Classical Information and Bits

The simplest variable which can carry information is the *bit*: a variable which can take only two values, 0 and 1. Any description—any ensemble—any set of discrete values—can be quantified by the number of bits needed to express it (as can any set of continuous values up to some precision $\delta > 0$).

Bits also have an important connection to classical logic, in which 0 represents *False* and 1 represents *True*. This connection is exemplified by the rules of *Boolean arithmetic*:

$$\begin{aligned} 0 \cdot 0 &= 0, & 0 + 0 &= 0, \\ 0 \cdot 1 &= 0, & 0 + 1 &= 1, \\ 1 \cdot 0 &= 0, & 1 + 0 &= 1, \\ 1 \cdot 1 &= 1, & 1 + 1 &= 1. \end{aligned}$$

Here the Boolean operation $+$ is equivalent to the logical operation "OR," and the Boolean operation $\cdot$ is equivalent to "AND."

A useful object in classical logic is the *Boolean function* $f(x_1, x_2, \ldots, x_n)$ which takes n bits as arguments and returns a single bit. Since n bits can take 2^n different values, there are 2^{2^n} distinct n-bit functions.

Any Boolean function can be "built up" from some standard set of smaller Boolean functions which take a fixed number of bits (usually 1, 2 or 3). Such standard functions are referred to as *logic gates*. A finite set of logic gates which enable one to build up *any* Boolean function are called *universal* (sometimes referred to as a *complete basis*).

AND, OR and NOT form such a complete basis; in fact, AND and NOT or OR and NOT suffice alone since $a \& b = \neg(\neg a \vee \neg b)$ and $a \vee b = \neg(\neg a \& \neg b)$. There are other universal sets. For instance, the NAND gate by itself is universal (if one is allowed to add *ancilla bits* in fixed states and *fanout*).

Formulas

A *formula* is an expression for a Boolean function in terms of the arguments $x_1, \ldots, x_n$ and the standard basis $\&, \vee, \neg$. An example would be

$$f(x_1, x_2) = (x_1 \& x_2) \vee \neg(x_1 \vee x_2),$$

which returns 1 if $x_1 = x_2$ and 0 otherwise. Any Boolean function can be expressed in terms of a formula (though this expression need not be very simple).

There are some standard forms for representing Boolean functions. For instance, *disjunctive normal form* represents a Boolean function in the form $(\cdot) \vee (\cdot) \vee \cdots \vee (\cdot)$, where the parentheses contain *conjunctions* of *literals* (either a variable x_i or its negation $\neg x_i$).

Similarly, there is *conjunctive normal form*, $(\cdot)\&(\cdot)\&\cdots\&(\cdot)$, where this time the parentheses contain *disjunctions* of literals. Our function $f(x_1, x_2)$ is

$$f(x_1, x_2) = (x_1 \& x_2) \vee (\neg x_1 \& \neg x_2)$$

in disjunctive normal form, and

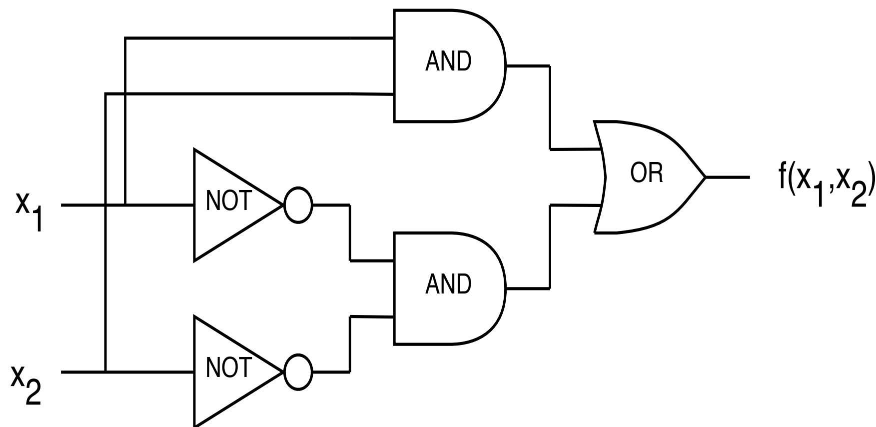
$$f(x_1, x_2) = (x_1 \vee \neg x_2) \& (\neg x_1 \vee x_2)$$

in conjunctive normal form.

In some cases, a complicated formula can be simplified by defining *auxiliary variables*. For instance, the formula $(x_1 \vee x_2 \vee \neg x_3)\&(x_4 \vee x_2 \vee \neg x_3)$ could be simplified by defining a variable $y = x_2 \vee \neg x_3$. We call this an *assignment*.

