3 Elements of Classical Computer Science

Computer science is a vast field, ranging from the very abstract and fundamental to the very applied and down-to-earth. It is impossible to summarize the status of the field for an audience of outsiders (such as physicists) on a few pages. The present chapter is intended to serve as an introduction to the most basic notions necessary to discuss logical operations, circuits, and algorithms. We will first introduce logic gates of two types: irreversible and reversible. Later we will discuss the Turing machine as a universal computer and the concept of complexity classes. All this will be done in an informal and highly non-rigorous style intended to provide our physicist readership with some rough idea about the subject.

3.1 Bits of history

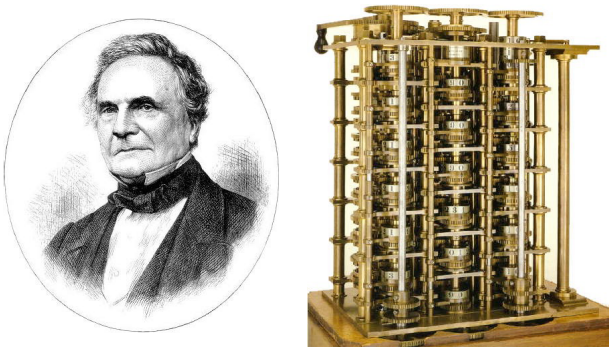

Figure 3.1: Charles Babbage (1791–1871) and a difference engine built according to his design.

The inventor of the first programmable computer is probably Charles Babbage (1791–1871) (→ Fig. 3.1). He was interested in the automatic computation of mathematical tables and designed the mechanical "analytical engine" in the 1830s. The engine was to be controlled and programmed by punchcards, a technique already known from the automatic Jacquard loom, but was never actually built. Babbage's unpublished notebooks were discovered in 1937 and the 31-digit accuracy "Difference Engine No. 2" was built to Babbage's specifications in 1991. (Babbage was also Lucasian professor of mathematics in Cambridge, like Newton, Stokes, Dirac, and Hawking, and he invented important practical devices such as the locomotive cowcatcher.)

The first computer programmer was probably Ada Augusta King, countess of Lovelace (1815–1852), daughter of the famous poet Lord Byron, who devised a program to compute Bernoulli numbers (recursively) with Babbage's engine. From this example we learn that the practice of devising algorithms for not-yet existing computers is considerably older than the quantum age.

Another important figure from 19th century Britain is George Boole (1815–1864) who in 1847 published his ideas for formalizing logical operations by using operations like AND, OR, and NOT on binary numbers.

Alan Turing (1912–1954) invented the Turing machine in 1936 in the context of the decidability problem posed by David Hilbert: Is it always possible to decide whether a given mathematical statement is true or not? (It is not, and Turing's machine helped to show that.)

3.2 Boolean algebra and logic gates

3.2.1 Bits and gates

Classical digital computers are based on Boolean logic. In this context, the "atoms" of information

are the binary digits, or *bits*. Every bit can assume one of two values, which are usually labeled 0 and 1 or true and false. In the computing hardware, bits are represented by easily distinguishable physical states, such as high or low voltage or the presence or absence of a charge or current, or the direction of a magnetization. Generally, information is then encoded in a string of bits, where the length of the string depends on the amount of information.

Information processing corresponds to manipulation of this information. Computations are defined by algorithms, i.e. sequences of elementary logical operations like NOT, OR, and AND, that act on (transform) strings of bits. Any transformation between two bit strings of finite length can be decomposed into one- and two-bit operations. (See [32]; a proof of the quantum version of this important fact will be sketched in Chapter 5.)

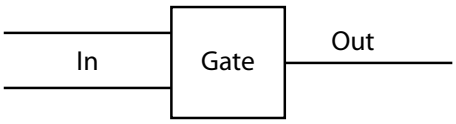

Figure 3.2: A logic gate with two input bits and one output bit.

Logic operations or gates can be characterized by the number of bits that they take as input and the number of bits they produce as output. Figure 3.2 shows a simple example with two input bits and one output bit. This representation of logic gates, where wires represent bits and boxes the gate operations leads naturally to what is called the *network model of computation* (often also called the *circuit model*).

The simplest type of logic gate operations are the one bit gates, which act on a single input bit and produce a single output bit. Four possible operations may be applied to a single bit: the bit may be left untouched (identity), it may be flipped (NOT), and it may be set to 0 or 1 unconditionally. The latter two operations are obviously irreversible.

