Learning Objectives:

- See that computation can be precisely modeled.
- Learn the computational model of *Boolean circuits / straight-line programs*.
- Equivalence of circuits and straight-line programs.
- Equivalence of AND/OR/NOT and NAND.
- Examples of computing in the physical world.

3 Defining computation

"there is no reason why mental as well as bodily labor should not be economized by the aid of machinery", Charles Babbage, 1852

"If, unwarned by my example, any man shall undertake and shall succeed in constructing an engine embodying in itself the whole of the executive department of mathematical analysis upon different principles or by simpler mechanical means, I have no fear of leaving my reputation in his charge, for he alone will be fully able to appreciate the nature of my efforts and the value of their results.", Charles Babbage, 1864

"To understand a program you must become both the machine and the program.", Alan Perlis, 1982

People have been computing for thousands of years, with aids that include not just pen and paper, but also abacus, slide rules, various mechanical devices, and modern electronic computers. A priori, the notion of computation seems to be tied to the particular mechanism that you use. You might think that the "best" algorithm for multiplying numbers will differ if you implement it in *Python* on a modern laptop than if you use pen and paper. However, as we saw in the introduction (Chapter 0), an algorithm that is asymptotically better would eventually beat a worse one regardless of the underlying technology. This gives us hope for a *technology independent* way of defining computation. This is what we do in this chapter. We will define the notion of computing an output from an input by applying a sequence of basic operations (see Fig. 3.3). Using this, we will be able to precisely define statements such as "function f can be computed by model X" or "function f can be computed by model X using s operations".

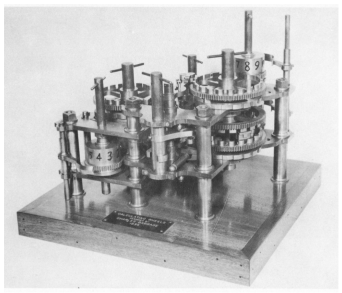


Figure 3.1: Calculating wheels by Charles Babbage. Image taken from the Mark I 'operating manual'

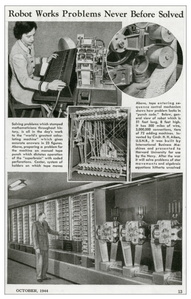
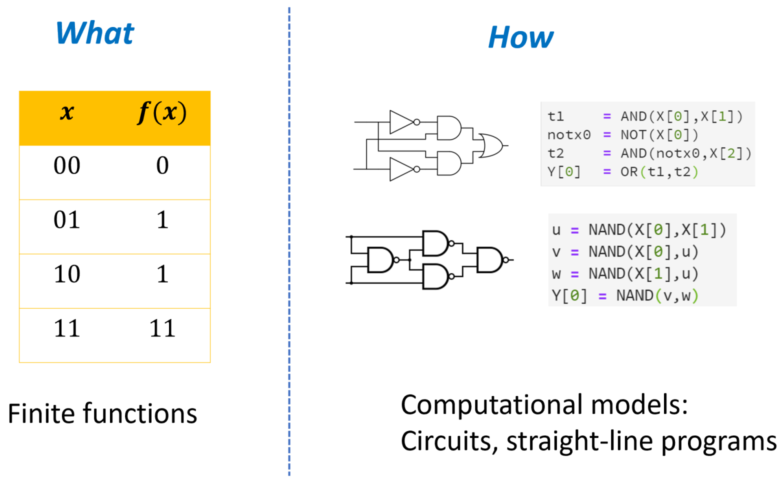

Robot Works Problems Never Before Solved

Figure 3.2: A 1944 *Popular Mechanics* article on the Harvard Mark I computer.



x	$f(x)$
00	0
01	1
10	1
11	11

```
t1    = AND(X[0],X[1])
notx0 = NOT(X[0])
t2    = AND(notx0,X[2])
Y[0]  = OR(t1,t2)
```

```
u = NAND(X[0],X[1])
v = NAND(X[0],u)
w = NAND(X[1],u)
Y[0] = NAND(v,w)
```

Figure 3.3: A function mapping strings to strings *specifies* a computational task, i.e., describes *what* is the desired relation between the input and the output. In this chapter we define models for *implementing* computational processes that achieve the desired relation, i.e., describe *how* to compute the output from the input. We will see several examples of such models using both Boolean circuits and straight-line programming languages.

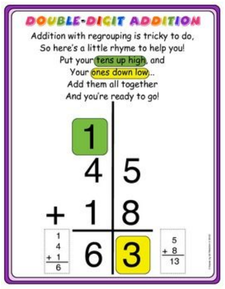
3.1 DEFINING COMPUTATION

The name "algorithm" is derived from the Latin transliteration of Muhammad ibn Musa al-Khwarizmi's name. Al-Khwarizmi was a Persian scholar during the 9th century whose books introduced the western world to the decimal positional numeral system, as well as to the solutions of linear and quadratic equations (see Fig. 3.4). However Al-Khwarizmi's descriptions of algorithms were rather informal by today's standards. Rather than use "variables" such as x, y, he used concrete numbers such as 10 and 39, and trusted the reader to be able to extrapolate from these examples, much as algorithms are still taught to children today.

Here is how Al-Khwarizmi described the algorithm for solving an equation of the form $x^2 + bx = c$:

> *[How to solve an equation of the form] "roots and squares are equal to numbers": For instance "one square , and ten roots of the same, amount to thirty-nine dirhems" that is to say, what must be the square which, when increased by ten of its own root, amounts to thirty-nine? The solution is this: you halve the number of the roots, which in the present instance yields five. This you multiply by itself; the product is twenty-five. Add this to thirty-nine' the sum is sixty-four. Now take the root of this, which is eight, and subtract from it half the number of roots, which is five; the remainder is three. This is the root of the square which you sought for; the square itself is nine.*

Figure 3.4: Text pages from Algebra manuscript with geometrical solutions to two quadratic equations. Shelfmark: MS. Huntington 214 fol. 004v-005r



Addition with regrouping is tricky to do,
So here's a little rhyme to help you!
Put your tens up high, and
Your ones down low...
Add them all together
And you're ready to go!

Figure 3.5: An explanation for children of the two digit addition algorithm

For the purposes of this book, we will need a much more precise way to describe algorithms. Fortunately (or is it unfortunately?), at least at the moment, computers lag far behind school-age children in learning from examples. Hence in the 20th century people came up with exact formalisms for describing algorithms, namely *programming languages*. Here is al-Khwarizmi's quadratic equation solving algorithm described in the *Python* programming language:

```
from math import sqrt
#Pythonspeak to enable use of the sqrt function to compute
 ↪  square roots.

def solve_eq(b,c):
    # return solution of x^2 + bx = c following Al
     ↪  Khwarizmi's instructions
    # Al Kwarizmi demonstrates this for the case b=10 and
     ↪  c= 39

    val1 = b / 2.0 # "halve the number of the roots"
    val2 = val1 * val1 # "this you multiply by itself"
    val3 = val2 + c # "Add this to thirty-nine"
    val4 = sqrt(val3) # "take the root of this"
    val5 = val4 - val1 # "subtract from it half the number
     ↪  of roots"
    return val5  # "This is the root of the square which
     ↪  you sought for"

# Test: solve x^2 + 10*x = 39
print(solve_eq(10,39))
# 3.0
```

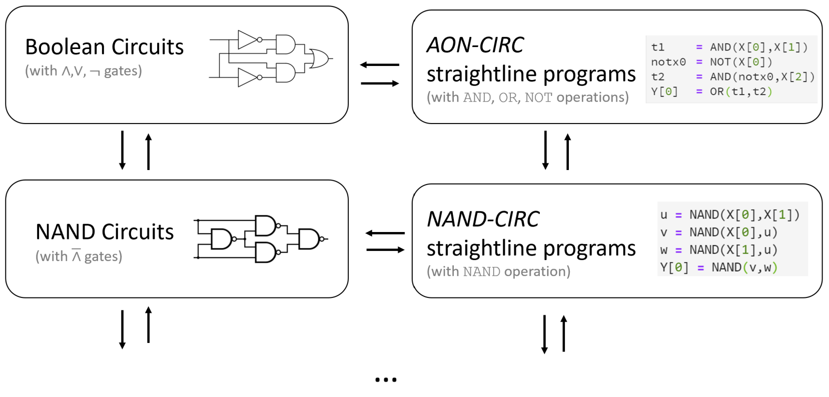
We can define algorithms informally as follows:

Informal definition of an algorithm: An *algorithm* is a set of instructions for how to compute an output from an input by following a sequence of "elementary steps".

An algorithm A *computes* a function F if for every input x, if we follow the instructions of A on the input x, we obtain the output $F(x)$.

In this chapter we will make this informal definition precise using the model of **Boolean Circuits**. We will show that Boolean Circuits are equivalent in power to **straight line programs** that are written in "ultra simple" programming languages that do not even have loops. We will also see that the particular choice of **elementary operations** is immaterial and many different choices yield models with equivalent power (see Fig. 3.6). However, it will take us some time to get there.

We will start by discussing what are "elementary operations" and how we map a description of an algorithm into an actual physical process that produces an output from an input in the real world.



```
t1    = AND(X[0],X[1])
notx0 = NOT(X[0])
t2    = AND(notx0,X[2])
Y[0]  = OR(t1,t2)
```

```
u = NAND(X[0],X[1])
v = NAND(X[0],u)
w = NAND(X[1],u)
Y[0] = NAND(v,w)
```

Figure 3.6: An overview of the computational models defined in this chapter. We will show several equivalent ways to represent a recipe for performing a finite computation. Specifically we will show that we can model such a computation using either a *Boolean circuit* or a *straight line program*, and these two representations are equivalent to one another. We will also show that we can choose as our basic operations either the set $\{AND, OR, NOT\}$ or the set $\{NAND\}$ and these two choices are equivalent in power. By making the choice of whether to use circuits or programs, and whether to use $\{AND, OR, NOT\}$ or $\{NAND\}$ we obtain four equivalent ways of modeling finite computation. Moreover, there are many other choices of sets of basic operations that are equivalent in power.

3.2 COMPUTING USING AND, OR, AND NOT.

An algorithm breaks down a *complex* calculation into a series of *simpler* steps. These steps can be executed in a variety of different ways, including:

- Writing down symbols on a piece of paper.
- Modifying the current flowing on electrical wires.
- Binding a protein to a strand of DNA.
- Responding to a stimulus by a member of a collection (e.g., a bee in a colony, a trader in a market).

To formally define algorithms, let us try to "err on the side of simplicity" and model our "basic steps" as truly minimal. For example, here are some very simple functions:

- $OR : \{0,1\}^2 \to \{0,1\}$ defined as

$$OR(a,b) = \begin{cases} 0 & a = b = 0 \\ 1 & \text{otherwise} \end{cases} \tag{3.1}$$

- $AND : \{0,1\}^2 \to \{0,1\}$ defined as

$$AND(a,b) = \begin{cases} 1 & a = b = 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

- $NOT : \{0,1\} \rightarrow \{0,1\}$ defined as

$$NOT(a) = \begin{cases} 0 & a = 1 \\ 1 & a = 0 \end{cases} \tag{3.3}$$

The functions *AND*, *OR* and *NOT*, are the basic logical operators used in logic and many computer system. In the context of logic, it is common to use the notation $a \wedge b$ for $AND(a,b)$, $a \vee b$ for $OR(a,b)$ and $\overline{a}$ and $\neg a$ for $NOT(a)$, and we will use this notation as well.

