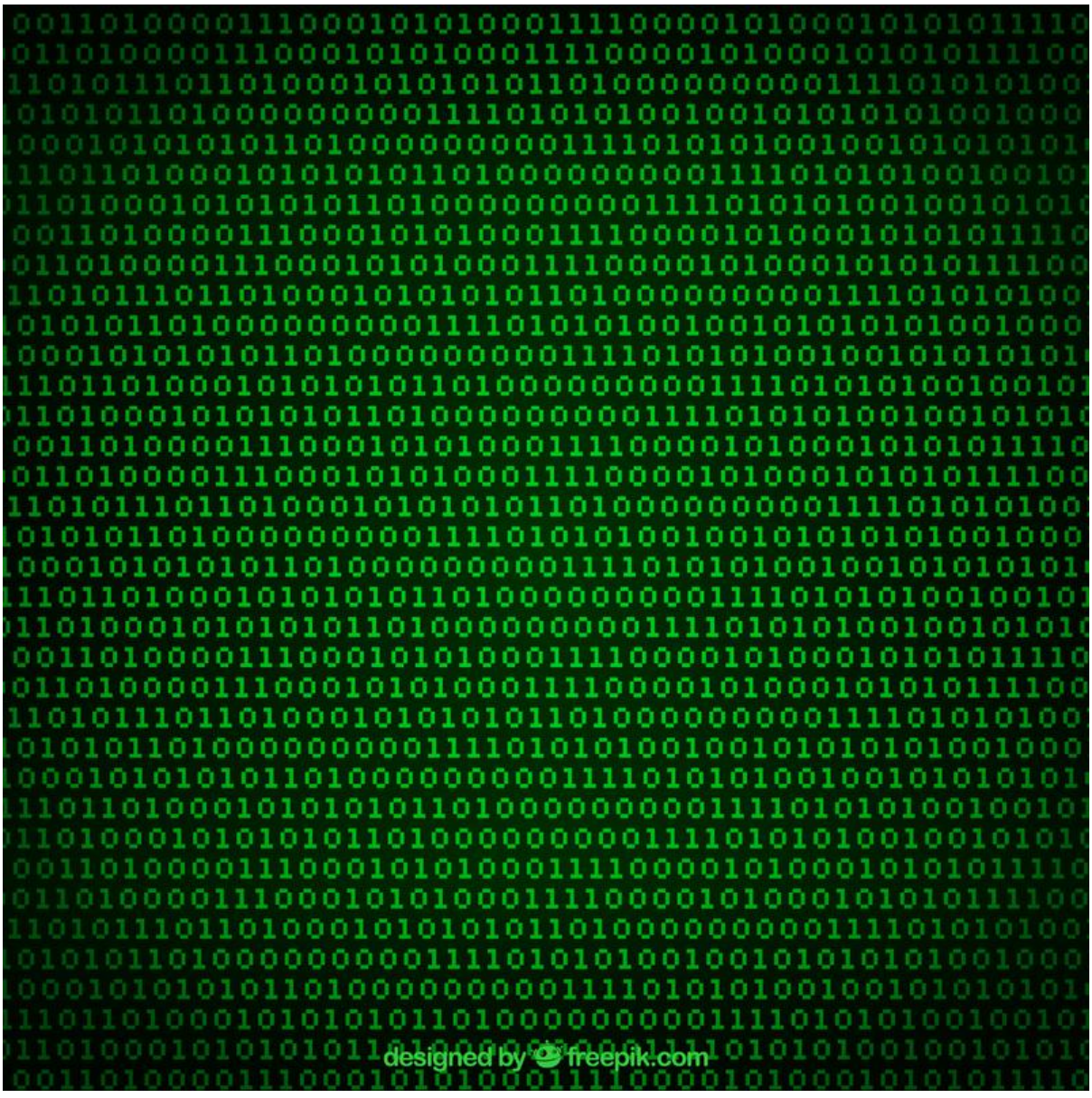
Digital Circuits 1: Binary, Boolean, and Logic

Created by Dave Astels



https://learn.adafruit.com/binary-boolean-and-logic

Last updated on 2022-12-01 03:10:50 PM EST

Table of Contents

Introduction

This is the first in a series of guides that will look at digital circuits: what they are, how they work, and how we can make use of them.

