Chapter 15

Models of Computation

Models of computation are the subject of the computer sciences (dt: Informatik), a branch of mathematics which developed from the 19th century meta-mathematical desire to formalize reasoning and cast arithmetics and other disciplines of mathematics into a purely logical framework. After the first promising results, mostly due to Gottlob Frege, the desire was promoted a challenge by David Hilbert when he formulated his famous problem set which became known as the "Hilbert Programme". Hilbert's challenge, the famous *Entscheidungsproblem*, was to decide whether there exists some general "mechanical procedure" which could, *in principle*, solve all problems of mathematics one after the other. Hilbert expected the answer to be "Yes!", but Kurt Gödel, Alonzo Church, Alan Turing any many other proved the answer to be "No!".

The challenge took the first serious hit with the Russel-Whitedhead paradox in

1902,[1] and was finally put to rest with the Gödel incompleteness theorem in 1931[2], yet despite its failure, it greatly contributed to the devolpment of formal languages and mathematical logic.[3]

In todays terminology, Hilberts mechanical procedure would be called an *algorithm*, and it was Turing's great invention of the celebrated *Turing machine* which captures the intuitive notion of an algorithm and casts it into a mathematical precise form. Befor Turing, recipes (algorithms) and cakes (solutions), say, were different things. Afte Turing, a recipe *is* a cake and vice versa.

The Turing machine, although formulated in the language of devices, with tapes, control units and read- and write head, is not a physical machine, but rather a mathematical model of computation. With respect to real devices, we are likely to

[1]The paradox is a variation of the Epimenides paradox who, as a Cretan, made the magnificent statement "All Cretans are liars". The paradox found its way into mathematics when, in their attempt to reduce arithmetics to logic, Bertrand Russell and Alfred North Withead were forced to contemplate on the mind boggling 'Set of all sets which do not contain themselves as an element'. If the set contains itself as an element, it does not by definition. If it does not contain itself as an element, it does by definition. Russel and Whitehead could rescue their programme, which was published "Principia Mathematica" in 1910–1913, but only at the expense of introducing an infinite hierarchy of sets (sets, classes, families etc).

[2]Published as Proposition VI in the 1931 paper "Über formal unentscheidbare Sätze der *Principia Mathematics* und verwandter Systeme I": Zu jeder ω-widerspruchsfreien rekursiven Klasse κ von Formeln gibt es rekursive Klassenzeichen r, so daß weder ν Gen r noch Neg(ν Gen r) zu Flg(κ) gehört (wobei ν die freie Variable aus r ist). In a nut shell: All consistent axiomatic formulations of number theory include undecidable propositions. For a nice exposition see the Bestseller and Pulitzer Price winner "Gödel, Escher, Bach: an Eternal Golden Brain" by Douglas R. Hofstadter, Basic Books, New York (1979). The German translation "Gödel, Escher, Bach: ein Endloses Geflochtenes Band" was published by Klett-Cotta in 1984 [ISBN 3-608-93037-X].

[3]One of the early proponents of the programme, Gottlob Frege, is now generally reckognized to be *the* founding father or modern logic and the philosophy of language. His "Begriffschrift" (1879) is still a model of concise formulation and clear thinking.

encounter the following *types* of computers[4]

- serial computer – what most of us have at home.
- parallel computer – what some of us have at work.
- distributed computing – what few did in order to crack RSA(129).
- DNA computing – what Adleman did first
- quantum computer – what we are working on

All these compute stuff, and all these are instances of the Turing machine.

With algorithm promoted a mathematical entity, the analyis of algorithm becomes a mathematical subject – the subject of complexity theory. Complexity theory classfies problems according to whether they are easy, hard, or even impossible to solve. The Halting problem, for example, is impossible to solve: there is no universal algorithm (mechanical procedure) which decides whether any given algorithm (which would prove the Goldbach conjecture, say) ever halts or not. This is Turing's nail in the coffin of Hilbert's challenge.

