The science of computing

first edition

by Carl Burch

Contents

Chapter 1

Introduction

Computer science is the study of algorithms for transforming information. In this course, we explore a variety of approaches to one of the most fundamental questions of computer science:

> What can computers do?

That is, by the course's end, you should have a greater understanding of what computers can and cannot do.

We frequently use the term **computational power** to refer to the range of computation a device can accomplish. Don't let the word *power* here mislead you: We're not interested in large, expensive, fast equipment. We want to understand the extent of what computers can accomplish, whatever their efficiency. Along the same vein, we would say that a pogo stick is more powerful than a truck: Although a pogo stick may be slow and cheap, one can use it to reach areas that a large truck cannot reach, such as the end of a narrow trail. Since a pogo stick can go more places, we would say that it is more powerful than a truck. When applied to computers, we would say that a simple computer can do everything that a supercomputer can, and so it is just as powerful.

1.1 Common misconceptions about computer science

Most students arrive to college without a good understanding of computer science. Often, these students choose to study computer science based on their misconceptions of the subject. In the worst cases, students continue for several semesters before they realize that they have no interest in computer science. Before we continue, let me point out some of the most common misconceptions.

Computer science is not primarily about applying computer technology to business needs. Many colleges have such a major called "Management Information Systems," closely related to a Management or Business major. Computer science, on the other hand, tends to take a scientist's view: We are interested in studying computation in itself. When we do study business applications of computers, the concentration is on the techniques underlying the software. Learning how to use the software effectively in practice receives much less emphasis.

Computer science is not primarily about building faster, better computers. Many colleges have such a major called "Computer Engineering," closely related to an Electrical Engineering major. Although computer science entails some study of computer hardware, it focuses more on computer software design, theoretical limits of computation, and human and social factors of computers.

Computer science is not primarily about writing computer programs. Computer science students learn how to write computer programs early in the curriculum, but the emphasis is not present in the "real" computer science courses later in the curriculum. These more advanced courses often depend on the understanding built up by learning how to program, but they rarely strive primarily to build programming expertise.

Computer science does not prepare students for jobs. That is, a good computer science curriculum isn't designed with any particular career in mind. Often, however, businesses look for graduates who have studied computer science extensively in college, because students of the discipline often develop skills and ways of thinking that work well for these jobs. Usually, these businesses want people who can work with others to understand how to use computer technology more effectively. Although this often involves programming computers, it also often does not.

Thinking about your future career is important, but people often concentrate too much on the quantifiable characteristics of hiring probabilities and salary. More important, however, is whether the career resonates with your interests: If you can't see yourself taking a job where a major in X is important, then majoring in X isn't going to prove very useful to your career, and it may even be a detriment. Of course, many students choose to major in computer science because of their curiosity, without regard to future careers.

1.2 Textbook goals

This textbook fulfills two major goals.

- It satisfies the curiosity of students interested in an overview of practical and theoretical approaches to the study of computation.
- Students who believe they may want to study computer science more intensively can get an overview of the subject to help with their discernment. In addition to students interested in majoring or minoring in computer science, these students include those who major in something else (frequently the natural sciences or mathematics) and simply want to understand computer science well also.

The course on which this textbook is based (CSCI 150 at the College of Saint Benedict and Saint John's University) has three major units, of roughly equal size.

- Students learn the fundamentals of how today's electronic computers work (Chapters 2 through 6 of this book). We follow a "bottom-up" approach, beginning with simple circuits and building up toward writing programs for a computer in assembly language.
- Students learn the basics of computer programming, using the specific programming language of Java (represented by the Java Supplement to this book).
- Students study different approaches to exploring the extent of computational power (Chapters 7 to 10 of this book).

Chapter 2

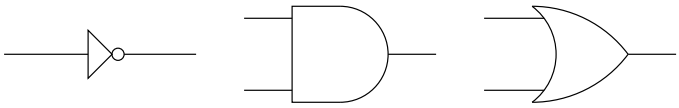
Logic circuits

We begin our exploration by considering the question of how a computer works. Answering this question will take several chapters. At the most basic level, a computer is an electrical circuit. In this chapter, we'll examine a system that computer designers use for designing circuits, called a **logic circuit**.

2.1 Understanding circuits

2.1.1 Logic circuits

A logic circuit consists of lines, representing wires, and peculiar shapes called **logic gates**. There are three types of logic gates:

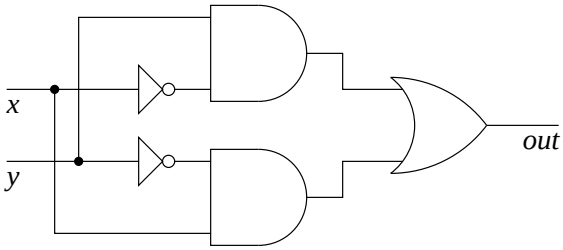


NOT gate AND gate OR gate

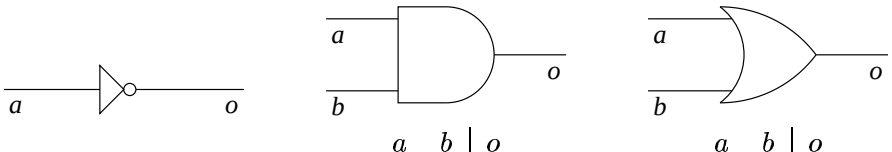
The relationship of the symbols to their names can be difficult to remember. I find it handy to remember that the word *AND* contains a *D*, and this is what an AND gate looks like. We'll see how logic gates work in a moment.

Each wire carries a single information element, called a **bit**. A bit's value can be either 0 or 1. In electrical terms, you can think of zero volts representing 0 and five volts representing 1. (In practice, there are many systems for representing 0 and 1 with different voltage levels. For our purposes, the details of voltage are not important.) The word *bit*, incidentally, comes from *B*inary dig*IT*; the term *binary* comes from the two (hence *bi-*) possible values.

Here is a diagram of one example logic circuit.



We'll think of a bit travelling down a wire until it hits a gate. You can see that some wires intersect in a small, solid circle: This circle indicates that the wires are connected, and so values coming into the circle



a	o
0	1
1	0

(a) NOT gate

a	b	o
0	0	0
0	1	0
1	0	0
1	1	1

(b) AND gate

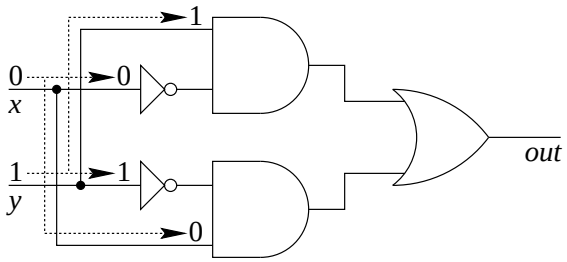
a	b	o
0	0	0
0	1	1
1	0	1
1	1	1

(c) OR gate

Figure 2.1: Logic gate behavior.

continue down all the wires connected to the circle. If two wires intersect with no circle, this means that one wire goes over the other, like an Interstate overpass, and a value on one wire has no influence on the other.

Suppose that we take our example circuit and send a 0 bit on the upper input (x) and a 1 bit on the lower input (y). Then these inputs would travel down the wires until they hit a gate.



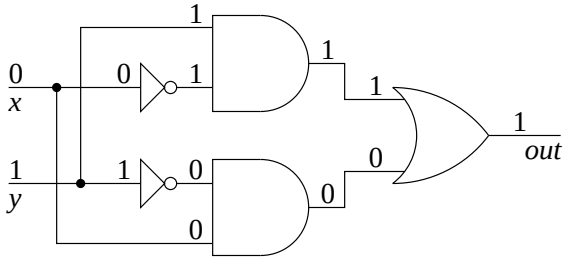
To understand what happens when a value reaches a gate, we need to define how the three gate types work.

NOT gate: Takes a single bit and produces the opposite bit (Figure 2.1(a)). In our example circuit, since the upper NOT gate takes 0 as an input, it will produce 1 as an output.

AND gate: Takes two inputs and outputs 1 only if both the first input *and* the second input are 1 (Figure 2.1(b)). In our example circuit, since both inputs to the upper AND gate are 1, the AND gate will output a 1.

OR gate: Takes two inputs and outputs 1 if either the first input *or* the second input are 1 (or if both are 1) (Figure 2.1(c)).

After the values filter through the gates based on the behaviors of Figure 2.1, the values in the circuit will be as follows.



Based on this diagram, we can see that when x is 0 and y is 1, the output out is 1.

By doing the same sort of propagation for other combinations of input values, we can build up a table of how this circuit works for different combinations of inputs. We would end up with the following results.

x	y	out
0	0	0
0	1	1
1	0	1
1	1	0

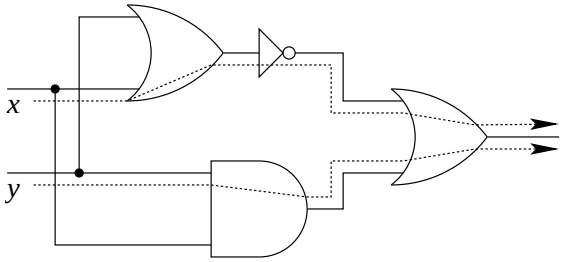
Such a table, representing what a circuit computes for each combination of possible inputs, is a **truth table**. The second row, which says that out is 1 if x is 0 and y is 1, corresponds to the propagation illustrated above.

2.1.2 Practical considerations

Logic gates are physical devices, built using transistors. At the heart of the computer is the **central processing unit** (CPU), which includes millions of transistors.

The designers of the CPU worry about two factors in their circuits: space and speed. The space factor relates to the fact that each transistor takes up space, and the chip containing the transistors is limited in size, so the number of transistors that fit onto a chip is limited by current technology. Since CPU designers want to fit many features onto the chip, they try to build their circuits with as few transistors as possible to accomplish the tasks needed. To reduce the number of transistors, they try to create circuits with few logic gates.

The second factor, speed, relates to the fact that transistors take time to operate. Since the designers want the CPU to work as quickly as possible, they work to minimize the **circuit depth**, which is the maximum distance from any input through the circuit to an output. Consider, for example, the two dotted lines in the following circuit, which indicate two different paths from an input to an output in the circuit.



The dotted path starting at x goes through three gates (an OR gate, then a NOT gate, then another OR gate), while the dotted path starting at y goes through only two gates (an AND gate and an OR gate). There are two other paths, too, but none of the paths go through more than three gates. Thus, we would say that this circuit's depth is 3, and this is a rough measure of the circuit's efficiency: Computing an output with this circuit takes about three times the amount of time it takes a single gate to do its work.

2.2 Designing circuits

In the previous section, we saw how logic circuits work. This is helpful when you want to understand the behavior of a circuit diagram. But computer designers face the opposite problem: Given a desired behavior, how can we build a circuit with that behavior? In this section, we look at a systematic technique for designing circuits. First, though, we'll take a necessary detour through the study of Boolean expressions.

2.2.1 Boolean algebra

Boolean algebra, a system of logic designed by George Boole in the middle of the nineteenth century, forms the foundation for modern computers. George Boole noticed that logical functions could be built from AND, OR, and NOT gates and that this observation leads one to be able to reason about logic in a mathematical system.

As Boole was working in the nineteenth century, of course, he wasn't thinking about logic circuits. He was examining the field of logic, created for thinking about the validity of philosophical arguments. Philosophers have thought about this subject since the time of Aristotle. Logicians formalized some common mistakes, such as the temptation to conclude that if A implies B, and if B holds, then A must hold also. ("Brilliant people wear glasses, and I wear glasses, so I must be brilliant.")

As a mathematician, Boole sought a way to encode sentences like this into algebraic expressions, and he invented what we now call **Boolean expressions**. An example of a Boolean expression is "$y\bar{x} + \bar{y}x$." A line over a variable (or a larger expression) represents a NOT; for example, the expression $\bar{y}$ corresponds to feeding y through a NOT gate. Multiplication (as with xy) represents AND. After all, Boole reasoned, the AND truth table (Figure 2.1(b)) is identical to a multiplication table over 0 and 1. Addition (as with $x + y$) represents OR. The OR truth table (Figure 2.1(c)) doesn't match an addition table over 0 and 1 exactly — although 1 plus 1 is 2, the result of 1 OR 1 is 1 — but, Boole decided, it's close enough to be a worthwhile analogy.

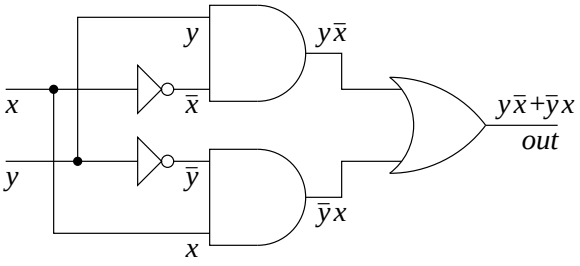
In Boolean expressions, we observe the regular order of operations: Multiplication (AND) comes before addition (OR). Thus, when we write $y\bar{x} + \bar{y}x$, we mean $(y\bar{x}) + (\bar{y}x)$. We can use parentheses when this order of operations isn't what we want. For NOT, the bar over the expression indicates the extent of the expression to which it applies; thus, $\overline{x + y}$ represents NOT $(x$ OR $y)$, while $\bar{x} + \bar{y}$ represents $(\text{NOT } x)$ OR $(\text{NOT } y)$.

A warning: Students new to Boolean expressions frequently try to abbreviate $\bar{x}\bar{y}$ as $\overline{xy}$ — that is, drawing a single line over the whole expresion, rather than two separate lines over the two individual pieces. This abbreviation is *wrong*. The first, $\bar{x}\bar{y}$, translates to $(\text{NOT } x)$ AND $(\text{NOT } y)$ (that is, both x and y are 0), while $\overline{xy}$ translates to NOT $(x$ AND $y)$ (that is, x and y aren't both 1). We could draw a truth table comparing the results for these two expressions.

x	y	$\bar{x}$	$\bar{y}$	$\bar{x}\bar{y}$	xy	$\overline{xy}$
0	0	1	1	1	0	1
0	1	1	0	0	0	1
1	0	0	1	0	0	1
1	1	0	0	0	1	0

Since the fifth column ($\bar{x}\bar{y}$) and the seventh column ($\overline{xy}$) aren't identical, the two expressions aren't equivalent.

Every expression directly corresponds to a circuit and vice versa. To determine the expression corresponding to a logic circuit, we feed expressions through the circuit just as values propagate through it. Suppose we do this for our circuit of Section 2.1.



The upper AND gate's inputs are y and $\bar{x}$, and so it outputs $y\bar{x}$. The lower AND gate outputs $\bar{y}x$, and the OR gate combines these two into $y\bar{x} + \bar{y}x$.

law		AND		OR
commutative		$AB = BA$		$A + B = B + A$
associative		$A(BC) = (AB)C$		$A + (B + C) = (A + B) + C$
identity		$A \cdot 1 = A$		$A + 0 = A$
distributive		$A(B + C) = AB + AC$	!	$A + BC = (A + B)(A + C)$
one/zero		$A \cdot 0 = 0$	!	$A + 1 = 1$
idempotency	!	$AA = A$	!	$A + A = A$
inverse	!	$A\overline{A} = 0$	!	$A + \overline{A} = 1$
DeMorgan's law	!	$\overline{AB} = \overline{A} + \overline{B}$	!	$\overline{A + B} = \overline{A}\,\overline{B}$
double negation			!	$\overline{\overline{A}} = A$

Figure 2.2: A sampler of important laws in Boolean algebra.

2.2.2 Algebraic laws

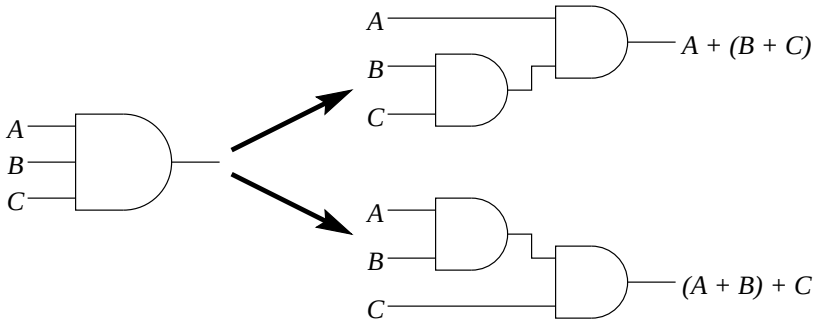
Boole's system for writing down logical expressions is called an *algebra* because we can manipulate symbols using laws similar to those of algebra. For example, the commutative law applies to both OR and AND. To prove that OR is commutative (that is, that $A + B = B + A$), we can complete a truth table demonstrating that for each possible combination of A and B, the values of $A + B$ and $B + A$ are identical.

A	B	$A + B$	$B + A$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

Since the third and fourth columns match, we would conclude that $A + B = B + A$ is a universal law.

Since OR (and AND) are commutative, we can freely reorder terms without changing the meaning of the expression. The commutative law of OR would allow us to transform $y\bar{x} + \bar{y}x$ into $\bar{y}x + y\bar{x}$, and the commutative law of AND (applied twice) allows us to transform $\bar{y}x + y\bar{x}$ to $x\bar{y} + \bar{x}y$.

Similarly, both OR and AND have an associative law (that is, $A + (B + C) = (A + B) + C$). Because of this associativity, we won't bother writing parentheses across the same operator when we write Boolean expressions. In drawing circuits, we'll freely draw AND and OR gates that have several inputs. A 3-input AND gate would actually correspond to two 2-input AND gates when the circuit is actually wired. There are two possible ways to wire this.



Because of the associative law for AND, it doesn't matter which we choose.

There are many such laws, summarized in Figure 2.2. This includes analogues to all of the important algebraic laws dealing with multiplication and addition. There are also many laws that *don't* hold with addition and multiplication; these are marked with an exclamation point in the table.

2.2.3 Converting a truth table

Now we can return to our problem: If we have a particular logical function we want to compute, how can we build a circuit to compute it? We'll begin with a description of the logical function as a truth table. Suppose we start with the following function for which we want a circuit.

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

Given such a truth table defining a function, we'll build up a Boolean expression representing the function. For each row of the table where the desired output is 1, we describe it as the AND of several factors.

x	y	z	out	description
0	0	1	1	$\bar{x}\bar{y}z$
0	1	0	1	$\bar{x}y\bar{z}$
1	1	0	1	$xy\bar{z}$
1	1	1	1	xyz

To arrive at a row's description, we choose for each variable either that variable or its negation, depending on which of these is 1 in that row. Then we take the AND of these choices. For example, if we consider the first of the rows above, we consider that since x is 0 in this row, $\bar{x}$ is 1; since y is 0, $\bar{y}$ is 1; and z is 1. Thus, our description is the AND of these choices, $\bar{x}\bar{y}z$. This expression gives 1 for the combination of values on this row; but for other rows, its value is 0, since every other row is different in some variable, and that variable's contribution to the AND would yield 0.

Once we have the descriptions for all rows where the desired output is 1, we observe the following: The value of the desired circuit should be 1 if the inputs correspond to the first 1-row, the second 1-row, the third 1-row, *or* the fourth 1-row. Thus, we'll combine the expressions describing the rows with an OR.

$$\bar{x}\bar{y}z + \bar{x}y\bar{z} + xy\bar{z} + xyz$$

Note that we do not include rows where the desired output is 0 — for these rows, we want none of the AND terms to yield 1, so that the OR of all terms gives 0.

The expression we get is called a **sum of products** expression. It is called this because it is the OR (a sum, if we understand OR to be like addition) of several ANDs (products, since AND corresponds to multiplication). We call this technique of building an expression from a truth table the **sum of products technique**.

This expression leads immediately to the circuit of Figure 2.3. In general, this technique allows us take any function over bits and build a circuit to compute that function. The existence of such a technique proves that circuits can compute any logical function.

Note, incidentally, that the depth of this circuit will always be three (or less), since every path from input to output will go through a NOT gate (maybe), an AND gate, and an OR gate. Thus, this technique shows that it's never necessary to design a circuit that is more than three gates deep. Sometimes, though, designers build deeper circuits because they are concerned not only with speed, but also with size: A larger circuit can often accomplish the same task using fewer gates.

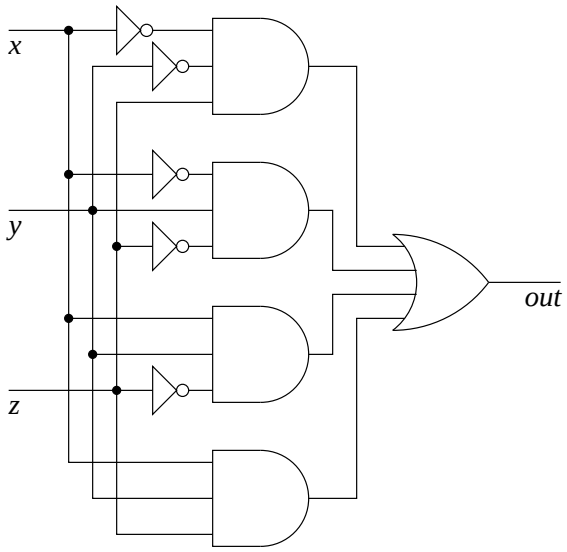


Figure 2.3: A circuit derived from a given truth table.