Each one of the functions AND, OR, NOT takes either one or two single bits as input, and produces a single bit as output. Clearly, it cannot get much more basic than that. However, the power of computation comes from *composing* such simple building blocks together.

■ **Example 3.1 — Majority from AND,OR and NOT.** Consider the function $MAJ : \{0,1\}^3 \rightarrow \{0,1\}$ that is defined as follows:

$$MAJ(x) = \begin{cases} 1 & x_0 + x_1 + x_2 \geq 2 \\ 0 & \text{otherwise} \end{cases} . \tag{3.4}$$

That is, for every $x \in \{0,1\}^3$, $MAJ(x) = 1$ if and only if the majority (i.e., at least two out of the three) of x's elements are equal to 1. Can you come up with a formula involving *AND*, *OR* and *NOT* to compute *MAJ*? (It would be useful for you to pause at this point and work out the formula for yourself. As a hint, although the *NOT* operator is needed to compute some functions, you will not need to use it to compute *MAJ*.)

Let us first try to rephrase $MAJ(x)$ in words: "$MAJ(x) = 1$ if and only if there exists some pair of distinct elements i, j such that both x_i and x_j are equal to 1." In other words it means that $MAJ(x) = 1$ iff *either* both $x_0 = 1$ *and* $x_1 = 1$, *or* both $x_1 = 1$ *and* $x_2 = 1$, *or* both $x_0 = 1$ *and* $x_2 = 1$. Since the *OR* of three conditions c_0, c_1, c_2 can be written as $OR(c_0, OR(c_1, c_2))$, we can now translate this into a formula as follows:

$$MAJ(x_0,x_1,x_2) = OR\left(AND(x_0,x_1) \ , \ OR(AND(x_1,x_2) \ , \ AND(x_0,x_2)) \right) \ . \tag{3.5}$$

Recall that we can also write $a \vee b$ for $OR(a,b)$ and $a \wedge b$ for $AND(a,b)$. With this notation, (3.5) can also be written as

$$MAJ(x_0,x_1,x_2) = ((x_0 \wedge x_1) \vee (x_1 \wedge x_2)) \vee (x_0 \wedge x_3) \ . \tag{3.6}$$

We can also write (3.5) in a "programming language" format, expressing it as a set of instructions for computing *MAJ* given the basic operations *AND*, *OR*, *NOT*:

```
def MAJ(X[0],X[1],X[2]):
    firstpair  = AND(X[0],X[1])
    secondpair = AND(X[1],X[2])
    thirdpair  = AND(X[0],X[2])
    temp       = OR(secondpair,thirdpair)
    return OR(firstpair,temp)
```

3.2.1 Some properties of AND and OR

Like standard addition and multiplication, the functions *AND* and *OR* satisfy the properties of *commutativity*: $a \vee b = b \vee a$ and $a \wedge b = b \wedge a$ and *associativity*: $(a \vee b) \vee c = a \vee (b \vee c)$ and $(a \wedge b) \wedge c = a \wedge (b \wedge c)$. As in the case of addition and multiplication, we often drop the parenthesis and write $a \vee b \vee c \vee d$ for $((a \vee b) \vee c) \vee d$, and similarly OR's and AND's of more terms. They also satisfy a variant of the distributive law:

Solved Exercise 3.1 — Distributive law for AND and OR. Prove that for every $a, b, c \in \{0,1\}$, $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.

■

Solution:

We can prove this by enumerating over all the 8 possible values for $a, b, c \in \{0,1\}$ but it also follows from the standard distributive law. Suppose that we identify any positive integer with "true" and the value zero with "false". Then for every numbers $u, v \in \mathbb{N}$, $u + v$ is positive if and only if $u \vee v$ is true and $u \cdot v$ is positive if and only if $u \wedge v$ is true. This means that for every $a, b, c \in \{0,1\}$, the expression $a \wedge (b \vee c)$ is true if and only if $a \cdot (b + c)$ is positive, and the expression $(a \wedge b) \vee (a \wedge c)$ is true if and only if $a \cdot b + a \cdot c$ is positive, But by the standard distributive law $a \cdot (b + c) = a \cdot b + a \cdot c$ and hence the former expression is true if and only if the latter one is.

■

3.2.2 Extended example: Computing *XOR* from *AND*, *OR*, and *NOT*

Let us see how we can obtain a different function from the same building blocks. Define $XOR : \{0,1\}^2 \rightarrow \{0,1\}$ to be the function $XOR(a, b) = a + b \mod 2$. That is, $XOR(0,0) = XOR(1,1) = 0$ and $XOR(1,0) = XOR(0,1) = 1$. We claim that we can construct *XOR* using only *AND*, *OR*, and *NOT*.

As usual, it is a good exercise to try to work out the algorithm for *XOR* using *AND*, *OR* and *NOT* on your own before reading further.

The following algorithm computes *XOR* using *AND*, *OR*, and *NOT*:

Algorithm 3.2 — XOR **from** $AND/OR/NOT$.

```
Input: $a, b \in \{0,1\}$.
Output: $XOR(a,b)$
1: $w1 \leftarrow AND(a,b)$
2: $w2 \leftarrow NOT(w1)$
3: $w3 \leftarrow OR(a,b)$
4: return $AND(w2,w3)$
```

Lemma 3.3 For every $a, b \in \{0,1\}$, on input a, b, Algorithm 3.2 outputs $a + b \mod 2$.

Proof. For every a, b, $XOR(a,b) = 1$ if and only if a is *different* from b. On input $a, b \in \{0,1\}$, Algorithm 3.2 outputs $AND(w2,w3)$ where $w2 = NOT(AND(a,b))$ and $w3 = OR(a,b)$.

- If $a = b = 0$ then $w3 = OR(a,b) = 0$ and so the output will be 0.
- If $a = b = 1$ then $AND(a,b) = 1$ and so $w2 = NOT(AND(a,b)) = 0$ and the output will be 0.
- If $a = 1$ and $b = 0$ (or vice versa) then both $w3 = OR(a,b) = 1$ and $w1 = AND(a,b) = 0$, in which case the algorithm will output $OR(NOT(w1), w3) = 1$.

■

We can also express Algorithm 3.2 using a programming language. Specifically, the following is a *Python* program that computes the *XOR* function:

```
def AND(a,b): return a*b
def OR(a,b):  return 1-(1-a)*(1-b)
def NOT(a):   return 1-a

def XOR(a,b):
    w1 = AND(a,b)
    w2 = NOT(w1)
    w3 = OR(a,b)
    return AND(w2,w3)

# Test out the code
```

```
print([f"XOR({a},{b})={XOR(a,b)}" for a in [0,1] for b in
 ↪  [0,1]])
# ['XOR(0,0)=0', 'XOR(0,1)=1', 'XOR(1,0)=1', 'XOR(1,1)=0']
```

Solved Exercise 3.2 — Compute XOR on three bits of input. Let $XOR_3 : \{0,1\}^3 \rightarrow \{0,1\}$ be the function defined as $XOR_3(a,b,c) = a + b + c \mod 2$. That is, $XOR_3(a,b,c) = 1$ if $a+b+c$ is odd, and $XOR_3(a,b,c) = 0$ otherwise. Show that you can compute XOR_3 using AND, OR, and NOT. You can express it as a formula, use a programming language such as Python, or use a Boolean circuit.

■

Solution:

Addition modulo two satisfies the same properties of *associativity* ($(a+b)+c = (a+b)+c$) and *commutativity* ($a+b = b+a$) as standard addition. This means that, if we define $a \oplus b$ to equal $a+b \mod 2$, then

$$XOR_3(a,b,c) = (a \oplus b) \oplus c \tag{3.7}$$

or in other words

$$XOR_3(a,b,c) = XOR(XOR(a,b),c) \; . \tag{3.8}$$

Since we know how to compute XOR using AND, OR, and NOT, we can compose this to compute XOR_3 using the same building blocks. In Python this corresponds to the following program:

```
def XOR3(a,b,c):
    w1 = AND(a,b)
    w2 = NOT(w1)
    w3 = OR(a,b)
    w4 = AND(w2,w3)
    w5 = AND(w4,c)
    w6 = NOT(w5)
    w7 = OR(w4,c)
    return AND(w6,w7)

# Let's test this out
print([f"XOR3({a},{b},{c})={XOR3(a,b,c)}" for a in [0,1]
 ↪  for b in [0,1] for c in [0,1]])
# ['XOR3(0,0,0)=0', 'XOR3(0,0,1)=1', 'XOR3(0,1,0)=1',
 ↪  'XOR3(0,1,1)=0', 'XOR3(1,0,0)=1', 'XOR3(1,0,1)=0',
 ↪  'XOR3(1,1,0)=0', 'XOR3(1,1,1)=1']
```

■

P

Try to generalize the above examples to obtain a way to compute $XOR_n : \{0,1\}^n \rightarrow \{0,1\}$ for every n using at most $4n$ basic steps involving applications of a function in $\{AND, OR, NOT\}$ to outputs or previously computed values.

3.2.3 Informally defining "basic operations" and "algorithms"

We have seen that we can obtain at least some examples of interesting functions by composing together applications of *AND*, *OR*, and *NOT*. This suggests that we can use *AND*, *OR*, and *NOT* as our "basic operations", hence obtaining the following definition of an "algorithm":

Semi-formal definition of an algorithm: An *algorithm* consists of a sequence of steps of the form "compute a new value by applying *AND*, *OR*, or *NOT* to previously computed values".

An algorithm A *computes* a function F if for every input x to F, if we feed x as input to the algorithm, the value computed in its last step is $F(x)$.

There are several concerns that are raised by this definition:

1. First and foremost, this definition is indeed too informal. We do not specify exactly what each step does, nor what it means to "feed x as input".

2. Second, the choice of *AND*, *OR* or *NOT* seems rather arbitrary. Why not *XOR* and *MAJ*? Why not allow operations like addition and multiplication? What about any other logical constructions such `if/then` or `while`?

3. Third, do we even know that this definition has anything to do with actual computing? If someone gave us a description of such an algorithm, could we use it to actually compute the function in the real world?

P

These concerns will to a large extent guide us in the upcoming chapters. Thus you would be well advised to re-read the above informal definition and see what you think about these issues.

A large part of this book will be devoted to addressing the above issues. We will see that:

1. We can make the definition of an algorithm fully formal, and so give a precise mathematical meaning to statements such as "Algorithm A computes function f".

2. While the choice of *AND*/*OR*/*NOT* is arbitrary, and we could just as well chose some other functions, we will also see this choice does not matter much. We will see that the we would obtain the same computational power if we used instead for addition and multiplication, and essentially every other operation that could be reasonably thought of as a basic step.

3. It turns out that we can and do compute such "*AND*/*OR*/*NOT* based algorithms" in the real world. First of all, such an algorithm is clearly well specified, and so can be executed by a human with a pen and paper. Second, there are a variety of ways to *mechanize* this computation. We've already seen that we can write Python code that corresponds to following such a list of instructions. But in fact we can directly implement operations such as *AND*, *OR*, and *NOT* via electronic signals using components known as *transistors*. This is how modern electronic computers operate.

In the remainder of this chapter, and the rest of this book, we will begin to answer some of these questions. We will see more examples of the power of simple operations to compute more complex operations including addition, multiplication, sorting and more. We will also discuss how to *physically implement* simple operations such as *AND*, *OR* and *NOT* using a variety of technologies.

