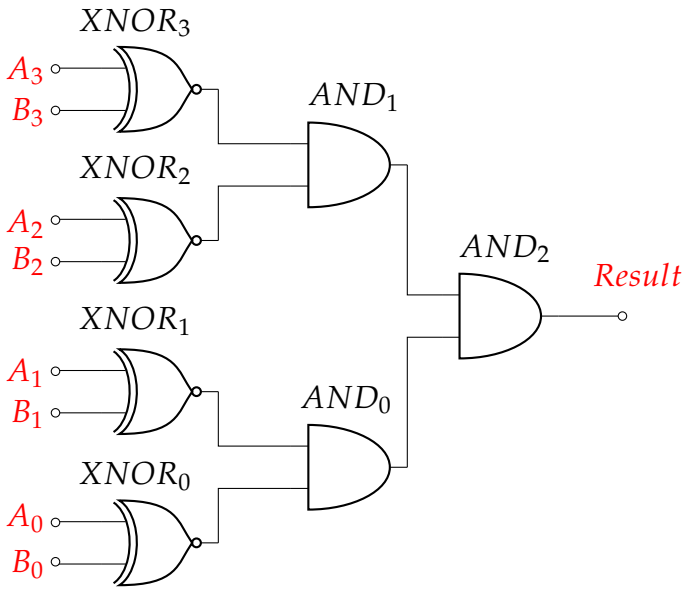



Figure 28: A circuit to provide a yes/no answer to the question, "are two four-bit numbers, A and B, equal?" Each XNOR gate compares the bits in each position of each number, and passes the answer to an AND gate input. The AND results ripple forward to a final answer.

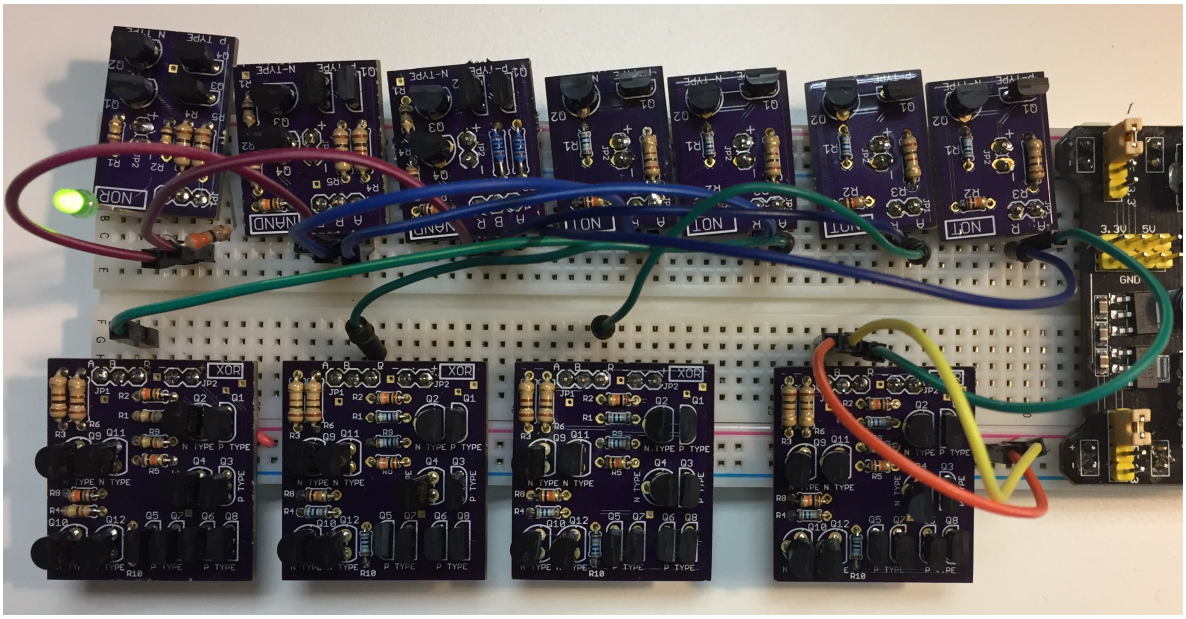



Figure 29: Testing for equality with four XNOR gates. Here, XNOR gates are created with XOR gates that pass the output (green wires) to a NOT gate. An XOR gate only goes logic high if the inputs are different. Thus, NOT XOR means both inputs are the same! Each XNOR output (blue wires) passes to a NAND gate. Recall that NAND gates only return zero if both inputs are 1. The output of each NAND gate (purple wires) passes to one input of a NOR gate. So if both inputs to a given NAND gate are "1", the NAND gate will show "0" as its result. NOR gates only return 1 if both inputs are zero. The result: the NOR gate lights up an LED if all inputs are equal. It is no different than the AND-gate cascade shown above in its result, it just uses negative logic to get there. Here, both inputs are equal to 1, so the LED is lit.