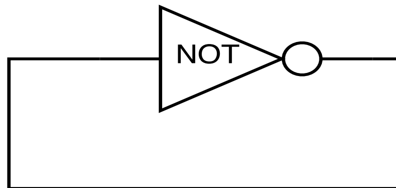
Circuits

It is very common to represent Boolean functions graphically as *circuits*. In this case, the logical values passed are indicated by *wires*, and the basic functions by *gates*. Our example function can be represented by the circuit:



In a computing context, the wires can be thought of as *memory*; each gate is an assignment. Note that the input variables were each used twice; we call this kind of duplication *fanout*. We also allow wires to cross each other, switching the relative positions of two variables. We call this a *crossover* or *swap*.

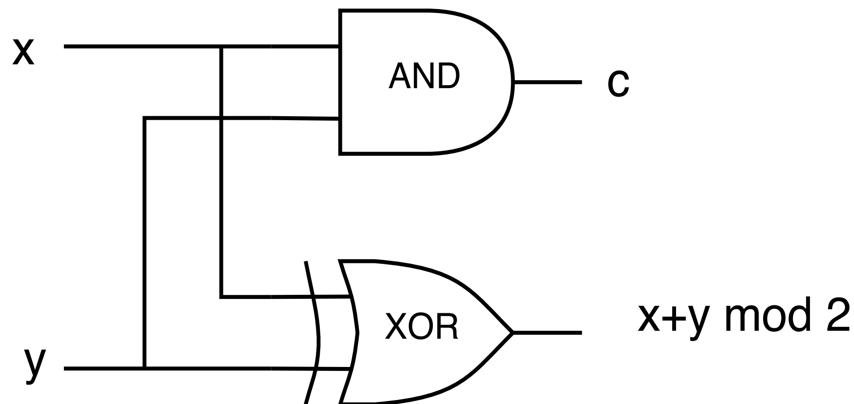
All circuits must be *acyclic*. Otherwise we could have uninterpretable constructions such as this:



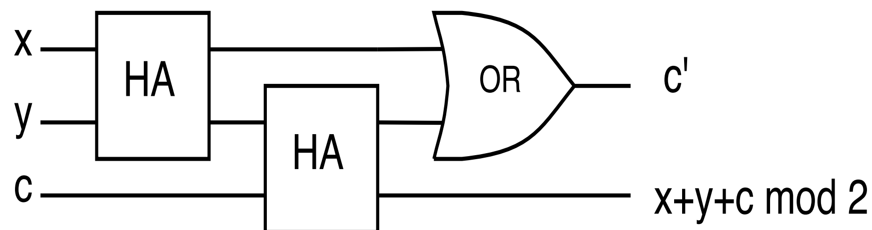
We can define several important quantifiers for a circuit. Define the *size* of a circuit to be the total number of assignments (or gates). The *depth* of a circuit is the *maximum number of gates along any path from the input to the output*. The circuit shown for $f(x_1, x_2)$ had size 5 and depth 3. Depth expresses how *parallelized* a computation is. There is often a trade-off between depth and size.

Another sometimes-useful quantity is the *width*: the maximum number of wires leading to the output at any given time, not including the input variables. This gives a measure of the *space* used by a calculation. However, there is a strong trade-off between width and size; remarkably, *any* Boolean function can be calculated by a circuit with bounded width (though this has a large cost in size).

Here are a few examples of useful circuits. The *half-adder*:

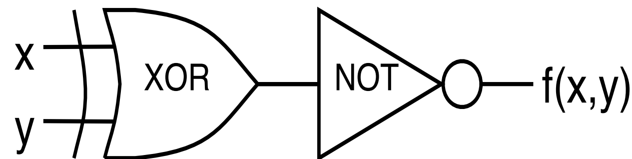


and the *full-adder*:

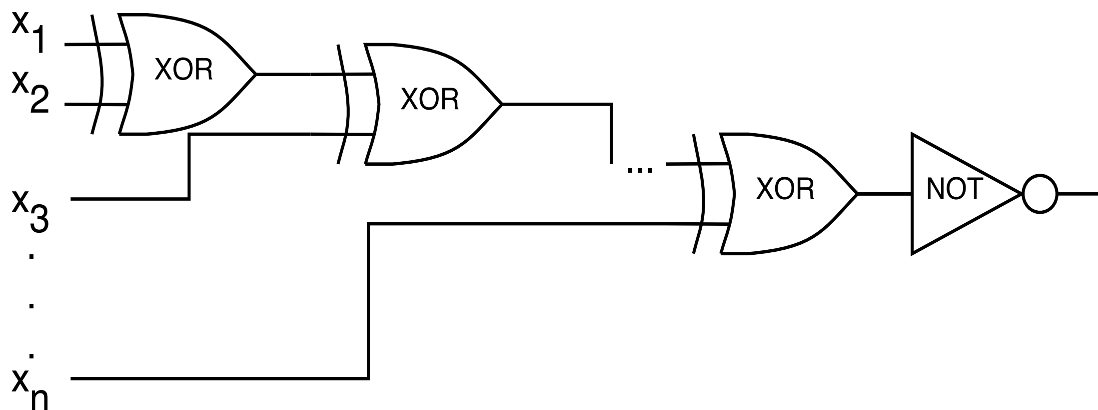


By stringing these together, we can add arbitrary n-bit numbers.

Our already existing function $f(x_1, x_2)$ is also a useful circuit: it can be expressed more efficiently in terms of the XOR gate:



Its generalizations give the negation of the *parity* of an arbitrary input string.



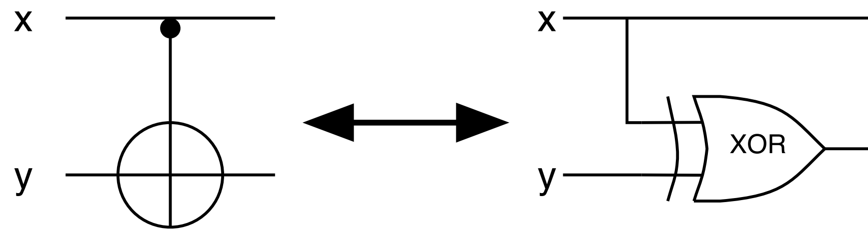
Reversible gates

Most of the standard gates used in classical logic are *irreversible* (or *uninvertible*). From the single bit output of an AND, OR, or XOR gate, it is impossible to deduce the values of the two input bits. The one widely-used reversible classical gate is the NOT gate: given an output of 1, one knows the input was 0, and vice-versa.

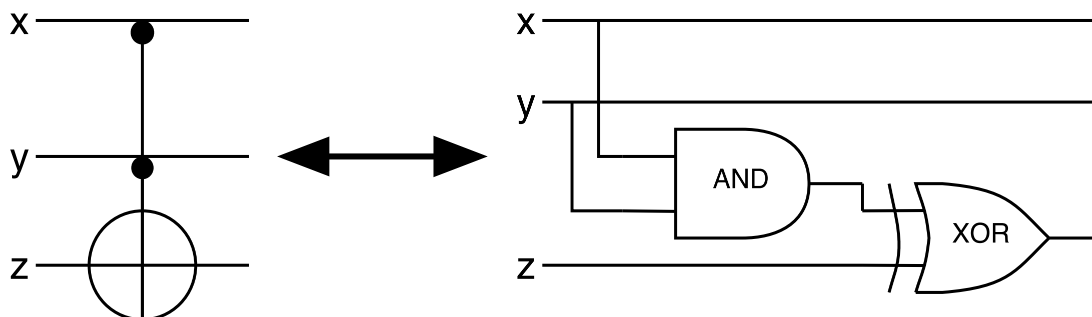
What are the requirements for a gate to be reversible? If it takes n bits as input, there are 2^n possible input values. In order to invert the gate, there must be 2^n distinct outputs. Therefore, an n-bit gate must also have n outputs. (This immediately rules out all of the standard two-bit gates.)

Since an n-bit gate takes the 2^n possible inputs to the 2^n possible outputs, any reversible gate must be a *permutation* on 2^n elements.

The only nontrivial reversible gate on one bit is the NOT gate. What about gates on two bits? The canonical example is the CNOT:



The CNOT is obviously reversible; it permutes the inputs $10 \leftrightarrow 11$. Another obviously reversible two-bit gate is the crossover or SWAP gate, which permutes $10 \leftrightarrow 01$. A generalization to three bits is the controlled-controlled-not or the *Toffoli gate*:



This switches $110 \leftrightarrow 111$.

Reversible circuits

Before we go further with reversible gates, we should state our final goal: to design fully *reversible circuits* consisting entirely of fully reversible gates. Is that even possible?

First, we must generalize to consider functions which take n bits to n bits. (We can think of these as a set of n functions $f_j(x_1, \ldots, x_n)$ for $j = 1, \ldots n$, each of which returns one bit.) If we allow the use of ancillas, we see that any Boolean function $f(x_1, \ldots, x_n)$ can be imbedded in a reversible function by adding an ancilla y:

$$x_1, \ldots, x_n, y \rightarrow x_1, \ldots, x_n, y \oplus f(x_1, \ldots, x_n).$$

This is clearly reversible; indeed, it is its own inverse. We get the value of f by setting the initial state of the ancilla to $y = 0$.