3.3 BOOLEAN CIRCUITS

Boolean circuits provide a precise notion of "composing basic operations together". A Boolean circuit (see Fig. 3.9) is composed of *gates* and *inputs* that are connected by *wires*. The *wires* carry a signal that represents either the value 0 or 1. Each gate corresponds to either the *OR*, *AND*, or *NOT* operation. An *OR gate* has two incoming wires, and one or more outgoing wires. If these two incoming wires carry the signals a and b (for $a, b \in \{0,1\}$), then the signal on the outgoing wires will be $OR(a,b)$. *AND* and *NOT* gates are defined similarly. The *inputs* have only outgoing wires. If we set a certain input to a value $a \in \{0,1\}$, then this value is propagated on all the wires outgoing from it. We also designate some gates as *output gates*, and their value corresponds to the result of evaluating the circuit. For example, Fig. 3.8 gives such a circuit for the *XOR* function, following Section 3.2.2. We evaluate an n-input Boolean circuit C on an input $x \in \{0,1\}^n$ by placing the bits of x on the inputs, and then propagating the values on the wires until we reach an output, see Fig. 3.9.

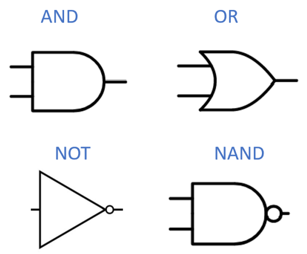


Figure 3.7: Standard symbols for the logical operations or "gates" of *AND*, *OR*, *NOT*, as well as the operation *NAND* discussed in Section 3.5.

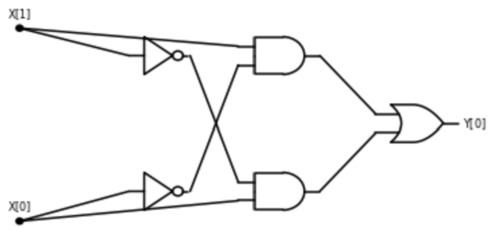


Figure 3.8: A circuit with *AND*, *OR* and *NOT* gates for computing the *XOR* function.

Remark 3.4 — Physical realization of Boolean circuits. Boolean circuits are a *mathematical model* that does not necessarily correspond to a physical object, but they can be implemented physically. In physical implementation of circuits, the signal is often implemented by electric potential or *voltage* on a wire, where for example voltage above a certain level is interpreted as a logical value of 1, and below a certain level is interpreted as a logical value of 0. Section 3.4 discusses physical implementation of Boolean circuits (with examples including using electrical signals such as in silicon-based circuits, but also biological and mechanical implementations as well).

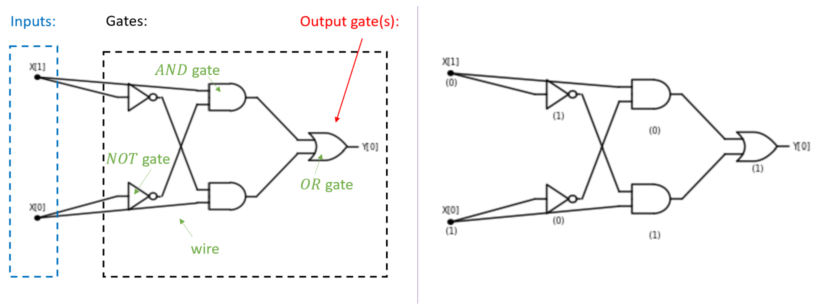


Figure 3.9: A *Boolean Circuit* consists of *gates* that are are connected by *wires* to one another and the *inputs*. The left side depicts a circuit with 2 inputs and 5 gates, one of which is designated the output gate. The right side depicts the evaluation of this circuit on the input $x \in \{0,1\}^2$ with $x_0 = 1$ and $x_1 = 0$. The value of every gate is obtained by applying the corresponding function (*AND*, *OR*, or *NOT*) to values on the wire(s) that enter it. The output of the circuit on a given input is the value of the output gate(s). In this case, the circuit computes the *XOR* function and hence it outputs 1 on the input 10.

Solved Exercise 3.3 — All equal function. Define $ALLEQ : \{0,1\}^4 \to \{0,1\}$ to be the function that on input $x \in \{0,1\}^4$ outputs 1 if and only if $x_0 = x_1 = x_2 = x_3$. Give a Boolean circuit for computing *ALLEQ*.

Solution:

Another way to describe the function *ALLEQ* is that it outputs 1 on an input $x \in \{0,1\}^4$ if and only if $x = 0^4$ or $x = 1^4$. We can phrase the condition $x = 1^4$ as $x_0 \wedge x_1 \wedge x_2 \wedge x_3$ which can be computed using three AND gates. Similarly we can phrase the condition $x = 0^4$ as $\overline{x}_0 \wedge \overline{x}_1 \wedge \overline{x}_2 \wedge \overline{x}_3$ which can be computed using four NOT gates and three AND gates. The output of *ALLEQ* is the OR of these two conditions, which results in the circuit of 4 NOT gates, 6 AND gates, and one OR gate presented in Fig. 3.10.

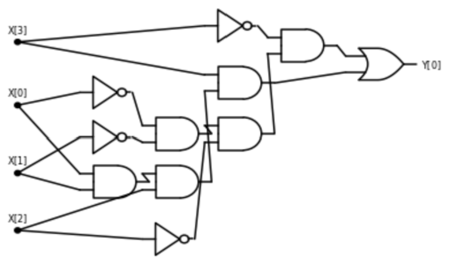


Figure 3.10: A Boolean circuit for computing the *all equal* function $ALLEQ : \{0,1\}^4 \to \{0,1\}$ that outputs 1 on $x \in \{0,1\}^4$ if and only if $x_0 = x_1 = x_2 = x_3$.

3.3.1 Boolean circuits: a formal definition

We defined Boolean circuits informally as obtained by connecting *AND*, *OR*, and *NOT* gates via wires so as to produce an output from an input. However, to be able to prove theorems about the existence or non-existence of Boolean circuits for computing various functions we need to:

1. Formally define a Boolean circuit as a mathematical object.

2. Formally define what it means for a circuit C to compute a function f.

We now proceed to do so. We will define a Boolean circuit as a labeled *Directed Acyclic Graph* (*DAG*). The *vertices* of the graph correspond to the gates and inputs of the circuit, and the *edges* of the graph correspond to the wires. A wire from an input or gate u to a gate v in the circuit corresponds to a directed edge between the corresponding vertices. The inputs are vertices with no incoming edges, while each gate has the appropriate number of incoming edges based on the function it computes. (That is, *AND* and *OR* gates have two in-neighbors, while *NOT* gates have one in-neighbor.) The formal definition is as follows (see also Fig. 3.11):

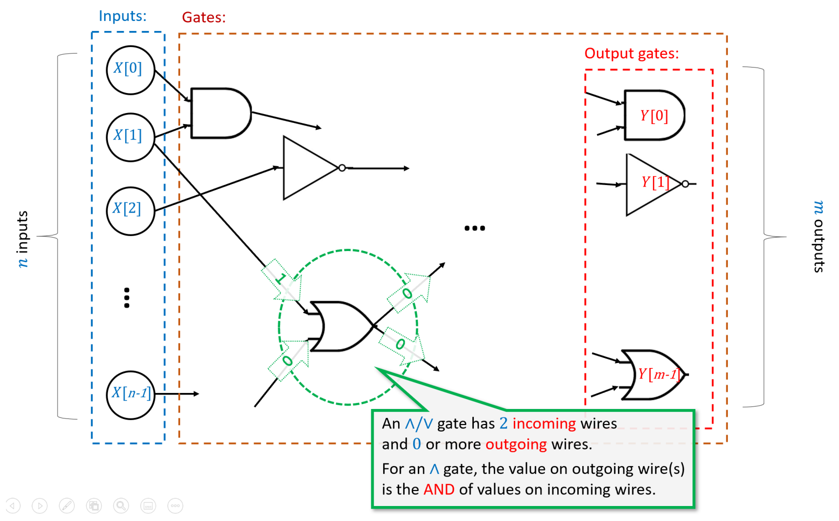

Inputs:
Gates:
Output gates:
X[0]
X[1]
X[2]
X[n-1]
Y[0]
Y[1]
Y[m-1]
n inputs
m outputs
1
0
0
0
An ∧/∨ gate has 2 incoming wires and 0 or more outgoing wires.
For an ∧ gate, the value on outgoing wire(s) is the AND of values on incoming wires.

Figure 3.11: A *Boolean Circuit* is a labeled directed acyclic graph (DAG). It has n *input* vertices, which are marked with `X[0]`,…, `X[`$n-1$`]` and have no incoming edges, and the rest of the vertices are *gates*. An *AND*, *OR*, or *NOT* gate has two or one incoming edges. If the circuit has m outputs, then m of the gates are known as *outputs* and are marked with `Y[0]`,…,`Y[`$m-1$`]`. When we evaluate a circuit C on an input $x \in \{0,1\}^n$, we start by setting the value of the input vertices to $x_0, \ldots, x_{n-1}$ and then propagate the values, assigning to each gate g the result of applying the operation of g to the values of g's in-neighbors. The output of the circuit is the value assigned to the output gates.

Definition 3.5 — Boolean Circuits. Let n, m, s be positive integers with $s \geq m$. A *Boolean circuit* with n inputs, m outputs, and s gates, is a labeled directed acyclic graph (DAG) $G = (V, E)$ with $s+n$ vertices satisfying the following properties:

- Exactly n of the vertices have no in-neighbors. These vertices are known as *inputs* and are labeled with the n labels `X[0]`, …, `X[`$n-1$`]`.

- The other s vertices are known as *gates*. Each gate is labeled with $\wedge$, $\vee$ or $\neg$. Gates labeled with $\wedge$ (*AND*) or $\vee$ (*OR*) have two in-neighbors. Gates labeled with $\neg$ (*NOT*) have one in-neighbor. We will allow parallel edges (and so for example an *AND* gate can have both its in-neighbors be the same vertex).
- Exactly m of the gates are also labeled with the m labels `Y[`0`]`, ..., `Y[`$m-1$`]` (in addition to their label $\wedge/\vee/\neg$). These are known as *outputs*.

P

This is a non-trivial mathematical definition, so it is worth taking the time to read it slowly and carefully. As in all mathematical definitions, we are using a known mathematical object — a directed acyclic graph (DAG) — to define a new object, a Boolean circuit. This might be a good time to review some of the basic properties of DAGs and in particular the fact that they can be *topologically sorted*, see Section 1.6.

If C is a circuit with n inputs and m outputs, and $x \in \{0,1\}^n$, then we can compute the output of C on the input x in the natural way: assign the input vertices `X[`0`]`, ..., `X[`$n-1$`]` the values $x_0, \ldots, x_{n-1}$, apply each gate on the values of its in-neighbors, and then output the values that correspond to the output vertices. Formally, this is defined as follows:

Definition 3.6 — Computing a function via a Boolean circuit. Let C be a Boolean circuit with n inputs and m outputs. For every $x \in \{0,1\}^n$, the *output* of C on the input x, denoted by $C(x)$, is defined as the result of the following process:

We let $h : V \to \mathbb{N}$ be the *minimal layering* of C (aka *topological sorting*, see Theorem 1.26). We let L be the maximum layer of h, and for $\ell = 0, 1, \ldots, L$ we do the following:

- For every v in the ℓ-th layer (i.e., v such that $h(v) = \ell$) do:
  - If v is an input vertex labeled with `X[`i`]` for some $i \in [n]$, then we assign to v the value x_i.
  - If v is a gate vertex labeled with $\wedge$ and with two in-neighbors u, w then we assign to v the *AND* of the values assigned to u and w. (Since u and w are in-neighbors of v, they are in lower layer than v, and hence their values have already been assigned.)

