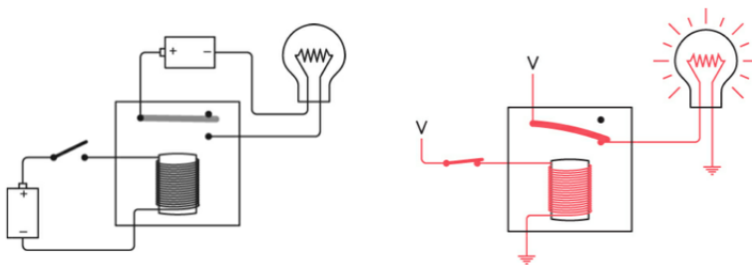


Logic gates

We want to explain the process of binary addition as a computer does it. This write-up is based on Charles Petzold's wonderful book, *Code*. Nearly all the figures below are taken from the book.

We start with a setup with two circuits that interact. Each circuit has a battery and a switch. In the first circuit, when the switch is closed, an electromagnet is energized by running current through a wire coiled in a series of loops. This produces a magnetic field which can then be made to open and close a switch in the second circuit.

When the second switch is closed, the indicator light is on.



The drawing on the right shows the ON condition and also abstracts the batteries and the circuits to voltages V that are connected to ground through the circuit elements.

We may call the first voltage a *signal* that energizes the electromagnet, closing the second switch and lighting the signal light.

But the light will be on only if the signal or voltage is also present in


the second circuit. Both signals are required. As a result, if a strong voltage is continually present at the beginning of the second circuit, what comes through faithfully reflects a series of ON and OFF signals in the first one. The circuit boosts the first signal, it is an *amplifier*.

The components shown in the box make a *relay*. Relays were originally used as signal boosters in telegraph lines. Imagine that the line in the first circuit is tens of miles long and the signal reaching the relay is weak. The relay boosts the signal back to the desired level so that it may continue on its way.

Mechanical relays have been replaced by other devices with similar function, first by vacuum tubes and then later by transistors. The basic function in tubes and transistors is still as we have drawn it above.

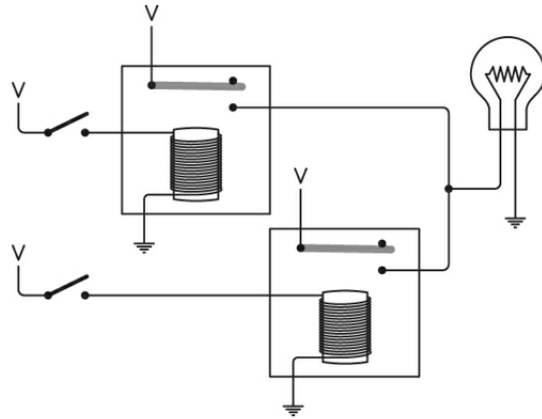
We will use the relay as a logic gate. We can represent its action as taking two input signals and computing the result of the logical AND operation. Let us symbolize the inputs and output as 0 (no signal) or 1 (signal present).

input 1	input 2	AND
0	0	0
0	1	0
1	0	0
1	1	1



The standard symbol for an AND gate is shown to the right of the logic table. We only get a signal out if both inputs are present.

An OR gate is simpler. Just connect the output of two independent inputs, like so:



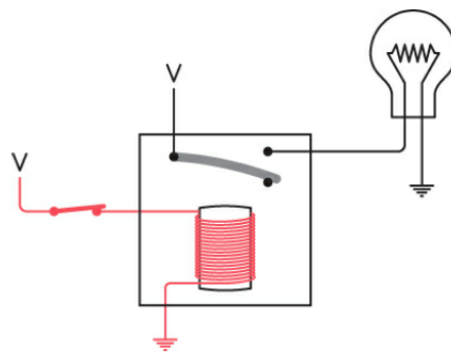
If either switch is connected, the light bulb will light.

in 1	in 2	OR
0	0	0
0	1	1
1	0	1
1	1	1



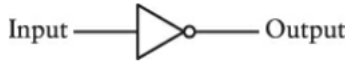
The symbol for the OR gate has a rounded base or input region (to the left) and a pointy output region (to the right).

The third fundamental gate is called an inverter. Whatever the signal is, the output is opposite.



Here is the logic table.

in	NOT
0	1
1	0



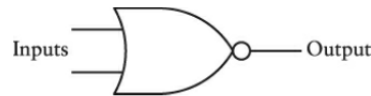
The triangular thingie is a generic symbol called a buffer. The important part is that little circle. It symbolizes the inverter. Crucially, an inverter can be added to any other gate to invert the output. Here is NAND (not and):

in 1	in 2	AND	NAND
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0



and NOR (not or):

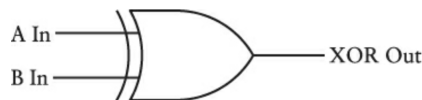
in 1	in 2	OR	NOR
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0



Besides AND and NAND, OR and NOR, there is a third pair, namely, XOR and XNOR.

XOR stands for exclusive or. We want the signal to come through if either one of the inputs is present, but not when both are present. The logic table and symbol are:

in 1	in 2	XOR
0	0	0
0	1	1
1	0	1
1	1	0



Cryptography gives a great illustration of where this operation is handy. Suppose I have a message, the *plaintext*, in binary, and also a *key* of random values, also in binary.