We'll start with the basics and build up to more and more powerful circuits, all based on what we've covered in earlier parts of the series. The plan is to end up with a simple 8 bit processor designed, built, and functioning.

As we go, in true maker fashion, we'll build what we need to help in our exploration. Schematics, build videos, parts lists, and where to find everything will be provided so I hope you will follow along and find out what really goes on in that little black chip on your Arduino, Feather, Trinket, or whatever board you use. It's fun and exciting, and it's the building blocks that all of computing is built on. At least until the not too distant future.*

That's where we're headed, but we'll start simply. With numbers.

* Quantum Computing

Key image Designed by Freepik () and used with permission.

Number systems

Decimal: base 10

We have 10 fingers, so the number 10 comes naturally to us. We count and do math using decimal numbers. Another term for that is base-10. All that really means is that we use 10 unique digits and when we put those digits together to make bigger numbers, each place moving to the left is worth ten times the place to its right. We say there is the one's place, the ten's place, the hundred's place, the thousand's place, and so on.

For example, the number

3571

has a 1 in the one's place, 7 in the ten's place, 5 in the hundred's place, and 3 in the thousand's place. We could represent is like so:

1 * 1 + 7 * 10 + 5 * 100 + 3 * 1000

Notice that each subsequent place has a multipler that is a power of 10 (I use the fairly standard ^ as the exponentiation operator):

1 * 10^0 + 7 * 10^1 + 5 * 10^2 + 3 * 10^3

This is why it's called base-10: it's based on 10.

There's nothing inherently special about basing a number system on 10. We have 10 fingers, so it's natural for us.

But there are other number systems that work better in some situations.

Binary

Binary is the foundation of information representation in a digital computer. Binary is also known as base-2. In binary we use 2 unique digits: 0 and 1. Similar to decimal, each place is "worth" a power of two: 2^0 (the 1's place), 2^1 (the 2's place), 2^2 (the 4's place), 2^3 (the 8's place), and so on.

What does the earlier (in decimal) 3571 look like in binary?

110111110011

That's

1*2^0 + 1*2^1 + 0*2^2 + 0*2^3 + 1*2^4 + 1*2^5 + 1*2^6 + 1*2^7 + 1*2^8 + 0*2^9 + 1*2^10 + 1*2^11

That's a lot more verbose. So why is this used in computers? It turns out that deep inside, computers are made up of very simple circuits. So simple that everything is in terms on or off. That corresponds directly with 1 and 0. Hence binary corresponds directly to how computers store information.

A potential problem is that a string of ones and zeros could easily be a decimal number. We typically prefix binary numbers with `0b` to make it clear, especially in source code. In this case the example number would be written as `0b110111110011`.

Octal

Binary is rather verbose and tedious to work with, so other, more compact number systems have been adopted.

In octal (aka base-8) we use the digits 0-7 (i.e. 8 unique digits and we have those laying around from base-10, so why not reuse them). Additionally, each place is a power of 8: 1's, 8's, 64's, etc.

Octal was common in the early days of computing when computers commonly had 12, 24, or 36 bit words instead of the 8, 16, 32, or 64 bits that have been common since then. To represent a single octal digit in binary takes exactly 3 digits. Considering that those early word lengths were multiples of 3 long, octal was a natural choice. Our earlier example (3571, aka 110111110011) in octal would be 6763.

    3 * 8^0 + 6 * 8^1 + 7 * 8^2 + 6 * 8^3

You can see this translation quite clearly if you separate the binary number in groups of 3 digits, starting at the right: 110 111 110 011.

Because octal numbers look a lot like decimal ones, it's we generally prefix them with `0o`. So our example would usually be written as `0o6763`. Again, this is standard in source code.

Hexidecimal

Also known as hex, hexadecimal is base-16. That is, each digit is worth increasing powers of 16.

Unlike octal where we had plenty of digits from decimal to make use of, we need 16 unique digits for hex. We have plenty of alphabetic characters just lying around so we can borrow a few. In fact for hex we use the digits 0-9 plus the letters A-F (or just as commonly: a-f).

0-9 have their usual meanings, but the letters have the following decimal values:

- `A - 10`
- `B - 11`
- `C - 12`
- `D - 13`

- `E - 14`
- `F - 15`

Similarly to octal, a hex digit represents exactly 4 binary digits. If we take a binary number and divide it into groups of 4 digits (starting at the right) each group corresponds to a hex digit.