  - If v is a gate vertex labeled with $\vee$ and with two in-neighbors u, w then we assign to v the OR of the values assigned to u and w.
  - If v is a gate vertex labeled with $\neg$ and with one in-neighbor u then we assign to v the negation of the value assigned to u.

- The result of this process is the value $y \in \{0,1\}^m$ such that for every $j \in [m]$, y_j is the value assigned to the vertex with label `Y[`j`]`.

Let $f : \{0,1\}^n \rightarrow \{0,1\}^m$. We say that the circuit C *computes* f if for every $x \in \{0,1\}^n$, $C(x) = f(x)$.

3.3.2 Equivalence of circuits and straight-line programs

We have seen two ways to describe how to compute a function f using *AND*, *OR* and *NOT*:

- A *Boolean circuit*, defined in Definition 3.5, computes f by connecting via wires *AND*, *OR*, and *NOT* gates to the inputs.
- We can also describe such a computation using a *straight-line program* that has lines of the form `foo = AND(bar,blah)`, `foo = OR(bar,blah)` and `foo = NOT(bar)` where `foo`, `bar` and `blah` are variable names. (We call this a *straight-line program* since it contains no loops or branching (e.g., if/then) statements.)

We now formally define the AON-CIRC programming language ("AON" stands for *AND*/*OR*/*NOT*) which has the above operations, and show that it is equivalent to Boolean circuits.

Definition 3.7 — AON-CIRC Programming language. An *AON-CIRC program* is a string of lines of the form `foo = AND(bar,blah)`, `foo = OR(bar,blah)` and `foo = NOT(bar)` where `foo`, `bar` and `blah` are variable names. [1] Variables of the form `X[`i`]` are known as *input* variables, and variables of the form `Y[`j`]` are known as *output* variables. In every line, the variables on the righthand side of the assignment operators must either be input variables or variables that have already been assigned a value before.

If an AON-CIRC program P has input variables `X[0]`,...,`X[`$n-1$`]` and output variables `Y[0]`,..., `Y[`$m-1$`]` then for every $x \in \{0,1\}^n$, we define the *output of P on input x*, denoted by $P(x)$, to be the string $y \in \{0,1\}^m$ corresponding to the values of the output variables `Y[0]` ,..., `Y[`$m-1$`]` in the execution of P where we initialize the input variables `X[0]`,...,`X[`$n-1$`]` to the values $x_0, \ldots, x_{n-1}$.

We say that such an AON-CIRC program P *computes* a function $f : \{0,1\}^n \to \{0,1\}^m$ if $P(x) = f(x)$ for every $x \in \{0,1\}^n$.

AON-CIRC is not a practical programming language: it was designed for pedagogical purposes only, as a way to model computation as composition of *AND*, *OR*, and *NOT*. However, AON-CIRC can still be easily implemented on a computer. The following solved exercise gives an example of an AON-CIRC program.

[1] We follow the common programming languages convention of using names such as `foo`, `bar`, `baz`, `blah` as stand-ins for generic identifiers. A variable identifier in our programming language can be any combination of letters, numbers, underscores, and brackets. The appendix contains a full formal specification of our programming language.

Solved Exercise 3.4 Consider the following function $CMP : \{0,1\}^4 \to \{0,1\}$ that on four input bits $a, b, c, d \in \{0,1\}$, outputs 1 iff the number represented by (a, b) is larger than the number represented by (c, d). That is $CMP(a, b, c, d) = 1$ iff $2a + b > 2c + d$.

Write an AON-CIRC program to compute *CMP*.

■

Solution:

Writing such a program is tedious but not truly hard. To compare two numbers we first compare their most significant digit, and then go down to the next digit and so on and so forth. In this case where the numbers have just two binary digits, these comparisons are particularly simple. The number represented by (a, b) is larger than the number represented by (c, d) if and only if one of the following conditions happens:

1. The most significant bit a of (a, b) is larger than the most significant bit c of (c, d).

   or

2. The two most significant bits a and c are equal, but $b > d$.

Another way to express the same condition is the following: the number (a, b) is larger than (c, d) iff $a > c$ **OR** $(($**NOT** $(c < a))$ **AND** $b > d)$.

For binary digits α, β, the condition $\alpha > \beta$ is simply that $\alpha = 1$ and $\beta = 0$ or $AND(\alpha, NOT(\beta)) = 1$. Together these observations can be used to give the following AON-CIRC program to compute *CMP*:

```
temp_1 = NOT(X[2])
temp_2 = OR(X[0],temp_1)
temp_3 = NOT(X[0])
temp_4 = OR(X[2],temp_3)
temp_5 = NOT(X[3])
```

```
temp_6 = OR(X[1],temp_5)
temp_7 = AND(temp_6,temp_4)
Y[0] = OR(temp_2,temp_7)
```

We can also present this 8-line program as a circuit with 8 gates, see Fig. 3.12.

■

It turns out that AON-CIRC programs and Boolean circuits have exactly the same power:

Theorem 3.8 — Equivalence of circuits and straight-line programs. Let $f : \{0,1\}^n \rightarrow \{0,1\}^m$ and $s \geq m$ be some number. Then f is computable by a Boolean circuit of s gates if and only if f is computable by an AON-CIRC program of s lines.

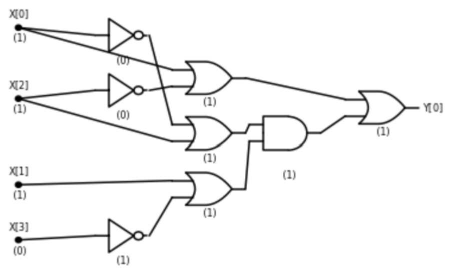


Figure 3.12: A circuit for computing the *CMP* function. The evaluation of this circuit on $(1, 1, 1, 0)$ yields the output 1, since the number 3 (represented in binary as 11) is larger than the number 2 (represented in binary as 10).

Proof Idea:

The idea is simple - AON-CIRC programs and Boolean circuits are just different ways of describing the exact same computational process. For example, an *AND* gate in a Boolean circuit corresponding to computing the *AND* of two previously-computed values. In a AON-CIRC program this will correspond to the line that stores in a variable the `AND` of two previously-computed variables.

⋆

P

This proof of Theorem 3.8 is simple at heart, but all the details it contains can make it a little cumbersome to read. You might be better off trying to work it out yourself before reading it. Our GitHub repository contains a "proof by Python" of Theorem 3.8: implementation of functions `circuit2prog` and `prog2circuits` mapping Boolean circuits to AON-CIRC programs and vice versa.

Proof of Theorem 3.8. Let $f : \{0,1\}^n \rightarrow \{0,1\}^m$. Since the theorem is an "if and only if" statement, to prove it we need to show both directions: translating an AON-CIRC program that computes f into a circuit that computes f, and translating a circuit that computes f into an AON-CIRC program that does so.

We start with the first direction. Let P be an s line AON-CIRC that computes f. We define a circuit C as follows: the circuit will have n inputs and s gates. For every $i \in [s]$, if the i-th line has the form `foo = AND(bar,blah)` then the i-th gate in the circuit will be an AND gate that is connected to gates j and k where j and k correspond to the last lines before i where the variables `bar` and `blah` (respectively) where written to. (For example, if $i = 57$ and the last line `bar` was written to is 35 and the last line `blah` was written to is 17 then the two in-neighbors of gate 57 will be gates 35 and 17.) If either `bar` or `blah` is an input variable then we connect the gate to the corresponding input vertex instead. If `foo` is an output variable of the form `Y[`j`]` then we add the same label to the corresponding gate to mark it as an output gate. We do the analogous operations if the i-th line involves an `OR` or a `NOT` operation (except that we use the corresponding *OR* or *NOT* gate, and in the latter case have only one in-neighbor instead of two). For every input $x \in \{0,1\}^n$, if we run the program P on x, then the value written that is computed in the i-th line is exactly the value that will be assigned to the i-th gate if we evaluate the circuit C on x. Hence $C(x) = P(x)$ for every $x \in \{0,1\}^n$.

For the other direction, let C be a circuit of s gates and n inputs that computes the function f. We sort the gates according to a topological order and write them as $v_0, \ldots, v_{s-1}$. We now can create a program P of s lines as follows. For every $i \in [s]$, if v_i is an AND gate with in-neighbors v_j, v_k then we will add a line to P of the form `temp_`i `= AND(temp_`j`,temp_`k`)`, unless one of the vertices is an input vertex or an output gate, in which case we change this to the form `X[.]` or `Y[.]` appropriately. Because we work in topological ordering, we are guaranteed that the in-neighbors v_j and v_k correspond to variables that have already been assigned a value. We do the same for OR and NOT gate. Once again, one can verify that for every input x, the value $P(x)$ will equal $C(x)$ and hence the program computes the same function as the circuit.

■

3.4 PHYSICAL IMPLEMENTATIONS OF COMPUTING DEVICES (DIGRESSION)

Computation is an abstract notion that is distinct from its physical *implementations*. While most modern computing devices are obtained by mapping logical gates to semi-conductor based transistors, over history people have computed using a huge variety of mechanisms, including mechanical systems, gas and liquid (known as *fluidics*), biological and chemical processes, and even living creatures (e.g., see Fig. 3.14 or this video for how crabs or slime mold can be used to do computations).

In this section we will review some of these implementations, both so you can get an appreciation of how it is possible to directly translate Boolean circuits to the physical world, without going through the entire stack of architecture, operating systems, and compilers, as well as to emphasize that silicon-based processors are by no means the only

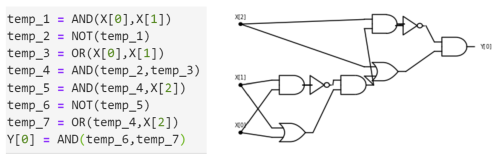

```
temp_1 = AND(X[0],X[1])
temp_2 = NOT(temp_1)
temp_3 = OR(X[0],X[1])
temp_4 = AND(temp_2,temp_3)
temp_5 = AND(temp_4,X[2])
temp_6 = NOT(temp_5)
temp_7 = OR(temp_4,X[2])
Y[0] = AND(temp_6,temp_7)
```

Figure 3.13: Two equivalent descriptions of the same AND/OR/NOT computation as both an AON program and a Boolan circuit.

way to perform computation. Indeed, as we will see in Chapter 22, a very exciting recent line of works involves using different media for computation that would allow us to take advantage of *quantum mechanical effects* to enable different types of algorithms.

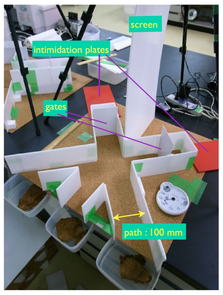


Figure 3.14: Crab-based logic gates from the paper "Robust soldier-crab ball gate" by Gunji, Nishiyama and Adamatzky. This is an example of an AND gate that relies on the tendency of two swarms of crabs arriving from different directions to combine to a single swarm that continues in the average of the directions.

3.4.1 Transistors

A *transistor* can be thought of as an electric circuit with two inputs, known as *source* and *gate* and an output, known as the *sink*. The gate controls whether current flows from the source to the sink. In a *standard transistor*, if the gate is "ON" then current can flow from the source to the sink and if it is "OFF" then it can't. In a *complementary transistor* this is reversed: if the gate is "OFF" then current can flow from the source to the sink and if it is "ON" then it can't.