The letters of the message have been converted to binary according to their position in the alphabet.

```
text: A      T      T      A      C      K
p:  00001 10100 10100 00001 00011 01011
```

The key was generated randomly.

```
k:  00001 00110 01101 01110 10111 11111
```

The encryption process is to compute the XOR, bit by bit, of the plaintext and the key. For each column in the data for p and k , the XOR of the first two rows is written to the third row.

```
text: A      T      T      A      C      K
p:  00001 10100 10100 00001 00011 01011
k:  00001 00110 01101 01110 10111 11111
c:  00000 10010 11001 01111 10100 10100
```

The ciphertext appears random as well, if you don't know the key. However, if you have the secret key, you can XOR the key and the ciphertext, to regenerate the plaintext.

I will leave it to you to figure out the table and symbol for the negation of XOR, the XNOR gate.

addition

With that introduction, we come to the heart of the problem. We want to implement binary addition using logic gates.

The sum of two binary numbers is something like

$$01 + 01 = 10$$

One plus one equals two. For the *ones place*, the two inputs are both 1 so the resulting sum has 0 in the ones place. However, there is also the carry bit.

The 1 symbol in the two's column of the answer comes from first adding 0 plus 0, then plus 1 from the carry bit.

The sum part (the ones place of the answer) has this logic:

in 1	in 2	SUM
0	0	0
0	1	1
1	0	1
1	1	0

We notice that this is just XOR.

Before, we skipped showing the circuit corresponding to the XOR operation. Now we will construct it, not from relays, but by combining gates.

Below is written the logic table for the NAND and NOR operations.

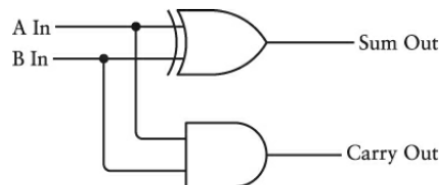
To be clear, the first two columns below are the inputs, the third is the output from the NAND operation on the inputs, and the fourth is the output from the OR operation on the inputs (it doesn't use anything from the third column).

in 1	in 2	NAND	OR
0	0	1	0
0	1	1	1
1	0	1	1
1	1	0	1

I hope you can see that if these two outputs are given as inputs to an AND gate, the result will be what we want.

The 1 symbol in the two's column of the answer to our simple problem comes from adding 0 plus 0, plus 1 from the carry bit, since the ones place of the addends has two 1's in it.

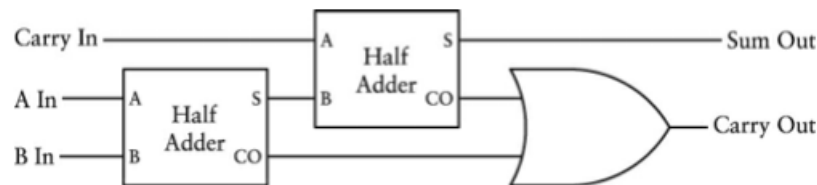
Here is how the combined pieces work together:



The XOR gate is exactly what we want for the sum.

To get the carry bit we simply use an AND gate. The combination is called a half-adder.

We showed how to build XOR from three gates: NAND, NOR plus AND. To combine a carry bit plus the two inputs and get the SUM out plus the carry bit, takes two half-adders and an OR gate.



So that's two half-adders, each with $\text{XOR} = \text{NAND} + \text{OR} + \text{AND}$, plus one AND. Finally, there is another OR gate.

Since there are three logical inputs, there are $8 = 2^3$ possible input states. Let us go through one of the eight.

Suppose that A is 1 and B is 0, and there is a carry bit input.

Then the bottom (first) half adder gives sum 1 and carry 0. The second half-adder has input $A = 1$ and $B = 1$ and the output is the final sum

0 and the intermediate carry 1.

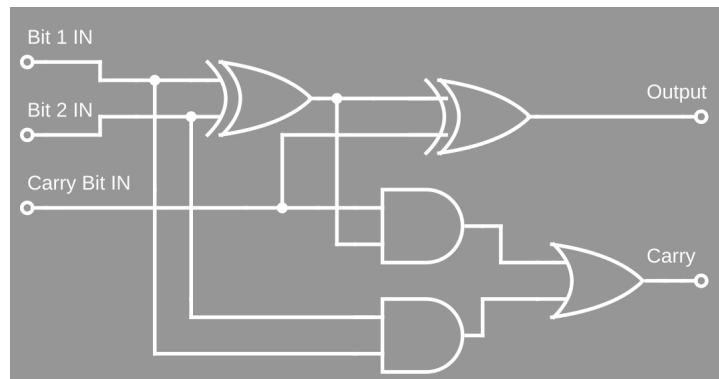
The last step takes the carry output from the second half-adder and does OR with the first carry output 1, yielding a final carry out of 1.

We leave it as an exercise to show that this circuit gives the correct outputs for the other seven possible input states.

The book goes on to combine adders to build something that will add bytes. In that device, an input carry of 0 must be provided to the first half-adder.

You must also decide what to do with the carry bit output from the last place. If this happens, it's called *overflow*. For example, in a 16 bit machine dealing with binary integers that are *unsigned*, the largest integer is $2^{16} - 1 =$ decimal 65535 or binary 1111 1111 1111 1111.

Here is another diagram I found on the web that shows all the gates for an adder.



While it is differently laid out, all the components are the same as what we had before.

There is a total of nine elementary logic gates (since each XOR is built from three gates).