More generally, if we let x denote an n-bit number and $f(x)$ be a function from n-bit numbers to m-bit numbers, we calculate f reversibly by a circuit that yields

$$(x, 0, 0) \rightarrow (f(x), g(x)),$$

where the first zero is an m-bit number and the second zero represents $L - m - n$ extra bits of "scratch space" for some value of $L > m + n$, and $g(x)$ is some $(L - m)$-bit number which contains the extra information needed to make the circuit invertible. Since we are usually uninterested in $g(x)$ per se, it is often referred to as "garbage."

In some cases we can eliminate most of the garbage, by a process known as "garbage collection." Here is an example of how this works.

Suppose that as part of a circuit we wish to calculate a one-bit function $f(p)$ where p is an m-bit number, and $p = p(x)$ where x is the n-bit input number. We need to retain x and $f(p)$ for later in the calculation, but we don't need p. We use m bits of "scratch space" to hold p, n bits to hold x, and one bit to receive the value $f(p)$. A reversible function could look like this:

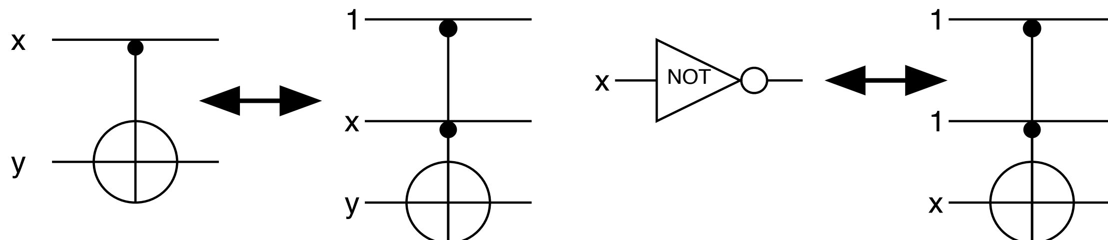
$$(x, 0, 0) \to (x, p(x), 0) \to (x, p(x), f(p))$$

$$\to (x, 0, f(p)).$$

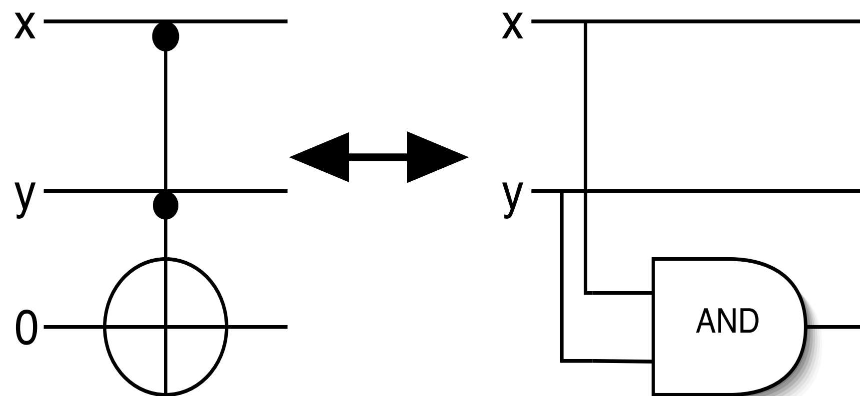
Once we had the value of $f(p)$, we "uncomputed" p, freeing up the scratch space to be used again. This is always possible if we computed $f(p)$ reversibly.

The next question is: can we build a reversible circuit entirely out of reversible gates? This might be possible if the CNOT and NOT gates, together with the ability to add ancillas in known initial states, were universal for computation. Unfortunately, they are not! In particular, it is impossible to realize the Toffoli gate using only CNOT, NOT, and ancillas.

However, the Toffoli gate (together with ancillas) *is* universal for computation. We see that we can easily construct circuits for the NOT and CNOT:



However, we can also do any other two-bit gate as well, by imbedding them inside a reversible circuit.



The OR gate can be realized from the AND and NOT gates. So the Toffoli gate plus ancillas is universal for computation.

Any Boolean circuit can be made reversible at a cost in size which is polynomial in the number of input bits. One simply replaces all the irreversible gates in the original circuit by appropriate arrangements of Toffoli gates and ancillas, and includes extra output bits (which contain the "garbage").