Quite generally, a problem's classification is in terms of the resources consumed by the algorithm which solves the problem. Resources are the running time (number of elementary steps) and storage space (number of bits involved). A problem is easy if the number of elementary steps necessary for its solution is a polynomial of the

[4]Which was not foreseen for quite some time. In the words of Thomas Watson, chairman of IBM in 1943: "There is a world market for maybe five computers". Evidently, the statement had little impact on most scientist who continued to ponder on the "what's" and "ifs" of computation. So much for the relevance of the statements of chief economists ...

size of the input, measured in bits. Otherwise it is hard.[5] With todays algorithms, multiplication is easy, but factorization is hard.

With all computers being essentially instances of a Turing machine, what would be the benefits of a quantum computer? The answer turns out to lay in algorithmic efficiency – there are problems which are hard on a classical computer, but which become easy on a quantum computer. For the problem of factorization, for example, Shor has found an algorithm which is efficient on a quantum computer, but turns completely inefficient if simulated on a classical computer.

15.1 The Turing machine

A Turing machine consists of a tape, a read-write head and a finite-state control unit. The tape is divided into cells, each cell holding a symbol from a finite alphabet Σ. The cells are numbered $\ldots, -2, -1, 0, 1, 2, \ldots$. The right part of the tape contain the input data $x \in \Sigma^*$. The left part eventually contains the output data $y \in \Sigma^*$. The read-write head serves to read and write the tape, one cell a time. It can be moved one step (cell) to the left (L) or right (R). The control unit coordinates the actions (write, move) of the machine. It is specified by a finite set of internal states $s_1, s_2, \ldots, s_n$. Also included in this set are the "start"-state q_s, and "halt"-state q_h.

The machines function is specified by a finite set of instructions, called the programme, each instruction being of the form $\langle s_i\sigma \Rightarrow \sigma' m s_j\rangle$, with $s_i, s_j \in Q$, $\sigma, \sigma' \in \Sigma$, and m a motion of the head, $m \in \{L, R\}$. The instruction $\langle s_{17}\mathtt{1} \Rightarrow \mathtt{0}Rq_4,$

[5]The following algorithm, written in pseudo-code, solves an easy problem – the problem of dividing 34 by 2: `print 17`. Check! Other problems may be more daring, for example the problem of counting from 1 to, say, N. According to complexity theory, this problem is difficult – it is exponentially hard.

for example, means “if in state s_{17} reading `1` from current cell then (over)write by `0`, move head one position to the right, and change state to s_4”.

The machine starts with the finite-state control in state s_{S}, the read-write head positioned on the cell numbered 0, say, and input data $x \in \Sigma^*$ – a string of length l – in cells 1 to l. The computation then proceeds in a step-by-step manner according to the programme. Once the state control is in state s_{H}, computation halts, with the right part of the tape containing the output data – a string of lenght l, say – in cells -1 to $-m$.

Definition A function $f : \Sigma^* \to \Sigma^*$ is computable by the Turing machine M, if, for all $x \in \Sigma^*$ input to M, it eventually halts with output $y = f(x) \in \Sigma^*$.

Church-Turing thesis The class of functions computable by a Turing machine corresponds exactly to the class of functions which we would naturally regard as being computable by an algorithm.

And although this class of functions in infinite, it is countable, and thus pretty much limited: there is plenty of room for both, inspiration and transpiration, in the mathematical and natural sciences.

The left side of an instruction is coded into a prime number

$$s_i\mathtt{0} \rightsquigarrow p_{2i} \tag{15.1}$$
$$s_i\mathtt{1} \rightsquigarrow p_{2i-1} \tag{15.2}$$

where p_j is the jth prime number.

The right side of an instruction is coded

$$\begin{aligned} s_{\mathrm{H}} &\rightsquigarrow 0 && (15.3)\\ \mathtt{0}Ls_j &\rightsquigarrow 4j-3 && (15.4)\\ \mathtt{0}Rs_j &\rightsquigarrow 4j-2 && (15.5)\\ \mathtt{1}Ls_j &\rightsquigarrow 4j-1 && (15.6)\\ \mathtt{1}Ls_j &\rightsquigarrow 4j && (15.7)\end{aligned}$$

Thus a particular set of instructions, which specifies a given Turing machine, is coded a natural number

$$n_T = {p_1}^{a_1} \cdot {p_2}^{a_2} \cdots p_i^{a_i} \cdots . \qquad (15.8)$$

And vice versa – to any given natural number n, there corresponds one and only one Turing machine, the instructions of which can be decoded from the prime number decomposition of n.