# 11 Building an Adding Machine

All of these little units, experiments, and instruction have been moving towards a goal – to make a primitive electronic calculator. Here, we will be able to add two four-bit numbers using only full adders, toggle switches, and LED blocks, all wired together on a breadboard. At the end of this unit, you will be able to enter two numbers in binary using toggle switches, and read the result on an LED readout, also in binary. There's no reason you have to stop at 4 bits – the 8-LED block means you could add two 7-bit numbers if you have seven full adders and more toggle switches, without *overflowing* the capabilities of the setup. An "overflow" is a type of error, and means the answer is not correctly displayed on the LED output bar. Either you have tried to put numbers into the system that cannot be represented by the computer because they are too large, or else the result of the two numbers is too large to display.

Anyway, let's build a calculator!

## Wiring up the adders

Put two sets of toggle switches on a breadboard and enough LEDs to display the sum (that is, at least 4 LEDs). Wire up power and ground to the switches and LEDs. If you're using your own LEDs, be sure and run a 220Ω or 330Ω resistor from the negative terminal of each LED to ground. Wire the switch outputs to the LED inputs, moving from right to left, because the rightmost digit is the smallest number value in our example, either 0 or 1.

It may be helpful to use one color LED for each bank of switches, and a different color for the "result" display. The 4-toggle LED boards included in this workshop have built-in LEDs to make it easy.

Run an additional wire from each switch output, in order, to the input of a full adder. Wire up each "carry-out" from one full adder to the next (leftward!) "carry-in" of the next full adder. The left-most carry-out digit should run to another LED, perhaps an LED of a different color, to indicate an overflow.

Next, run each adder "sum" wire to each output LED's positive terminal. In doing this you will have wired up each column of output, meaning you can read off the sum of the two numbers in binary.