3.2.2 2-bit logic gates

At the next level of complexity are the 2-bit logic gates. We first discuss one-bit functions of a two-bit argument:

$$(x, y) \longrightarrow f(x, y) \text{ where } x, y, f = 0 \text{ or } 1.$$

Logic gates of this type are called *Boolean functions*. The four possible inputs 00, 01, 10, 11 can each be mapped to one of two possible outputs 0 and 1; the function is completely characterized by the string of four output bits $(f(00), f(01), f(10), f(11))$. Since there are $2^4 = 16$ different output strings, we have 16 possible Boolean functions of two binary variables. Most of these gates are *irreversible* since the output is shorter than the input.

x	y	x OR y	x AND y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

The binary operations OR and AND are defined by their *truth tables*, see the table above. Some elementary logic operations have useful algebraic representations:

- NOT $x \to 1 - x$
- x AND $y \to xy$
- x XOR $y \to x \oplus y$, i.e. addition modulo 2 .

All other operations, such as IMPLIES or XOR can be constructed from the elementary operations NOT, OR, and AND. As an example for the reduction of a logical operation to more elementary operations consider

$$x \text{ XOR } y = (x \text{ OR } y) \text{ AND } \text{ NOT } (x \text{ AND } y).$$

(XOR is also often denoted by $\oplus$, because it is equivalent to addition modulo 2.)

We now return to the 16 Boolean functions of two bits. We number them according to the four-bit output string as given in the above truth table,

read from top to bottom and interpreted as a binary number. For example AND outputs 0001=1 and OR outputs 0111=7. We can thus characterize each gate or function by a number between 0 and 15 and look at them in order. Some examples are:

0: The absurdity, e.g. $(x \text{ AND } y) \text{ AND } \text{ NOT } (x \text{ AND } y)$.

1: x AND y

2: x AND (NOT y)

3: x, which can be written in a more complicated way: $x = x$ OR $(y$ AND NOT $y)$

4: (NOT x) AND y

5: $y = ...$(see x above)

8: (NOT x) AND (NOT y) =: (x NOR y)

9: ((NOT x) AND (NOT y)) OR (x AND y) = NOT (x XOR y) =: (x EQUALS y)

All others can be obtained by negating the above; notable are

13: NOT (x AND (NOT y)) =: x IMPLIES y

14: NOT (x AND y) =: x NAND y

15: The banality, for example $(x \text{ AND } y) \text{ OR } \text{ NOT } (x \text{ AND } y)$.

We have thus seen that all Boolean functions can be constructed from the elementary Boolean operations. Furthermore, since

$$x \text{ OR } y = (\text{ NOT } x) \text{ NAND } (\text{ NOT } y),$$

we see that we *only* need NAND (as defined by line **14**) and NOT to achieve any desired classical logic gate with two input bits and one output bit.

In order to connect an arbitrary number n of input lines to m output lines we need, in addition to the logic gates shown schematically in Figure 3.2, the ability to COPY the contents of one bit to a different bit while keeping the original bit. This is usually symbolized by a branching line in a network diagram, which symbolizes a branching wire with equal voltage levels at the three terminals. While copying a classical bit is thus a trivial operation, copying a quantum bit turns out to be impossible! This *no-cloning theorem* will be discussed in Chapter 4; it is at the heart of the schemes developed for *secure quantum communication* to be discussed in Chapter 13.

3.2.3 Minimum set of irreversible gates

We would like to reduce the number of gates needed to perform an arbitrary bit string operation to the absolute minimum. Being able to build a network using the smallest possible set of different elements is desirable from a theoretical point of view. In practice, however, it is usually more advisable to employ a larger variety of gates in order to keep the total size of the network smaller. We note that

$$\begin{aligned} x \text{ NAND } y &= \text{ NOT } (x \text{ AND } y) \\ &= (\text{ NOT } x) \text{ OR } (\text{ NOT } y) \\ &= 1 - xy. \end{aligned}$$

If we can copy x to another bit, we can use NAND to achieve NOT:

$$x \text{ NAND } x = 1 - x^2 = 1 - x = \text{ NOT } x$$

(where we have used $x^2 = x$ for $x = 0, 1$). Alternatively, if we can prepare a constant bit 1:

$$x \text{ NAND } 1 = 1 - x = \text{ NOT } x.$$

We can also express AND and OR by NAND only:

$$\begin{aligned} & (x \text{ NAND } y) \text{ NAND } (x \text{ NAND } y) \\ = \; & 1 - (1 - xy)^2 \\ = \; & 1 - (1 - xy) = xy = x \text{ AND } y \end{aligned}$$

and

$$\begin{aligned} & (x \text{ NAND } x) \text{ NAND } (y \text{ NAND } y) \\ = \; & (\text{ NOT } x) \text{ NAND } (\text{ NOT } y) \\ = \; & 1 - (1 - x)(1 - y) = x \oplus y - xy \\ = \; & x \text{ OR } y. \end{aligned}$$

Thus the combination of the NAND gate and the COPY operation (which is not a gate in the strict sense defined above) forms a *universal set* of (irreversible) classical gates. A different universal set of two gates is given by NOR and COPY, for example.