15.2 The circuit model

Conceptually more simple than the Turing machine – yet fully equivalent a model of computation – is provided by the *circuit model*. A circuit consists of *wires* and *gates*. The wires carry the information around. They represent the movement of the cebits through space, say, or through time. The gates process the information. They perform elementary computational tasks.

Simple circuits are displayed in the figures below. Note that none of these circuits displays a loop or any other kind of "feed-back". This is important, as feed-back may cause instabilities or other kinds of uncontrolled behavior. A circuit without loops is called *acyclic*. We will be dealing only with acyclic circuits models, leaving cyclic models (of quantum circuits) for Diploma thesis or the like.

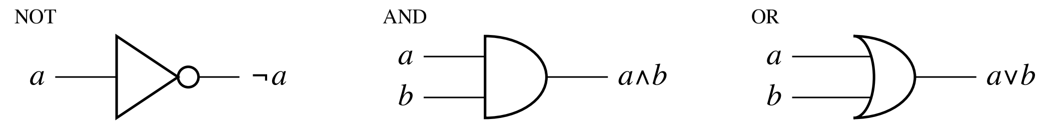


Figure 15.1: The most common elementary gates.

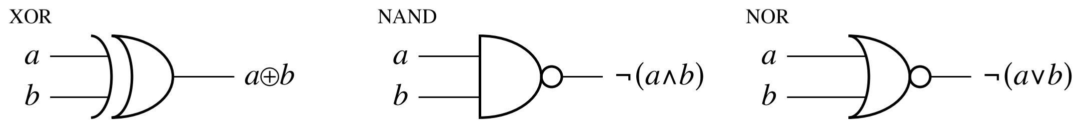


Figure 15.2: More Popular gates.

15.2.1 Elementary gates

A logical gate is a function $f : \{0,1\}l \to \{0,1\}^k$ from some fixed number l of *input bits* to some fixed number k of *output bits*. The NOT, for example, is defined by $f(x) = 1 \oplus x$, where $\oplus$ denotes addition modulo 2. Most common are the NOT, the AND, and the OR – see Fig. 15.1. Other elementary gates are the XOR, the NAND and the NOR – – see Fig. 15.2.

The set of gates dipicted in Fig. 15.1, if complemented by the FANOUT and the SWAP, is universal: its members can be used to build any given n-to-l bit circuit. The HALF-ADDER, for example, is composed from a AND, a XOR, a FANOUT, and a

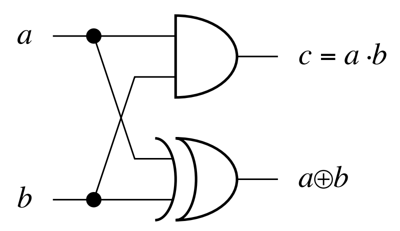


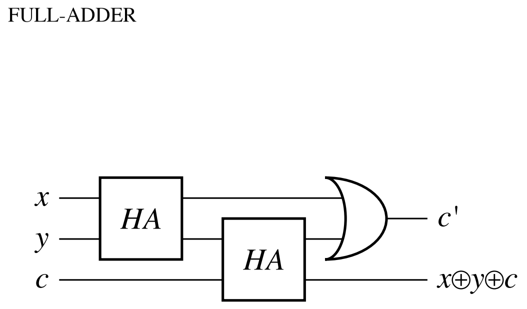
Figure 15.3: The HALF-ADDER. The full dot represents a FANOUT, the crossing a SWAP.

SWAP, see Fig. 15.3. It takes two bits, x and y as input, and produces output $x \oplus y$ together with a carry bit c. A cascade of two HALF-ADDERS and a OR may be used to build a FULL-ADDER, and a cascade of a HALF-ADDER and FULL-ADDER may be used to build a circuit $add(x, y)$ for the addition of two integers – see Fig. 15.4.

A smaller yet still universal a set of gates is provided by the NAND, the SWAP and the FANOUT – see Fig. 15.5.