- `0 - 0000`
- `1 - 0001`
- `2 - 0010`
- `3 - 0011`
- `4 - 0100`
- `5 - 0101`
- `6 - 0110`
- `7 - 0111`
- `8 - 1000`
- `9 - 1001`
- `A - 1010`
- `B - 1011`
- `C - 1100`
- `D - 1101`
- `E - 1110`
- `F - 1111`

Using the same example:

110111110011 becomes 1101 1111 0011 which becomes DF3

    3 * 16^0 + F * 16^1 + D * 16^2

or, using decimal values for the places:

    3 * 16^0 + 15 * 16^1 + 13 * 16^2

As before, we use a prefix to denote a hex number. No, not `0h` for some reason. Instead we use `0x`.

Since modern computers use word lengths that are multiples of 8, hex is pretty standard. In fact there are often situations where the fact that value is byte (8 bits), such as the red, green, and blue values in a color specification. In cases like this it is standard to use hexidecimal numbers. Instead of `(192, 255, 128)` you would

usually write `(0xC0, 0xFF, 0x80)`. The same is also commonly done with 16 and 32 bit numbers. This makes it explicit that the value needs to fit in a certain number of bits and can't be just anything.

Why?

So if decimal is we use ubiquitously throughout life, why the fascination with binary and similar system (e.g. hexadecimal)? It comes down to the fact (mentioned earlier) that computers work in binary internally. If we are going to study the internal structures of computers, in this case digital logic, it's way easier to use binary (or similar) system because that's the basis of everything in that world.

Boolean Logic

Logic

We're not talking about philosophical logic: modus ponens and the like. We're talking about boolean logic aka digital logic.

Boolean logic gets it's name from George Boole who formulated the subject in his 1847 book The Mathematical Analysis of Logic. Boole defined an algebra (not shockingly, called Boolean Algebra) for manipulating combinations of True and False values. True and False (we'll use T and F as a shorthand)... sounds similar to 1 and 0, or on and off. It should be no surprise that boolean algebra is a foundation of digital circuit design.

Basic Operations

AND

Conjunction: the result is T if, and only if, all inputs are T; if any input is F, the result is F.

OR

Disjunction: the result is T if any (including all) inputs are T; the result is F if, and only if, all inputs are F.

NOT

Negation/Inversion: the result is T if the input is F, and F if the input is T.

Truth Tables

A very useful tool when working with boolean logic is the truth table. For simplicity and consistancy, we'll use A, B, C, and so on for inputs and Q for output. Truth tables list all possible combinations of inputs. At this point we limit our discussions to 2 inputs, though logical operations can have any number. For each combination the corresponding result is noted. As we explore more complex logic circuits we will find that there can be multiple outputs as well.

Here are tables for AND, OR, and NOT.

Combinations

These basic operations can be combined in any number of ways to build, literally, everything else in a computer.

One of the more common combinations is NAND, this is simply NOT AND. Likewise N OR is NOT OR.

NAND

A	B	Q
F	F	T
F	T	T
T	F	T
T	T	F

NOR

A	B	Q
F	F	T
F	T	F
T	F	F
T	T	F

Exclusive OR/NOR

One more pair of operations that is particularly useful (as we'll see in a later guide) is exclusive-or and it's negation: exclusive-nor.

XOR

A	B	Q
F	F	F
F	T	T
T	F	T
T	T	F

XNOR

A	B	Q
F	F	T
F	T	F
T	F	F
T	T	T

The XOR is like an OR with the exception that Q is F if all the inputs are T, and T only if some inputs are T. You could describe this as Q being T when not all inputs are the same.

Of course, XNOR is the opposite: Q is T if, and only if, all inputs are the same.

We will see these gates later when we explore adders.

The Magical NAND

NAND is particularly interesting. Above, we discussed AND, OR, and NOT as fundamental operations. In fact NAND is THE fundamental operation, in that everything else can be made using just NAND: we say that NAND is functionally complete. NOR is also functionally complete but when we consider implementation circuits, NOR generally requires more transistors than NAND.

Look at the truth table for NAND. What happens if the inputs are the same? NAND acts like NOT. So you can make NOT from a NAND by connecting all its inputs together.