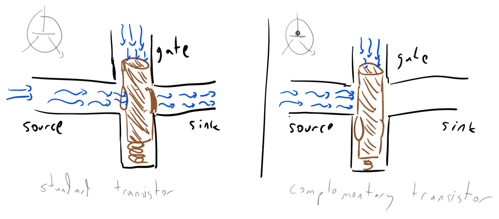


Figure 3.15: We can implement the logic of transistors using water. The water pressure from the gate closes or opens a faucet between the source and the sink.

There are several ways to implement the logic of a transistor. For example, we can use faucets to implement it using water pressure (e.g. Fig. 3.15). This might seem as merely a curiosity but there is a field known as fluidics concerned with implementing logical operations using liquids or gasses. Some of the motivations include operating in extreme environmental conditions such as in space or a battlefield, where standard electronic equipment would not survive.

The standard implementations of transistors use electrical current. One of the original implementations used *vacuum tubes*. As its name implies, a vacuum tube is a tube containing nothing (i.e., a vacuum) and where a priori electrons could freely flow from source (a wire) to the sink (a plate). However, there is a gate (a grid) between the two, where modulating its voltage can block the flow of electrons.

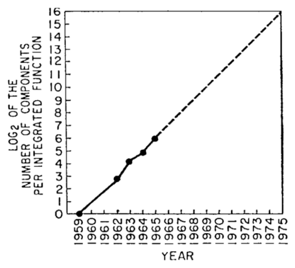


Figure 3.16: The number of transistors per integrated circuits from 1959 till 1965 and a prediction that exponential growth will continue at least another decade. Figure taken from "Cramming More Components onto Integrated Circuits", Gordon Moore, 1965

Early vacuum tubes were roughly the size of lightbulbs (and looked very much like them too). In the 1950's they were supplanted by *transistors*, which implement the same logic using *semiconductors* which are materials that normally do not conduct electricity but whose conductivity can be modified and controlled by inserting impurities ("doping") and an external electric field (this is known as the *field effect*). In the 1960's computers were started to be implemented using *integrated circuits* which enabled much greater density. In 1965, Gordon Moore predicted that the number of transistors per integrated circuit would double every year (see Fig. 3.16), and that this would lead to "such wonders as home computers —or at least terminals connected to a central computer— automatic controls for automobiles, and personal portable communications equipment". Since then, (adjusted versions of) this so-called "Moore's law" have been running strong, though exponential growth cannot be sustained forever, and some physical limitations are already becoming apparent.

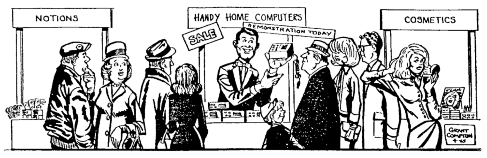


Figure 3.17: Cartoon from Gordon Moore's article "predicting" the implications of radically improving transistor density.

3.4.2 Logical gates from transistors

We can use transistors to implement various Boolean functions such as *AND*, *OR*, and *NOT*. For each a two-input gate $G : \{0,1\}^2 \rightarrow \{0,1\}$, such an implementation would be a system with two input wires x, y and one output wire z, such that if we identify high voltage with "1" and low voltage with "0", then the wire z will equal to "1" if and only if applying G to the values of the wires x and y is 1 (see Fig. 3.19 and Fig. 3.20). This means that there exists a AND/OR/NOT circuit to compute a function $g : \{0,1\}^n \rightarrow \{0,1\}^m$, then we can compute g in the physical world using transistors as well.

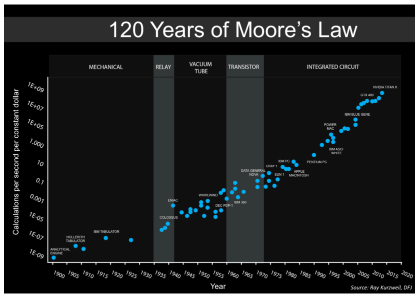


Figure 3.18: The exponential growth in computing power over the last 120 years. Graph by Steve Jurvetson, extending a prior graph of Ray Kurzweil.

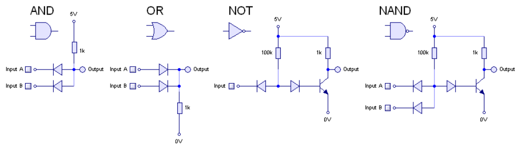


Figure 3.19: Implementing logical gates using transistors. Figure taken from Rory Mangles' website.

3.4.3 Biological computing

Computation can be based on biological or chemical systems. For example the *lac* operon produces the enzymes needed to digest lactose only if the conditions $x \wedge (\neg y)$ hold where x is "lactose is present" and y is "glucose is present". Researchers have managed to create transistors, and from them logic gates, based on DNA molecules (see also Fig. 3.21). One motivation for DNA computing is to achieve increased parallelism or storage density; another is to create "smart biological agents" that could perhaps be injected into bodies, replicate themselves, and fix or kill cells that were damaged by a disease such as cancer. Computing in biological systems is not restricted of course to DNA. Even larger systems such as flocks of birds can be considered as computational processes.

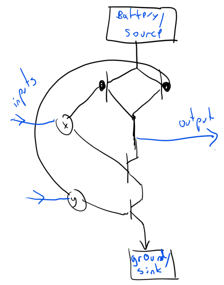


Figure 3.20: Implementing a NAND gate (see Section 3.5) using transistors.

3.4.4 Cellular automata and the game of life

Cellular automata is a model of a system composed of a sequence of *cells*, which of which can have a finite state. At each step, a cell updates its state based on the states of its *neighboring cells* and some simple rules. As we will discuss later in this book (see Section 7.4), cellular automata such as Conway's "Game of Life" can be used to simulate computation gates .

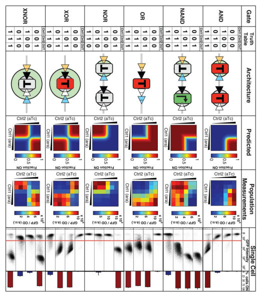


Figure 3.21: Performance of DNA-based logic gates. Figure taken from paper of Bonnet et al, Science, 2013.

3.4.5 Neural networks

One computation device that we all carry with us is our own *brain*. Brains have served humanity throughout history, doing computations that range from distinguishing prey from predators, through making scientific discoveries and artistic masterpieces, to composing witty 280 character messages. The exact working of the brain is still not fully understood, but one common mathematical model for it is a (very large) *neural network*.

A neural network can be thought of as a Boolean circuit that instead of *AND*/*OR*/*NOT* uses some other gates as the basic basis. For example, one particular basis we can use are *threshold gates*. For every vector $w = (w_0, \ldots, w_{k-1})$ of integers and integer t (some or all of whom could be negative), the *threshold function corresponding to* w, t is the function $T_{w,t} : \{0,1\}^k \rightarrow \{0,1\}$ that maps $x \in \{0,1\}^k$ to 1 if and only if $\sum_{i=0}^{k-1} w_i x_i \geq t$. For example, the threshold function $T_{w,t}$ corresponding to $w = (1,1,1,1,1)$ and $t = 3$ is simply the majority function MAJ_5 on $\{0,1\}^5$. Threshold gates can be thought of as an approximation for *neuron cells* that make up the core of human and animal brains. To a first approximation, a neuron has k inputs and a single output and the neurons "fires" or "turns on" its output when those signals pass some threshold.

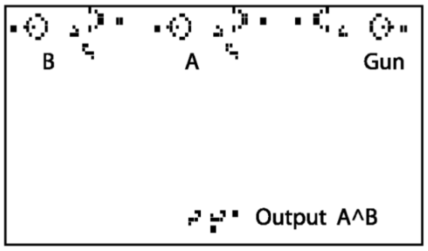


Figure 3.22: An AND gate using a "Game of Life" configuration. Figure taken from Jean-Philippe Rennard's paper.

Many machine learning algorithms use *artificial neural networks* whose purpose is not to imitate biology but rather to perform some computational tasks, and hence are not restricted to threshold or other biologically-inspired gates. Generally, a neural network is often described as operating on signals that are real numbers, rather than 0/1 values, and where the output of a gate on inputs $x_0, \ldots, x_{k-1}$ is obtained by applying $f(\sum_i w_i x_i)$ where $f : \mathbb{R} \rightarrow \mathbb{R}$ is an an activation function such as rectified linear unit (ReLU), Sigmoid, or many others (see Fig. 3.23). However, for the purposes of our discussion, all of the above are equivalent (see also Exercise 3.11). In particular we can reduce the setting of real inputs to binary inputs by representing a real number in the binary basis, and multiplying the weight of the bit corresponding to the i^{th} digit by 2^i.

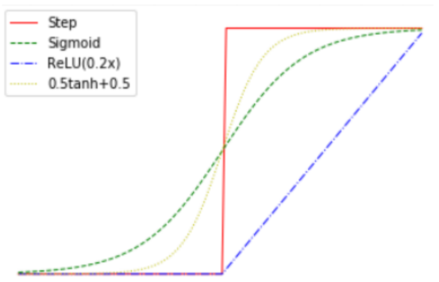


Figure 3.23: Common activation functions used in Neural Networks, including rectified linear units (ReLU), sigmoids, and hyperbolic tangent. All of those can be thought of as continuous approximations to simple the step function. All of these can be used to compute the NAND gate (see Exercise 3.11). This property enables neural networks to (approximately) compute any function that can be computed by a Boolean circuit.

3.4.6 A computer made from marbles and pipes

We can implement computation using many other physical media, without any electronic, biological, or chemical components. Many suggestions for *mechanical* computers have been put forward, going back at least to Gottfried Leibniz' computing machines from the 1670s and Charles Babbage's 1837 plan for a mechanical "Analytical Engine". As one example, Fig. 3.24 shows a simple implementation of a NAND (negation of AND, see Section 3.5) gate using marbles going through pipes. We represent a logical value in $\{0,1\}$ by a pair of pipes, such that there is a marble flowing through exactly one of the pipes. We call one of the pipes the "0 pipe" and the other the "1 pipe", and so the identity of the pipe containing the marble determines the logical value. A NAND gate corresponds to a mechanical object with two pairs of incoming pipes and one pair of outgoing pipes, such that for every $a, b \in \{0,1\}$, if two marble are rolling toward the object in the a pipe of the first pair and the b pipe of the second pair, then a marble will roll out of the object in the $NAND(a,b)$-pipe of the outgoing pair. In fact, there is even a commercially-available educational game that uses marbles as a basis of computing, see Fig. 3.26.

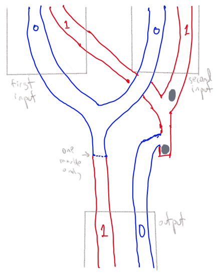


Figure 3.24: A physical implementation of a NAND gate using marbles. Each wire in a Boolean circuit is modeled by a pair of pipes representing the values 0 and 1 respectively, and hence a gate has four input pipes (two for each logical input) and two output pipes. If one of the input pipes representing the value 0 has a marble in it then that marble will flow to the output pipe representing the value 1. (The dashed line represent a gadget that will ensure that at most one marble is allowed to flow onward in the pipe.) If both the input pipes representing the value 1 have marbles in them, then the first marble will be stuck but the second one will flow onwards to the output pipe representing the value 0.