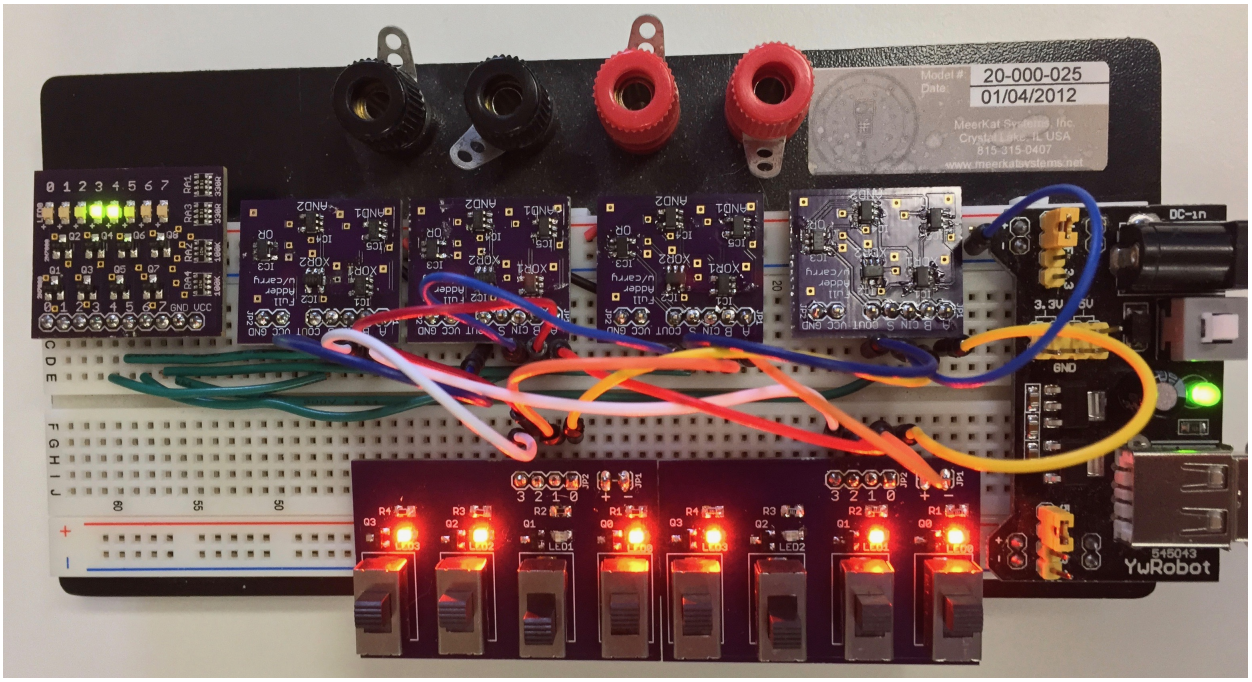



Figure 30: A properly wired 4-bit adder, with a carryout. This setup can count to 30. Here, it is adding $(8+4+0+1)$ and $(8+0+2+1)$, and the result is 24. The input values are color coded in pairs, so the 1s are yellow, the 2s are orange, the 4s are red, and the 8s are white. The carry-out and carry-in jumpers are blue, and the sum lines (that go to the LEDs) are green.

# A Appendices

## Higher Order Exponents

You can multiply a number times itself as many times as you want. Understanding a little more about exponents (the number of times you multiply a number times itself) will make understanding the language we use to discuss computers quite a bit easier. Just as we mention kilobytes and megabytes as units of storage, there are also less common units of storage that invent new multiplier terms for bits and bytes, because of the "powers of 2." So let's learn about base-2 exponents– the "powers of two".

| Powers of Two | | | Notes |
|---|---|---|---|
| $2^0$ | = | 1 | This one is strange, sort of. But it works! Any number "to the zeroth power" is equal to 1. See below for more. |
| $2^1$ | = | 2 | |
| $2^2$ | = | 4 | |
| $2^3$ | = | 8 | |
| $2^4$ | = | 16 | Since $2^2 = 4$, $2^2 \times 2^2 = 16$ |
| $2^5$ | = | 32 | This is why you can count to 31 on one hand. |
| $2^6$ | = | 64 | |
| $2^7$ | = | 128 | |
| $2^8$ | = | 256 | 8-bit computers were the first machines really adopted by consumers. Also, 8 bits makes up one *byte* of computer memory, so each byte can take on up to 256 values. |
| $2^9$ | = | 512 | |
| $2^{10}$ | = | 1,024 | 1,000 usually gets the prefix *kilo-*, like a kilogram is 1000 grams. A *kilobyte* is 1024 bytes. |
| $2^{16}$ | = | 65,536 | |
| $2^{20}$ | = | 1,048,576 | $2^{10}$ bytes is a kilobyte; $(2^{10} \times 2^{10})$ bytes is a *megabyte*. |
| $2^{24}$ | = | 16,777,216 | Most computer displays can show up to 16 million colors, using red, green, and blue, all in combination. Each piece of the color can have 256 ($2^8$) levels, from zero (black) to 255 (100% red, or green, or blue) |
| $2^{30}$ | = | 1,073,741,824 | $2^{30}$ bytes is a *gibibyte*, or a bit more than a *gigabyte*, which is 1000 megabytes. |
| $2^{32}$ | = | 4,294,967,296 | |
| $2^{40}$ | = | 1,099,511,627,776 | $2^{40}$ bytes is a *tebibyte*, or a few percent more than 1000 gigabytes – a *terabyte.* |
| $2^{50}$ | = | 1,125,899,906,842,624 | $2^{50}$ bytes is *pebibyte*. A petabyte is so large that one pebibyte is enough to store the DNA of the entire population of the USA…and then clone them, *twice.* |

*Explanation:* The reason that any number to the zeroth power is equal to one comes from the way we subtract exponents when dividing. You know that 8 divided by 4 equals 2; written another way, $2^3 \div 2^2 = 2^1$. Notice that the exponents change by subtraction, but the equation is the same! You can *divide* base-exponent numbers by *subtracting* the exponents (*extra-special historical trivia: this is how slide rules work*; see Figure 4 for a picture). And since any number divided by itself equals one, as in $2^3 \div 2^3 = 1$, subtracting the exponents gives $2^0$.