NAND is simply NOT AND, so what is NOT NAND? It's equivalent to NOT NOT AND. The NOTs cancel out and we are left with AND. So we can make an AND from 2 NAN Ds, using one to negate the output of the other.

What about OR, our other basic operation? The truth tables shows what A NAND B is. What happens if we negate A and B (denoted by placing a bar over A and B)? We have (NOT A) NAND (NOT B). Let's write out the truth table for that.

A	B	$\overline{A}$	$\overline{B}$	$\overline{A}$ NAND $\overline{B}$	A OR B
F	F	T	T	F	F
F	T	T	F	T	T
T	F	F	T	T	T
T	T	F	F	T	T

That's interesting; it looks just like the truth table for A OR B.

This is a known phenomenon in boolean algebra: De Morgan's law. It was originally formulated by a Augustus De Morgan, a 19th century mathematician. It defines a relationship between AND and OR. Specifically if you negate an AND or OR expression, it is equivalent to using the other operation with the inputs negated:

NOT (A AND B) = (NOT A) OR (NOT B)

and

NOT (A OR B) = (NOT A) AND (NOT B)

How can we use this? Well, NOT (A AND B) is the same as A NAND B so

A NAND B = (NOT A) OR (NOT B)

if we now replace A with (NOT A) and B with (NOT B) we get

(NOT A) NAND (NOT B) = (NOT (NOT A)) OR (NOT (NOT B))

Knowing from above that pairs of NOTs cancel out, we are left with:

(NOT A) NAND (NOT B) = A OR B

BOOM!

NOT can be made with a NAND, so with 3 NANDs we can make OR. Specifically:

(A NAND A) NAND (B NAND B) = A OR B

Logic Gates

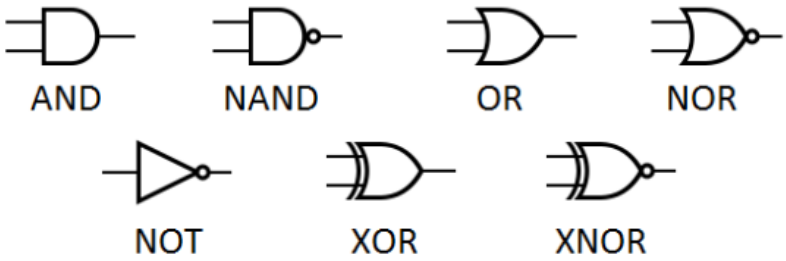
From math to silicon

One neat thing about what we've been discussing so far is that it converts directly into silicon and circuitry.

Earlier we used True (T) and False (F) to indicate logic states. When we talk about actual circuits, it's more typical to talk about high (H) and low (L) signals.There's some leeway as to what voltages are considered high and low, but we can idealise it to high being Vcc and low being Gnd.

Just like any kind of circuits we need a way to describe logic circuits. And just like other circuits, there is an assortment of diagramming symbols we can use. In the same way there are standard symbols for resistors, capacitors, and the like, there are symbols for the different gates (and as well as more complex digital building blocks

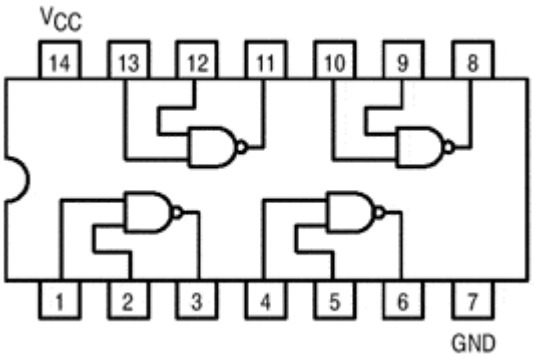
like counters and shift registers. Semiconductor companies such as Texas Instruments, Motorola, Fairchild, etc. publish "Data Books" on their lines of ICs. I've always found these to be a superb resource for knowing what chips are available, their pinouts, and functional characteristics.