3.5 THE NAND FUNCTION

The *NAND* function is another simple function that is extremely useful for defining computation. It is the function mapping $\{0,1\}^2$ to $\{0,1\}$ defined by:

$$NAND(a,b) = \begin{cases} 0 & a=b=1 \\ 1 & \text{otherwise} \end{cases} . \tag{3.9}$$

As its name implies, *NAND* is the NOT of AND (i.e., $NAND(a,b) = NOT(AND(a,b))$), and so we can clearly compute *NAND* using *AND* and *NOT*. Interestingly, the opposite direction holds as well:

Theorem 3.9 — NAND computes AND,OR,NOT. We can compute *AND*, *OR*, and *NOT* by composing only the *NAND* function.

Proof. We start with the following observation. For every $a \in \{0,1\}$, $AND(a,a) = a$. Hence, $NAND(a,a) = NOT(AND(a,a)) = NOT(a)$. This means that *NAND* can compute *NOT*. By the principle of "double negation", $AND(a,b) = NOT(NOT(AND(a,b)))$, and hence we can use *NAND* to compute *AND* as well. Once we can compute *AND* and *NOT*, we can compute *OR* using "De Morgan's Law": $OR(a,b) = NOT(AND(NOT(a),NOT(b)))$ (which can also be written as $a \vee b = \overline{\overline{a} \wedge \overline{b}}$) for every $a,b \in \{0,1\}$. ■

P

Theorem 3.9's proof is very simple, but you should make sure that **(i)** you understand the statement of the theorem, and **(ii)** you follow its proof. In particular, you should make sure you understand why De Morgan's law is true.

We can use *NAND* to compute many other functions, as demonstrated in the following exercise.

Solved Exercise 3.5 — Compute majority with NAND. Let $MAJ : \{0,1\}^3 \to \{0,1\}$ be the function that on input a,b,c outputs 1 iff $a+b+c \geq 2$. Show how to compute *MAJ* using a composition of *NAND*'s.

Solution:

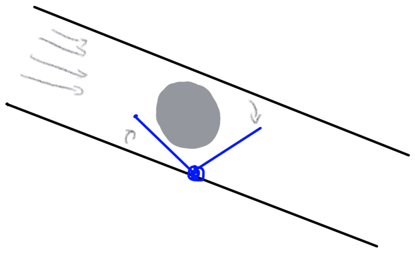


Figure 3.25: A "gadget" in a pipe that ensures that at most one marble can pass through it. The first marble that passes causes the barrier to lift and block new ones.

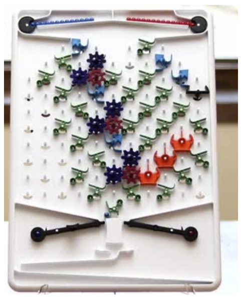


Figure 3.26: The game "Turing Tumble" contains an implementation of logical gates using marbles.

Recall that (3.5) states that

$$MAJ(x_0,x_1,x_2) = OR\left(AND(x_0,x_1)\ ,\ OR(AND(x_1,x_2)\ ,\ AND(x_0,x_2))\right)\ . \tag{3.10}$$

We can use Theorem 3.9 to replace all the occurrences of AND and OR with $NAND$'s. Specifically, we can use the equivalence $AND(a,b) = NOT(NAND(a,b))$, $OR(a,b) = NAND(NOT(a), NOT(b))$, and $NOT(a) = NAND(a,a)$ to replace the righthand side of (3.10) with an expression involving only $NAND$, yielding that $MAJ(a,b,c)$ is equivalent the (somewhat unwieldy) expression

$$\begin{aligned} NAND\Big(& NAND\Big(NAND(NAND(a,b), NAND(a,c)), \\ & NAND(NAND(a,b), NAND(a,c)) \Big), \\ & NAND(b,c) \Big) \end{aligned} \tag{3.11}$$

The same formula can also be expressed as a circuit with NAND gates, see Fig. 3.27.

■

3.5.1 NAND Circuits

We define *NAND Circuits* as circuits in which all the gates are NAND operations. Such a circuit again corresponds to a directed acyclic graph (DAG) since all the gates correspond to the same function (i.e., NAND), we do not even need to label them, and all gates have in-degree exactly two. Despite their simplicity, NAND circuits can be quite powerful.

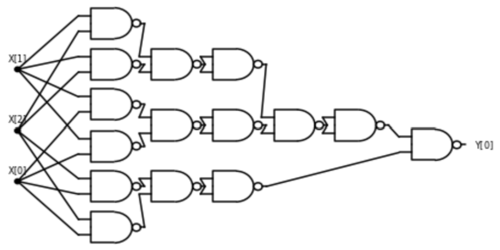


Figure 3.27: A circuit with NAND gates to compute the Majority function on three bits

■ **Example 3.10 — $NAND$ circuit for XOR.** Recall the XOR function which maps $x_0, x_1 \in \{0,1\}$ to $x_0 + x_1 \mod 2$. We have seen in Section 3.2.2 that we can compute XOR using AND, OR, and NOT, and so by Theorem 3.9 we can compute it using only $NAND$'s. However, the following is a direct construction of computing XOR by a sequence of NAND operations:

1. Let $u = NAND(x_0,x_1)$.
2. Let $v = NAND(x_0,u)$
3. Let $w = NAND(x_1,u)$.
4. The XOR of x_0 and x_1 is $y_0 = NAND(v,w)$.

One can verify that this algorithm does indeed compute XOR by enumerating all the four choices for $x_0, x_1 \in \{0,1\}$. We can also represent this algorithm graphically as a circuit, see Fig. 3.28.

In fact, we can show the following theorem:

Theorem 3.11 — NAND is a universal operation. For every Boolean circuit C of s gates, there exists a NAND circuit C' of at most $3s$ gates that computes the same function as C.

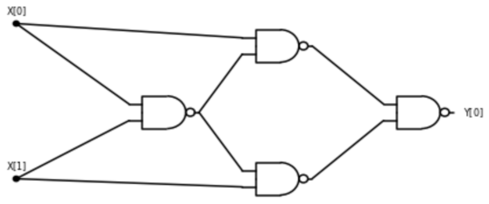


Figure 3.28: A circuit with NAND gates to compute the XOR of two bits.

Proof Idea:

The idea of the proof is to just replace every AND, OR and NOT gate with their NAND implementation following the proof of Theorem 3.9.

⋆

Proof of Theorem 3.11. If C is a Boolean circuit, then since, as we've seen in the proof of Theorem 3.9, for every $a, b \in \{0,1\}$

- $NOT(a) = NAND(a,a)$
- $AND(a,b) = NAND(NAND(a,b), NAND(a,b))$
- $OR(a,b) = NAND(NAND(a,a), NAND(b,b))$

we can replace every gate of C with at most three $NAND$ gates to obtain an equivalent circuit C'. The resulting circuit will have at most $3s$ gates.

■

Big Idea 3 Two models are *equivalent in power* if they can be used to compute the same set of functions.

3.5.2 More examples of NAND circuits (optional)

Here are some more sophisticated examples of NAND circuits:

Incrementing integers. Consider the task of computing, given as input a string $x \in \{0,1\}^n$ that represents a natural number $X \in \mathbb{N}$, the representation of $X+1$. That is, we want to compute the function $INC_n : \{0,1\}^n \rightarrow \{0,1\}^{n+1}$ such that for every $x_0, \ldots, x_{n-1}$, $INC_n(x) = y$ which satisfies $\sum_{i=0}^{n} y_i 2^i = \left(\sum_{i=0}^{n-1} x_i 2^i\right) + 1$. (For simplicity of notation, in this example we use the representation where the least significant digit is first rather than last.)

The increment operation can be very informally described as follows: *"Add 1 to the least significant bit and propagate the carry"*. A little more precisely, in the case of the binary representation, to obtain the increment of x, we scan x from the least significant bit onwards, and flip all 1's to 0's until we encounter a bit equal to 0, in which case we flip it to 1 and stop.

Thus we can compute the increment of $x_0, \ldots, x_{n-1}$ by doing the following:

Algorithm 3.12 — Compute Increment Function.

```
Input: $x_0, x_1, \ldots, x_{n-1}$ representing the number $\sum_{i=0}^{n-1} x_i \cdot 2^i$
    # we use LSB-first representation
Output: $y \in \{0,1\}^{n+1}$ such that $\sum_{i=0}^{n} y_i \cdot 2^i = \sum_{i=0}^{n-1} x_i \cdot 2^i$
 1: Let $c_0 \leftarrow 1$        # we pretend we have a "carry" of 1 initially
 2: for $i = 0, \ldots, n-1$ do
 3:     Let $y_i \leftarrow XOR(x_i, c_i)$.
 4:     if $c_i = x_i = 1$ then
 5:         $c_{i+1} = 1$
 6:     else
 7:         $c_{i+1} = 0$
 8:     end if
 9: end for
10: Let $y_n \leftarrow c_n$.
```

Algorithm 3.12 describes precisely how to compute the increment operation, and can be easily transformed into *Python* code that performs the same computation, but it does not seem to directly yield a NAND circuit to compute this. However, we can transform this algorithm line by line to a NAND circuit. For example, since for every a, $NAND(a, NOT(a)) = 1$, we can replace the initial statement $c_0 = 1$ with $c_0 = NAND(x_0, NAND(x_0, x_0))$. We already know how to compute *XOR* using NAND and so we can use this to implement the operation $y_i \leftarrow XOR(x_i, c_i)$. Similarly, we can write the "if" statement as saying $c_{i+1} \leftarrow AND(y_i, x_i)$, or in other words $c_{i+1} \leftarrow NAND(NAND(y_i, x_i), NAND(y_i, x_i))$. Finally, the assignment $y_n = c_n$ can be written as $y_n = NAND(NAND(c_n, c_n), NAND(c_n, c_n))$. Combining these observations yields for every $n \in \mathbb{N}$, a *NAND* circuit to compute INC_n. For example, Fig. 3.29 shows how this circuit looks like for $n = 4$.

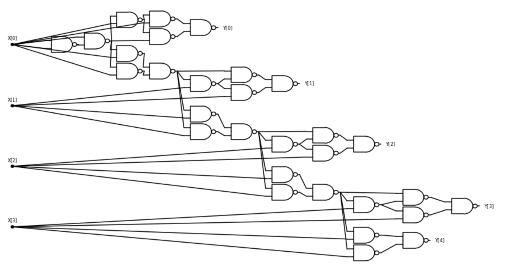


Figure 3.29: NAND circuit with computing the *increment* function on 4 bits.

From increment to addition. Once we have the increment operation, we can certainly compute addition by repeatedly incrementing (i.e., compute $x + y$ by performing $INC(x)$ y times). However, that would be quite inefficient and unnecessary. With the same idea of keeping track of carries we can implement the "grade-school" addition algorithm and compute the function $ADD_n : \{0,1\}^{2n} \rightarrow \{0,1\}^{n+1}$ that on input $x \in \{0,1\}^{2n}$ outputs the binary representation of the sum of the numbers represented by $x_0, \ldots, x_{n-1}$ and $x_{n+1}, \ldots, x_n$:

Algorithm 3.13 — Addition using NAND.

```
Input: $u \in \{0,1\}^n, v \in \{0,1\}^n$ representing numbers in
    LSB-first binary representation.
Output: LSB-first binary representation of $x + y$.
 1: Let $c_0 \leftarrow 0$
 2: for $i = 0, \ldots, n-1$ do
 3:     Let $y_i \leftarrow u_i + v_i \mod 2$
 4:     if $u_i + v_i + c_i \geq 2$ then
 5:         $c_{i+1} \leftarrow 1$
 6:     else
 7:         $c_{i+1} \leftarrow 0$
 8:     end if
 9: end for
10: Let $y_n \leftarrow c_n$
```

Once again, Algorithm 3.13 can be translated into a NAND circuit. The crucial observation is that the "if/then" statement simply corresponds to $c_{i+1} \leftarrow MAJ_3(u_i, v_i, v_i)$ and we have seen in in Solved Exercise 3.5 that the function $MAJ_3 : \{0,1\}^3 \rightarrow \{0,1\}$ can be computed using *NAND*s.