2.3 Simplifying circuits

The circuit generated by the sum-of-products technique can be quite large. This is impractical: Additional gates cost money, so CPU designers want to use gates as economically as possible to make room for more features or to reduce costs. Thus, we'd like to make a smaller circuit accomplishing the same thing, if possible.

We can simplify a circuit by taking the corresponding expression and reducing it using laws of Boolean algebra. We will insert this simplification step into our derivation process. Thus, our approach for converting a truth table into a circuit will now have three steps.

1. Build a sum of products expression from the truth table.

2. Simplify the expression using laws of Boolean algebra (as in Figure 2.2).

3. Draw the circuit corresponding to this simplified expression.

In the rest of this section, we look at a particular technique for simplifying a Boolean expression. Our technique for simplifying expressions works *only for sum-of-products expressions* — that is, the expression must be an OR of terms, each term being an AND of variables or their negations.

Suppose we want to simplify the expression

$$\bar{x}\bar{y}z + \bar{x}y\bar{z} + xy\bar{z} + xyz\ .$$

1. We look through the terms looking for all pairs that are identical in every way, except for one variable that is NOTted in one and not NOTted in the other. Our example has two such pairs.

   $\bar{x}y\bar{z}$ and $xy\bar{z}$ differ only in x.
   $xy\bar{z}$ and xyz differ only in z.

   If no such pairs are found, we are done.

2. We create a new expression. This expression includes a single term for each pair from step 1; this term keeps everything that the pair shares in common. The expression also keeps any terms that do not participate in any pairs from step 1. For our example, this would lead us to the following.

   $$\bar{x}\bar{y}z + y\bar{z} + xy$$

   The $\bar{x}\bar{y}z$ term arises because this term does not belong to a pair from step 1; we include $y\bar{z}$ due to the $\bar{x}y\bar{z} + xy\bar{z}$ pair, in which y and $\bar{z}$ are the common factors; and we include xy due to the $xy\bar{z} + xyz$ pair. Note that we include a term for *every pair*, even if some pairs overlap (that is, if two pairs include the same term).

   The following reasoning underlies this transformation.

   - Using the law that $A + A = A$, we can duplicate the $xy\bar{z}$ term, which appears in two of the pairs.

     $$\bar{x}\bar{y}z + \bar{x}y\bar{z} + xy\bar{z} + xy\bar{z} + xyz$$

   - Because of the associative and commutative laws of OR, we can rearrange and insert parentheses so that the pairs are together.

     $$\bar{x}\bar{y}z + (\bar{x}y\bar{z} + xy\bar{z}) + (xy\bar{z} + xyz)$$

     We'll concentrate on the first pair ($\bar{x}y\bar{z} + xy\bar{z}$) in the following. (The reasoning for the other pair, $xy\bar{z} + xyz$, proceeds similarly.)

   - $\bar{x}y\bar{z} + xy\bar{z}$ has two terms with $y\bar{z}$ in common. Using the distributive law of AND over OR, we get $(\bar{x} + x)y\bar{z}$.
   - We can apply the law $A + \overline{A} = 1$ to get $1 \cdot y\bar{z}$.
   - Finally, we apply AND's identity law ($1 \cdot A = A$) to get $y\bar{z}$.

3. If there are duplicates among the terms, we can remove the duplicates. This is justified by the Boolean algebra law that $A + A = A$. (There are no duplicates in this case.)

4. Return to step 1 to see if there are any pairs in this new expression. (In our working example, there are no more pairs to be found.)

Thus, we end up with the simplified expression

$$\bar{x}\bar{y}z + y\bar{z} + xy\ .$$

From this we can derive the smaller circuit of Figure 2.4 that works the same as that of Figure 2.3. In this case we have replaced the 10 gates of Figure 2.3 with only 7 in Figure 2.4.

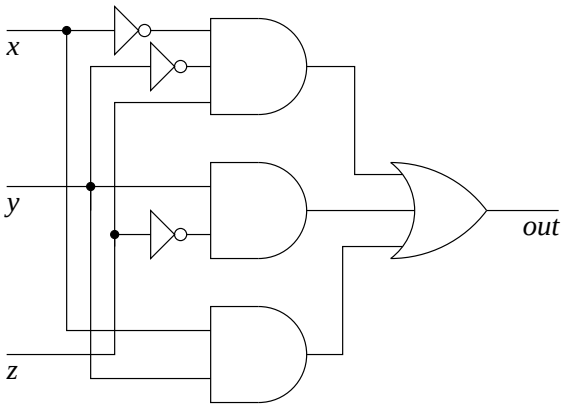


Figure 2.4: A simplified circuit equivalent to Figure 2.3.

Chapter 3

Data representation

Since computer science is the study of algorithms for *transforming information*, an important element of the discipline is understanding techniques for representing information. In this chapter, we'll examine some techniques for representing letters, integers, real numbers, and multimedia.

3.1 Binary representation

One of the first problems we need to confront is how wires can represent many values, if a wire can only carry a 0 or 1 based on its electrical voltage.

3.1.1 Combining bits

A wire can carry one bit, which can represent up to two values. But suppose we have several values. For example, suppose we want to remember the color of a traffic light (red, amber, or green). What can we do?

We can't do this using a single wire, since a wire carries a single bit, which can represent only two values (0 and 1). Thus, we will need to add more bits. Suppose we use two wires. Then we can assign colors to different combinations of bits on the two wires.

wire 1	wire 2	meaning
0	0	red
0	1	amber
1	0	green

For example, if wire 1 carries 0 and wire 2 carries 1, then this would indicate that the light is amber. In fact, with two bits we can represent up to *four* values: The fourth combination, $\langle 1, 1 \rangle$, was unnecessary for the traffic light.

If we want to represent one of the traditional colors of the rainbow (red, orange, yellow, green, blue, indigo, violet), then two bits would not be enough. But three bits would be: With three bits, there are four distinct values where the first bit is 0 (since there are four different combinations for the other two bits) and four distinct values where the first bit is 1, for a total of eight combinations, which is enough for the rainbow's seven colors.

In general, if we have k bits, we can represent 2^k distinct values through various combinations of the bits' values. In fact, this fact is so important that, to emphasize it, we will look at a formal proof.

Theorem 1 *We can represent 2^k distinct values using different combinations of k bits.*

Proof: For each bit, we have two choices, 0 or 1. We make the k choices independently, so we can simply multiply the number of choices for each bit.

$$\overbrace{2 \cdot 2 \cdot \cdots \cdot 2}^{k \text{ times}} = 2^k$$

There are, therefore, 2^k different combinations of choices.

Computers often deal with English text, so it would be nice to have some way of assigning distinct values to represent each possible printable character on the keyboard. (We consider lower-case and capital letters as different characters.) How many bits do we need to represent the keyboard characters?

If you count the symbols on the keyboard, you'll find that there are 94 printable characters. Six bits don't provide enough distinct values, since they provide only $2^6 = 64$ different combinations, but seven is plenty ($2^7 = 128$). Thus, we can represent each of the 94 possible printable characters using seven bits. Of course, seven bits actually permit up to 128 different values; the extra 34 can be dedicated to the space key, the enter key, the tab key, and other convenient characters.

Many computers today use **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) for reprenting characters. This seven-bit coding defines the space as 0100000, the capital A as 1000001, the zero digit as 0110000, and so on. Figure 3.1 contains a complete table of the ASCII codes. (Most of the extra 34 values, largely obscure abbrevations in the first column of Figure 3.1, are rarely used today.)

Computers deal with characters often enough that designers organize computer data around the size of their representation. Computer designers prefer powers of two, however. (This preference derives from the fact that k bits can represent up to 2^k different values.) Thus, they don't like dividing memory up into units of seven bits (as required for the ASCII code). Instead, they break memory into groups of eight bits ($2^3 = 8$), each of which they call a **byte**. The extra bit is left unused when representing characters using ASCII.

When they want to talk about large numbers of bytes, computer designers group them into **kilobytes** (KB). The prefixes *kilo-* come from the metric prefix for a thousand (as in *kilometer* and *kilogram*) However, because computers deal in binary, it's more convenient to deal with powers of 2, and so the prefix *kilo-* in kilobyte actually stands for the closest power of 2 to 1000, which is $2^{10} = 1024$. These abbreviations extend upward.*

kilobyte	KB	$2^{10}=$	1,024 bytes
megabyte	MB	$2^{20}=$	1,048,576 bytes
gigabyte	GB	$2^{30}=$	1.074 billion bytes
terabyte	TB	$2^{40}=$	1.100 trillion bytes

3.1.2 Representing numbers

We can already represent integers from zero to one using a single bit. To represent larger numbers, we need to use combinations of bits. The most convenient technique for assigning values to combinations is based on **binary notation** (also called **base 2**).

You're already familiar with **decimal notation** (also called **base 10**). You may remember the following sort of diagram from grade school.

$$\underset{1000}{1}\ \underset{100}{0}\ \underset{10}{2}\ \underset{1}{4}$$

That is, in representing the number 1024, we put a 4 in the ones place, a 2 in the tens place, a 0 in the hundreds places, and a 1 in the thousands place. This system is called *base 10* because there are 10 possible

*Sometimes, manufacturers use powers of 10 instead of 2 for marketing purposes. Thus, they may advertise a hard disk as holding 40 GB, when it actually holds only 37.3 GB, or 40 billion bytes.

0	0000000	*NUL*	32	0100000	*SP*	64	1000000	@	96	1100000	‘
1	0000001	*SOH*	33	0100001	!	65	1000001	A	97	1100001	a
2	0000010	*STX*	34	0100010	"	66	1000010	B	98	1100010	b
3	0000011	*ETX*	35	0100011	#	67	1000011	C	99	1100011	c
4	0000100	*EOT*	36	0100100	$	68	1000100	D	100	1100100	d
5	0000101	*ENQ*	37	0100101	%	69	1000101	E	101	1100101	e
6	0000110	*ACK*	38	0100110	&	70	1000110	F	102	1100110	f
7	0000111	*BEL*	39	0100111	’	71	1000111	G	103	1100111	g
8	0001000	*BS*	40	0101000	(	72	1001000	H	104	1101000	h
9	0001001	*HT*	41	0101001	)	73	1001001	I	105	1101001	i
10	0001010	*NL*	42	0101010	*	74	1001010	J	106	1101010	j
11	0001011	*VT*	43	0101011	+	75	1001011	K	107	1101011	k
12	0001100	*NP*	44	0101100	,	76	1001100	L	108	1101100	l
13	0001101	*CR*	45	0101101	-	77	1001101	M	109	1101101	m
14	0001110	*SO*	46	0101110	.	78	1001110	N	110	1101110	n
15	1001111	*SI*	47	0101111	/	79	1001111	O	111	1101111	o
16	0010000	*DLE*	48	0110000	0	80	1010000	P	112	1110000	p
17	0010001	*DC*1	49	0110001	1	81	1010001	Q	113	1110001	q
18	0010010	*DC*2	50	0110010	2	82	1010010	R	114	1110010	r
19	0010011	*DC*3	51	0110011	3	83	1010011	S	115	1110011	s
20	0010100	*DC*4	52	0110100	4	84	1010100	T	116	1110100	t
21	0010101	*NAK*	53	0110101	5	85	1010101	U	117	1110101	u
22	0010110	*SYN*	54	0110110	6	86	1010110	V	118	1110110	v
23	0010111	*ETB*	55	0110111	7	87	1010111	W	119	1110111	w
24	0011000	*CAN*	56	0111000	8	88	1011000	X	120	1111000	x
25	0011001	*EM*	57	0111001	9	89	1011001	Y	121	1111001	y
26	0011010	*SUB*	58	0111010	:	90	1011010	Z	122	1111010	z
27	0011011	*ESC*	59	0111011	;	91	1011011	[	123	1111011	{
28	0011100	*FS*	60	0111100	<	92	1011100	\	124	1111100	|
29	0011101	*GS*	61	0111101	=	93	1011101	]	125	1111101	}
30	0011110	*RS*	62	0111110	>	94	1011110	^	126	1111110	~
31	0011111	*US*	63	0111111	?	95	1011111	_	127	1111111	*DEL*

Figure 3.1: The ASCII character assignments.

digits for each place (0 through 9) and because the place values go up by factors of 10 ($1 \to 10 \to 100 \to 1000 \cdots$).

In binary notation, we have only two digits (0 and 1) and the place values go up by factors of 2. So we have a ones place, a twos place, a fours place, an eights place, a sixteens place, and so on. The following diagrams a number written in binary notation.

$$\underset{8}{1}\ \underset{4}{0}\ \underset{2}{1}\ \underset{1}{1}$$

This value, $1011_{(2)}$, represents a number with 1 eight, 0 fours, 1 two, and 1 one: $1{\cdot}8{+}0{\cdot}4{+}1{\cdot}2{+}1{\cdot}1 = 11_{(10)}$. (The parenthesized subscripts indicate whether the number is in binary notation or decimal notation.)

We'll often want to convert numbers between their binary and decimal representations. We saw how to convert binary to decimal with $1011_{(2)}$. Here's another example: Suppose we want to identify what $100100_{(2)}$ represents. We first determine what places the 1's occupy.

$$\underset{32}{1}\ \underset{16}{0}\ \underset{8}{0}\ \underset{4}{1}\ \underset{2}{0}\ \underset{1}{0}$$

We then add up the values of these places to get a base-10 value: $32 + 4 = 36_{(10)}$.

To convert a number from decimal to binary, we repeatedly determine the largest power of two that fits into the number and subtract it, until we reach zero. The binary representation has a 1 bit in each place whose value we subtracted. Suppose, as an example, we want to convert $88_{(10)}$ to binary. We observe the largest power of 2 less than 88 is 64, so we decide that the binary expansion of 88 has a 1 in the 64's place, and we subtract 64 to get $88 - 64 = 24$. Then we see than the largest power of 2 less than 24 is 16, so we decide to put a 1 in the 16's place and subtract 16 from 24 to get 8. Now 8 is the largest power of 2 that fits into 8, so we put a 1 in the 8's place and subtract to get 0. Once we reach 0, we write down which places we filled with 1's.

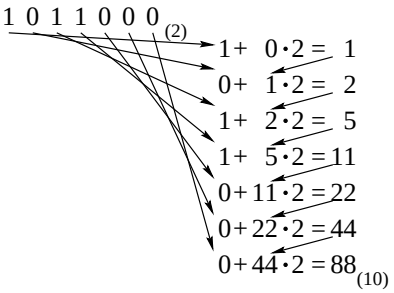
$$\underset{64}{1}\ \underset{32}{\_}\ \underset{16}{1}\ \underset{8}{1}\ \underset{4}{\_}\ \underset{2}{\_}\ \underset{1}{\_}$$

We put a zero in each empty place and conclude that the binary representation of $88_{(10)}$ is $1011000_{(2)}$.

3.1.3 Alternative conversion algorithms

In the previous section, we saw a procedure (an *algorithm*) for converting between binary notation and decimal notation, and we saw another procedure for converting between decimal notation and binary notation. Those algorithms work well, but there are alternative algorithms for each that some people prefer.

From binary To convert a number written in binary notation to decimal notation, we begin by thinking "0," and we go through the binary representation left-to-right, each time adding that bit to twice the number we are thinking. Suppose, for example, that we want to convert $1011000_{(2)}$ into decimal notation.



We end up with the answer $88_{(10)}$.

This algorithm is based on the following reasoning. A five-bit binary number $10110_{(2)}$ corresponds to $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. This latter expression is equivalent to the polynomial

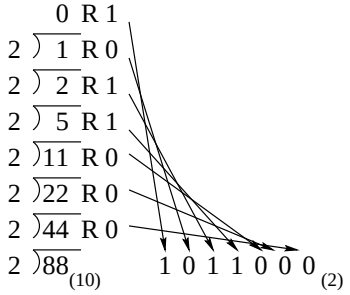
$$ax^4 + bx^3 + cx^2 + dx + e\,,$$

where $a = 1$, $b = 0$, $c = 1$, $d = 1$, $e = 0$, and $x = 2$. We can convert this polynomial into an alternative form.[†]

$$ax^4 + bx^3 + cx^2 + dx + e = (((ax + b)x + c)x + d)x + e$$

In the algorithm, we're computing based on the alternative form instead of the original polynomial.

To binary To convert a number in the opposite direction, we repeatedly divide a number by 2 until we reach 0, each time taking note of the remainder. When we string the remainders in reverse order, we get the binary representation of the original number. For example, suppose we want to convert $88_{(10)}$ into binary notation.



After going through the repeated division and reading off the remainders, we arrive at $1011000_{(2)}$.

Understanding how this process works is simply a matter of observing that it reverses the double-and-add algorithm we just saw for converting from binary to decimal.

3.1.4 Other bases

Binary notation, which tends to lead to very long numerical representations, is cumbersome for humans to remember and type. But using decimal notation obscures the relationship to the individual bits. Thus, when the identity of the individual bits is important, computer scientists often compromise by using a power of two as the base: The popular alternatives are base eight (**octal**) and base sixteen (**hexadecimal**).

The nice thing about these bases is how easily they translate into binary. Suppose we want to convert $173_{(8)}$ to binary. One possible way is to convert this number to base 10 and then convert that answer to binary. But doing a direct conversion turns out to be much simpler: Since each octal digit corresponds to a three-bit binary sequence, we can replace each octal digit of $173_{(8)}$ with its three-bit sequence.

$$\underset{64}{1}\,\underset{8}{7}\,\underset{1}{3} = \underset{64}{001}\,\underset{8}{111}\,\underset{1}{011}$$

Thus, we conclude $173_{(8)} = 001111011_{(2)}$.

To convert the other way, we split the binary number we want to convert into groups of three (starting from the 1's place), and then we replace each three-bit sequence with its corresponding octal digit. Suppose we want to convert $1011101_{(2)}$ to octal.

$$\underset{64}{1} \,\Big|\, \underset{32}{0}\,\underset{16}{1}\,\underset{8}{1} \,\Big|\, \underset{4}{1}\,\underset{2}{0}\,\underset{1}{1} = \underset{64}{1}\,\underset{8}{3}\,\underset{1}{5}$$

[†]The first known description of this is in 1299 by a well-known Chinese mathematician Chu Shih-Chieh (1270?–1330?). An obscure Englishman, William George Horner (1786–1837), later rediscovered the principle known today as Horner's method.

Thus, we conclude $1011101_{(2)} = 135_{(8)}$.

Hexademical, frequently called *hex* for short, works the same way, except that we use groups of four bits instead. One slight complication is that hexadecimal requires 16 different digits, and we have only 10 available. Computer scientists use Latin letters to fill the gap. Thus, after *0* through *9* come *A* through *F*.

0	0	0000	4	4	0100	8	8	1000	C	12	1100
1	1	0001	5	5	0101	9	9	1001	D	13	1101
2	2	0010	6	6	0110	A	10	1010	E	14	1110
3	3	0011	7	7	0111	B	11	1011	F	15	1111

As an example of a conversion from hex to decimal, suppose we want to convert the number $F5_{(16)}$ to base 10. We would replace the F with $1111_{(2)}$ and the 5 with $0101_{(2)}$, giving its binary equivalent $11110101_{(2)}$.

3.2 Integer representation

We can now examine how computers remember integers on the computer. (Recall that integers are numbers with no fractional part, like 2, 105, or -38.)

3.2.1 Unsigned representation

Modern computers usually represent numbers in a fixed amount of space. For example, we might decide that each byte represents a number. A byte, however, is very limiting: The largest number we can fit is $11111111_{(2)} = 255_{(10)}$, and we often want to deal with larger numbers than that.

Thus, computers tend to use groups of bytes called **words**. Different computers have different word sizes. Very simple machines have 16-bit words; today, most machines use 32-bit words, though some computers use 64-bit words. (The term *word* comes from the fact that four bytes (32 bits) is equivalent to four ASCII characters, and four letters is the length of many useful English words.) Thirty-two bits is plenty for most numbers, as it allows us to represent any integer from 0 up to $2^{32} - 1 = 4{,}294{,}967{,}295$. But the limitation is becoming increasingly irritating, and so people are beginning to move to 64-bit computers. (This isn't because people are dealing with larger numbers today than earlier, so much as the fact that memory has become much cheaper, and so it seems silly to continue trying to save money by skimping on bits.)

The representation of an integer using binary representation in a fixed number of bits is called an **unsigned representation**. The term comes from the fact that the only numbers representable in the system have no negative sign.