Landauer's Principle

A good question to ask at this point is: *why* try to make circuits reversible? The motivation comes out of what turns out to be a closely related problem: is there a necessary power cost to do a computation? Must one do work to calculate? The answer, perhaps surprisingly, is no.

The reason lies in a principle first discovered by Rolf Landauer: there is no necessary entropy cost to copy a bit, or measure one; but *erasure* of a bit must produce entropy of at least $k_B T \ln 2$ at temperature T. This is because erasure is an irreversible process, and must inevitably increase entropy (like all irreversible processes).

This is a remarkable result: in computation, *physical* reversibility is related to *logical* reversibility. This result has nothing directly to do with quantum mechanics. But if we wish to do computation using quantum systems, the implications are important.

Any classical Boolean circuit can be replaced by a reversible circuit. This means that if we replace the input bits with *q-bits* in the states $|0\rangle, |1\rangle$, and the reversible gates with appropriate unitary transformations, we have constructed a *quantum* circuit which does the same calculation.

Indirectly, we have shown that any computation that can be done by a classical circuit can be done by a quantum circuit with no more than a polynomial increase in complexity. (More on this later.)

Turing machines

Now we will look at a very different model of computation: the *Turing machine*. A Turing machine is a hypothetical computer, conceived by Alan Turing, which is the baseline for our ideas of computability and noncomputability.

The machine consists of three parts. It has a finite number of *internal states*, labeled by a variable q. It has an *infinite tape*, on which are written symbols x drawn from a finite alphabet. Without loss of generality, we can assume them to be 0 and 1. There is also a *read-write head*, which can read a symbol from the tape at the current position, replace it with a new symbol, and move the tape either right or left (or neither).

The operation of the machine is given by its *program*. This is a series of lines of the form (q, x, q', x', m), where q and q' are states, x and x' are symbols, and m is ± 1 or 0. At the start of each step, the computer checks its internal state q and the symbol at the current tape position x. If it has a program line starting with (q, x), it then shifts its internal state to q', replaces the symbol by x', and moves the tape by an amount m.

There are two special states of the machine. The *start* state q_0 is where it always begins the computation. The *halt* state q_h brings the computation to an end. At the start of the computation, the read-write head is at position 0 on the tape (as far left as possible), and the tape contains a finite amount of data (0's and 1's); past the end of the data the tape is all 0's. We can think of this data as being some binary number X of length N.

If the machine tries to back up past position 0, or if there is no program line corresponding to the current (q, x), the machine goes to q_h and halts.

This may seem like a very abstract model of a computer, but it captures the essence of what we mean by computation. The *Church-Turing Thesis* states that any computable function can be computed by an appropriate Turing machine. It has been shown to be equivalent in power to many other models of computation: recursively-enumerable functions, the lambda calculus, and the random-access machine, for three. There are many variations of the Turing machines: machines with two or more tapes, reversible machines, etc. Different variations may be useful for different questions, but they are all equivalent in computing power.

The Universal Turing Machine

For a give number of internal states n, there are a finite number of possible Turing machines. We can therefore label every possible Turing machine with a unique number T. Turing showed that a *universal Turing machine* (UTM) exists.

When this machine is run, its tape initially contains a number T and a number X, either separated by a special symbol or a special binary code. The UTM then will produce the same output that would be produced by Turing machine T given input X. We can think of T as being a description of the machine's *program*, and X as being the data.

This is similar to the way our own computers work—except that they don't have an infinite tape!

The Halting Problem and Uncomputability

If one believes in the Church-Turing thesis, then there are functions which are *uncomputable*. The standard example of that is the *halting problem*.

Turing machines are by no means guaranteed to halt after a finite time. Suppose a Turing machine exists which, given T, calculates $f(T)$: $f(T) = 0$ if machine T halts, and $f(T) = 1$ if machine T never halts. There is then a machine which, given input T, halts if $f(T) = 1$ and goes into an infinite loop if $f(T) = 0$. This machine must itself have a number T'. What then happens if the machine is given its *own* number T'? Clearly this is a logical contradiction. Therefore, no such machine can exist, and $f(T)$ is an *uncomputable function*.

Other functions have been shown to be uncomputable, usually by being shown equivalent to the halting problem. For instance, *Hilbert's Tenth Problem* asks for an algorithm which, given a polynomial equation with integer coefficients states whether it has integer solutions (so-called *Diophantine equations*). It has been shown that such an algorithm cannot exist; it is equivalent to the halting problem.

There is a large gulf between computable and uncomputable functions. But for practical purposes, the important question is often whether a given function $f(x)$ be calculated *efficiently* by a Turing machine, or not? This is what we turn to next.

Next time: Computational Complexity