The operations NAND and COPY can both be performed by a single two-bit to two-bit gate, if we can prepare a bit in state 1. This is the NAND/NOT gate:

$$\begin{aligned}(x, y) &\longrightarrow (1 - x, 1 - xy) \\ &= (\text{ NOT } x, x \text{ NAND } y).\end{aligned} \tag{3.1}$$

The truth table is

x	y	NOT x	x NAND y
0	0	1	1
0	1	1	1
1	0	0	1
1	1	0	0

The NOT and NAND functions are obviously achieved by ignoring the second and first output bit, respectively. For $y = 1$ we obtain COPY, combined with a NOT which can be inverted by the same gate.

3.2.4 Minimum set of reversible gates

Although we know how to construct a universal set of irreversible gates there are good reasons to study the reversible alternative. Firstly, quantum gates *are* reversible, and secondly, reversible computation is in principle possible without dissipation of energy.

A reversible computer evaluates an invertible n-bit function of n bits. Note that every irreversible function can be made reversible at the expense of additional bits: the irreversible (for $m < n$) function mapping n bits to m bits

$$x(n \text{ bits}) \longrightarrow f(m \text{ bits})$$

is replaced by the obviously reversible function mapping $n + m$ bits to $n + m$ bits

$$(x, m \text{ times } 0) \longrightarrow (x, f).$$

The *reversible* n-bit functions are *permutations* among the 2^n possible bit strings; there are $(2^n)!$ such functions. For comparison, the number of *arbitrary* n-bit functions is $(2^n)^{(2^n)}$ (Each of the 2^n input strings can be mapped to every possible output string). The number of reversible 1,- 2-, and 3-bit gates is 2, 24, and 40320, respectively.

While irreversible classical computation gets by with two-bit operations, reversible classical computation needs three-bit gates in order to be universal. This can be seen by observing that the 24 reversible two-bit gates are all linear, that is, they can be written in the form [32]

$$\begin{pmatrix} x \\ y \end{pmatrix} \longrightarrow \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} a \\ b \end{pmatrix},$$

where all matrix and vector elements are bits and all additions are modulo 2. As the two reversible one-bit gates are also obviously linear, any combination of one- and two-bit operations applied to the components of a n-bit vector $\vec{x}$ can only yield a result linear in $\vec{x}$. On the other hand, for $n \geq 3$ there are invertible n-bit gates which are *not* linear, for example, the Toffoli gate to be discussed below. In Chapter 5 we will see that quantum computing, although reversible too, does not need gates acting on three quantum bits to be universal. Furthermore all quantum gates will have to be strictly linear because quantum mechanics is a linear theory.

3.2.5 The CNOT gate

One of the more interesting reversible classical two-bit gates is the controlled NOT, or CNOT, also known as "reversible XOR", which makes the XOR operation reversible by storing one argument:

$$(x, y) \longrightarrow (x, x \text{ XOR } y). \tag{3.2}$$

The following table shows why (3.2) is called CNOT:

x	y	x	x XOR y
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

The target bit y is flipped if and only if the control bit $x = 1$. A second application of CNOT restores the initial state, so this gate is its own

The CNOT gate can be used to copy a classical bit, because it maps

$$(x, 0) \longrightarrow (x, x). \tag{3.3}$$

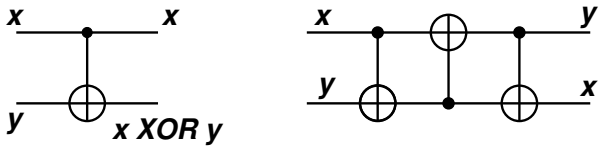

Figure 3.3: Left: Single CNOT gate. Right: SWAP gate.

The network combining three XOR gates in Figure 3.3 achieves a SWAP of the two input bits:

$$\begin{aligned}(x, y) &\longrightarrow (x, x \text{ XOR } y)\\ &\longrightarrow ((x \text{ XOR } y) \text{ XOR } x, x \text{ XOR } y)\\ &\longrightarrow (y, y \text{ XOR } (x \text{ XOR } y)) = (y, x)\end{aligned}$$

Thus the reversible XOR can be used to copy and move bits around.