But what about negative integers? After all, there are some perfectly respectable numbers below 0. We'll examine two techniques for representing integers both negative and positive: sign-magnitude representation and two's-complement representation.

3.2.2 Sign-magnitude representation

Sign-magnitude representation is the more intuitive technique. Here, we let the first bit indicate whether the number is positive or negative (the number's *sign*), and the rest tells how far the number is from 0 (its *magnitude*). Suppose we are working with 8-bit sign-magnitude numbers.

3 would be represented as 00000011
-3 would be represented as 10000011

For $-3_{(2)}$, we use 1 for the first bit, because the number is negative, and then we place 3 into the remaining seven bits.

What's the range of integers we can represent with an 8-bit sign-magnitude representation? For the largest number, we'd want 0 for the sign bit and 1 everywhere else, giving us 01111111, or $127_{(10)}$. For the smallest number, we'd want 1 for the sign bit and 1 everywhere else, giving us $-127_{(10)}$. An 8-bit sign-magnitude representation, then, can represent any integer from $-127_{(10)}$ to $127_{(10)}$.

This range of integers includes 255 values. But we've seen that 8 bits can represent up to 256 different values. The discrepency arises from the fact that the representation includes two representations of the number zero (0 and -0, represented as 00000000 and 10000000).

Arithmetic using sign-magnitude representation is somewhat more complicated than we might hope. When you want to see if two numbers are equal, you would need additional circuitry so that -0 is understood as equal to 0. Adding two numbers requires circuitry to handle the cases of when the numbers' signs match and when they don't match. Because of these complications, sign-magnitude representation is not often used for representing integers. We'll see it again, however, when we get to floating-point numbers in Section 3.3.2.

3.2.3 Two's-complement representation

Nearly all computers today use the **two's-complement representation** for integers. In the two's-complement system, the topmost bit's value is the negation of its meaning in an unsigned system. For example, in an 8-bit unsigned system, the topmost bit is the 128's place.

$$\overline{128}\ \overline{64}\ \overline{32}\ \overline{16}\ \overline{8}\ \overline{4}\ \overline{2}\ \overline{1}$$

In an 8-bit two's-complement system, then, we negate the meaning of the topmost bit to be -128 instead.

$$\overline{-128}\ \overline{64}\ \overline{32}\ \overline{16}\ \overline{8}\ \overline{4}\ \overline{2}\ \overline{1}$$

To represent the number $-100_{(10)}$, we would first choose a 1 for the -128's place, leaving us with $(-100) - (-128) = 28$. (We are using the repeated subtraction algorithm described in Section 3.1.2. Since the place value is negative, we subtract a negative number.) Then we'd choose a 1 for the 16's place, the 8's place, and the 4's place to reach 0.

$$\underset{-128}{1}\ \underset{64}{0}\ \underset{32}{0}\ \underset{16}{1}\ \underset{8}{1}\ \underset{4}{1}\ \underset{2}{0}\ \underset{1}{0}$$

Thus, the 8-bit two's-complement representation of $-100_{(10)}$ would be 10011100.

3 would be represented as 00000011
-3 would be represented as 11111101

What's the range of numbers representable in an 8-bit two's-complement representation? To arrive at the largest number, we'd want 0 for the -128's bit and 1 everywhere else, giving us 01111111, or $127_{(10)}$. For the smallest number, we'd want 1 for the -128's bit and 0 everywhere else, giving $-128_{(10)}$. In an 8-bit two's-complement representation, we can represent any integer from $-128_{(10)}$ up to $127_{(10)}$. (This range includes 256 integers. There are no duplicates as with sig-magnitude representation.)

It's instructive to map out the bit patterns (in order of their unsigned value) and their corresponding two's-complement values.

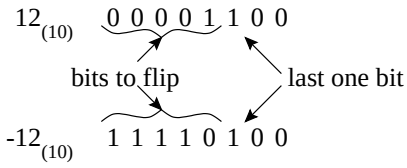
bit pattern	value
00000000	0
00000001	1
00000010	2
⋮	
01111110	126
01111111	127
10000000	−128
10000001	−127
⋮	
11111110	−2
11111111	−1

Notice that the two's-complement representation wraps around: If you take the largest number, 01111111, and add 1 to it as if it were an unsigned number, and you get 10000000, the smallest number. This wrap-around behavior can lead to some interesting behavior. In one game I played as a child (back when 16-bit computers were popular), the score would go up progressively as you guided a monster through a maze. I wasn't very good at the game, but my little brother mastered it enough that the score would hit its maximum value and then wrap around to a very negative value! Trying to get the largest possible score — without wrapping around — was an interesting challenge.

Negating two's-complement numbers

For the sign-magnitude representation, it's easy to represent the negation of a number: You just flip the sign bit. For the two's-complement representation, however, the relationship between the representation of a number and the representation of its negation is not as obvious as one might like.

The following is a handy algorithm for relating the representation of a number to the representation of its negation: You start at the right, copying bits down, until you reach the first 1, beyond which you flip every bit. The representation of 12, for example, is 00001100, so its two's complement representation will be 11110100 — the lower three bits (below and including the lowest 1) are identical, while the rest are all different.



Why this works is not immediately obvious. To understand it, we first need to observe that if we have a negative number $-x$ and we interpret its two's-complement representation in the unsigned format, we end up with $256 - x$. This is because the two's-complement representation of $-x$ will have a 1 in the uppermost bit, representing -128, but when we interpret it as an unsigned number, we interpret this bit as representing 128, which is 256 more than before. The other bits' values remain unchanged. Since the value of this uppermost bit has increased by 256, the unsigned interpretation of the bit pattern is worth 256 more than the two's-complement interpretation.

We can understand our negation algorithm as being a two-step process.

1. We flip all the bits (from 00001100 becomes 11110011, for example).

2. We add one to the number (which would give 11110100 in our example).

Adding one to a number flips bits from the right, until we reach a zero. Since we already flipped all the bits in the first step, this second step flips these bits back to their original values.

Now we can observe that if the original number is x when interpreted as an unsigned number, we will have $256 - x$ after going through the process. The first step of flipping all bits is equivalent to subtracting each bit from 1, which is the same as subtracting x from the all-ones number (255). Thus, after the first step, we have the value $255 - x$. The second step adds one to this, giving us $(255 - x) + 1 = 256 - x$. When the unsigned representation of $256 - x$ is interpreted as a two's-complement number, we understand it to be $-x$.

Adding two's-complement numbers

One of the nice things about two's-complement numbers is that you can add them just as you add regular numbers. Suppose we want to add -3 and 5.

$$\begin{array}{rr} & 1001011 \\ + & 0011001 \\ \hline \end{array}$$

We can attempt to do this using regular addition, akin to the technique we traditionally use in adding base-10 numbers.

$$\begin{array}{rr} & 11111\ 1 \\ & 11111101 \\ + & 00000101 \\ \hline & 100000010 \end{array}$$

We get an extra 1 in the ninth bit of the answer, but if we ignore this ninth bit, we get the correct answer.

We can reason that this is correct as follows: Say one of the two numbers is negative and the other is positive. That is, one of the two numbers has a 1 in the -128's place, and the other has 0 there. If there is no carry into the -128's place, then the answer is OK, because that means we got the correct sum in the last 7 bits, and then when we add the -128's place, we'll maintain the -128 represented by the 1 in that location in the negative number.

If there is a carry into the -128's place, then this represents a carry of 128 taken from summing the 64's column. This carry of 128 (represented by a carry of 1), added to the -128 in that column for the negative number (represented by a 1 in that column), should give us 0. This is exactly what we get we add the carry of 1 into the leftmost column to the 1 of the negative number in this column and then throw away the carry ($1 + 1 = \not{1}0_2$).

A similar sort of analysis will also work when we are adding two negative numbers or two positive numbers. Of course, the addition only works if you end up with something in the representable range. If you add 120 and 120 with 8-bit two's-complement numbers, then the result of 240 won't fit. The result of adding 120 and 120 as 8-bit numbers would turn out to be -16!

For dealing with integers that can possibly be negative, computers generally use two's-complement representation rather than sign-magnitude. It is somewhat less intuitive, but it allows simpler arithmetic. (It is true that negation is somewhat more complex, but only slightly so.) Moreover, the two's-complement representation avoids the problem of multiple representations of the same number (0).

3.3 General numbers

Representing numbers as fixed-length integers has some notable limitations. It isn't suitable for very large numbers that don't fit into 32 bits, like 6.02×10^{23}, nor can it handle numbers that have a fraction, like 3.14. We'll now turn to systems for handling a wider range of numbers.

3.3.1 Fixed-point representation

One possibility for handling numbers with fractional parts is to add bits after the decimal point: The first bit after the decimal point is the halves place, the next bit the quarters place, the next bit the eighths place, and so on.

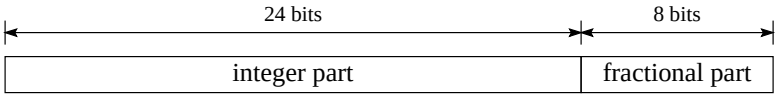
$$\frac{}{4}\ \frac{}{2}\ \frac{}{1}\ .\ \frac{}{\frac{1}{2}}\ \frac{}{\frac{1}{4}}\ \frac{}{\frac{1}{8}}$$

Suppose that we want to represent $1.625_{(10)}$. We would want 1 in the ones place, leaving us with 0.625. Then we want 1 in the halves place, leaving us with $0.625 - 0.5 = 0.125$. No quarters will fit, so put a 0 there. We want a 1 in the eighths place, and we subtract 0.125 from 0.125 to get 0.

$$\frac{0}{4}\ \frac{0}{2}\ \frac{1}{1}\ .\ \frac{1}{\frac{1}{2}}\ \frac{0}{\frac{1}{4}}\ \frac{1}{\frac{1}{8}}$$

So the binary representation of 1.625 would be $1.101_{(2)}$.

The idea of **fixed-point representation** is to split the bits of the representation between the places to the left of the decimal point and places to the right of the decimal point. For example, a 32-bit fixed-point representation might allocate 24 bits for the integer part and 8 bits for the fractional part.



To represent 1.625, we would then write

$$00000000\,00000000\,00000001\,10100000\ .$$

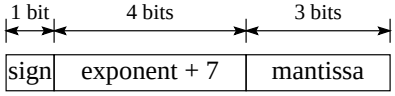
The first three bytes give the 1, and the last byte gives the representation of 0.625.

Fixed-point representation works well as long as you work with numbers within the given range. The 32-bit fixed-point representation described above can represent any multiple of $\frac{1}{256}$ from 0 up to $2^{24} \approx 16.7$ million. But programs frequently need to work with numbers from a much broader range. For this reason, fixed-point representation isn't used very often in today's computing world.[‡]

3.3.2 Floating-point representation

Floating-point representation is an alternative technique based on scientific notation. Because we're working with computers, we'll base our scientific notation on powers of 2, not 10 as is traditional. For example, the binary representation of $5.5_{(10)}$ is $101.1_{(2)}$. When we convert this to binary scientific notation, we move the decimal point to the left two places, giving us $1.011_{(2)} \times 2^2$. (This is just like converting $101.1_{(10)}$ to scientific notation: It would be $1.011_{(10)} \times 10^2$.)

To represent a number written in scientific notation in bits, we'll decide how to split up the representation to fit it into a fixed number of bits. First, let us define the two parts of scientific representation: In $1.011_{(2)} \times 10^2$, we call $1.011_{(2)}$ the **mantissa** (or the **significand**), and we call 2 the **exponent**. In this section we'll use 8 bits to store such a number, divided as follows.



[‡]Financial software is a notable exception. Here, designers often want all computations to be precise to the penny, and in fact they should always be rounded to the nearest penny. There is no reason to deal with very large amounts (like trillions of dollars) or fractions of a penny. Such programs use a variant of fixed-point representation that represents each amount as an integer multiple of $\frac{1}{100}$, just as the fixed-point representation represents numbers as an integer multiple of $\frac{1}{256}$.

We use the first bit to represent the sign (1 for negative, 0 for positive), the next four bits for the sum of 7 and the actual exponent (we add 7 to allow for negative exponents), and the last three bits for the fraction of the mantissa. Note that we omit the digit to the left of the decimal point: Since the mantissa has only one nonzero bit to the left of the decimal point, and the only nonzero bit is 1, we know that the bit to the left of the decimal point must be a 1. There's no point in wasting space in inserting this 1 into our bit pattern. We include only the bits of the mantissa to the right of the decimal point.

We call this a *floating-point representation* because the values of the mantissa bits "float" along with the decimal point, based on the exponent's given value. This is in contrast to *fixed-point* representation, where the decimal point is always in the same place among the bits given.

Continuing our example of $5.5_{(10)} = 1.011_{(2)} \times 2^2$, we add 7 to 2 to arrive at $9_{(10)} = 1001_{(2)}$ for the exponent bits. Into the mantissa bits we place the bits following the decimal point of the scientific notation, 011. This gives us

$$0\,1001\,011$$

as the 8-bit floating-point representation of $5.5_{(10)}$.

Suppose we want to represent $-96_{(10)}$.

1. First we convert our desired number to binary: $-1100000_{(2)}$.
2. Then we convert this to binary scientific notation: $-1.100000_{(2)} \times 2^6$.
3. Then we fit this into the bits.
   (a) We choose 1 for the sign bit if the number is negative. (It is, in this case.)
   (b) We add 7 to the exponent and place the result into the four exponent bits. (In this case, we arrive at $6 + 7 = 13_{(10)} = 1101_{(2)}$.)
   (c) The three mantissa bits are the first three bits following the leading 1: 100. (If there are more than three bits, then rounding will be necessary.)

   Thus we end up with $1\,1101\,100$.

Conversely, suppose we want to decode the number $0\,0101\,100$.

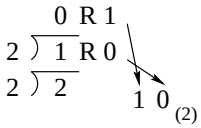
1. We observe that the number is positive, and the exponent bits represent $0101_{(2)} = 5_{(10)}$. This is 7 more than the actual exponent, and so the actual exponent must be -2. Thus, in binary scientific notation, we have $1.100_{(2)} \times 2^{-2}$.
2. We convert this to binary: $1.100_{(2)} \times 2^{-2} = 0.011_{(2)}$.
3. We convert the binary into decimal: $0.011_{(2)} = \frac{1}{4} + \frac{1}{8} = \frac{3}{8} = 0.375_{(10)}$.

Alternative conversion algorithm

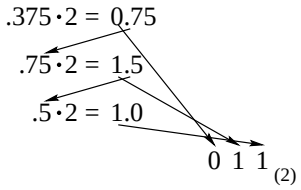
The process described above for converting from decimal to binary representation relies implicitly on the repeated subtraction algorithm of Section 3.1.2. For example, we arrive at $101.1_{(2)}$ for $5.5_{(10)}$ by subtracting 4, then 1, then 0.5. If we wanted to convert $2.375_{(10)}$, we would choose a 1 for the 2's place (leaving 0.375), the $1/4$'s place (leaving 0.125), and the $1/8$'s place (leaving 0), giving us $10.011_{(2)}$.

Alternatively, we can use a process inspired by the repeated division algorithm of Section 3.1.3. Here, we convert to binary in two steps.

- We take the portion to the left of the decimal point and use the repeated division algorithm of Section 3.1.3. In the case of $2.375_{(10)}$, we would take the 2 to the left and repeatedly divide until we reach 0, reading the remainders to arrive at $10_{(2)}$.



- We take the portion to the right of the decimal point and repeatedly multiply it by 2, each time extracting the bit to the right of the decimal point, until we reach 0 (or until we have plenty of bits to fill out the mantissa bits). In the example of $2.375_{(10)}$, the fractional part is $.375_{(10)}$. We repeatedly multiply this by 2, each time taking out the integer part as the next bit of the binary fraction, arriving at 011.



  These bits are the bits following the decimal point in the binary representation. Placed with the bits before the decimal point determined in the previous step, we would conclude with $1.011_{(2)}$.

Representable numbers

This 8-bit floating-point format can represent a wide range of both small numbers and large numbers. To find the smallest possible positive number we can represent, we would want the sign bit to be 0, we would place 0 in all the exponent bits to get the smallest exponent possible, and we would put 0 in all the mantissa bits. This gives us 0 0000 000, which represents

$$1.000_{(2)} \times 2^{0-7} = 2^{-7} \approx 0.0078_{(10)} .$$

To determine the largest positive number, we would want the sign bit still to be 0, we would place 1 in all the exponent bits to get the largest exponent possible, and we would put 1 in all the mantissa bits. This gives us 0 1111 111, which represents

$$1.111_{(2)} \times 2^{15-7} = 1.111_{(2)} \times 2^{8} = 111100000_{(2)} = 480_{(10)} .$$

Thus, our 8-bit floating-point format can represent positive numbers from about $0.0078_{(10)}$ to $480_{(10)}$. In contrast, the 8-bit two's-complement representation can only represent positive numbers between 1 and 127.

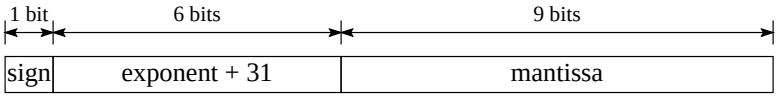
But notice that the floating-point representation can't represent all of the numbers in its range — this would be impossible, since eight bits can represent only $2^8 = 256$ distinct values, and there are infinitely many real numbers in the range to represent. Suppose we tried to represent $17_{(10)}$ in this scheme. In binary, this is $10001_{(2)} = 1.0001_{(2)} \times 2^4$. When we try to fit the mantissa into the mantissa portion of the 8-bit representation, we find that the final 1 won't fit: We would be forced to round. In this case, computers would ignore the final 1 and use the pattern 0 1011 000.[§] That rounding means that we're not representing the number precisely. In fact, 1 1011 000 translates to

$$1.000_{(2)} \times 2^{11-7} = 1.000_{(2)} \times 2^{4} = 10000_{(2)} = 16_{(10)} .$$

[§]Computers generally round to the nearest possibility. But, when we are exactly between two possibilities, as in this case, most computers follow the policy of rounding so that the final mantissa bit is 0. This detail of exactly how the rounding occurs is not important to our discussion, however.

Thus, in our 8-bit floating-point representation, 17 *equals* 16! That's pretty irritating, but it's a price we have to pay if we want to be able to handle a large range of numbers with such a small number of bits.

While a floating-point representation can't represent all numbers precisely, it *does* give us a guaranteed number of significant digits. For this 8-bit representation, we get a single digit of precision, which is pretty limited. To get more precision, we need more mantissa bits. Suppose we defined a similar 16-bit representation with 1 bit for the sign bit, 6 bits for the exponent plus 31, and 9 bits for the mantissa.



This representation, with its 9 mantissa bits, happens to provide three significant digits. Given a limited length for a floating-point representation, we have to compromise between more mantissa bits (to get more precision) and more exponent bits (to get a wider range of numbers to represent). For 16-bit floating-point numbers, the 6-and-9 split is a reasonable tradeoff of range versus precision.

IEEE standard

Nearly all computers today follow the the **IEEE standard**, published in 1980, for representing floating-point numbers. This standard is similar to the 8-bit and 16-bit formats we've explored already, but the standard deals with longer lengths to gain more precision and range. There are three major varieties of the standard, for 32 bits, 64 bits, and 80 bits.

format	sign bits	exponent bits	mantissa bits	exponent excess	significant digits
Our 8-bit	1	4	3	7	1
Our 16-bit	1	6	9	31	3
IEEE 32-bit	1	8	23	127	6
IEEE 64-bit	1	11	52	1,023	15
IEEE 80-bit	1	15	63	16,383	19

All of these formats use an offset for the exponent, called the **excess**. In all of these formats, the excess is halfway up the range of numbers that can fit into the exponent bits. For the 8-bit format, we had 4 exponent bits; the largest number that can fit into 4 bits is $2^4 - 1 = 15$, and so the excess is $7 \approx 15/2$. The IEEE 32-bit format has 8 exponent bits, and so the largest number that fits is 255, and the excess is $127 \approx 255/2$.

The IEEE standard formats *generally* follow the rules we've outlined so far, but there are two exceptions: the denormalized numbers and the nonnumeric values. We'll look at these next.

Denormalized numbers (optional)

The first special case is for dealing with very small values. Let's go back to the 8-bit representation we've been studying. If we plot the small numbers that can be represented exactly on the number line, we get the distribution illustrated in Figure 3.2(a). The smallest representable positive number is $2^{-7} = \frac{1}{128}$ (bit pattern 00000000), and the largest representable negative number is $-2^{-7} = \frac{-1}{128}$ (bit pattern 10000000). These are small numbers, but when we look at Figure 3.2(a), we see an anomaly: There is a relatively large gap between them. And — notice — there is no exact representation of one of the most important numbers of all: zero!