## Numbers Below Zero?

Representing *positive* numbers is easy for a computer: it counts upwards from zero. Representing *negative* numbers is harder, because *taking away values by adding them* is a little awkward. To represent a negative, designers use the leading (left-most) bit to declare "positive" (zero) or "negative" (one). Note that the *range* of numbers that can be represented does not change – for a 4-digit binary number, whether counting from zero to 15, or from -8 to 7, the total space used on the number line is still 16 numbers, in order.

> *Problem:* If the computer doesn't know it is supposed to use that first number to determine whether a number is negative or not, how does it evaluate the number?

## Complement of a Number

In mathematics, the *complement* of a binary number is "the value obtained by inverting all the bits in the binary representation of the number (swapping 0s for 1s and vice versa)." This number is called the *ones' complement* of the number.[§]

Here's an example:

| Number | + | − |
|---|---|---|
| 0 | 0000 | 1111 |
| 1 | 0001 | 1110 |
| 2 | 0010 | 1101 |
| 3 | 0011 | 1100 |
| 4 | 0100 | 1011 |
| 5 | 0101 | 1010 |
| 6 | 0110 | 1001 |
| 7 | 0111 | 1000 |

## Subtraction

Subtraction is similar to addition – very similar, since the process is "adding a negative number". To this point, we haven't seen negative numbers, because numbers below zero are harder to create or see, compared to numbers between zero and 15, or some other positive number. To get a negative number, computer scientists decided to create a pattern for describing negative numbers, that computers can correctly interpret. They decided to use the first bit (1 or 0, just a "yes" or "no" indicator) of the number as the *sign bit*: that is, if the first bit, in a "signed integer", is 1, then the number is a negative number. So if we have a four bit number, and the first bit is only for the "is this number negative" indicator, the number represents numbers in the range $[-7, 8]$ – still 16 values, but counting from

[§]Wikipedia page on Ones' Complement.

−7 instead of zero. Also note that if you don't tell the computer the number is signed, it will happily assume the number is *unsigned* and give you the wrong answer[¶].

To subtract, we first create the *two's complement* of the number we are subtracting (the "subtrahend"). We leave the number from which we are subtracting (the "minuend") alone.

Two's complement works like this: you take the *complement* of the number, and add one. *Complement* means you swap all the zeroes for ones, and all the ones for zeroes. Put another way, you run each bit through a NOT gate, then add a one to the result, using a set of full adders.

Let's take the two's complement of 7.

```
       0111      7

       1000           ones' complement of 7
       0001           add one to get two's complement of 7
===========   ====
       1001     -8 + 1 = -7
```

So here's a simple example of subtraction:

```
       0110      6
-      0011      3
===========   ====

       1100           ones' complement of 3
       0001           add one to get two's complement
===========
       1101           (-8 + 1 + 0 + 4) = -3 ... good. Now we can add:


       0110      6
+      1101     -3
===========   ====
       0011      3      (see how the sign bit was converted to positive?)
                        (the overflowed bit doesn't matter in this case)
```

[¶]Well, really, it will give you the *right* answer to a question that is different from the question you thought you were asking!

And here's a slightly more interesting one, with 8 bit numbers:

```
  0000 0110      6
- 0001 0011     19
===========   ====

  0001 0011     19

  1110 1100             ones' complement
  0000 0001             add one
===========   ====
  1110 1101     (-128 + 64 + 32 + 0 + 8 + 4 + 0 + 1) = -19


  0000 0110      6
+ 1110 1101    -19
===========   ====
  1111 0011    -13   (-128 + 64 + 32 + 16 + 0 + 0 + 2 + 1) = BOOM!
```

Designing logic circuits for subtraction using the twos' complement is surprisingly elegant. Let's review what happens to execute a twos' complement subtraction operation:

- Get the ones' complement of the subtrahend (invert each bit with a NOT function)
- Add 1
- Add the resulting value to the minuend

Building a flexible subtraction logic circuit depends on two features of our existing gates: first, that full adders can accept an incoming "carry" bit; and second, that XOR gates can be used as NOT gates by tying one of the inputs to positive voltage. Therefore, the logic circuit works like this:

- Pass each bit of the subtrahend through XOR gates with one of the inputs pulled up to positive voltage level; this takes the ones' complement of the number
- The result passes into one side of the full adder line we use for adding numbers
- Set the carry bit on the right-most full adder (i.e. the zero or 1 register) to high – *this action adds 1 to the incoming number*

The result of the line of adders is the answer to the subtraction question.