3.5.3 The NAND-CIRC Programming language

Just like we did for Boolean circuits, we can define a programming-language analog of NAND circuits. It is even simpler than the AON-CIRC language since we only have a single operation. We define the *NAND-CIRC Programming Language* to be a programming language where every line has the following form:

```
foo = NAND(bar,blah)
```

where `foo`, `bar` and `blah` are variable identifiers.

■ **Example 3.14 — Our first NAND-CIRC program.** Here is an example of a NAND-CIRC program:

```
u = NAND(X[0],X[1])
v = NAND(X[0],u)
w = NAND(X[1],u)
Y[0] = NAND(v,w)
```

P

Do you know what function this program computes? Hint: you have seen it before.

We can formally define the notion of computation by a NAND-CIRC program in the natural way:

Definition 3.15 — Computing by a NAND-CIRC program. Let $f : \{0,1\}^n \to \{0,1\}^m$ be some function, and let P be a NAND-CIRC program. We say that P *computes* the function F if:

1. P has n input variables `X[`0`]`, … ,`X[`$n-1$`]` and m output variables `Y[`0`]`,…,`Y[`$m-1$`]`.
2. For every $x \in \{0,1\}^n$, if we execute P when we assign to `X[`0`]`, … ,`X[`$n - 1$`]` the values $x_0, \ldots, x_{n-1}$, then at the end of the execution, the output variables `Y[`0`]`,…,`Y[`$m - 1$`]` have the values $y_0, \ldots, y_{m-1}$ where $y = f(x)$.

As before we can show that NAND circuits are equivalent to NAND-CIRC programs (see Fig. 3.30):

Theorem 3.16 — NAND circuits and straight-line program equivalence. For every $f : \{0,1\}^n \to \{0,1\}^m$ and $s \geq m$, f is computable by a NAND-CIRC program of s lines if and only if f is computable by a NAND circuit of s gates.

We omit the proof of Theorem 3.16 since it follows along exactly the same lines as the equivalence of Boolean circuits and AON-CIRC program (Theorem 3.8). Given Theorem 3.16 and Theorem 3.11, we know that we can translate every s-line AON-CIRC program P into an equivalent NAND-CIRC program of at most $3s$ lines. In fact, this translation can be easily done by replacing every line of the form `foo = AND(bar,blah)`, `foo = OR(bar,blah)` or `foo = NOT(bar)` with the equivalent 1-3 lines that use the `NAND` operation. Our GitHub repository contains a "proof by code": a simple Python program `AON2NAND` that transforms an AON-CIRC into an equivalent NAND-CIRC program.

```
temp_1 = NAND(X[0],X[1])
temp_2 = NAND(X[0],temp_1)
temp_3 = NAND(X[1],temp_1)
temp_4 = NAND(temp_2,temp_3)
temp_5 = NAND(temp_4,X[2])
temp_6 = NAND(temp_4,temp_5)
temp_7 = NAND(X[2],temp_5)
Y[0] = NAND(temp_6,temp_7)
```

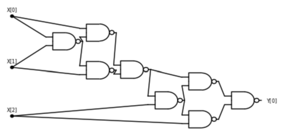


Figure 3.30: A NAND program and the corresponding circuit. Note how every line in the program corresponds to a gate in the circuit.

R

Remark 3.17 — Is the NAND-CIRC programming language Turing Complete? (optional note). You might have heard of a term called "Turing Complete" that is sometimes used to describe programming languages. (If you haven't, feel free to ignore the rest of this remark: we define this term precisely in Chapter 7.) If so, you might wonder if the NAND-CIRC programming language has this property. The answer is **no**, or perhaps more accurately, the term "Turing Completeness" is not really applicable for the NAND-CIRC programming language. The reason is that, by design, the NAND-CIRC programming language can only compute *finite* functions $F : \{0,1\}^n \rightarrow \{0,1\}^m$ that take a fixed number of input bits and produce a fixed number of outputs bits. The term "Turing Complete" is only applicable to programming languages for *infinite* functions that can take inputs of arbitrary length. We will come back to this distinction later on in this book.

3.6 EQUIVALENCE OF ALL THESE MODELS

If we put together Theorem 3.8, Theorem 3.11, and Theorem 3.16, we obtain the following result:

Theorem 3.18 — Equivalence between models of finite computation. For every sufficiently large s, n, m and $f : \{0,1\}^n \rightarrow \{0,1\}^m$, the following conditions are all equivalent to one another:

- f can be computed by a Boolean circuit (with $\wedge, \vee, \neg$ gates) of at most $O(s)$ gates.
- f can be computed by an AON-CIRC straight-line program of at most $O(s)$ lines.
- f can be computed by a NAND circuit of at most $O(s)$ gates.
- f can be computed by a NAND-CIRC straight-line program of at most $O(s)$ lines.

By "$O(s)$" we mean that the bound is at most $c \cdot s$ where c is a constant that is independent of n. For example, if f can be computed by a Boolean circuit of s gates, then it can be computed by a NAND-CIRC program of at most $3s$ lines, and if f can be computed by a NAND circuit of s gates, then it can be computed by an AON-CIRC program of at most $2s$ lines.

Proof Idea:

We omit the formal proof, which is obtained by combining Theorem 3.8, Theorem 3.11, and Theorem 3.16. The key observation is that the results we have seen allow us to translate a program/circuit that computes f in one of the above models into a program/circuit that computes f in another model by increasing the lines/gates by at most a constant factor (in fact this constant factor is at most 3).

⋆

Theorem 3.8 is a special case of a more general result. We can consider even more general models of computation, where instead of AND/OR/NOT or NAND, we use other operations (see Section 3.6.1 below). It turns out that Boolean circuits are equivalent in power to such models as well. The fact that all these different ways to define computation lead to equivalent models shows that we are "on the right track". It justifies the seemingly arbitrary choices that we've made of using AND/OR/NOT or NAND as our basic operations, since these choices do not affect the power of our computational model. Equivalence results such as Theorem 3.18 mean that we can easily translate between Boolean circuits, NAND circuits, NAND-CIRC programs and the like. We will use this ability later on in this book, often shifting to the most convenient formulation without making a big deal about it. Hence we will not worry too much about the distinction between, for example, Boolean circuits and NAND-CIRC programs.

In contrast, we will continue to take special care to distinguish between *circuits/programs* and *functions* (recall Big Idea 2). A function corresponds to a *specification* of a computational task, and it is a fundamentally different object than a program or a circuit, which corresponds to the *implementation* of the task.

3.6.1 Circuits with other gate sets

There is nothing special about AND/OR/NOT or NAND. For every set of functions $\mathcal{G} = \{G_0, \ldots, G_{k-1}\}$, we can define a notion of circuits that use elements of $\mathcal{G}$ as gates, and a notion of a "$\mathcal{G}$ programming language" where every line involves assigning to a variable `foo` the result of applying some $G_i \in \mathcal{G}$ to previously defined or input variables. Specifically, we can make the following definition:

Definition 3.19 — General straight-line programs. Let $\mathcal{F} = \{f_0, \ldots, f_{t-1}\}$ be a finite collection of Boolean functions, such that $f_i : \{0,1\}^{k_i} \rightarrow \{0,1\}$ for some $k_i \in \mathbb{N}$. An $\mathcal{F}$ *program* is a sequence of lines, each of which assigns to some variable the result of applying some $f_i \in \mathcal{F}$ to k_i other variables. As above, we use `X[`i`]` and `Y[`j`]` to denote the input and output variables.

We say that $\mathcal{F}$ is a *universal set of operations* (also known as a universal gate set) if there exists a $\mathcal{F}$ program to compute the function $NAND$.

AON-CIRC programs correspond to $\{AND, OR, NOT\}$ programs, NAND-CIRC programs corresponds to $\mathcal{F}$ programs for the set $\mathcal{F}$ that only contains the $NAND$ function, but we can also define $\{IF, ZERO, ONE\}$ programs (see below), or use any other set.

We can also define $\mathcal{F}$ *circuits*, which will be directed graphs in which each *gate* corresponds to applying a function $f_i \in \mathcal{F}$, and will each have k_i incoming wires and a single outgoing wire. (If the function f_i is not *symmetric*, in the sense that the order of its input matters then we need to label each wire entering a gate as to which parameter of the function it corresponds to.) As in Theorem 3.8, we can show that $\mathcal{F}$ circuits and $\mathcal{F}$ programs are equivalent. We have seen that for $\mathcal{F} = \{AND, OR, NOT\}$, the resulting circuits/programs are equivalent in power to the NAND-CIRC programming language, as we can compute $NAND$ using $AND/OR/NOT$ and vice versa. This turns out to be a special case of a general phenomena— the *universality* of $NAND$ and other gate sets — that we will explore more in depth later in this book.

■ **Example 3.20 — IF,ZERO,ONE circuits.** Let $\mathcal{F} = \{IF, ZERO, ONE\}$ where $ZERO : \{0,1\} \rightarrow \{0\}$ and $ONE : \{0,1\} \rightarrow \{1\}$ are the constant zero and one functions,[2] and $IF : \{0,1\}^3 \rightarrow \{0,1\}$ is the function that on input (a,b,c) outputs b if $a = 1$ and c otherwise. Then $\mathcal{F}$ is universal.

Indeed, we can demonstrate that $\{IF, ZERO, ONE\}$ is universal using the following formula for $NAND$:

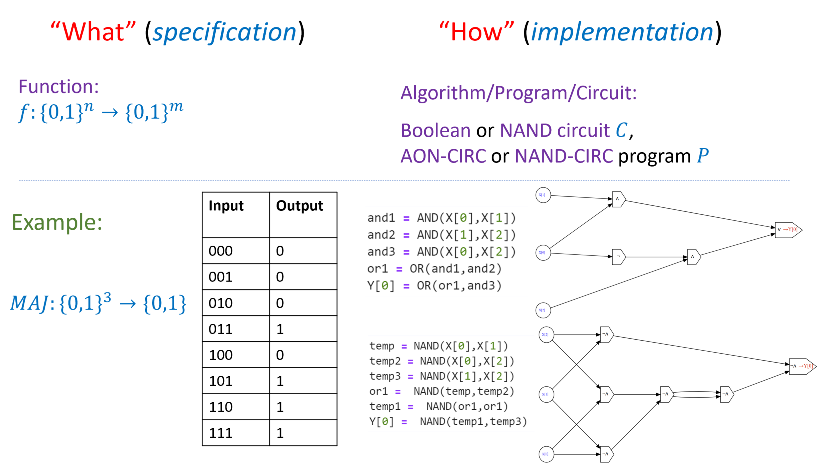
$$NAND(a,b) = IF(a, IF(b, ZERO, ONE), ONE) \ . \tag{3.12}$$

[2] One can also define these functions as taking a length zero input. This makes no difference for the computational power of the model.

There are also some sets $\mathcal{F}$ that are more restricted in power. For example it can be shown that if we use only AND or OR gates (without NOT) then we do *not* get an equivalent model of computation. The exercises cover several examples of universal and non-universal gate sets.