To deal with this, the IEEE standard defines the **denormalized numbers**. The idea is to take the most closely clustered numbers illustrated in Figure 3.2(a) and spread them more evenly across 0. This will give us the diagram in Figure 3.2(b).

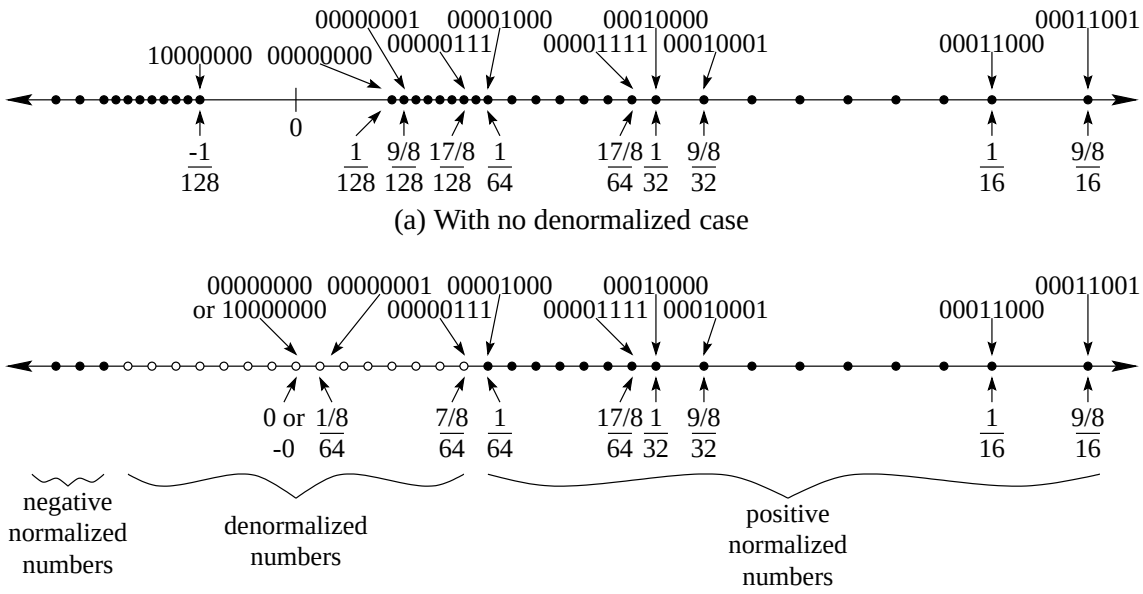


Figure 3.2: Distribution of small floating-point numbers with and without a denormalized case.

Those closely-clustered numbers in Figure 3.2(a) are those whose exponent bits are all 0. We'll change the meanings of these numbers as follows: When all exponent bits are 0, then the exponent is -6, and the mantissa has an implied 0 before it.[¶] Consider the bit pattern 0 0000 010: In a floating-point format incorporating the denormalized case, this represents $0.010_{(2)} \times 2^{-6} = 1 \times 2^{-8} = \frac{1}{256}$. (Without the denormalized case, this would represent $1.010_{(2)} \times 2^{-7}$. The changes are in the bit before the mantissa's decimal point and in the exponent of -7.)

Suppose we want to represent $0.005_{(10)}$ in our 8-bit floating-point format with a denormalized case. We first convert our number into the form $x \times 2^{-6}$. In this case, we would get $0.320_{(10)} \times 2^{-6}$. Converting $0.320_{(10)}$ to binary, we get approximately $0.0101001_{(2)}$. In the 8-bit format, however, we have only three mantissa bits, and so we would round this to $0.011_{(2)}$. Thus, we have $0.011_{(2)} \times 2^{-6}$, and so our bit representation would be 0 0000 011. This is just an approximation to the original number of $0.005_{(10)}$: It is about $0.00586_{(10)}$. Without the denormalized case, the best approximation would be much further off ($0.00781_{(10)}$).

How would we represent 0? We go through the same process: Converting this into the form $x \times 2^{-6}$, we get 0.0×2^{-6}. This translates into the bit representation 0 0000 000.

Why -6 for the exponent? It would make more intuitive sense to use -7, since this is what the all-zeroes exponent is normally. We use -6, however, because we want a smooth transition between the normalized values and the denormalized values. The least positive normalized value is 1×2^{-6} (bit pattern 0 0000 000). If we used -7 for the denormalized exponent, then the largest denormalized value would be $0.111_{(2)} \times 2^{-7}$, which is roughly half of the smallest positive normalized value. By using the same exponent as for the smallest normalized case, the standard spreads the denormalized numbers evenly from the smallest positive normalized number to 0. Figure 3.2(b) diagrams this: The open circles, representing values handled by the denormalized case, are spread evenly between the solid circles, representing the numbers handled by the normalized case.

[¶] The word *denormalized* comes from the fact that the mantissa is not in its *normal* form, where a nonzero digit is to the left of the decimal point.

The denormalized case works the same for the IEEE standard floating-point formats, except that the exponent varies based on the format's excess. In the 32-bit standard, for example, the denormalized case is still the case when all exponent bits are zero, but the exponent it represents is -126 (since the normalized case involves an excess-127 exponent, and so the lowest exponent for normalized numbers is $1 - 127 = -126$).

Nonnumeric values (optional)

The IEEE standard's designers were concerned with some special cases — particularly, computations where the answer doesn't fit into the range of defined numbers. To address such possibilities, they reserved the all-ones exponent for the **nonnumeric values**. They designed two types of nonnumeric values into the IEEE standard.

- If the exponent is all ones and the mantissa is all zeroes, then the number represents infinity or negative infinity, depending on the sign bit. Essentially, these two values are to represent numbers that have gone out of bounds. This value results from an overflow; for example, if you doubled the largest positive value, you would get infinity. Or if you divide 1 by a tiny number, you would get either infinity or negative infinity.
- If the exponent is all ones, and the mantissa has some non-zero bits, then the number represents "not a number," written as NaN. This represents an error condition; some situations where this occurs include finding the square root of -1, computing the tangent of $\pi/2$, and dividing 0 by 0.

3.4 Representing multimedia

Programs often deal with much more complex data than characters, integers, and real numbers. In this section, we'll look at the representation of images, video, and sound.

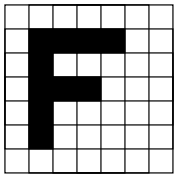
3.4.1 Images: The PNM format

Most techniques for representing images work by first breaking the image into a grid of **pixels**; each pixel is a small atom of color in the overall image. The word *pixel* comes from *pic*ture *el*ement. (This isn't the only way to represent an image: An important alternative is to represent the image by component shapes. This works well for computer-generated drawings, but it works poorly for photographs.)

There are many different formats for representing images. We'll look at one of the simplest: the PNM format. (PNM stands for *P*ortable a*N*y*M*ap.) In this format, we represent a picture as a sequence of ASCII characters.

```
P1
7 7
0 0 0 0 0 0 0
0 1 1 1 1 0 0
0 1 0 0 0 0 0
0 1 1 1 0 0 0
0 1 0 0 0 0 0
0 1 0 0 0 0 0
0 0 0 0 0 0 0
```

The file begins with the format type — in this case, it is "P1" to indicate a black-and-white image. The next line provides the width and height of the image in pixels. The rest of the file contains a sequence of ASCII 0's and 1's, representing each pixel starting in the upper left corner of the image and going in left-to-right, top-down order. A 1 represents a black pixel, while a 0 represents a white pixel. In this example, the image is a 7×7 image that looks like the following.



Representing images as text in this way is simple, but it is also very wasteful: Each pixel of the image requires 16 bits (one for the 0 or 1 ASCII code, and one for the ASCII code for the space that separates it from the next pixel's number). PNM has an alternative binary format where we spend only one bit per pixel. Here is an example; since an ASCII representation of the file would make no sense, we have to look at the binary contents directly.

byte	value	description
1	01010000	ASCII code for *P*
2	00110100	ASCII code for *4*
3	00001010	ASCII code for line break
4	00110111	ASCII code for *7*
5	00100000	ASCII code for space
6	00000111	ASCII code for *7*
7	00001010	ASCII code for line break
8	00000000	first eight pixels (all of row 1, plus a pixel of row 2)
9	11110001	next eight pixels (rest of row 2, plus 2 pixels of row 3)
10	00000011	next eight pixels (rest of row 3, plus 3 pixels of row 4)
11	10000100	next eight pixels (rest of row 4, plus 4 pixels of row 5)
12	00001000	next eight pixels (rest of row 5, plus 5 pixels of row 6)
13	00000000	next eight pixels (rest of row 6, plus 6 pixels of row 7)
14	00000000	last pixels (rest of row 7, the rest padded with 0's)

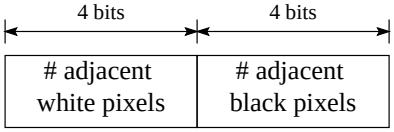
The file begins the same as with the ASCII file (except the header is now "P4" instead of "P1", which in the PNM format indicates that the data within the file is in binary (not ASCII) black-and-white format). When we reach the data describing individual pixels, however, it switches to packing the pixels together into bytes.

With the earlier ASCII version, the picture took 105 bytes to represent; this version takes only 14. This technique is *not* what the term *image compression* describes, however. For black-and-white images, the "uncompressed" representation is where you have one bit for each pixel, just as we have in this binary PNM format. A "compressed" image is one where the image is described using less than one bit per pixel.

3.4.2 Run-length encoding

Describing a black-and-white image with less than one bit per pixel may sound at first like an impossibility. It can be achieved in many cases, however, by taking advantage of the fact that useful images generally have some form of pattern.

We'll look at one particularly simple compression technique called **run-length encoding**. Suppose we begin our file the same as the PNM format, with a file format descriptor followed by the image's dimensions. When we get to describing the pixels' values, however, we repeatedly use bytes in the following format.



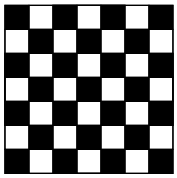
Here, we have four bits describing the number of adjacent white pixels as we go through the image in left-to-right, top-down order; then we have four bits describing the number of adjacent black pixels. [1]

If we were to encode the same *F* image we've been examining, it would look as follows. (This would not actually be a valid PNM file; the PNM format does not allow for such compression techniques.)

byte	value	description
1	01010000	ASCII code for *P*
2	00110111	ASCII code for *7*
3	00001010	ASCII code for line break
4	00110111	ASCII code for *7*
5	00100000	ASCII code for space
6	00000111	ASCII code for *7*
7	00001010	ASCII code for line break
8	10000100	image has eight white pixels, followed by four black pixels
9	00110001	image has three white pixels, followed by one black pixels
10	01100011	image has six white pixels, followed by three black pixels
11	01000001	image has four white pixels, followed by one black pixels
12	01100001	image has six white pixels, followed by one black pixels
13	11000000	image has twelve white pixels, followed by no black pixels

You can see that this scheme has shaved one byte from the uncompressed binary PNM format, which had 14 bytes for the same image.

Run-length encoding isn't always effective. Suppose we want to encode a checkerboard pattern.



Run-length encoding would look at this and see that each "run" of white pixels and of black pixels is only 1 pixel long. Thus our scheme would spend repeatedly use eight bits (00010001) to describe two adjacent pixels, whereas the uncompressed scheme would take only two bits for the same pixels. Thus the "compressed" image would actually be about four times larger than the uncompressed version! In many cases, however, run-length encoding turns out to be an effective compression technique, particularly in images that have large white or black regions.

Popular image formats, like the GIF and JPEG formats, usually do not use run-length encoding. They use more complex compression techniques that we will not explore here. Like run-length encoding, these techniques also take advantage of repetition in images, but they identify more complex repeated patterns in the image than the simple one-pixel repetition handled by run-length encoding.

3.4.3 General compression concepts

We've seen that run-length encoding isn't 100% effective. That is, sometimes it gives us something that is the same size or even longer than the original. This is disappointing. Why not study a *perfect* compression technique, one that always compresses a file?

There is a very good reason for this: Perfect compression is impossible. The following argument is a mathematical proof for this important fact.

[1] Using four bits, we can represent numbers only up to 15. If there are more than 15 adjacent pixels of the same color, we can describe them as several groups of 15 or fewer pixels, interleaved with groups of 0 pixels of the other color.

Theorem 2 *No perfect compression technique exists.*

Proof: Suppose you described a perfect compression technique to me, and I had an n-bit file I wanted to compress. I could apply your technique to my file n times, and each time your technique (being perfect) would give me a shorter file. I would end up with a zero-bit file. Now, to be a reasonable compression technique, there must be some corresponding decompression technique to arrive at the original. I can decompress this zero-bit file n times to arrive at the original file.

Now suppose I have a *different* n-bit file and I compress it n times. This will give me a zero-bit file again, and we've already seen that when we decompress a zero-bit file n times, it gives me the first file I compressed. Thus, a file compressed does not always decompress to the same thing; in other words, your proposed compression technique doesn't work.

Some image formats (including the JPEG format) use **lossy compression**. In lossy compression, we are willing to "lose" some of the original information for the sake of shorter files. The above proof relies on the fact that an "effective" compression algorithm must be able to restore a compressed file to the exact original. With lossy compression techniques, we forgo this requirement, and thus it's possible to have a lossy compression algorithm that always reduces a file's size.

The trick is to make a lossy compression algorithm that doesn't lose any important information. Luckily, images — particularly photographs — tend to be very rich in extraneous information: A picture of a tree, for example, would have a different shade of green for virtually every leaf, and we don't need all those different shades to understand we're looking at a tree.

3.4.4 Video

In principle, storing video isn't very different from images: You can simply store an image every $\frac{1}{24}$ seconds. (Since movies have 24 frames per second, this would give film-quality animation.)

The problem is that this eats up lots of space: Suppose we have a 90-minute 864×1152 video, where we represent each pixel with three bytes. (The 864×1152 resolution would give a 12×16-inch picture.) This would require

$$90 \text{ min} \times \frac{60 \text{ sec}}{\text{min}} \times \frac{24 \text{ frames}}{\text{sec}} \times \frac{864 \times 1152 \text{ pixels}}{\text{frame}} \times \frac{3 \text{ bytes}}{\text{pixel}} = 360 \text{ GB} .$$

We would need 23 DVDs to store the 90-minute movies. (A DVD can hold 15.9 GB.) If we wanted to use CDs, we would need 567 of them!

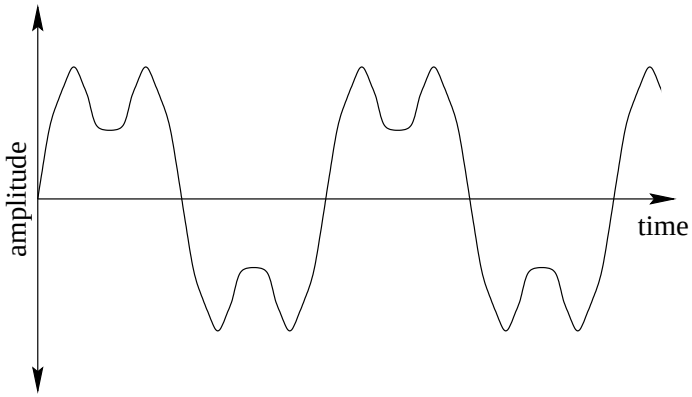
Compression is a necessity when you're dealing with video. Luckily, lossless compression is not important: Since each frame shows for only a fraction of a second, imperfections in a single frame aren't going to show up.

One simple compression technique is similar to the run-length compression technique, but we look for run-lengths along the time dimension. In this technique, we list only the pixels in each frame that differ from the previous frame. Listing each pixel involves giving the coordinates of each pixel, plus the color of that pixel.

This works well when the camera is still, when the background tends to remain unchanged. When the camera pans or zooms, however, it causes problem. To provide for these cases also, popular video compression formats (such as MPEG) provide for a more complex specification of the relationship between the pixels of two adjacent frames.

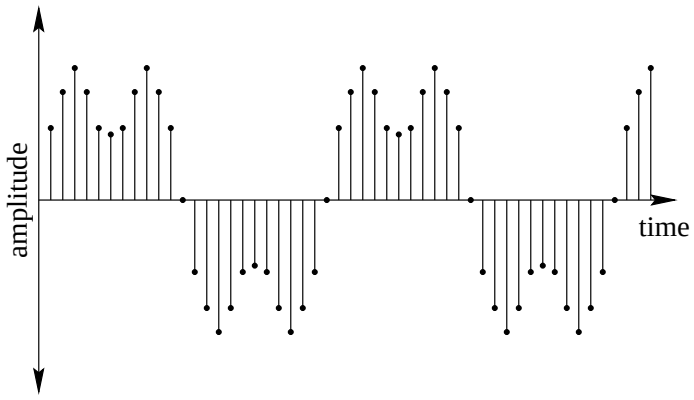
3.4.5 Sound

Sound is another important medium to represent in modern computing systems. Sound consists of vibrations in the air, which can be modeled as a graph of amplitude over time.



(This is a very simple, regular amplitude graph. The graphs you frequently see, especially for human speech, tend to be much more irregular and spiky.)

One way of representing the sound is to divide the amplitude graph into regular intervals and to measure the graph's height at each interval.



Then, we can store the height of the subsequent intervals. A system can interpolate between the sampled points to reproduce the original sound.

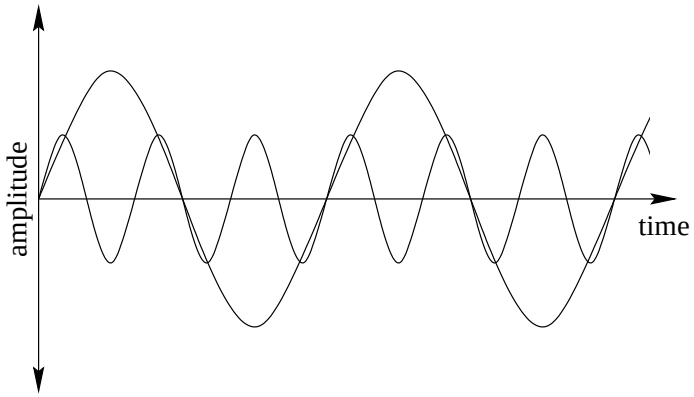
This technique of sampling is exactly the one used by CDs, which include samples of the amplitude graph taken 44,100 times a second. (The designers chose 44,100 times because the human ear can sense tones with a frequency of up to 22 KHz. That is, the human ear can handle tones that go from down to up to down again 22,000 times a second. By sampling at twice that rate, they get both the "down" and the "up" of such cycles so that the sound producer, which interpolates between the samples, can reproduce the sound wave.) CDs use 16 bits to represent each sample's height, and they include samples for both the left and the right speaker to achieve a stereo effect, for a total of 32 bits per sample. Thus, to store music in the CD format, you need 10 megabytes for each minute of sound:

$$\frac{44,100 \text{ samples}}{\text{sec}} \times \frac{32 \text{ bits}}{\text{sample}} \times \frac{\text{byte}}{8 \text{ bits}} \times \frac{\text{MB}}{2^{20} \text{ bytes}} \times \frac{60 \text{ sec}}{\text{min}} = \frac{10 \text{ MB}}{\text{min}} .$$

Since CDs typically store around 650 MB, they can hold around 65 minutes of sound. (Some CDs can hold slightly more data, providing for somewhat longer recordings.)

CDs provide high-quality sound through a high sampling rate, but this contains much extraneous information that most ears can't hear. It leaves a lot of room for more efficient representations of sound data.

The MP3 audio format is a particularly popular alternative. It uses a more complex understanding of how the human ear perceives sound. The human ear works by recognizing dominant frequencies in the sound it receives. Thus, to convert a sound to MP3 format, computers analyze a sound for the dominant sine waves that add up to the original wave.



Computers would ignore any frequencies beyond the ear's limit of 22 KHz, and they give some preference to waves in the range where the ear is most sensitive (2–4 KHz). Then, it stores the frequencies for these sine waves in the MP3 file. The result is a file representing the most important data that is needed to reproduce a sound for the human ear. Through this, and through some more minor techniques, the MP3 files tends to be approximately $1/11$ the size of the simple sampling technique used for CDs.

Chapter 4

Computational circuits

We saw several simple circuits in Chapter 2. But all these circuits did was to compute some weird function of a combination of bits. Why would anybody want to do that? I can forgive you if you felt underwhelmed.

Now that we understand the basics of data representation, we can explore more useful circuits for performing computation. In this chapter, we examine circuits for adding numbers together and circuits to remember data.

4.1 Integer addition

First we'll examine a circuit for adding integers, something that we can certainly agree that a computer needs to do.

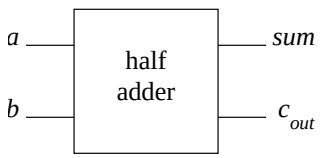
To work toward such a circuit, we first think about how we would do this on paper. Of course, we already understand how to do this for base-10 numbers. The computer will add binary numbers, but we can still use a similar approach. Suppose, for example, that we want to add $253_{(10)} = 11111101_{(2)}$ and $5_{(10)} = 101_{(2)}$.

$$\begin{array}{rr} & {\scriptstyle 11111\ \ 1} \\ & 11111101 \\ + & \quad 101 \\ \hline & 100000010 \end{array}$$

The result here is $100000010_{(2)} = 258_{(10)}$.