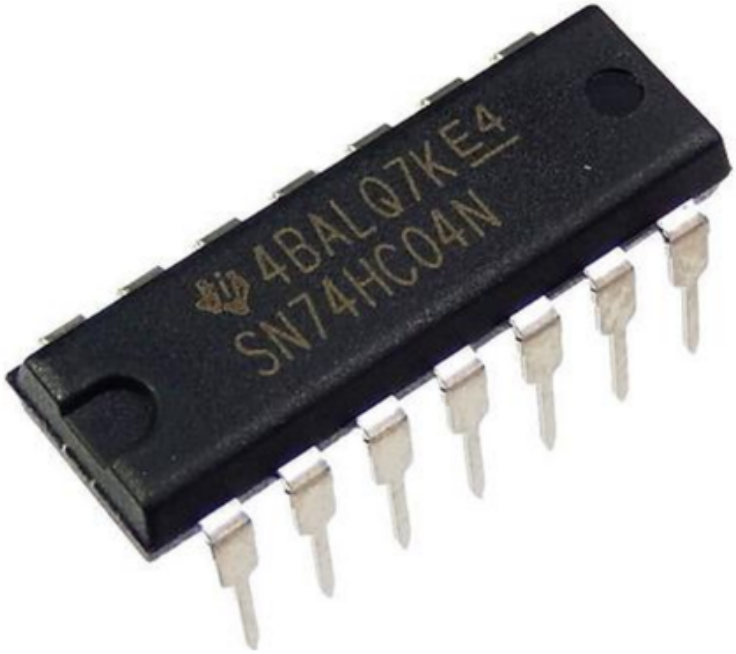
Be sure to learn these symbols, as we will be using them extensively in the guides to come in this series.

If you actually want to follow along and build the circuits we will be exploring, you'll need some logic ICs. I like using TTL (short for Transistor-Transistor-Logic, which refers to an implementation approach). TTL chips are robust, fairly impervious to static discharge, and still quite readily available. Digikey has a wide assortment as does Jameco and other component retailers.

When you get a logic gate IC, you usually don't get a single gate. Depending on the exact configuration of inputs, you might get 2, 3, 4, or even 6 gates on a single chip. For example here's configuration of a 7400, a chip that contains 4 2-input NAND gates.

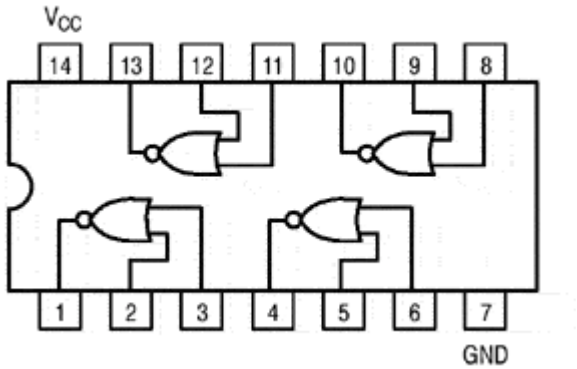


Physically it comes in a 14 pin chip. These are available in surface mount packages, but it's a lot easier to work with on a breadboard if you go oldschool with DIP (Dual Inline Package) chips. The 7400 is packaged in a DIP-14, that looks something like (except that this is a 7404, hex NOT gate):



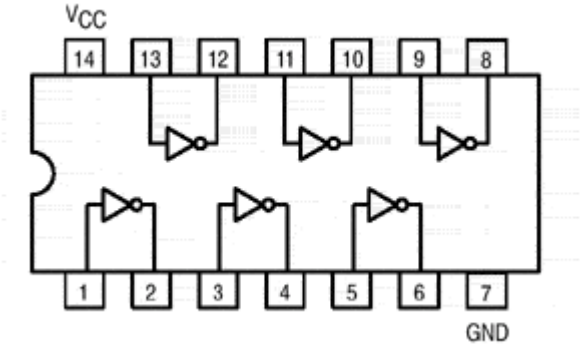
Note the dimple in one end. That corresponds to the notch in the pinout diagram. It denotes the end of the chip with pin 1. Pin numbering starts there and proceeds counter-clockwise around the chip. Some chips will have a small circular depression near pin 1 instead. In any case, there will always be a clear way of telling where pin 1 is.

The other simple, 2-input gates will be similar, but be sure to check the datasheets to confirm pinouts as sometimes they differ. For example, here's the configuration of the 7402, a quad 2-input NOR gate:

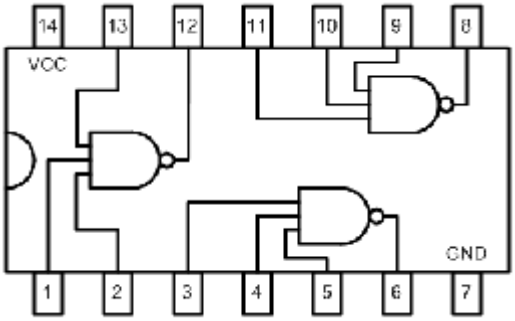


That said, Vcc and GND are *usually* on the same corner pins: 7 & 14, 8 & 16, etc depending on the size of the chip. But always double check.