# A Bit More About Capacitors

The Leyden jar was one of the first capacitors invented: metal foil was placed inside a glass jar, and wrapped around the outside of a glass jar, but neither foil gets near the top of the jar. The glass barrier between the foil sheets (inside and outside) allows a charge to build up between the foil without allowing the electrical charge to move through the glass. Leyden jars didn't hold much charge, but the concept it demonstrated has not changed.

Capacitors are usually made with two metal plates that are on top of each other (or wrapped around each other), but that do not actually touch. When powered, they allow energy to be stored inside an electrical field. Because the plates need a lot of area to store even a small amount of charge, the plates are usually rolled up into some other shape, such as a cylinder. Sometimes, other shapes of capacitors are used for special purposes.

*Experiment:* make a variable capacitor from aluminum foil, paper, and a paper towel tube. The paper should go around the roll only once. Cut the foil one-quarter or one-half inch smaller than the paper on all sides. Tape a small wire to one corner of the foil, if possible using metallic/conductive tape[a]. Tape the foil to the paper. Tape one edge of the paper, foil side down, against the cardboard tube. Make another paper/foil combination. Wrap the paper/foil around the cardboard tube, but tape the paper only to itself, and just loose enough to slide a little.
Another example, using a coke can, can be found here.

It is also possible to make a regular capacitor with the paper-foil layers, separated by flat sheets of cardboard. Keep track of the "up" and "down" sides! Note that it is pretty easy to gang each "plate" together and make capacitor with a larger value; tying all the anodes together, and all the cathodes together, makes a larger capacitor.

[a] While it is possible to solder to aluminum foil, really there are much better and safer ways to attach a wire to the foil for the purposes of this workshop.

# Why NAND Gates and NOR Gates are "Universal"

It turns out that it is possible to make *any* gate from NAND gates. The same is true for NOR gates. A very simple example can be seen, where we can create an OR gate from two NOR gates.

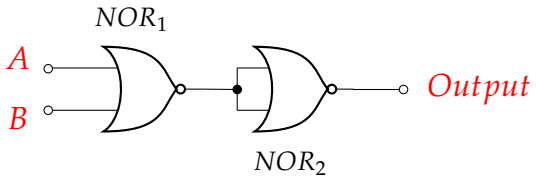



Figure 31: An OR gate from from two NOR gates. $NOR_2$ has its inputs tied together, making it a NOT gate. The gate then becomes "NOT-NOT-OR", and that is equivalent to OR.

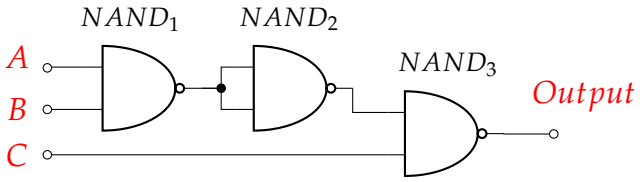



Figure 32: A three-input NAND gate created only from two-input NAND gates. The simplest version of a three-input NAND uses a single AND gate and a single NAND gate, but this example shows the potential to use NAND gates as universal gates, just like the NOR gates, above. $NAND_2$ is configured as a NOT gate, making $NAND_1$ and $NAND_2$ equal to an AND gate.

# Memory

Logic circuits need to store the numbers they are working with in order to do more than one thing. Central Processing Units (CPUs), Arithmetic Logic Units (ALUs) or Floating-Point Units (FPUs) each work with numbers, and need to fetch them from somewhere and put them somewhere when they are done. The "somewhere" is *memory*. Keeping with the use of binary math and binary logic, a high voltage would be a 1 and a low voltage would be a 0. Through the use of these little stored charges, computers can keep track of many, many pieces of information. No matter what the circuits are doing, everything being stored is composed of some count of zeros and ones.

There have been *many* forms of memory through the years, including punched paper cards, and even a wiggly wire! The most common memory that a CPU uses these days is usually called "dynamic RAM" (DRAM), and each bit consists of a circuit that is primarily a single capacitor and a single transistor, though it requires supporting circuitry and cannot be read without erasing the memory cell. Computers read eight, sixteen, 32, or 64 bits at a time from a row of memory and pass the information (numbers or instruction codes) from the memory on to be processed by the CPU.