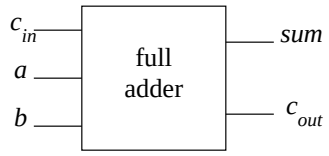
In Section 3.2.3 we saw that to add numbers in a two's-complement format, we can simply perform regular addition as if they are unsigned and then throw out any extra bits (like the uppermost bit in the above example). The above, then, could also be understood as the computation of $-3 + 5$, which would yield a solution of 2. Thus, though we will build a circuit for unsigned integers, our circuit will apply to adding two's-complement signed integers too.

The addition technique is fine on paper, but the computer doesn't have the luxury of a pencil and paper to perform computation like this. We need a circuit. We'll break the design process into two smaller pieces. The first, called a **half adder**, is for the rightmost column. (The box is intentionally drawn empty; we'll see what circuit it represents soon.)



It takes two inputs, representing the two bits in the rightmost column of our addition, and it has two outputs, representing the bit to place in the sum's rightmost column (sum) and the bit to carry to the next column (c_{out}).

For each of the other columns, we'll have three inputs: the carry from the previous column (c_{in}) and the two bits in the current column. We'll call the circuit to add these three bits together a **full adder**.



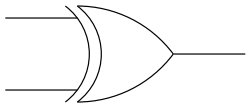
The two outputs have the same meaning as with the half adder.

The half adder

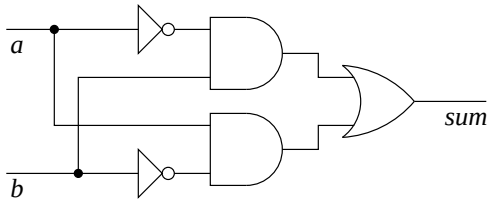
To build the half adder, we consider the four possible combinations of bits for the two inputs. We can draw a truth table for this.

a	b	c_{out}	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

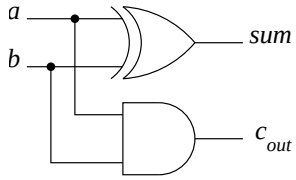
Notice that the c_{out} output is the AND function on a and b. The sum output is called the **exclusive-or** function (abbreviated **XOR**), so named because it is like an OR, but it *excludes* the possibility of both inputs being 1. We draw a XOR gate as an OR gate with a shield on its inputs.*



To design a XOR gate using AND, OR, and NOT gates, we observe that the sum-of-products expression is $\bar{a}b + a\bar{b}$, from which we can construct a circuit.



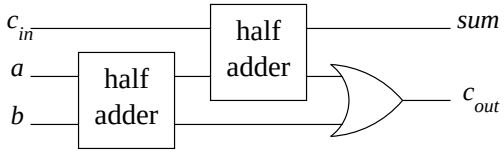
With a XOR gate built, we can now put together our half adder.



*Incidentally, many people use use a circled plus sign to represent XOR in Boolean expressions. In this system, $x \oplus y$ represents x XOR y.

The full adder

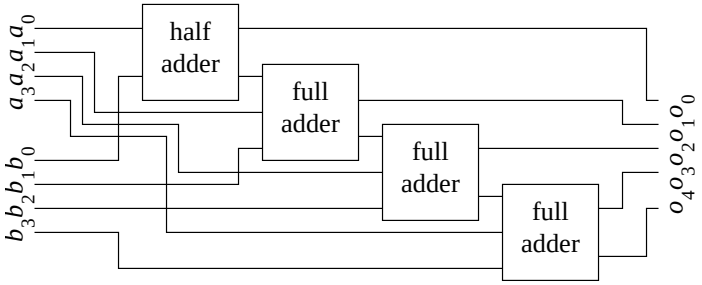
To design our full adder, we combine two half adders. The first half adder finds the sum of the first two input bits, and the second sums the first half adder's output with the third input bit.



Technically, you might say, we should use another half adder to add the carry bits from the two half adders together. But since the carry bits can't both be 1, an OR gate works just as well. (We prefer to use an OR gate because it is only one gate, and a half adder uses several; thus, the OR gate is cheaper.)

Putting it together

To build our complete circuit, we'll combine a half adder and several full adders, with the carry bits strung between them. For example, here is a four-bit adder.



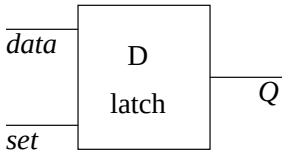
This diagram supposes that the first input is of the form $a_3a_2a_1a_0$ — that is, we call the 1's bit of the four-bit number a_0, the 2's bit a_1, the 4's bit a_2, and the 8's bit a_3. (If we were dealing with two's-complement numbers, a_3 would represent the -8's bit, and the circuit would add properly.) Similarly, the second input is of the form $b_3b_2b_1b_0$. Numbering the bits this way — starting with 0 for the 1's bit — may seem confusing: This numbering system comes from the fact that $2^0 = 1$ and so a_0 is for the 1's bit, while a_1 is for the 2's bit since $2^1 = 2$. Each bit a_k stands for the 2^k's bit. Designers conventionally use this system for numbering bits.

4.2 Circuits with memory

For a computer to be able to work interactively with a human, it must have some form of memory. In working toward this goal, we'll begin by examining a particular type of circuit called a *latch*.

4.2.1 Latches

It would be nice to have some circuit that remembers a single bit, with a single output representing this bit, and two inputs allowing us to alter the circuit's value when we choose.



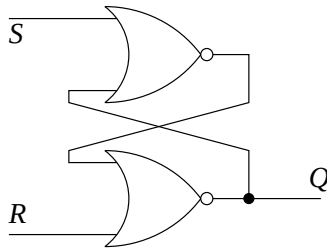
We'll call the two inputs set and $data$. When set is 0, the circuit should do nothing except continue emitting what it remembers. But when set becomes 1, the circuit should begin remembering $data$'s value instead.

set	$data$	memory
0	0	unchanged
0	1	unchanged
1	0	0
1	1	1

Such a circuit is called a **D latch**. It's called a *latch* because it holds a value. The *D* designation refers to the particular way the set and $data$ inputs work. (In particular, *D* stands for **D**ata.) In this subsection we'll see how to build such a latch.

SR latch

We begin by considering the following little circuit.



The OR gates with circles after them are NOR gates. They're a combination of an OR gate with a NOT gate attached: Given the inputs x and y, a NOR gate outputs the value $\overline{x + y}$.

This circuit — with its output Q going into the upper NOR gate, whose output loops back to the gate computing Q — is peculiar: We haven't seen a circuit with such loops before. This loop is what will give rise to our memory.

So what does this circuit do? We can fill out the following table for what Q this circuit will compute given various combinations of R, S, and the current value of Q. We include the current value of Q (labeled "old Q") among the input columns of the table because Q's value loops back as an input to one of the gates.

S	R	old Q	new Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	ignore
1	1	1	ignore

To see how we arrive at this table, let's take the first row as an example, when S and R are both 0, and when Q is currently 0. In this case, the lower NOR gate must be emitting a 0, since that is what Q is. This 0, and the S input of 0, are the inputs to the upper NOR gate, so the upper NOR gate emits a 1. Tracing this around, this 1 is an input to the lower NOR gate, along with the R input of 1, so the lower NOR gate emits a 0. We can continue tracing this around, and the output of the lower NOR gate will continue being 0; thus, we write 0 for the new Q value in the first row.

Now let's say we change the S input to be 1 — this moves us to the fifth row of the table, when S is 1, R is 0, and Q is 0. Now look at the upper NOR gate: It receives the S input of 1 and the Q input of 0, so the upper gate emits 0. But this changes the output of the lower NOR gate: With the 0 input from the upper NOR gate, and the R input of 0, the lower NOR gate emits 1. Now this 1 goes up to the upper NOR gate, and, with the S input of 1, the NOR gate continues to output 0. Now the circuit is again in a stable state, but with Q now being 1. Thus a 1 is in the last column for the fifth row. We can continue this sort of analysis to complete the other five rows labeled in the above truth table.

As for the last two rows, we're simply going to avoid them. We'll assume nobody will ever set both R and S inputs to 1, since such inputs won't be useful to us.

Examining the other rows of the table, we notice that if both R and S are 0 (the first two rows), then Q remains unchanged; that is, it remembers a bit. If S is 0 and R is 1 (the third and fourth rows), then Q becomes 0 regardless of its previous value. And if S is 1 and R is 0 (the fifth and sixth rows), then Q becomes 1 regardless of its previous value. We can tabulate this as follows.

S	R	memory
0	0	unchanged
0	1	0
1	0	1
1	1	ignore

This circuit is called an **SR latch**. Again, it's a *latch* because it holds a bit. The S and R refer to the traditional names of the inputs. The names R and S derive from the fact that when S is 1, the remembered bit is **S**et to 1, and when R is 1, the remembered bit is **R**eset to 0.

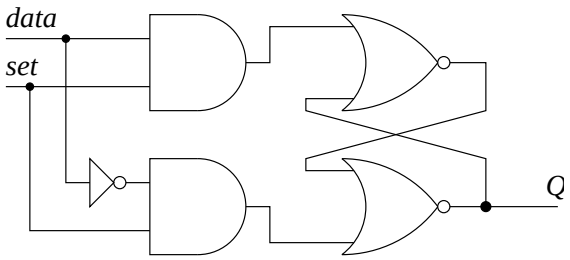
D latch

With an SR latch in hand, we can build the D latch we set out to design, which you can recall has inputs of set and $data$. What we'll do is translate the various combinations of set and $data$ to the required S and R inputs corresponding to desired behavior; from this, we can build a circuit incorporating an SR latch.

set	$data$	desired Q	S	R
0	0	old Q	0	0
0	1	old Q	0	0
1	0	0	0	1
1	1	1	1	0

For the first row of this table, we had already decided that we want the new Q to remain the same when set is 0. We've seen that the way to accomplish this using an SR latch is to set both S and R to 0. Thus, you see 0 and 0 in the last two columns of the first row. Deriving the other rows proceeds similarly.

Based on this table, we can determine that S should be $set \cdot data$, while R should be $set \cdot \overline{data}$. We use this to build a circuit giving our desired D latch behavior.

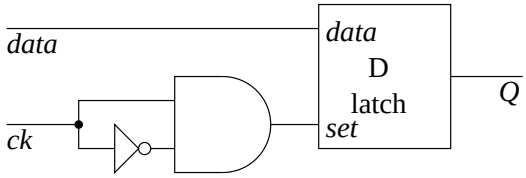


4.2.2 Flip-flops

The D latch gives us the ability to remember a bit, but in practice it's more convenient to have components whose values change only at the instant that the set input changes to 1. This reduces confusion about what happens when $data$ changes if set is still 1. Such a circuit — whose value changes only at the instant that its set input changing values — is called a **flip-flop**. For these circuits, we will call the set input the *clock*.

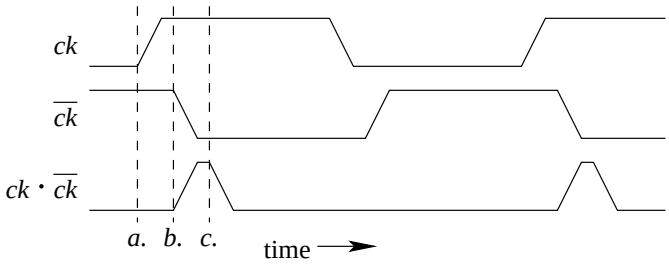
D flip-flop

Consider the following circuit, called a **D flip-flop**.



Notice what this circuit does to compute the set input to the D latch: It computes $ck \cdot \overline{ck}$. (The ck name here stands for *clock*.) This is weird: According to the law $A \cdot \overline{A} = 0$ from Boolean algebra, the AND gate would always output 0. What's the point of having a latch if its set input is always 0?

This apparent pointlessness is explained by considering the fact that gates are physical devices, and they take time to respond to inputs. To understand how this circuit really works, it's useful to look at the following illustration, called a **timing diagram**.



The horizontal axis represents time. The upper line of the diagram (labeled ck) indicates that ck begins at 0, then changes to 1, then back to 0, then back to 1. The first change to 1 occurs at instant a in time (diagrammed with a vertical dashed line with a label below it). Since electricity is a physical quantity, voltage cannot change instantaneously, so in this diagram each change in value is diagrammed with a slanted line.

Let's look at what happens at time a: The outputs of the NOT and AND gates do not immediately change when ck changes, because they take time to sense the change and react. The beginning of the NOT gate's reaction appears in the diagram at time b. More surprisingly, the AND gate reacts at time b, too: You can see from the diagram that between a and b, the AND gate sees a 1 from ck and a 1 from $\overline{ck}$. The AND gate's behavior is to output a 1 in this circumstance, so at time b it begins emitting a 1. By time c, it detects that the NOT gate is now at 0, and so the AND gate's output changes back to 0. Thus, the AND gate outputs 1 for a brief instant whenever ck changes from 0 to 1.

In the flip-flop circuit, then, when ck changes from 0 to 1, the set input to the D latch instantaneously becomes 1, and the D latch will remember whatever $data$ holds at that instant. Then its set input switches back to 0 again, so that further changes to $data$ do not influence the latch (until, that is, ck changes from 0 to 1 in its next cycle).

In circuit diagrams, we represent the D flip-flop as follows.

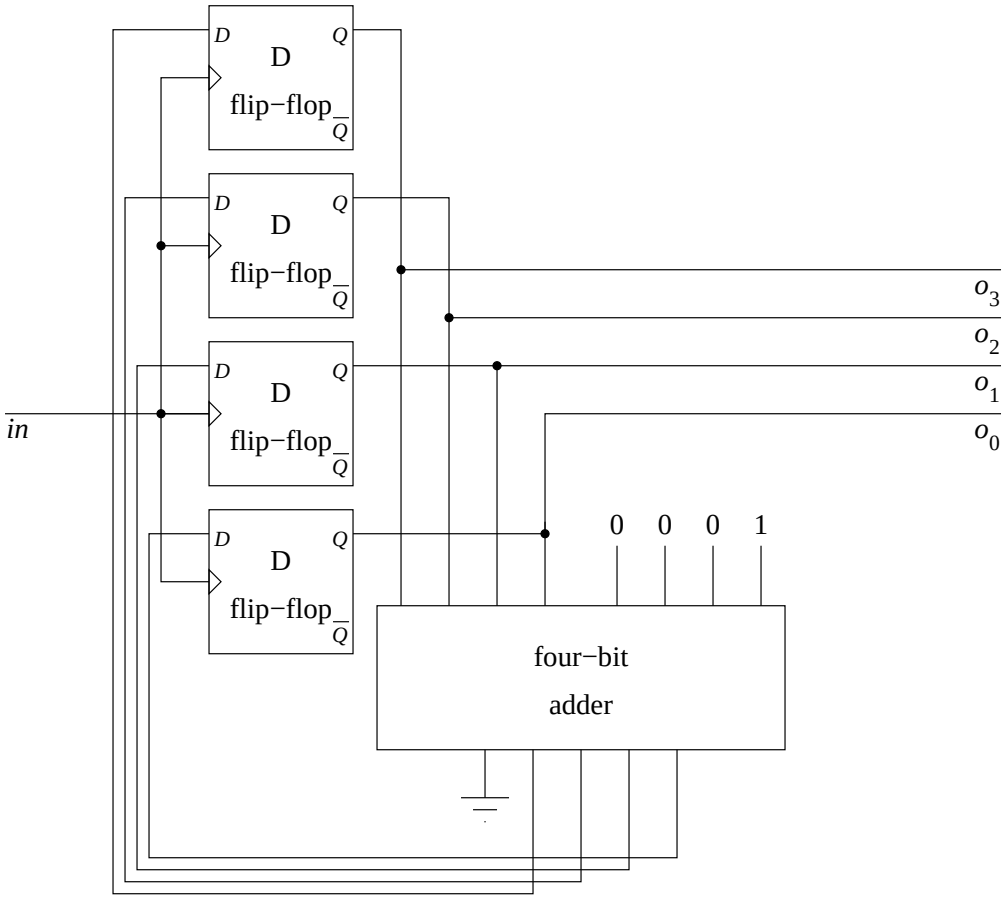
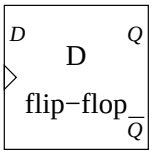


Figure 4.1: A four-bit counter.



The triangle is traditional way of denoting an input for which the component acts only when the input changes from 0 to 1. Notice that this component also outputs $\overline{Q}$. The flip-flop outputs this value because it is easy to compute. (The upper NOR gate in the underlying SR latch generates it.) It's often convenient to take advantage of this additional output in circuits.

4.2.3 Putting it together: A counter

Suppose we want a circuit that counts how many times the user has given it a 1 bit as an input. Such a simple circuit would be useful, for example, in a turnstile to count people entering. To do this, we'll use four flip-flops, to remember the current count in binary. (Our counter will only count up to 15, the largest four-bit number, and then it will reset to 0. If we want to count higher, we would need more flip-flops.) And we'll include a four-bit adder to compute the next value for these four flip-flops. Figure 4.1 contains a diagram of our circuit.

To get a feel for how the circuit of Figure 4.1 works, suppose the in input is 0, and all the D flip-flops

hold 0. Then these outputs would be fed into the four-bit adder, which also takes its other input of $0001_{(2)}$, and would output $0000_{(2)} + 0001_{(2)} = 00001_{(2)}$. The lower four bits of this output are fed into the D flip-flops' D inputs, but the flip-flops' values don't change, because their clock inputs (wired to in) are all 0. (The upper bit of the adder's output is ignored — in the circuit, we acknowledge this by representing that the output is grounded.)

When the in input changes to 1 again, then the flip-flops' values will suddenly change their remembered values to 1, and the circuit's outputs will reflect this. Also, the four-bit adder would now receive 0001 for its upper four-bit input, so that the adder would output $0001_{(2)} + 0001_{(2)} = 00010_{(2)}$. This goes into the flip-flops, but the flip-flops values won't change again, because flip-flops change their value only at the instant that in becomes 1, and that time has long past before $0010_{(2)}$ reaches them.

This last point, by the way, illustrates the point of using flip-flops instead of latches. Suppose we used latches instead. Because the set input would still be 1 at this point, the latches would begin remembering $0010_{(2)}$. And this would go through the adder, and the $0011_{(2)}$ would go into the latches. This would go through the adder, and $0100_{(2)}$ would go into the latches. The circuit would count incredibly fast until finally set would go to 0. We wouldn't be able to predict where it stops.[†] Using flip-flops, however, the count goes up only once each time the input goes to 1.

4.3 Sequential circuit design (optional)

Circuits whose output is dependent solely on the current inputs of the circuit are called **combinational circuits**. All of the circuits that we studied in Chapter 2 are combinational circuits, as is the adder circuit in this chapter. Other circuits, in which the output may depend on past inputs also, are **sequential circuits**. Flip-flops are a simple example of sequential circuits. A counter is a more complex example.

When we studied combinational circuits, we examined a systematic technique for designing them: You take the truth table, which was the specification of the circuit's design, from there you get a sum-of-products Boolean expression, which you can minimize and then use to build a circuit.

In this section, we look at a systematic way for designing sequential circuits. It is a four-step process.

1. Draw a state transition diagram outlining how the circuit should change state as it receives inputs. The number of states will dictate how many flip-flops the circuit must have.

2. Generate a table saying how the flip-flop values should change in each step, based on the flip-flops' current values and the inputs to the circuit. Also, generate a table relating the flip-flops' values to the desired output of the circuit.

3. Derive combinational circuits to compute the inputs to each flip-flop and to compute each circuit output.

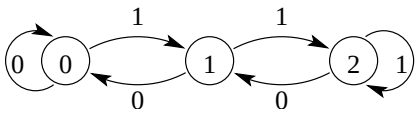
4. Combine these derived circuits together into a sequential circuit.

4.3.1 An example

As an example of this process, suppose we want a circuit using D flip-flops with two inputs and with two outputs. One of the inputs ck is a clock; the second, dir, says whether the circuit should count up or down (1 representing up). The circuit should count up to 2, and it should not display wraparound behavior — that is, when the circuit is at 0 and dir says to count down, the circuit remains at 0; and when the circuit is at 2 and dir says to count up, the circuit remains at 2.

[†] In fact, since the gates aren't all identically fast, and the wires aren't all identically long, the changes in latches' values would be much more erratic.

Step one: Drawing a state transition diagram In this case, the circuit should "remember" one of three things: The counter could be at 0, it could be at 1, or it could be at 2. Based on this, we can draw a picture of how what it will remember should change.