3.6.2 Specification vs. implementation (again)

As we discussed in Section 2.5.1, one of the most important distinctions in this book is that of *specification* versus *implementation* or separating "what" from "how" (see Fig. 3.31). A *function* corresponds to the *specification* of a computational task, that is *what* output should be produced for every particular input. A *program* (or circuit, or any other way to specify *algorithms*) corresponds to the *implementation* of *how* to compute the desired output from the input. That is, a program is a set of instructions how to compute the output from the input. Even within the same computational model there can be many different ways to compute the same function. For example, there is more than one NAND-CIRC program that computes the majority function, more than one Boolean circuit to compute the addition function, and so on and so forth.


“What” (*specification*)
Function:
$f:\{0,1\}^n \rightarrow \{0,1\}^m$
“How” (*implementation*)
Algorithm/Program/Circuit:
Boolean or NAND circuit C,
AON-CIRC or NAND-CIRC program P
Example:
$MAJ:\{0,1\}^3 \rightarrow \{0,1\}$

Input	Output
000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

```
and1 = AND(X[0],X[1])
and2 = AND(X[1],X[2])
and3 = AND(X[0],X[2])
or1 = OR(and1,and2)
Y[0] = OR(or1,and3)
```

```
temp = NAND(X[0],X[1])
temp2 = NAND(X[0],X[2])
temp3 = NAND(X[1],X[2])
or1 =  NAND(temp,temp2)
temp1 =  NAND(or1,or1)
Y[0] =  NAND(temp1,temp3)
```

Figure 3.31: It is crucial to distinguish between the *specification* of a computational task, namely *what* is the function that is to be computed and the *implementation* of it, namely the algorithm, program, or circuit that contains the instructions *how* to map and input to an output. The same function could be computed in many different ways.

Confusing specification and implementation (or equivalently *functions* and *programs*) is a common mistake, and one that is unfortunately encouraged by the common programming-language terminology of referring to parts of programs as “functions”. However, in both the theory and practice of computer science, it is important to maintain this distinction, and it is particularly important for us in this book.

✓ **Lecture Recap**

- An *algorithm* is a recipe for performing a computation as a sequence of “elementary” or “simple” operations.
- One candidate definition for “elementary” operations is the set *AND*, *OR* and *NOT*.
- Another candidate definition for an “elementary” operation is the *NAND* operation. It is an operation that is easily implementable in the physical world in a variety of methods including by electronic transistors.
- We can use *NAND* to compute many other functions, including majority, increment, and others.
- There are other equivalent choices, including the sets $\{AND, OR, NOT\}$ and $\{IF, ZERO, ONE\}$.
- We can formally define the notion of a function $F : \{0,1\}^n \rightarrow \{0,1\}^m$ being computable using the *NAND-CIRC Programming language*.
- For every set of basic operations, the notions of being computable by a circuit and being computable by a straight-line program are equivalent.

3.7 EXERCISES

Exercise 3.1 — Compare 4 bit numbers. Give a Boolean circuit (with AND/OR/NOT gates) that computes the function $CMP_8 : \{0,1\}^8 \rightarrow \{0,1\}$ such that $CMP_8(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3) = 1$ if and only if the number represented by $a_0a_1a_2a_3$ is larger than the number represented by $b_0b_1b_2b_3$.

■

Exercise 3.2 — Compare n bit numbers. Prove that there exists a constant c such that for every n there is a Boolean circuit (with AND/OR/NOT gates) C of at most $c \cdot n$ gates that computes the function $CMP_{2n} : \{0,1\}^{2n} \rightarrow \{0,1\}$ such that $CMP_{2n}(a_0 \cdots a_{n-1}b_0 \cdots b_{n-1}) = 1$ if and only if the number represented by $a_0 \cdots a_{n-1}$ is larger than the number represented by $b_0 \cdots b_{n-1}$.

■

Exercise 3.3 — OR,NOT is universal. Prove that the set $\{OR, NOT\}$ is *universal*, in the sense that one can compute NAND using these gates.

■

Exercise 3.4 — AND,OR is not universal. Prove that for every n-bit input circuit C that contains only AND, and OR gates, as well as gates that compute the constant functions 0 and 1, C is *monotone*, in the sense that if $x, x' \in \{0,1\}^n$, $x_i \leq x'_i$ for every $i \in [n]$, then $C(x) \leq C(x')$.

Conclude that the set $\{AND, OR, 0, 1\}$ is *not* universal.

■

Exercise 3.5 — XOR is not universal. Prove that for every n-bit input circuit C that contains only XOR, gates, as well as gates that compute the constant functions 0 and 1, C is *affine or linear modulo two*, in the sense that there exists some $a \in \{0,1\}^n$ and $b \in \{0,1\}$ such that for every $x \in \{0,1\}^n$, $C(x) = \sum_{i=0}^{n-1} a_ix_i + b \mod 2$.

Conclude that the set $\{XOR, 0, 1\}$ is *not* universal.

■

Exercise 3.6 — MAJ,NOT is universal. Prove that $\{MAJ, NOT\}$ is a universal set of gates.

■

Exercise 3.7 — NOR is universal. Let $NOR : \{0,1\}^2 \rightarrow \{0,1\}$ defined as $NOR(a,b) = NOT(OR(a,b))$. Prove that $\{NOR\}$ is a universal set of gates.

■

Exercise 3.8 — Lookup is universal. Prove that $\{LOOKUP_1, 0, 1\}$ is a universal set of gates where 0 and 1 are the constant functions $LOOKUP_1 : \{0,1\}^3 \rightarrow \{0,1\}$ satisfies $LOOKUP_1(a,b,c)$ equals a if $c = 0$ and equals b if $c = 1$.

■

Exercise 3.9 — Bound on universal basis size (challenge). Prove that for every subset B of the functions from $\{0,1\}^k$ to $\{0,1\}$, if B is universal then there is a B-circuit of at most $O(k)$ gates to compute the *NAND* function (you can start by showing that there is a B circuit of at most $O(k^{16})$ gates).[3]

[3] Thanks to Alec Sun for solving this problem.

■

Exercise 3.10 — Threshold using NANDs. Prove that there is some constant c such that for every $n > 1$, and integers $a_0, \ldots, a_{n-1}, b \in \{-2^n, -2^n + 1, \ldots, -1, 0, +1, \ldots, 2^n\}$, there is a NAND circuit with at most $c\dot{n}^4$ gates that computes the *threshold* function $f_{a_0,\ldots,a_{n-1},b} : \{0,1\}^n \rightarrow \{0,1\}$ that on input $x \in \{0,1\}^n$ outputs 1 if and only if $\sum_{i=0}^{n-1} a_i x_i > b$.

■

Exercise 3.11 — NANDs from activation functions. We say that a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ is a *NAND approximator* if it has the following property: for every $a, b \in \mathbb{R}$, if $\min\{|a|, |1-a|\} \leq 1/3$ and $\min\{|b|, |1-b|\} \leq 1/3$ then $|f(a,b) - NAND(\lfloor a \rceil, lfloorb\rceil)| \leq 1/3$ where we denote by $\lfloor x \rceil$ the integer closest to x. That is, if a, b are within a distance $1/3$ to $\{0,1\}$ then we want $f(a,b)$ to equal the *NAND* of the values in $\{0,1\}$ that are closest to a and b respectively. Otherwise, we do not care what the output of f is on a and b.

In this exercise you will show that you can construct a NAND approximator from many common activation functions used in deep neural networks. As a corollary you will obtain that deep neural networks can simulate NAND circuits. Since NAND circuits can also simulate deep neural networks, these two computational models are equivalent to one another.

1. Show that there is a NAND approximator f defined as $f(a,b) = L(ReLU(L'(a,b)))$ where $L' : \mathbb{R}^2 \rightarrow \mathbb{R}$ is an *affine* function (of the form $L'(a,b) = \alpha a + \beta b + \gamma$ for some $\alpha, \beta, \gamma \in \mathbb{R}$), L is an affine function (of the form $L(y) = \alpha y + \beta$ for $\alpha, \beta \in \mathbb{R}$), and $ReLU : \mathbb{R} \rightarrow \mathbb{R}$, is the function defined as $ReLU(x) = \max\{0, x\}$.

2. Show that there is a NAND approximator f defined as $f(a,b) = L(sigmoid(L'(a,b)))$ where L', L are affine as above and $sigmoid : \mathbb{R} \rightarrow \mathbb{R}$ is the function defined as $sigmoid(x) = e^x/(e^x + 1)$.

3. Show that there is a NAND approximator f defined as $f(a,b) = L(tanh(L'(a,b)))$ where L', L are affine as above and $tanh : \mathbb{R} \rightarrow \mathbb{R}$ is the function defined as $tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$.

4. Prove that for every NAND-circuit C with n inputs and one output that computes a function $g : \{0,1\}^n \rightarrow \{0,1\}$, if we replace every

gate of C with a NAND-approximator and then invoke the resulting circuit on some $x \in \{0,1\}^n$, the output will be a number y such that $|y - g(x)| \leq 1/3$.

■

Exercise 3.12 — Majority with NANDs efficiently. Prove that there is some constant c such that for every $n > 1$, there is a NAND circuit of at most $c \cdot n$ gates that computes the function $MAJ_n : \{0,1\}^n \to \{0,1\}$ is the majority function on n input bits. That is $MAJ_n(x) = 1$ iff $\sum_{i=0}^{n-1} x_i > n/2$.[4]

■

[4] *Hint:* One approach to solve this is using recursion and analyzing it using the so called "Master Theorem".

Exercise 3.13 — Output at last layer. Prove that for every $f : \{0,1\}^n \to \{0,1\}$, if there is a Boolean circuit C of s gates that computes f then there is a Boolean circuit C' of at most s gates such that in the minimal layering of C', the output gate of C' is in placed the last layer. See footnote for hint.[5]

■

[5] *Hint:* Vertices in layers beyond the output can be safely removed without changing the functionality of the circuit.

3.8 BIOGRAPHICAL NOTES

The excerpt from Al-Khwarizmi's book is from "The Algebra of Ben-Musa", Fredric Rosen, 1831.

Charles Babbage (1791-1871) was a visionary scientist, mathematician, and inventor (see [Swa02; CM00]). More than a century before the invention of modern electronic computers, Babbage realized that computation can be in principle mechanized. His first design for a mechanical computer was the *difference engine* that was designed to do polynomial interpolation. He then designed the *analytical engine* which was a much more general machine and the first prototype for a programmable general purpose computer. Unfortunately, Babbage was never able to complete the design of his prototypes. One of the earliest people to realize the engine's potential and far reaching implications was Ada Lovelace (see the notes for Chapter 6).

Boolean algebra was first investigated by Boole and DeMorgan in the 1840's [Boo47; De 47]. The definition of Boolean circuits and connection to electrical relay circuits was given in Shannon's Masters Thesis [Sha38]. (Howard Gardener called Shannon's thesis "possibly the most important, and also the most famous, master's thesis of the [20th] century".) Savage's book [Sav98], like this one, introduces the theory of computation starting with Boolean circuits as the first model. Jukna's book [Juk12] contains a modern in-depth exposition of Boolean circuits, see also [Weg87].

The NAND function was shown to be universal by Sheffer [She13], though this also appears in the earlier work of Peirce, see [Bur78]. Whitehead and Russell used NAND as the basis for their logic in their magnum opus *Principia Mathematica* [WR12]. In her Ph.D thesis, Ernst [Ern09] investigates empirically the minimal NAND circuits for various functions. Nissan and Shocken's book [NS05] builds a computing system starting from NAND gates and ending with high level programs and games ("NAND to Tetris"); see also the website nandtotetris.org.