15.2.2 Reversible gates

With the exception of the unary NOT and ID (wire), the other elementary gates are *irreversible* – one can not infer the input from the output. There are, however, reversible gates which in principle allow to run the gate operation "backwards". The most common reversible gates – the CNOT, the FREDKIN, and the TOFFOLI – are depicted in Fig. 15.6. These gates typically involve one or more *control bits* – bits which control the operation on the other bits, called the *target bits*. In the controlled-not CNOT, for example, the target bit is flipped if the control bit is set to 1. Otherwise it is left unchanged.



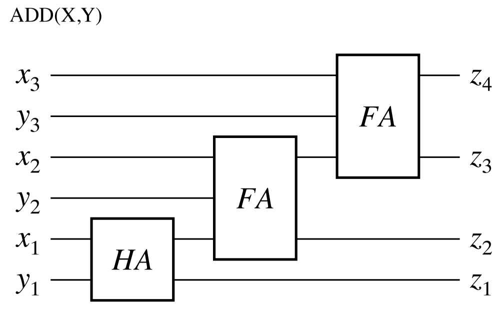


Figure 15.4: The FULL-ADDER (left) and an adder for two three-bit numbers (right). Left: the value of the carry-bit c is given by $c' = xy \oplus yc \oplus cx \oplus xyx \oplus xyc$. Right: the integer $z = \sum_{i=1}^{4} z_i 2^{i-1}$ is the arithmetic sum of $x = \sum_{i=1}^{2} x_i 2^{i-1}$ and $y = \sum_{i=1}^{2} y_i 2^{i-1}$, where $x_i, y_i, z_i \in \{0, 1\}$.

Both, the Toffoli gate and the Fredkin gate are universal by themselves, i.e. without special need for routing gates like FAN-OUT or SWAP. Figure 15.7 depicts the configuration of the Toffoli gate for the implementation of the NAND and the FANOUT. Figure 15.8 depicts the configuration of the FREDKIN for the implementation of the AND, the NOT (which also covers the FANOUT), and the SWAP. Quite typically, the implementations involve *ancilla bits* – bits input in some fixed state 0 or 1 – and produces some *garbage bits* – bits which are not needed for the computation.

Ancilla bits, and in particular the garbage bits poses some real threat to our resources in space (and energy) which are, after all, pretty much limited. Fortunately, the overhead in space scales linear in the gate operations, and the overhead in energy can be reduced to zero.

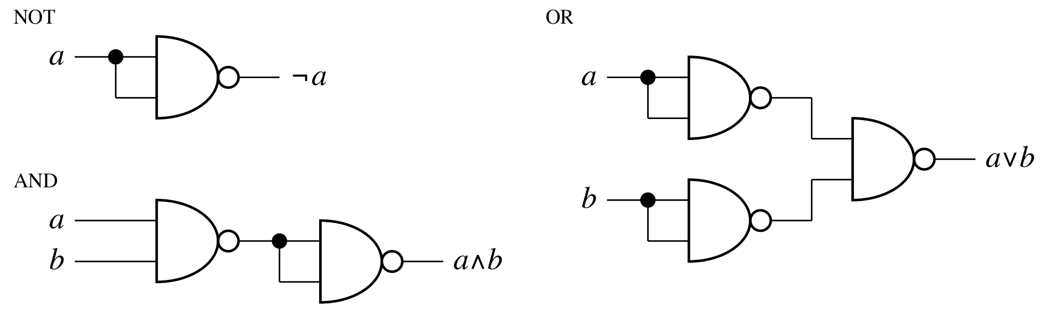


Figure 15.5: Constructing elementary gates from the NAND.

15.3 Complexity

Computational complexity is the study of the time and space resources required to solve computational problems. The task is to to prove lower bounds on the resources required by the best possible algorithm for solving a problem, even if that algorithm is not explicitely known. A problem is regarded as easy (tractable or feasible) if an algorithm for solving the problem using polynomial resources exists, and as hard (intractable or infeasible) if the best possible algorithm requires exponential (super-polynomial) resources.

There are essentially three properties that may be varied in the definition of a complexity class: the resources of interest (time, space,. . .), the type of problem being considered (decision problem, optimization problem,. . .), and the underlying computational method (deterministic TM, probabilistic TM, quantum computer,. . .).