The arrows in this picture represent transitions between states. If, for example, the circuit is at state 0, and the clock changes while input dir is 1, then the counter's value should change to 1, and so the circuit should move to state 1. Thus we see an arrow labeled 1 extending from state 0 to state 1.

Because there are three states, and because two bits can represent three different values, we'll use two flip-flops. (Of course, two bits can actually handle up to four different values. We don't have any use for that last value here, though.) We name the outputs of the two flip-flops Q_1 and Q_0, and we create a table relating flip-flop values to states in the diagram.

state	Q_1	Q_0
0	0	0
1	0	1
2	1	0

Step two: Generating tables Based on our state diagram, we can generate a table of how states should change based on the current state and the dir input. (We don't include the clock input: It will simply be wired to the clock input of each flip-flop.)

dir	old Q_1	old Q_0	new Q_1	new Q_0
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	d	d
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	d	d

This table is a direct translation of the state diagram. In the first row, for example, dir, Q_1, and Q_0 are all currently 0. The current values of Q_1 and Q_0 indicate that we are in state 0, according to the translation table determined in the previous step. According to the state transition diagram, if we're in state 0, and dir is 0, then we should move to state 0. Looking again at the translation table from the previous step, we see that the new values for Q_1 and Q_0 should be 0.

For the second row, the current Q_1 and Q_0 values indicate that we are in state 1. Since this row is for the case that dir is 0, we look into the state transition diagram for an arrow starting at state 1 and labeled 0, and we observe that this arrow leads to state 0. The translation table indicates that state 0 is indicated by having Q_1 and Q_0 both be 0, and so that is what you see in the last two columns of the second row.

The entries marked d in this table stand for *don't-care*. They occur here because, when Q_1 and Q_0 are 1, we are in an undefined state, and we don't care about the circuit's behavior then. In fact, something would happen should the circuit ever reach this state, but if we've designed it properly, it never will get there. We don't commit to a behavior for this undefined state now, because that will maintain our freedom later to choose whatever behavior keeps the final circuit simplest.

We should also draw a table saying how the circuit's output relates to the flip-flop's values. In this case, we've chosen the flip-flop values to correspond exactly to the desired outputs, so this part is easy.

Q_1	Q_0	o_1	o_0
0	0	0	0
0	1	0	1
1	0	1	0
1	1	d	d

For other problems, the relationship between the flip-flops' values and the table would be more complex.

Step three: Derive combinational circuits For the third step, we derive combinational circuits for computing the flip-flops' new values and the outputs. For example, for the second flip-flop's value (new Q_1), we can look at our table and see the following. Note that we're removing the "new Q_0" column from before; the circuit will compute that column simultaneously, and so we can't use it in determining "new Q_1."

dir	old Q_1	old Q_0	new Q_1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	d
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	d

Following the procedure of Chapter 2, we can derive a circuit for this table. Our sum-of-products expression would be

$$dir \cdot \overline{Q_1} \cdot Q_0 + dir \cdot Q_1 \cdot \overline{Q_0} \,. = dir \cdot Q_1 \cdot \overline{Q_0} + dir \cdot \overline{Q_1} \cdot Q_0$$

This does not simplify, and so we can stop there.[‡] (I've commuted the two terms to make the circuit diagram later prettier.)

Computing the expression for "new Q_0" proceeds similarly, and we would end up with

$$\overline{dir} \cdot Q_1 \cdot \overline{Q_0} + dir \cdot \overline{Q_1} \cdot \overline{Q_0} \,.$$

Similarly, the expression for o_1 would be $Q_1\overline{Q_0}$, for o_0 it would be $\overline{Q_1}Q_0$.

Step four: Put the circuits together In the final step, we join these circuits based on the previous step into one overall circuit. Figure 4.2 shows this derived circuit. The top dotted box computes the "new Q_1" expression derived in the previous step. Its output, notice, is looped around back to Q_1 to be stored the next time ck changes. Also notice how the circuit inside the dotted box saves unneeded NOT gates by using the flip-flops' $\overline{Q}$ outputs when appropriate.

Similarly, the second dotted box computes "new Q_0," the third dotted box computes the output o_1, and the bottom dotted box computes the output o_0.

4.3.2 Another example

Let's look at another example. This time, we want a circuit with a single input — the clock — and a single output. The output should be 1 every fourth time the clock input changes from 0 to 1.

[‡]If we were to choose the lower d in the table to be a 1, we could simplify the circuit. In this case, we're not going to worry about finding the smallest possibility, though.

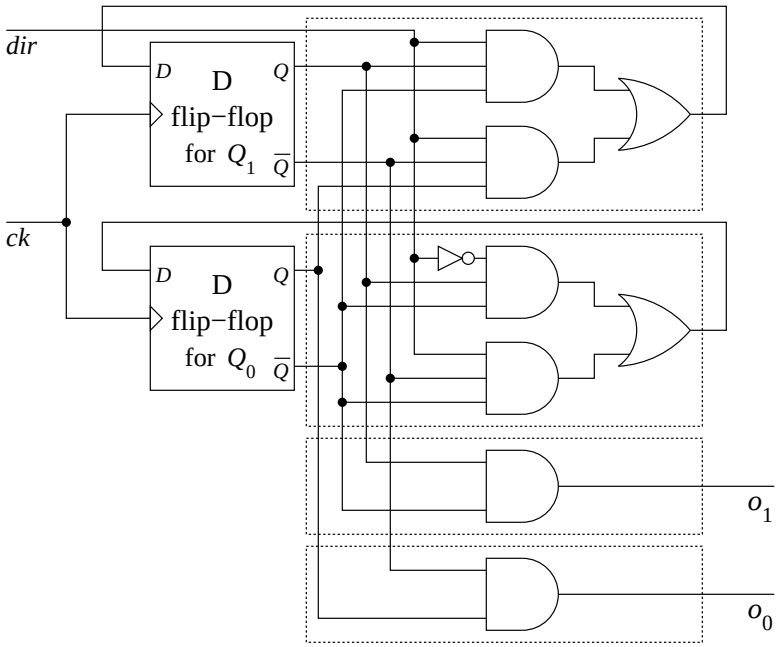
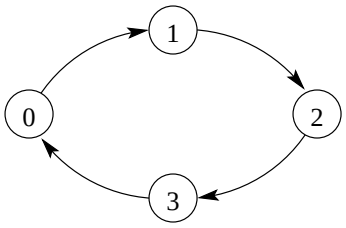


Figure 4.2: A sequential circuit counting up and down between 0 and 2.

Step one: Drawing a state transition diagram In this case, the circuit should "remember" one of four things: The current output could be at 1, it could be at 0 on the first clock pulse, it could be at 0 on the second clock pulse, or it could be at 0 on the third clock pulse. Based on this, we can draw a picture of how what it will remember should change.



This diagram illustrates that the circuit should cycle through the four states each time the clock changes. In contrast to the last circuit we designed, which had an input other than the clock which affected how the circuit was to modify its state, this circuit has no inputs other than the clock. Thus, the state transition diagram for this circuit doesn't need labels on the arrows between states.

Because there are four states, we'll use two flip-flops. We name the outputs of the two flip-flops Q_1 and Q_0, and we create a table relating flip-flop values to states in the diagram.

state	Q_1	Q_0
0	0	0
1	0	1
2	1	0
3	1	1

Step two: Generating tables This problem specifies no circuit inputs other than the clock input, and so the only columns on the left side of our table are the current flip-flops' values. Based on our state diagram,

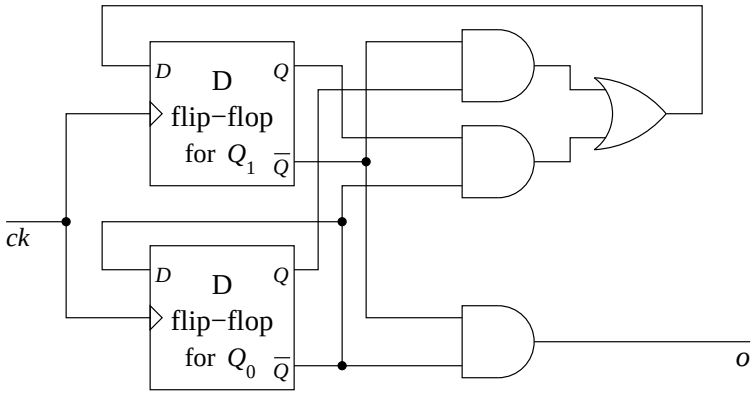


Figure 4.3: A sequential circuit whose value is 1 every fourth clock pulse.

we can generate a table of how states should change based on the current state.

old Q_1	old Q_0	new Q_1	new Q_0
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

The table for the circuit's output corresponds to the desired output for each state.

Q_1	Q_0	o
0	0	1
0	1	0
1	0	0
1	1	0

Step three: Derive combinational circuits Based on the tables from the previous steps, we derive Boolean expressions for the flip-flops' new values and for the circuit's output.

$$\begin{aligned} \text{new } Q_1 &= \overline{Q_1}Q_0 + Q_1\overline{Q_0} \\ \text{new } Q_0 &= \overline{Q_1}\,\overline{Q_0} + Q_1\overline{Q_0} = \overline{Q_0} \\ o &= \overline{Q_1}\,\overline{Q_0} \end{aligned}$$

Note that the expression for the new Q_0 simplified.

Step four: Put the circuits together Figure 4.3 shows the derived circuit.

Chapter 5

Computer architecture

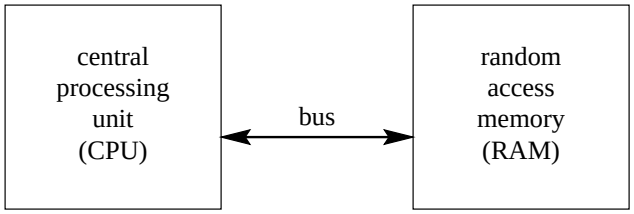
Thus far, we've seen how to build circuits to perform simple computation. A computer, however, is a much more complex device. In this chapter, we'll examine the level on which computers operate programs, and we'll get some feel for how this can be done via circuits.

5.1 Machine design

To work with a concrete computer design, we'll examine a computer called *HYMN*, a simple design invented for teaching purposes.* The name stands for *HY*pothetical *M*achi*N*e. (While studying a "real" industrial-strength computer sounds nice at first, the added complexity interferes with understanding the essential concepts.)

5.1.1 Overview

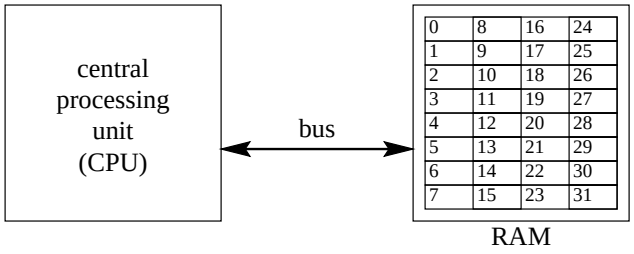
Modern computers, including HYMN, include two major parts: the **central processing unit**, or **CPU**, and **random access memory**, or **RAM**. The CPU performs the computation, while the RAM stores long-term information.



A bundle of wires called a **bus** connects these two pieces. The bus gives the CPU an avenue for communicating with memory to retrieve and store data when needed for computation.

RAM is the simpler of the two pieces: It is simply an array of bytes. Although modern computers have millions or even trillions of bytes of RAM, the RAM in HYMN holds only 32 bytes.

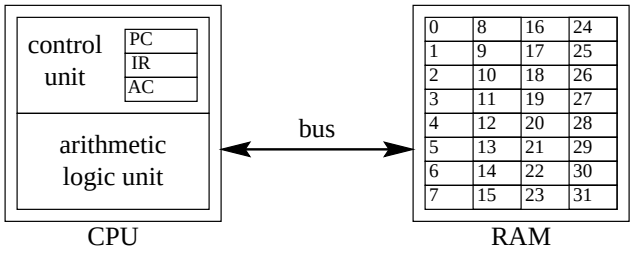
*HYMN's design comes from Noreen Herzfeld's book *Computer Concepts and Applications for Non-Majors* (manuscript, 2002).



Since each byte has 8 bits, and we can use a D flip-flop to remember each bit of RAM, we could build this RAM using $32 \cdot 8 = 256$ D flip-flops.

Each byte of RAM has a number for describing it, called its **address**; when the CPU wants to retrieve data from RAM, it sends the address of the desired byte on the bus. Sometimes, when talking about memory, we'll use notation such as "$\mathbf{M}[7]$," which represents the byte whose address is 7 (at the bottom of the RAM's leftmost column in the picture).

In most modern computers, the CPU is a single chip including thousands or millions of logic gates. The CPU's design can be split into two major pieces, the control unit and the arithmetic logic unit.



The **control unit** controls the overall structure of the computation performed, while the **arithmetic logic unit** (or **ALU**) is for performing arithmetic and logical operations. For HYMN, the only arithmetic and logical operations provided by the ALU are addition, subtraction, and identification of whether a number is positive or zero (or neither). In more sophisticated CPUs, the ALU would also include circuitry for other arithmetic operations like multiplication and division and for logical operations like AND, OR, and NOT.

As the CPU performs its task, it will remember data. Each location on the CPU for storing a piece of data is called a **register**. HYMN's design calls for three registers.

The accumulator (abbreviated *AC*) holds temporary data being used for computation.

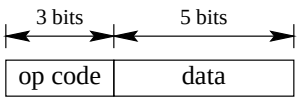
The program counter (abbreviated *PC*) tracks the address of the instruction to execute next.

The instruction register (abbreviated *IR*) holds the current instruction being executed.

You can think of the registers as the computer's "short-term memory" and RAM as its "long-term memory."

5.1.2 Instruction set

Each instruction in the program will be encoded as a value in RAM. In HYMN's design, each instruction is eight bits long, including three bits describing the instruction code (the **op code** — *op* is short for *operation*) and five bits containing additional data for the instruction.

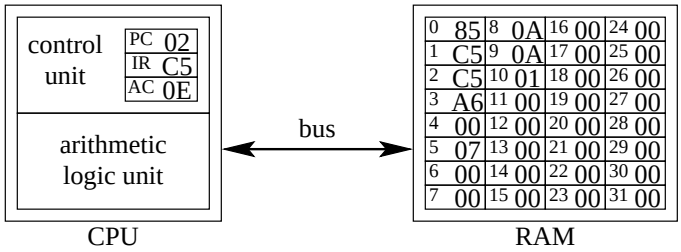


code	op	behavior
000	HALT	nothing further happens (computer halts)
001	JUMP	PC $\leftarrow$ $data$
010	JZER	**if** AC $= 0$ **then** PC $\leftarrow$ $data$ **else** PC $\leftarrow$ PC $+ 1$
011	JPOS	**if** AC > 0 **then** PC $\leftarrow$ $data$ **else** PC $\leftarrow$ PC $+ 1$
100	LOAD	AC $\leftarrow$ $\mathbf{M}[data]$; PC $\leftarrow$ PC $+ 1$
101	STORE	$\mathbf{M}[data] \leftarrow$ AC; PC $\leftarrow$ PC $+ 1$
110	ADD	AC $\leftarrow$ AC $+ \mathbf{M}[data]$; PC $\leftarrow$ PC $+ 1$
111	SUB	AC $\leftarrow$ AC $- \mathbf{M}[data]$; PC $\leftarrow$ PC $+ 1$

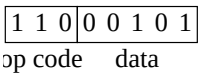
Figure 5.1: The HYMN instruction set.

The op code designates one of HYMN's eight possible instruction types, which are tabulated with their behaviors in Figure 5.1.

For example, suppose our HYMN computer were running with the following values in registers and memory. (All values are written in hexadecimal.)



The current instruction the computer wants to execute is normally in the IR; at this point, IR holds $C5_{(16)}$, or $11000101_{(2)}$. To execute this instruction, the control unit would first divide the instruction into its two pieces.



It interprets the first three bits, 110, as being the operation's code; based on the row labeled 110 in Figure 5.1, we see that we're looking at an ADD instruction.

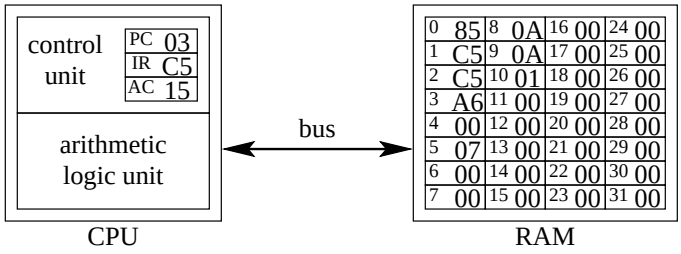
$$110 \quad \text{ADD} \quad \text{AC} \leftarrow \text{AC} + \mathbf{M}[data]; \text{PC} \leftarrow \text{PC} + 1$$

This says that the computer should do two things to perform this operation.

AC $\leftarrow$ AC $+ \mathbf{M}[data]$: The computer computes AC $+ \mathbf{M}[data]$ and places the result into AC. To compute the value, it looks first at the last five bits of the instruction to determine $data$; in this case, the last five bits give the number $00101_{(2)} = 5_{(10)}$. Then, it determines $\mathbf{M}[data]$ by looking in memory at address 5; the memory currently contains $07_{(16)}$. Finally, it adds this value ($07_{(16)}$) to the current value in AC (that is, $0E_{(16)}$), to arrive at the result $15_{(16)}$. The computer places this value into AC.

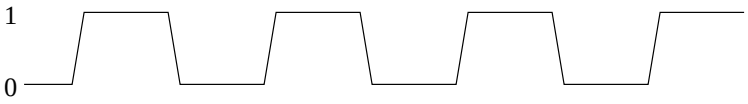
PC $\leftarrow$ PC $+ 1$: The computer takes the current value of PC (that is, $02_{(16)}$) and adds 1. It places the result, $03_{(16)}$, into PC.

Thus, after completing the instruction, the computer holds the following data instead. (The only values that have changed are those in AC and PC.)



5.1.3 The fetch-execute cycle

Computers incorporate a **clock** for sending signals to the CPU telling it when to move forward in its computation. The clock's job is to simply emit a signal oscillating between 0 and 1.



Each oscillation, from 0 to 1 and back to 0, is called a **pulse**. CPU specifications often include a measure of how fast this clock can go: A 3GHz (*three gigahertz*) computer, for example, contains a CPU that will work as long as the clock doesn't go faster than three billion (*giga-*) pulses a second.[†]

Doing a single instruction is a two-step process, called the **fetch-execute cycle**. First, the computer *fetches* the next instruction to execute. Then, the computer *executes* this instruction. Through repeating this process *ad infinitum*, the computer completes its execution.

For HYMN, the PC register is for holding the address of the next instruction to execute, and the IR is for holding the current instruction. Thus, during the fetch process, the HYMN CPU will take the contents of PC, send it to RAM via the bus, and the CPU will take RAM's response and place it into the IR.

The execute process involves taking the current value stored in the IR (which was placed there in the preceding fetch), determining that instruction's op code by examining its first three bits, and performing the action as specified in the corresponding row of Figure 5.1.

5.1.4 A simple program

When we want to run a program, we put the program into RAM before starting the CPU. For example, we might place the following into the memory and then start the CPU.

addr	value	*op*	*data*
0	$10000101_{(2)}$ ($85_{(16)}$)	*LOAD*	*5*
1	$11000101_{(2)}$ ($C5_{(16)}$)	*ADD*	*5*
2	$11000101_{(2)}$ ($C5_{(16)}$)	*ADD*	*5*
3	$10100110_{(2)}$ ($A6_{(16)}$)	*STORE*	*6*
4	$00000000_{(2)}$ ($00_{(16)}$)	*HALT*	—
5	$00001100_{(2)}$ ($07_{(16)}$)	*7*	—
6	$00000000_{(2)}$ ($00_{(16)}$)	*0*	—

[†]There are several factors that play into this speed limitation. One is that electrical signals take time, and a too-fast clock could demand that the computer use information before the computation prompted by the previous pulse has had time to propagate through the circuit, in which case the circuit would use wrong information. Another factor is that a faster clock pushes the gates to work faster; if the gates perform too much computation, they can literally overheat and burn the CPU. Computers with fast clocks often have elaborate cooling systems to prevent overheating.

The following table represents what happens as the computer begins running. Each row represents the contents of the registers (written in hexademical) at the beginning of a clock pulse, in which the computer performs either the fetch or execute process.