One type of computer memory, called *static RAM*, uses at least six transistors per bit. Since DRAM is simpler (it uses only one transistor and one capacitor per bit), it is also cheaper. Therefore, to store one bit with 6 or more transistors is more expensive. So SRAM is mostly used for very important memory, like the memory very close to the core of the computer processor. Have a look at Figure 33[11], which is pretty complicated, but if you understand how the two types of transistors are turned on and off, it will make sense. To keep this diagram simple, no resistors are shown. If you look closely at the $Q_1/Q_2$ and $Q_3/Q_4$ transistors, you can see that they are acting like inverters (that is, each pair makes a NOT gate). When WL (the "word line") goes high, $Q_5$ and $Q_6$ open up, allowing access to the single bit stored in $BL$ and the inverse of that bit in $\overline{BL}$. Whether the word line is active (that is, whether or not you can write to the memory bit), it is possible to read the value of the bit–making SRAM very fast, because the system does not wait on the word line to activate.

SRAM uses 6 transistors per bit, making it very expensive compared to dynamic RAM, but each bit of SRAM can stand alone, with no other supporting circuitry except for the word line transistors. So, we use SRAM here because you can see the circuit, and see how it works. As a bonus, you can construct a bit of SRAM with opposing NOT gates, making it fairly easy to see what is happening.

We are going to implement some RAM using a *flip-flop* circuit, made of NAND gates. It uses even more transistors than the SRAM circuit, but it is simpler to create and uses logic gates with which you are already familiar. This "JK Flip-Flop" circuit uses two input NAND gates and also *three*-input NAND gates. 3-input NAND operates mostly the same way as a regular NAND gate, except that the inputs also include the value of the Q and not-Q output (value) lines. These lines are required before the output state (value) of the gate will change. And so is a "clock" signal line. By using a signal that turns on or turns

[11] Diagram and supporting information adapted from Wikipedia and Robert Entner's dissertation.

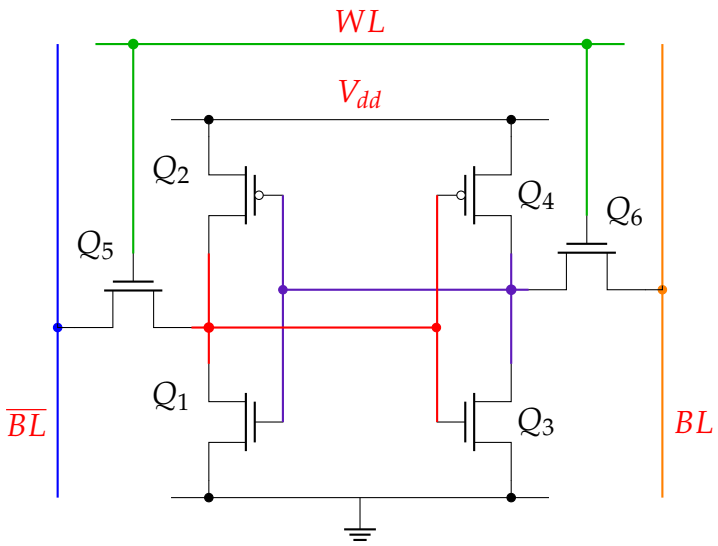



Figure 33: A static ram cell. The bit line ($BL$) on the right is the value of the bit, and the bit line on the left ($\overline{BL}$) is the complement (NOT) the value of that bit.

off everywhere at once, the computer memory can be more reliable. The clock line ticks on and off like the second hand of a clock, allowing the computer to read or write to the memory without wondering if the values are changing. So, the set- and reset- functions for the memory cell only activate when the clock signal is also high (has positive voltage on the line). This extra feature allows for certainty of reads and writes: it helps avoid rapid oscillation / instability or lock-ups should a circuit (or a third grader!) try to both set and reset the flip-flop at the same time. Furthermore, when powering up a simple SR flip-flop, there is no inherent guarantee of what the value would be (of course, it's possible to put a pull-up or pull-down resistor on the signal lines to create a natural "base level" of a set of inputs, though care must be taken to not accidentally create incompatible base conditions (such as accidentally setting both set and reset to high).