Since a NOT gate only needs 2 connections instead of 3, 6 will fit in a DIP-14. Here's the 7404:



A DIP-14 will fit 3 3-input gates, such as the 7410 3-input NAND:



Hands On

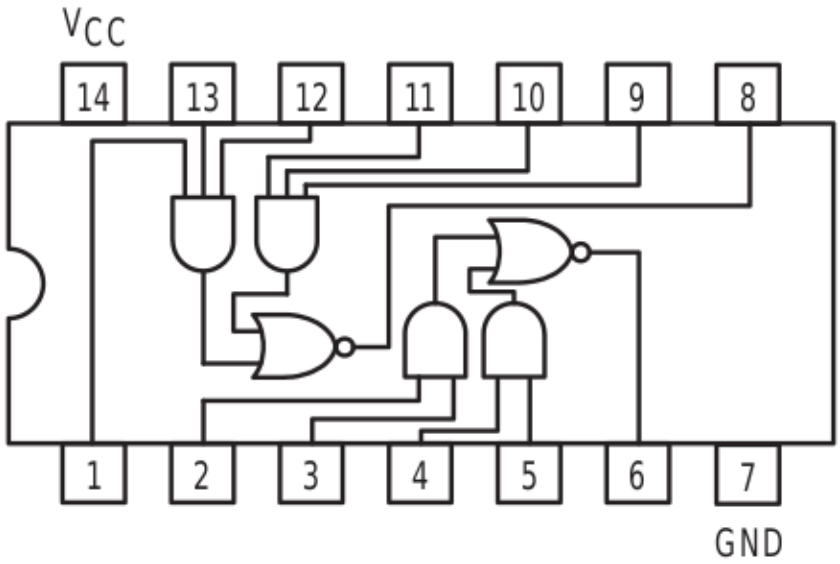
Using the various gate chips, experiment with connecting inputs to various combinations of logic levels and observe the outputs. You can either use a logic probe (see the next guide in the series for a DIY logic probe) or connect output to an LED (via a current limiting resistor 220-330 ohm)

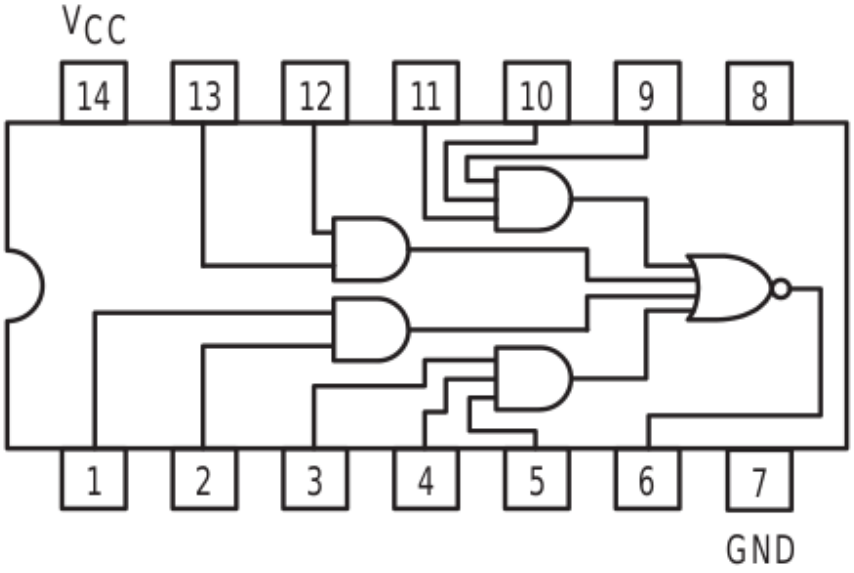
7400 - 2 input NAND
7402 - 2 input NOR
7404 - NOT
7408 - 2 input AND
7420 - 4 input NAND
7430 - 8 input NAND
7432 - 2 input OR