PC	IR	AC	action
00	00	00	The computer automatically starts with zero in each of its registers.
00	85	00	Fetch: CPU fetches the memory at address PC $= 0$ into IR.
01	85	07	Execute LOAD: CPU fetches the memory at address $data = 5$ into AC and places PC $+ 1$ into PC.
01	C5	00	Fetch: CPU fetches the memory at address PC $= 1$ into IR.
02	C5	0E	Execute ADD: CPU adds the memory at address $data = 5$ into AC and places PC $+ 1$ into PC.
02	C5	0E	Fetch: CPU fetches the memory at address PC $= 2$ into IR.
03	C5	15	Execute ADD: CPU adds the memory at address $data = 5$ into AC and places PC $+ 1$ into PC.
03	A6	15	Fetch: CPU fetches the memory at address PC $= 3$ into IR.
04	A6	15	Execute STORE: CPU stores AC $= 15_{(16)}$ into memory at address $data = 6$ and places PC $+ 1$ into PC.
04	00	15	Fetch: CPU fetches the memory at address PC $= 4$ into IR.
04	00	15	Execute HALT: CPU does nothing.
04	00	15	Fetch: CPU fetches the memory at address PC $= 4$ into IR.
04	00	15	Execute HALT: CPU does nothing.
	$\vdots$		The computer continues fetching the same HALT instruction and doing nothing. It has stopped performing useful computation.

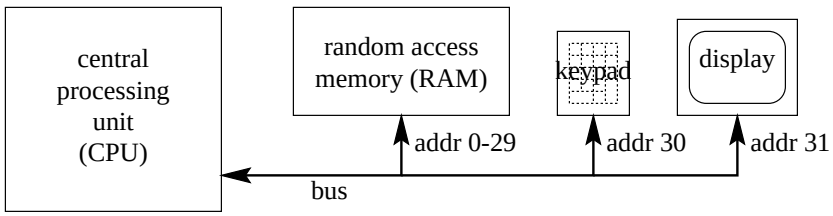
What the computer has accomplished here is to take the number at $\mathbf{M}[5]$, add $\mathbf{M}[5]$ to it, and add $\mathbf{M}[5]$ again, placing the result into $\mathbf{M}[6]$. Since we had 7 at $\mathbf{M}[5]$, the program placed $21_{(10)} = 15_{(16)}$ into $\mathbf{M}[6]$ before it halted.

5.2 Machine language features

So far, we have seen how the computer executes a straightforward program. In this section we'll consider more complex programming features that enable us to build more sophisticated programs with HYMN.

5.2.1 Input and output

While our program to place three times the contents of memory at address 5 into memory at address 6 is nice, it would be even better if could have a program that interacts with the user. To accomplish this, we'll modify HYMN's structure to include two new components — a keypad and a display — attached to the bus.



We dedicate a memory address to each of these devices: The keypad gets address 30, and the display gets address 31. RAM will not respond to these addresses.

When the CPU sends a request to load information from address 30 onto the bus, RAM doesn't respond. Instead, the keypad waits until the user types a number, and it sends that number to the CPU via the bus as its response. Similarly, when the CPU sends a request to store a number to address 31, the display handles the request (by showing the number on the screen).

The following program reads a number n from the user and displays $3n$ on the screen.

addr	value	*op*	*data*
0	$10011110_{(2)}$ ($9E_{(16)}$)	*LOAD*	*30*
1	$10100110_{(2)}$ ($A6_{(16)}$)	*STORE*	*6*
2	$11000110_{(2)}$ ($C6_{(16)}$)	*ADD*	*6*
3	$11000110_{(2)}$ ($C6_{(16)}$)	*ADD*	*6*
4	$10111111_{(2)}$ ($BF_{(16)}$)	*STORE*	*31*
5	$00000000_{(2)}$ ($00_{(16)}$)	*HALT*	—
6	$00000000_{(2)}$ ($00_{(16)}$)	*0*	—

It works by loading into AC a number the user types on the keypad (instruction 0), then storing this number in $\mathbf{M}[6]$ (instruction 1). Then it adds $\mathbf{M}[6]$ to the accumulator twice (instructions 2 and 3); now AC holds $3n$. It stores AC in $\mathbf{M}[31]$ (instruction 4) which effectively displays $3n$ on the screen, before halting (instruction 5).

5.2.2 Loops

HYMN includes three instructions that are useful for writing programs to perform a process repeatedly: JUMP, JPOS, and JZER. The JUMP instruction works by placing the $data$ of the instruction into the PC; thus, in the next fetch-execute cycle, the computer will fetch and then execute the instruction at the address given in the JUMP instruction. The effect of this is that the computer *jumps* to the instruction mentioned in the $data$ of the JUMP instruction, rather than merely continuing to the next instruction as with the LOAD, STORE, and ADD instructions.

The JPOS ("jump if positive") and JZER ("jump if zero") instructions are similar, except that for these the CPU will copy $data$ into PC only if the AC holds a positive number (for JPOS) or zero (for JZER). Otherwise, the CPU will increment PC so that the next instruction executes.

The following program, which uses the JPOS instruction, displays the numbers from 10 down to 1.

addr	value	*op*	*data*
0	$10011110_{(2)}$ ($9E_{(16)}$)	*LOAD*	*6*
1	$10111111_{(2)}$ ($BF_{(16)}$)	*STORE*	*31*
2	$11100101_{(2)}$ ($E5_{(16)}$)	*SUB*	*5*
3	$01100001_{(2)}$ ($61_{(16)}$)	*JPOS*	*1*
4	$00000000_{(2)}$ ($00_{(16)}$)	*HALT*	—
5	$00000001_{(2)}$ ($01_{(16)}$)	*1*	—
6	$00001010_{(2)}$ ($0A_{(16)}$)	*10*	—

To understand this program, let's trace through the process of HYMN executing it.

PC	IR	AC	action
00	00	00	The computer starts with zero in each register.
00	9E	00	Fetch: CPU fetches the memory at address PC $= 0$ into IR.
01	9E	0A	Execute LOAD: CPU fetches the memory at address $data = 6$ into AC and places PC $+ 1$ into PC.
01	BF	0A	Fetch: CPU fetches the memory at address PC $= 1$ into IR.

02	BF	0A	Execute STORE: CPU sends AC $=$ $A_{(16)}$ to address $31_{(10)}$ and places PC $+ 1$ into PC. Since address $31_{(10)}$ refers to the display, the display shows the decimal representation of $A_{(16)} = 10_{(10)}$.
02	E5	0A	Fetch: CPU fetches the memory at address PC $= 2$ into IR.
03	E5	09	Execute SUB: CPU subtracts the memory at address $data = 5$ from AC and places PC $+ 1$ into PC.
03	61	09	Fetch: CPU fetches the memory at address PC $= 3$ into IR.
01	61	09	Execute JPOS: Since AC is positive, CPU changes PC to $data = 1$.
01	BF	09	Fetch: CPU fetches the memory at address PC $= 1$ into IR.
02	BF	09	Execute STORE: CPU sends AC $= 9_{(16)}$ to address $31_{(10)}$ and places PC $+ 1$ into PC. Since address $31_{(10)}$ refers to the display, the display shows 9.
02	E5	09	Fetch: CPU fetches the memory at address PC $= 2$ into IR.
03	E5	08	Execute SUB: CPU subtracts the memory at address $data = 5$ from AC and places PC $+ 1$ into PC.
03	61	08	Fetch: CPU fetches the memory at address PC $= 3$ into IR.
01	61	08	Execute JPOS: Since AC is positive, CPU changes PC to $data = 1$.
	⋮		The computer continues repeating the instructions at addresses 1 through 3. Eventually, the CPU sends 1 to the display.
02	BF	01	Execute STORE: CPU sends AC $= 1_{(16)}$ to address $31_{(10)}$ and places PC $+ 1$ into PC. Since address $31_{(10)}$ refers to the display, the display shows 1.
02	E5	01	Fetch: CPU fetches the memory at address PC $= 2$ into IR.
03	E5	00	Execute SUB: CPU subtracts the memory at address $data = 5$ from AC and places PC $+ 1$ into PC.
03	61	00	Fetch: CPU fetches the memory at address PC $= 3$ into IR.
04	61	00	Execute JPOS: Since AC is *not* positive, CPU changes PC to PC $+ 1$.
04	00	00	Fetch: CPU fetches the memory at address PC $= 4$ into IR.
04	00	00	Execute HALT: CPU does nothing. It will continue fetching the same HALT instruction and doing nothing until the power is turned off.

Notice that the computer doesn't actualy go to another instruction in a JUMP, JPOS, or JZER. The instructions simply change the contents of the PC register, similarly to how a LOAD instruction changes the contents of the AC register. The actual "jump" occurs as a side effect of the fact that, in the next fetch phase, the computer fetches the next instruction to execute from the address just stored by the JUMP instruction into PC.

5.3 Assembly language

The representation of a program as a sequence of instructions written in the machine's encoding system is called **machine language**. Because the instructions' binary encoding is so foreign for humans, writing programs in machine language is laborious and difficult for programmers to manage. Thus, people prefer to write programs in **assembly language**, which uses mnemonic codes to describe the instructions. Then they can use a program called an **assembler**, which translates the mnemonic codes into the corresponding machine code.

5.3.1 Instruction mnemonics

A simple assembly language designed for HYMN allows us to write the name of an operation followed by a base-10 number to give the $data$. For a HALT instruction, for which $data$ is irrelevant, we omit the number.

Here is an example of a complete program written in HYMN's assembly language.

```
LOAD 6
STORE 31  # address 1: display AC on screen
SUB 5
JPOS 1
HALT
1         # address 5
10        # address 6
```

When the assembler sees a sharp ('#'), it ignores it and any characters after it in the same line. This is a **comment**; it is useless to the computer, but it can be useful for any human readers of the program.

The last two lines of this program illustrate an alternative way in this assembly language for describing what should go into memory: You can simply write the base-10 value that you want to place in memory.

5.3.2 Labels

Putting memory addresses directly in the program, as in the 6 of "LOAD 6," forces us to waste a lot of time counting lines in the assembly program. Worse, if we decide to add or remove a line from the program, we end up having to change many instructions' data.

To alleviate this pain, our assembly language allows us to "name" a byte with a **label**. To do this, we begin a line with the label's name, followed by a colon. This label, then, refers to the address of the data given within the line. In instructions where we want to refer to a memory address, then, we can instead write the line's name.

```
        LOAD start
again:  STORE 31  # display AC on screen
        SUB one
        JPOS again
        HALT
                  # (The assembler ignores blank lines like this.)
one:    1         # address 5
start:  10        # address 6
```

The assembler, when it translates this file, goes through a two-step process. First, it determines to which address each label refers. Then, it translates each individual line, substituting for each label the address to which it corresponds. Note that we can use labels for instructions (as "`again`" labels the "`STORE 31`" line) or for data (as "`one`" labels the "`1`" line). In general, HYMN doesn't distinguish between instructions and numbers — it simply treats data as instructions in some situations (such as the data in IR) and as numbers in other situations (such as the data in AC)

(The above assembly language program mixes capital and lower-case letters. The HYMN assembler actually treats lower-case letters and their capital equivalents identically. Thus, we could write this same program in all lower-case letters, all capital letters, or any mix we like.)

5.3.3 Pseudo-operations

Assemblers also often define **pseudo-ops**, which appear to be separate instructions in the machine language, but they actually translate to existing instructions. The HYMN assembler defines two of these: "READ" stands for "LOAD 30," and "WRITE" stands for "STORE 31." Thus, we could write our earlier program to read a number n and print $3n$ as follows.

```
    READ        # reads from the keypad into AC
    STORE n
    ADD n
    ADD n
    WRITE       # displays the contents of AC on screen
    HALT
n:  0
```

5.4 Designing assembly programs

Writing large assembly language programs is confusing: Keeping track of register contents and understanding the flow of control through all the jumps can be a nightmare. To alleviate the confusion, designers of assembly language programs use **pseudocode** to help understand a program's process for solving a problem.

5.4.1 Pseudocode definition

Pseudocode is an informal, formatted mixture of English and mathematics written to describe a computational process. Suppose, for example, that I want to describe the process of reading some number n from the user and then printing the sum of the integers up to n (i.e., $1 + 2 + \cdots + n$). The following is one way of writing pseudocode expressing a process for accomplishing this.

1. Read a number from the user, which we'll call n.
2. Initialize sum to 0.
3. Initialize i to 1.
4. Repeat the following while $i \leq n$:
    a. Increase sum by i.
    b. Increment i.
5. Display sum to the user.
6. Stop.

This is just one way of writing pseudocode, though. We could equally as well write the following to express the same process. (It happens that this book's pseudocode will look more like the following than the preceding example.)

Read n.
Initialize sum to 0.
Initialize i to 1.
while $i \leq n$, **do:**
    Increase sum by i.
    Increment i.
end while
Write sum.
Stop.

The important thing is that we're writing a step-by-step process for accomplish the desired task, with a different line representing each discrete step. But we're not worrying about the details of how to translate this to assembly language — we only want to describe the general process. Notice that the pseudocode does not refer to HYMN instructions, registers, or labels.

Pseudocode helps in trying to understand conceptually how to solve the problem. Assembly language designers, then, can follow three steps to develop their programs.

1. Develop pseudocode describing the procedure used.
2. Test the procedure by running through the pseudocode on paper.
3. Translate the pseudocode line by line into assembly language.

The second of these steps — testing the procedure — involves some mental calculation on some simple examples. We might, for our example, suppose that the user starts the program and types 5. What values would the variables take on as the pseudocode executes?

n	5										
sum		0		1		3		6		10		15
i			1		2		3		4		5		6

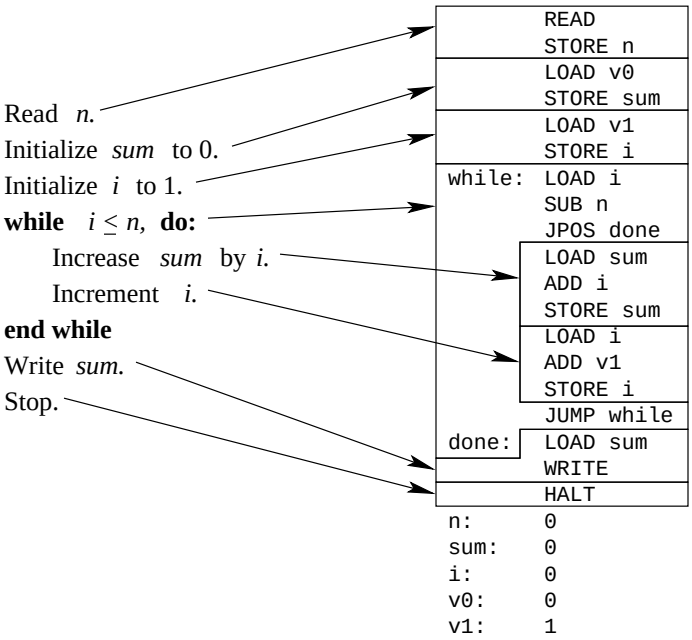
It would cease repeating the middle steps at this point, since it's no longer true that $i \leq n$. Thus, the pseudocode would continue down to displaying sum, which is 15. This is indeed the correct answer here ($1 + 2 + 3 + 4 + 5 = 15$).

One test is never enough to conclude anything, though. A good programmer would try something else. Often, it happens that a program is wrong for very small inputs. So let's suppose the user runs the program and types 1. Then what happens?

n	5			
sum		0		1
i			1		2

Now it would display $sum = 1$, which is indeed the sum of the numbers from 1 to 1.

Once we have our overall design down, we can proceed to a line-by-line translation, in which we take each line independently and create a translation of that line alone. The following diagram illustrates the process.



Each line translates to a handful of assembly language instructions. Most translations are straightforward.

The only non-obvious part of this translation is translating the line "**while** $i \leq n$, **do:**". This expresses that we want to repeat the steps inside several times, and so at the bottom, it will be necessary to jump back to the beginning. Thus we begin the translation with a `while` label, and at the end (for "**end while**") we place a "`JUMP while`" instruction. We place a `done` label on the line immediately following; we want to jump there when it's no longer the case that $i \leq n$ — that is, we want to jump to `done` when $i > n$. To test whether $i > n$, we can test instead whether $i - n > 0$. (You can see that this is equivalent by subtracting n from both sides of $i > n$.) Thus, at the top, before we go into the steps after the "**do:**", we see assembly code for computing $i - n$ in the accumulator, and then a JPOS instruction saying to jump to `done` if the result is positive.

As we perform this translation, we worry about translating each line alone, without worrying about the other lines. If the pseudocode is correct, then this will give us a correct program. Writing pseudocode allows

us to worry about the overall design issues first, and then the translation into assembly language should be a straightforward task.

5.4.2 Pseudocode examples

Learning to write pseudocode is a skill that requires looking at more than one example. In this section, we look at several more. As you read through these examples, try stepping through them with some small numbers to verify that they are correct (and that you understand them).

Even though this pseudocode follows a strict system (which we'll examine later), remember that such a systematic technique is not important to the pseudocode concept. The most important thing is to take the problem and separate it into discrete steps, each written in English on a different line.

Printing up to n Suppose we want to read a number n and print the integers counting up to n.

```
Read n.
Initialize i to 1.
while i ≤ n, do:
    Write i.
    Increment i.
end while
Stop.
```

Computing 2^n Suppose we want to read a number n and print 2^n.

```
Read n.
Initialize power to 1.
repeat n times:
    Double power.
end repeat
Write power.
Stop.
```

Multiplication Suppose we want to read two integers m and n and print their product, $m \cdot n$.

```
Read m.
Read n.
Initialize sum to 0.
repeat n times:
    Increase sum by m.
end repeat
Print sum.
Stop.
```

Fibonacci sequence Suppose we want to read an integer n and print the first n numbers in Fibonacci sequence. The Fibonacci sequence,

$$\langle 1, 1, 2, 3, 5, 8, 13, \ldots \rangle ,$$

begins with two 1's, and each successive number is the sum of the preceding two numbers (e.g., $13 = 5 + 8$).

Read n.
Initialize a to 1.
Write 1.
Initialize b to 1.
repeat $n - 1$ **times:**
    if $a > b$, **then:**
        Write b.
        Increase b by a.
    else:
        Write a.
        Increase a by b.
    end if
end repeat
Stop.

5.4.3 Systematic pseudocode

In general, pseudocode will be composed of three different constructs.

Imperative statements are English descriptions of single things to do. Frequently, the imperative statement will involve changing the value associated with a variable, as in "Read n" or "Double $power$." Imperative statements in pseudocode generally should not involve doing something several times. ("Let sum be the sum of the integers from 1 to n," should not appear in pseudocode, for example.)

Conditional statements say to do a sequence of steps only in particular conditions. In the pseudocode examples in the preceding section, this was represented by the **if**. . . **then** construct in the final example.

if . . ., **then:**
    ⋮
else:
    ⋮
end if

construct of the final example. This indicated to do the steps following **then** in one case (when $a > b$), and to do the steps following **else** in others (i.e., when $a \leq b$).

Repetition statements say to perform some sequence of steps repeatedly. The pseudocode examples we've seen include two types of such constructs.

while . . ., **do:**	**repeat** . . . **times:**
⋮	⋮
end while	**end repeat**

You can write good pseudocode for any task based on these three categories of constructs.

Once you've written and tested your pseudocode, you can mechanically translate it into a HYMN program. But you may be wondering: If this translation is so mechanical, then why not have the computer do it for us?

```
program SumConsecutive;

var
    Sum, I, N : integer;

begin
    readln(N);
    Sum := 0;
    I := 0;
    while I <= N do
      begin
        Sum := Sum + I;
        I := I + 1
      end;
    writeln(Sum)
end.
```

(a) Pascal

```
#include <stdio.h>

int main() {
    int n, sum, i;

    scanf("%d", &n);
    sum = 0;
    i = 0;
    while(i <= n) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    return 0;
}
```

(b) C

Figure 5.2: Example programs in high-level languages.

This is, in fact, the idea behind **high-level languages**, also called **programming languages**. Some popular programming languages include C, C++, and Java. These languages are basically dialects of pseudocode that have been defined narrowly enough that a computer can break it into pieces. Figure 5.2 gives some example programs written in two well-known high-level programming language, Pascal and C. You can see that they are more similar to pseudocode than they are to the assembly language translation.

A computer program called a **compiler** will read a program written in the high-level language and translates it into an assembly language program. The compiled program can then run on the computer. Compilers are complex programs, but they work very well in their translation, often generating better assembly language programs than humans can manage.

5.5 Features of real computers (optional)

While HYMN incorporates most of the concepts in computer design, it does skip over a few additional concepts. In this section, we examine a few of the major differences between HYMN and real computers.

5.5.1 Size

Typical computers are much bigger than HYMN in at least three ways. First, and most significantly, they have more RAM. HYMN allows only 32 bytes of RAM. This is a major limitation on the size of programs we can write and the amount of data the computer can remember. Computers found in the real world tend to have many megabytes (MB) or even gigabytes (GB) of RAM.