3.2.6 The Toffoli gate

We will show now that the functionality of the universal NAND/NOT gate (3.1) can be achieved by adding a three-bit gate to our toolbox, the *Toffoli gate* $\theta^{(3)}$, also known as controlled-controlled-NOT, (CCNOT) which maps

$$(x, y, z) \longrightarrow (x, y, xy \text{ XOR } z), \tag{3.4}$$

that is, z is flipped only if both x and y are 1.

Input

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Output

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	1
1	1	0

The nonlinear nature of the Toffoli gate is evident from the presence of the product xy. This gate forms by itself a universal set, provided that we can prepare fixed input bits and ignore output bits:

- For $z = 1$ we have $(x, y, 1) \longrightarrow (x, y, 1 - xy) = (x, y, x \text{ NAND } y)$.
- For $x = 1$ we obtain z XOR y which can be used to copy, swap, etc.
- For $x = y = 1$ we obtain NOT.

Thus we can do any computation reversibly. In fact it is even possible to avoid the dissipative step of memory clearing (in principle): store all "garbage" which is generated during the reversible computation, copy the end result of the computation and then let the computation run backwards to clean up the garbage without dissipation. Though this may save some energy dissipation, it has a price as compared to reversible computation with final memory clearing:

- The time (number of steps) grows from T to roughly $2T$.
- Additional storage space, growing roughly proportional to T, is needed.

However, there are ways [32] to split the computation up in a number of steps which are inverted individually, so that the additional storage grows only proportional to $\log T$, but in that case more computing time is needed.

3.2.7 The Fredkin gate

Another reversible three-bit gate which can be used to build a universal set of gates is the Fred-

kin gate [27]. While the Toffoli gate has two control bits and one target bit, the Fredkin gate has one control qubit and two target bits. The target bits are interchanged if the control bit is 1, otherwise they are left untouched. Table 3.1 shows the input and output of the Fredkin gate, where x is the control bit, and y and z are the target bits, respectively.

Input			Output		
x	y	z	x	y	z
1	1	1	1	1	1
1	1	0	1	0	1
1	0	1	1	1	0
1	0	0	1	0	0
0	1	1	0	1	1
0	1	0	0	1	0
0	0	1	0	0	1
0	0	0	0	0	0

Table 3.1: Truth table of the Fredkin gate.

To implement a reversible AND gate, for example, the z bit is set to 0 on input. On output z then contains x AND y, as can be seen in Table 3.1. If the other two bits x and y were discarded, this gate would be irreversible; keeping the input bits makes the operation reversible.

The NOT gate may also be embedded in the Fredkin gate: setting $y = 0$ and $z = 1$ on input we see that on output $z =$ NOT x, and $y = x$; thus we have implemented a COPY gate at the same time.

3.3 Universal computers

Computer science has developed some concepts that allow one to check computability and efficiency of algorithms in a way that does not depend on hardware implementation. They include some representations of computing machinery that have proved useful for this purpose, although nobody would actually build a computer according to this recipe. The most important example of such a "universal computer" is the Turing machine.

3.3.1 The Turing machine

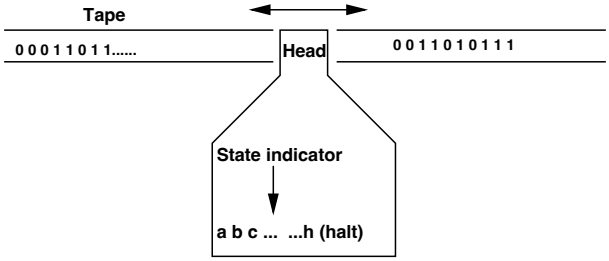

Figure 3.4: A Turing machine operating on a tape with binary symbols and possessing several internal states, including the *halt* state.

The Turing machine has no importance as a practical computing device. However, according to the Church-Turing hypothesis (see next subsection) every task that can be performed by some computer can also be performed by a Turing machine, hence its importance in theoretical computer science as the simplest example for a *universal computer*. The Turing machine acts on a tape (or string of symbols) as an input/output medium. It has a finite number of internal states. If the machine reads the symbol s from the tape while being in state G, it will replace s by another symbol s', change its state to G' and move the tape one step in direction d (left or right). The machine is completely specified by a *finite set of transition rules*

$$(s, G) \longrightarrow (s', G', d)$$

The machine has one special internal state, the "halt" state, in which the machine stops all further activity. On input, the tape contains the "program" and "input data"; on output, the result of the computation.

The (finite) set of transition rules for a given Turing machine T can be coded as a binary number