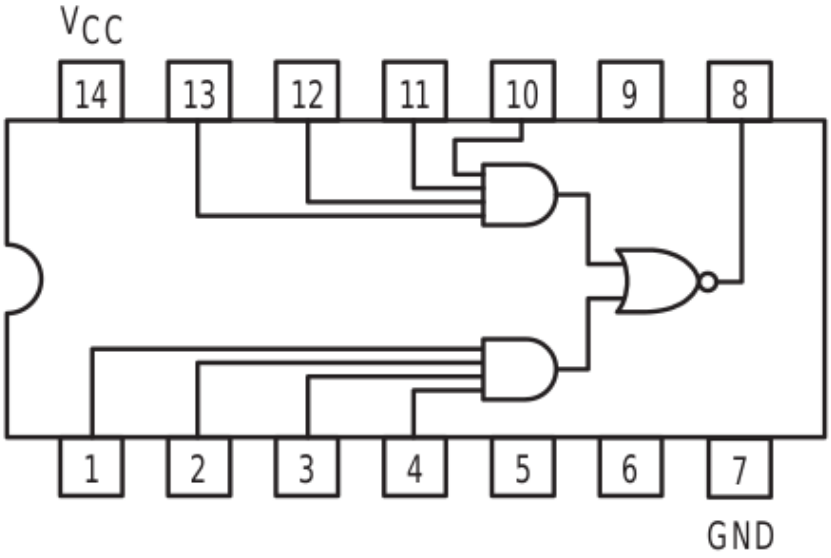
An interesting exercise is to get a bunch of 7400 chips and build the other types of gates (AND, OR, NOT, NOR, XOR, and XNOR) just using NANDs.

Next combine gates, connecting the output of some to the inputs of others. Note that you can not connect gate outputs together, and each input can be connected to a single output.* However a single output can be connected to multiple inputs.

One interesting combination is ORing the output of AND gates. This AND-OR combination is so common that there are chips such as 7451, 7454, and 7455 that combine AND, OR, and NOT. They are referred to as AND-OR-INVERT gates.







Practical concerns

A logic circuit has what's called a fan-out. That indicates how many logic inputs an output can be connected to and still work reliably. This is typically high enough with TTL that it won't be an issue for these experiments, but in more complex circuits it can be a concern. You can usually rely on a fan-out of at least 10 for TTL.

Another physical limitation is speed, or rather the lack of speed: latency. Each gate takes a minute amount of time to generate a new output when it's inputs change. This is it's latency. At this point you can completely ignore this, but as circuits get more complex it adds up as each gate takes a bit of time. Eventually a design will need to take latency effects into account.

* We'll look at open-collector outputs in a later guide.

Supplies, parts, and equipment

You can find TTL chips at the major electronics component retailers including Digikey and Jameco (which I have very happily dealt with). Be sure to specify that you want DIP chips, not SMT.

You'll need some solderless breadboard. Adafruit carries various sizes, but I've found that the large has plenty of space to experiment. It may seem like overkill but you'll end up needing it if you continue following along at home. And having breadboard space available is never a bad thing. I have a habit of throwing 3-packs of full and half sized ones onto Amazon orders.

You'll need an assortment of jumper wires to use with the breadboard. Some alternatives are listed on the right. The key is they need to be male-male to use the with breadboard. Having an few lengths can be handy.

Next you'll need a good (i.e. with stable, smooth output voltage) 5 volt power supply. I've listed a 2 amp model which will probably do well until toward the end of the series. If you don't mind the price, I've listed a 10 amp model as well that will do quite nicely.

Finally you will need a way to get that power onto the breadboard. If you get/have a 5v power supply with a standard 2.1mm plug, you can get the breadboard friendly barrel-jack I have listed.

There are other alternatives, just be sure it's at least a couple amps and provides a steady output.

Next time

I'll look at some tools that will be handy in the parts to come. Of course, being makers I'll provide circuits and examples of them so you can make your own.

Closing

Have fun and experiment. If you have questions you can find me on Twitter and Adafruit's Discord server.

There will soon be a video to run through techniques and some experiments.

Series Index

1. Binary, Boolean, and Logic ()
2. Some Tools ()
3. Combinational Circuits ()
4. Sequential Circuits ()
5. Memories ()
6. An EPROM Emulator ()
7. MCUs... how do they work? ()