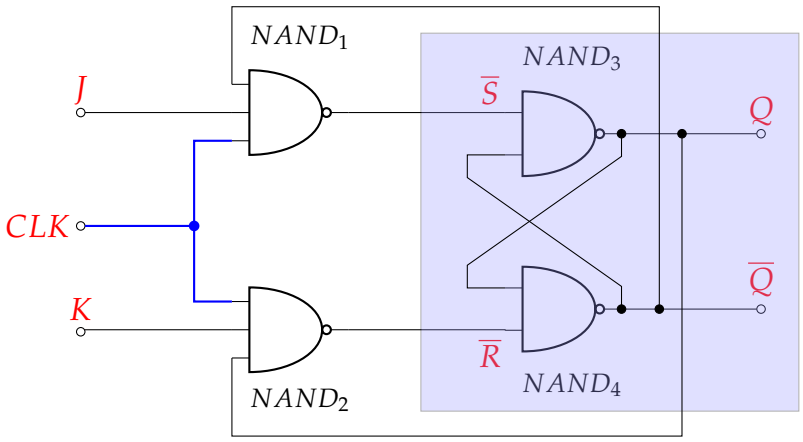



Figure 34: A gate-level schematic of one bit of memory, using a JK flip-flop circuit. Note that the two left-most NAND gates have *three* inputs.

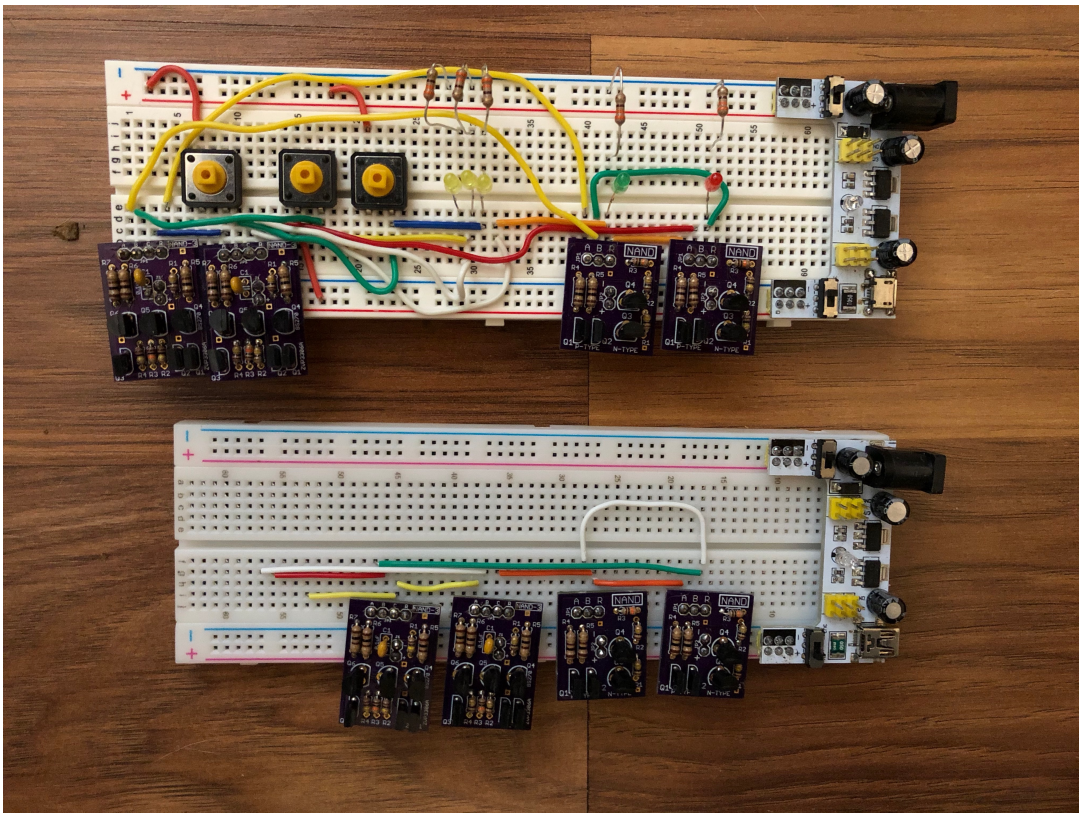
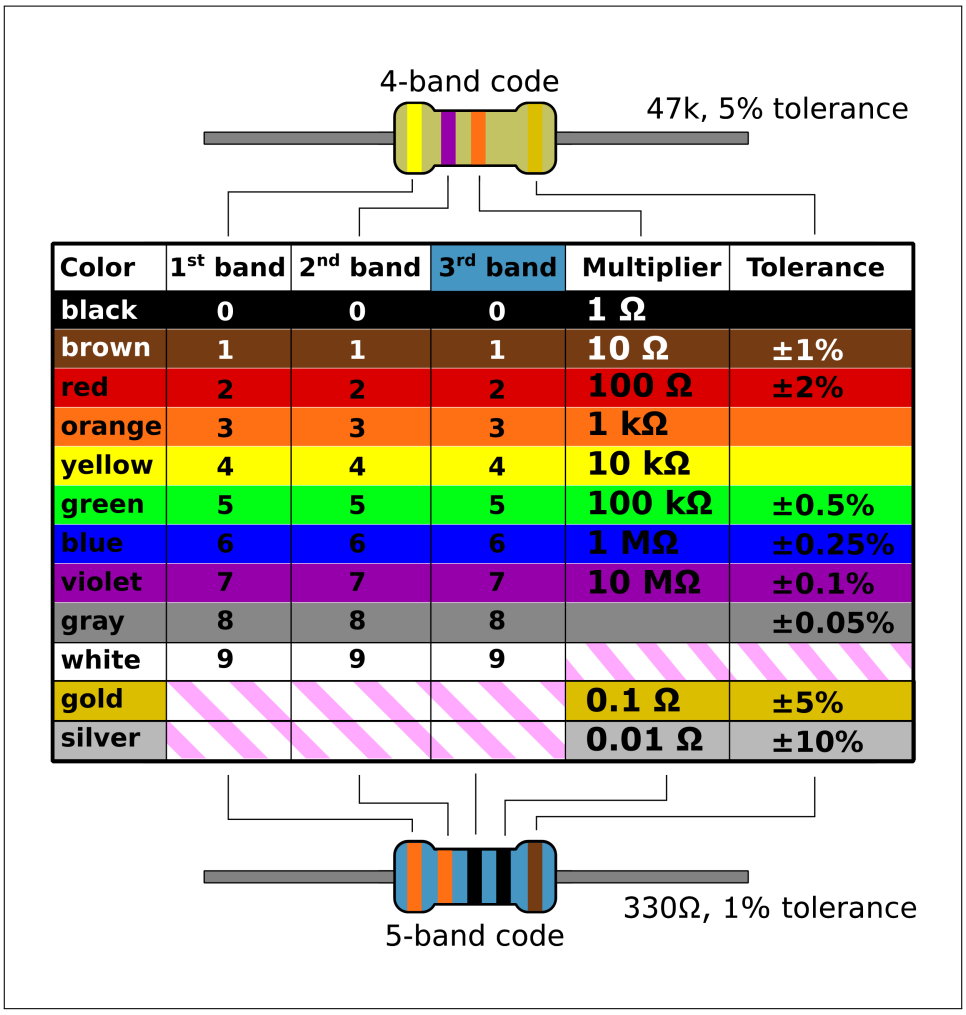


Figure 35: Two bits of data storage, using JK flip-flop gates.

# Resistor Chart

Resistors are marked with colored bands to make it easy to determine what resistance value they have, and how close each individual part's value is guaranteed to be to that stated value.




| Color | 1st band | 2nd band | 3rd band | Multiplier | Tolerance |
|---|---|---|---|---|---|
| black | 0 | 0 | 0 | 1 Ω | |
| brown | 1 | 1 | 1 | 10 Ω | ±1% |
| red | 2 | 2 | 2 | 100 Ω | ±2% |
| orange | 3 | 3 | 3 | 1 kΩ | |
| yellow | 4 | 4 | 4 | 10 kΩ | |
| green | 5 | 5 | 5 | 100 kΩ | ±0.5% |
| blue | 6 | 6 | 6 | 1 MΩ | ±0.25% |
| violet | 7 | 7 | 7 | 10 MΩ | ±0.1% |
| gray | 8 | 8 | 8 | | ±0.05% |
| white | 9 | 9 | 9 | | |
| gold | | | | 0.1 Ω | ±5% |
| silver | | | | 0.01 Ω | ±10% |

Figure 36: A resistor value color band chart. The 'tolerance' stripe (how close the value is guaranteed to be to the stated value) is on the right side.