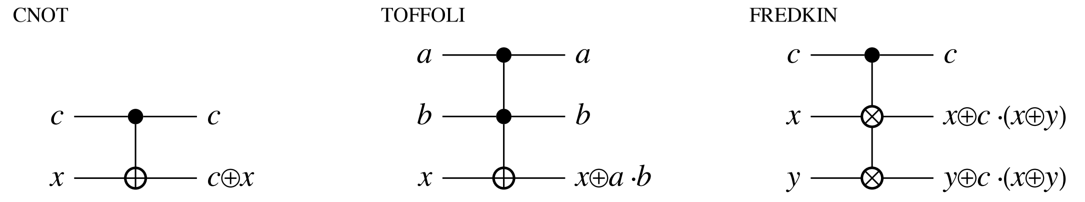


Figure 15.6: Reversible gates.

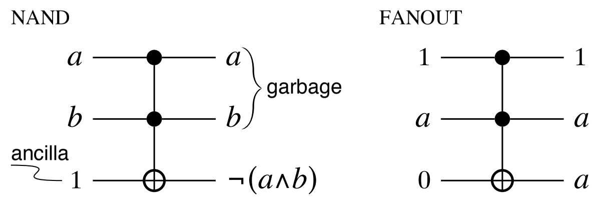


Figure 15.7: Constructing elementaries from Toffoli.

15.4 Energy of computation

In most of complexity theory only time and space are considered resources. Yet physics knows another resource of potential interest – energy. Surely, the operation of a real computer costs energy, energy which is dissipated in the environment in form of heat. To date the dominant source of energy consumption is ohmic resistance. With better materials being developed, however, this energy consumption can shrink to zero. On the microscopic level the laws of nature are invariant under time reversal, the dynamics is reversible, and energy is conserved.

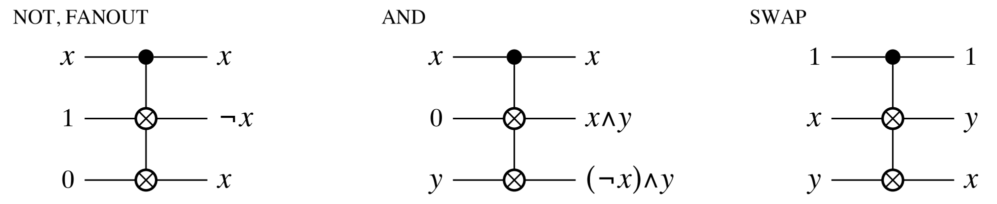


Figure 15.8: Constructing elementaries from Fredkin.

Yet, as was pointed out by Landauer in 1961, there still is unavoidable energy consumption which comes with the *erasure* of information. For a computer operating in an environment at temperature T, the erasure of one bit costs at least energy $k_\mathrm{B}T\ln 2$, where k_B is the Boltzmann constant.

Example A molecule in a bi-partite box can store one bit of information. Erasure means that the molecule is moved to the left side, say, irrespective of whether it started out on the left or on the right. Physically, erasure may be achieved by suddenly removing the partition which divides the box, and then slowly compressing the one-moleucle gas with a piston until the molecule is definitely on the left side. For an isothermal process at temperature T, the work $W = k_\mathrm{B}T\ln 2$ must be performed on the piston. The compression leads to a heat flow from the gas to the environment. In the end, the entropy of the gas is decreased – and the entropy of the environment is increased – by an amount $\Delta S = k_\mathrm{B}\ln 2$.

Thus the ultimate limit of energy consumption is set by the amount of information erased per usage of the computer. In a NAND, for example, one bit (on average) is erased per gate operation.

Yet irreversibility is not constitutive an ingredient of computation. The Fredkin and Toffoli gates, for example, are reversible gates. And since both of them are universal, reversible computation – that is computation which consumes no energy – is well possible.

But even with computation fully reversible, there still remains a power bill to pay. Befor computation starts, the register bits must be prepared in a proper input state. As this requires the erasure of the output from the previous computation, irreversibility slips in and Landauer's principle applies.

15.5 DNA Computing

[To be completed]