A second way in which a real computer is bigger is in the size of the instruction set. HYMN has only 8 types of instructions. Actual computers tend to have between 50 and 200 instruction types. These instructions allow the computer to incorporate a variety of useful arithmetic and logical operations, to compute with several data types (such as various integer lengths (16, 32, and 64 bits) and floating-point lengths), and to provide features useful for operating systems design.

```
        READ             # Read which prime to access from user.
        ADD primes_addr  # Computer memory address to access.
        LOAD_AC          # Load from that address. (LOAD_AC is not in HYMN.)
        WRITE            # And display that data.
        HALT
primes: 2
         3
         5
         7
         11
         13
primes_addr: primes
```

Figure 5.3: A pseudo-HYMN program illustrating an array.

Finally, while HYMN incorporates only three registers, a real computer would use many more registers. An Intel Pentium chip, which has fewer registers than most, has eight registers for holding 32-bit integers (each analogous to HYMN's accumulator), eight registers for 80-bit floating-point numbers, and many others for specific purposes (including one analogous to HYMN's PC and other internal registers analogous to HYMN's IR).

5.5.2 Accessing memory

HYMN's architecture incorporates a memory address into each LOAD, STORE, ADD, and SUB instructions. Real computers also provide the capability of accessing memory based on a register's value, called **indirect addressing**.

This capability is useful when you want a list (called an *array*) of several pieces of data in adjacent memory locations. The program might ask the user which number to access in the list, and the number the user types would go into a register. Based on this, the program can compute the address of the memory slot containing the data, placing the result into a register. Using indirect addressing, the program can access data.

As an example of how this might work, we can suppose there were a LOAD_AC instruction.

$$\text{LOAD\_AC} \quad \text{AC} \leftarrow \mathbf{M}[\text{AC}];\ \text{PC} \leftarrow \text{PC} + 1$$

Figure 5.3 contains a program that uses this hypothetical instruction. The program reads a number n from the user and displays the nth item of a list of prime numbers contained in memory.

5.5.3 Computed jumps

A similar thing happens with HYMN's JUMP, JPOS, and JZER instructions: The address to which to jump is incorporated directly in the instruction. Real computers also include the capability to jump based on a register's value.

This concept is useful for a **subroutine**, which is a piece of code designed to be used from other locations in a program. Suppose there is a particular type of computation that is useful in many places in the program; for example, computers rarely have an instruction to raise a integer to a power. Rather than duplicating the code for exponentiation several times within a program, a programmer can write a single exponentation subroutine and simply **call** the subroutine to make it happen.

To see how we might do this through HYMN, suppose that we have a program that includes a subroutine. Before the program JUMPs into the subroutine, it must first store the address of the instruction following

```
        READ             # Read m and n from user, storing them where
        STORE exp_m      #  exp subroutine expects.
        READ
        STORE exp_n
        LOAD_PC          # Load where to go after finishing subroutine
        JUMP exp         # Now jump into the exp subroutine.
        LOAD exp_val     # The exp subroutine will jump here when done.
        WRITE
        HALT

exp:    STORE exp_ret    # Store return address for safekeeping
         # (Subroutine code omitted. It would compute exp_m to the
         # (exp_n)th power and place the result into exp_val)
        LOAD exp_ret
        JUMP_AC          # (JUMP_AC is not in the HYMN definition.)
exp_m:   0
exp_n:   0
exp_val: 0
exp_ret: 0
```

Figure 5.4: A pseudo-HYMN program illustrating a subroutine.

the JUMP into some location, perhaps the AC. When we JUMP into the subroutine, it can perform its computation and, once it has completed, it can copy the AC value back into PC to return back to the instruction following the JUMP. To construct at a program illustrating this, we need two new instructions in HYMN's instruction set.

LOAD_PC	$\text{AC} \leftarrow \text{PC} + 2; \text{PC} \leftarrow \text{PC} + 1$
JUMP_AC	$\text{PC} \leftarrow \text{AC}$

Figure 5.4 contains a program using these hypothetical instructions to illustrate how a subroutine might be called.

Chapter 6

The operating system

Computers typically use a special piece of software called an **operating system**; the most popular operating systems for personal computers today are MacOS, Microsoft Windows, and Linux. In this chapter, we'll survey what this software does and how it accomplishes its tasks.

6.1 Disk technology

Before we explore operating systems, we need to look a little more carefully at the hardware in today's computers. In HYMN, we've already gotten an idea of the two most important components of today's computers — the CPU and the RAM — and how these work. Computers systems can have many other components, though, including such devices as display screens, keyboards, mice, speakers, hard disks, and CD-ROM drives. These devices are called **peripherals** because they are secondary to the primary components (the CPU and the RAM).

Among the peripherals, only one close to being vital to the modern computer's operation: the hard disk. We'll look briefly at how hard disks work before continuing.

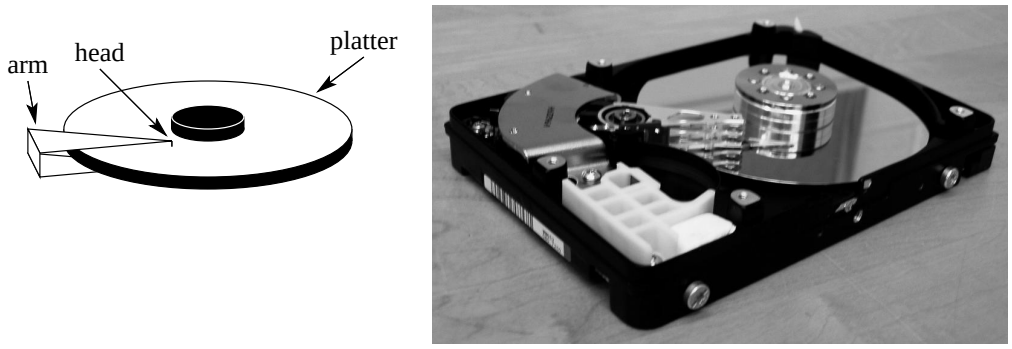
The hard disk is similar to RAM in its function: It stores data. But it has some important differences. Most notable is the fact that data stored on the hard disk persists even when the power is turned off. But, also, the hard disk is also much cheaper. The primary reason computers don't use it exclusively is because the technology is also much slower.

The technology underlying hard disks is significantly different from that of RAM. Hard disks include one or more platters of magnetic material, and each bit is stored in a particular tiny region of the disk, based on the current polarization of the magnetic charge within that region. The magnetic charge does not require any current to maintain its polarization, and this accounts for why data on the disk lasts for long periods without power.

For reading or writing the charge at a location at a disk, the disk has an *arm* which can move toward or away from the platter. With the platters rotating, and the arm moving to and from the center of the platter, the arm can access any position on the platter. At the end of the arm is the *head*, which has the ability to detect or change the magnetic charge at the point underneath it.

Figure 6.1 illustrates the internals of a hard disk. This is what you would see if you were to open up the box in which it is encased. (Normally, the disk is encased in a steel box, tightly sealed to prevent dust from getting in and interfering with the head and scratching the disk.) This particular disk has only one platter, with a head for each side. Disks frequently have two, three, or even more platters.

Disks tend to be slow. This is surprising when you consider that a high-quality disk revolves up to 15,000 times a minute. But then we perform a calculation of how long it would take. On average, the arm



(a) (b)

Figure 6.1: The internals of a hard disk. (This particular hard disk has only one platter.)

must wait for $1/2$ a revolution before the desired data comes around to be under it.

$$1/2 \text{ rev} \times \frac{\text{min}}{15,000 \text{ rev}} \times \frac{60 \text{ s}}{\text{min}} = \frac{1,000 \text{ ms}}{\text{s}} = 2 \text{ ms}$$

That time — 2 ms — may not sound like much, but you need to remember that the clocks on computers often run 2 billion pulses a second (2GHz) or faster. If the computer can complete an instruction every pulse — and most computers can —, this means 2 billion instructions a second. In those 2 ms it takes to load data from the disk, the computer can complete 4 million instructions. (By contrast, RAM tends to take nanoseconds to access. It's still slow relative to the CPU, but it only takes a few dozen instructions.)

To reduce the penalty of the time delay, disks usually read and write blocks of data, called, appropriately enough, *blocks*. Thus, the 2 ms delay is for reading a full block. (The bytes are packed so densely on the disk that waiting for the disk to rotate through the whole block is relatively fast.) In a typical disk, a block might hold 4 kilobytes.

6.2 Operating system definition

The **operating system** manages the computer's resources for the benefit of programs running on the computer.

6.2.1 Virtual machines

The operating system acts as a **virtual machine** for programs, giving designers of other programs the illusion that using the computer for display, input, file access, printing, etc., is much easier than it really is. While the operating system must worry about issues like where exactly bytes are distributed on a disk, for example, the programmer of other software can imagine a file as simply a sequence of bytes.

There are several virtual machines in a modern computing system, arranged in layers as in Figure 6.2. Without such layers of abstraction, designing a large computer system would be like designing a ship by listing how each individual board is cut and joined.

Above the operating system in Figure 6.2 is another virtual machine, represented by the programming language. The language is usually designed to work across a variety of operating systems, so that a single program written in the language will work with other systems. They also often provide the illusion of new capabilities; for example, some programming languages make it seem that the computer has a built-in

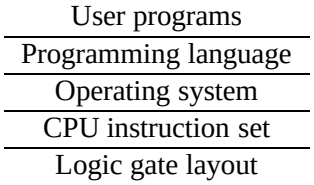


Figure 6.2: Layers of abstraction in a computer system.

capacity for exponentiating numbers, but many operating systems and instruction sets do not have such an operation.

Below the operating system is the CPU's instruction set. This, too, is a virtual machine, which allows designers to write an operating system without worrying about how the CPU's gates are actually arranged.

6.2.2 Benefits

The operating system, then, serves primarily as an intermediary between the programs and the computer hardware. As an intermediary, it provides three major benefits.

It abstracts complex computer resources. For example, when most programs want to store something on the disk, they want to be able to work with a sequence of bytes called a **file**. Each file has a distinct name, and different files can have different lengths. However, the disk inside a computer (on which files are stored) is a simple device that is not sophisticated enough to handle the concept of a file; all that a disk can do is to read and write fixed-size blocks of data to particular locations identified by number. The operating system creates the *file* abstraction to simplify access to the disk for other programs.

Another important abstraction is the **window** seen in graphical user interfaces. The window has no basis in computer display technology; the only way to draw something on the screen is to tell it how to color individual pixels. Allowing individual programs to send such messages to the screen would lead to chaos in an environment where multiple programs contend to display information to the user. To work around this, the operating system creates the *window* abstraction of a rectangular area of the display, and individual programs can request from the operating system a new window into which they can draw without any danger of contention with other programs for the same display space.

Other important abstractions include the *process* abstraction for a running program and the *connection* abstraction for network communication.

It provides hardware compatibility. When you first think of operating systems, you're likely to think that they cause *incompatibility* issues. After all, one of the first questions asked about new software or hardware is, does it work with my operating system? But in fact they reduce incompatibility problems: We don't recognize this because they reduce compatibility problems in the realm of computer hardware so effectively.

As an example of the hardware compatibility issue, consider the many types of disk designs available: There are many technologies (hard disks, floppy disks, CD-ROMs), and even if you choose just one technology, manufacturers often build their disks to work with different interfaces, usually to improve performance for their particular disk. Without an operating system, you would need code in each of your programs to support each type of disk interface. For each program you acquire, you'd have to check that it works with the specific disk types your computer has (as well as the same display, the same network device, the same printer, etc.). And, if you decide to buy a new disk, you would find

that it would be compatible with some of your programs (which already contain the relevant code) but not with others.

The operating system saves us from this chaos. Because each program accesses the disk (and display, network, printer, etc.) via the abstractions provided by the operating system, it's only important that the hardware be compatible with the operating system. Most operating systems provide a way of extending their hardware facilities through software called a **driver**, so that manufacturers who produce new hardware can distribute the necessary driver software along with their hardware. One can install the driver in the operating system once, and the hardware is immediately compatible with all of the programs running on the computer.

The operating system protects the overall computer system. While it may sound initially nice to give each programmer full freedom of access to the computer system, such trust opens a system to catastrophes. Among these catastrophes is the possibility that a user might download and run a program that appears useful or interesting but in fact does something like wipe the disk. (Such a program is called a **trojan horse**.) Even in programs written with good intentions, there are often errors ("bugs") that a user could accidentally trigger.

To prevent this, the operating system acts as an intermediary between each individual program and the rest of the system. A program requests something from the operating system using a **system call**, and the operating system verifies that the request is acceptable before completing it.

You can think of an operating system as the adult in the computer, parenting the young user programs. An adult often explains events at the child's level using metaphors (those are the abstractions) and performs tasks, like buying a piece of candy, that the child can't handle on its own.

6.3 Processes

Each instance of a running program is termed a **process**. Like other abstractions created by the operating system, processes don't exist in the hardware; they are a concept created by the operating system to allow it to work with active programs.

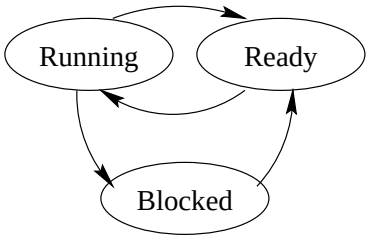
Today's sophisticated operating systems can have several processes active simultaneously. Often, these processes are programs that the user initiated (like a mail reader or Web browser), but they can also be processes that run silently in the background (like a program that archives e-mail sent from other computers). As I write this on my Linux system, the system is managing 80 active processes.

6.3.1 Context switching

The CPU has one thread of execution — that is, it does only one thing at once.* The OS must provide to each process the illusion that it "owns" the computer. Thus, the OS will switch processes on and off the CPU; the amount of time the OS runs before it interrupts a process is called the process' **time slice**. The OS designer will choose the time slice duration to be small enough that a human user can't distinguish the difference, but not so small that the OS spends much of its time rotating processes on and off the CPU.

During its life, a process cycles between three states.

*Actually, today's CPUs are much more complex than this; they often work on several instructions simultaneously. However, to keep the overall CPU design simple, most CPUs provide the illusion that the computer does only one instruction at a time. The operating system, built using the CPU's instruction set, is grounded on this illusion.



Running The CPU is currently executing instructions for the process.

Ready The process is prepared for the CPU to execute its instructions, but the CPU is doing something else.

Blocked The process cannot continue its computation, usually because it is waiting for a hardware device to send it information. For example, when a process asks to read something from a file, the disk can take several milliseconds to generate a response. During this time, the process cannot continue, and so the process is "blocked." While it is blocked, the computer could execute millions of instructions for other processes.

Many processes spend most of their time in the Blocked state, often because they are waiting for the user to give additional input via the mouse or keyboard. While my computer system has over 80 processes active right now, in fact only a small handful (often, just 1) are in the Ready or Running state.

The OS should be designed so that each program can be written as if it has sole control of the CPU. One of the most important elements of this is that each program should "own" the CPU's registers. With the HYMN architecture, we want to write programs so that a number placed into the accumulator will remain there until a subsequent instruction in the same program replaces it.

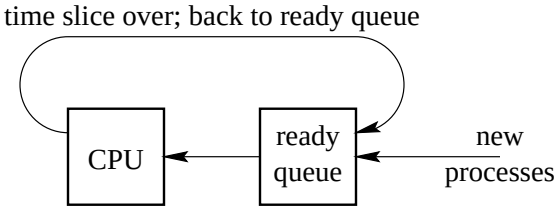
However, when the operating system switches to another program, that other program will have its own ideas of what should be in the registers. Thus, when the OS moves a process from the Running state to the Ready or Blocked state, it will have to save the current values in each of the registers to a place in memory. The OS maintains a **process table** in its memory to track data like this about each current process. Then, just before the OS moves a process into the Running state again, the OS can restore the registers stored in next process' process table entry. In this way, the next process continues from where it left off with the same register values that existed when that process moved out of the Running state. This procedure of saving one process's context (including its registers) and restoring another is called a **context switch**.

To illustrate, suppose our computer has two processes, A and B.

1. The computer runs program A for 10 milliseconds. (Ten milliseconds is a reasonable period for a time slice, based on the fact that humans cannot perceive time differences smaller than around 40 milliseconds. Of course, the latter fact is also why movies have a frame rate of 24 frames a second (24×40 ms ≈ 1 s).)
2. The operating system takes over. It saves the current register values (those that A had placed there) into A's entry of the process table.
3. The operating system restores the register values stored in B's entry of the process table.
4. The OS jumps into program B for 10 milliseconds.
5. The operating system takes over again. It saves the register values into B's process table entry.
6. The operating system restores the register values from A's process table entry.
7. The computer repeats the process.

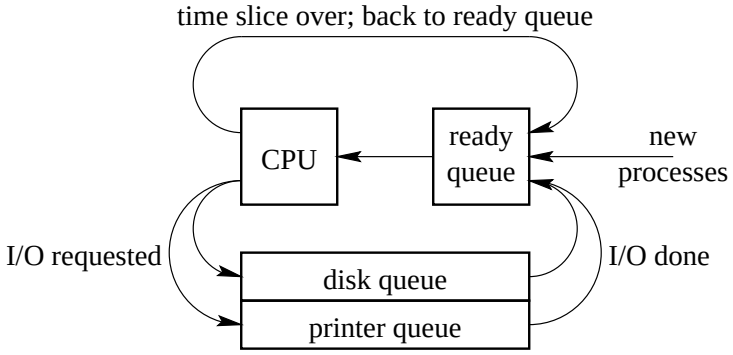
6.3.2 CPU allocation

Because there can be many processes in the Ready state at any time, the computer maintains a **ready queue** to track these processes.



The ready queue is where the processes politely line up, waiting for their turn with the CPU. (Technically, the processes can't really "do" anything like stand in line when they're not on the CPU. This is just a way of talking about how the operating system is managing what it knows about the processes.)

When there are I/O devices, like disks, keyboards, or printers, things get more complicated. (**I/O** stands for **I**nput/**O**utput.) With such devices, processes might enter the Blocked state waiting for information from them. The operating system will maintain an **I/O wait queue** for each of these devices.



A process requesting access to a device goes into the device's I/O wait queue until the device handles the request.

The example of Figure 6.3 illustrates how this works. Suppose we have a computer system with the timing assumptions of Figure 6.3(a), and we begin with the four processes of Figure 6.3(b) starting simultaneously in the ready queue. Figure 6.3(c) tabulates how the OS would manage these processes.[†] Note that it has taken a total of 29 ms for the computer to finish all the processes, with an average completion time of

$$\frac{19.5 + 29.0 + 13.0 + 22.0}{4} = 20.975 \text{ ms}$$

per process.

Now, for the sake of argument, suppose that our system only allows one process at any time, and so we must run the programs in sequence.

A takes $2 + 4 + 1 + 3 + 1$	$= 11$ ms, finishing after	11 ms
B takes $1 + 4 + 3 + 1 + 4 + 1$	$= 14$ ms, finishing after	$11 + 14 = 25$ ms
C takes 5	$= 5$ ms, finishing after	$25 + 5 = 30$ ms
D takes $1 + 3 + 5$	$= 9$ ms, finishing after	$30 + 9 = 39$ ms

[†]There is a slight ambiguity at time 18.0, when both B and D enter the ready queue: The choice of D entering first is arbitrary. Also, this table neglects several unimportant details; for example, at time 12.0, the OS would have to suspend C while the OS performs the task of moving D into the ready queue.

The time slice for a process is 3 ms.
The time to execute a context switch is 0.5 ms.
The printer takes 4 ms to respond to a request.
The disk takes 3 ms to respond to a request.

(a) Timing facts for a computer system.

A	B	C	D
run 2 ms	run 1 ms	run 5 ms	run 1 ms
print	print		use disk
run 1 ms	run 1 ms		run 5 ms
use disk	use disk		
run 1 ms	run 1 ms		
	print		
	run 1 ms		

(b) The details of four processes' work.

time	CPU	ready queue	disk queue	printer queue	comment
0.0		$ABCD$			(starting configuration)
0.5	A	BCD			A enters CPU
2.5		BCD		A	A requests printer
3.0	B	CD		A	B enters CPU
4.0		CD		AB	B requests printer
4.5	C	D		AB	C enters CPU
6.5	C	DA		B	A finishes with printer
7.5		DAC		B	C's time slice expires
8.0	D	AC		B	D enters CPU
9.0		AC	D	B	D requests disk
9.5	A	C	D	B	A enters CPU
10.5		CB	DA		A requests disk; B finishes printer
11.0	C	B	DA		C enters CPU
12.0	C	BD	A		D finishes with disk
13.0		BD	A		C ends
13.5	B	D	A		B enters CPU
14.5		D	AB		B requests disk
15.0	D	A	B		D enters CPU; A finishes with disk
18.0		ADB			D's time slice done; B finishes disk
18.5	A	DB			A enters CPU
19.5		DB			A ends
20.0	D	B			D enters CPU
22.0		B			D ends
22.5	B				B enters CPU
23.5				B	B requests printer
27.5		B			B finishes with printer
28.0	B				B enters CPU
29.0					B ends

(c) A timeline of the OS running the processes. (Queues begin from the left.)

Figure 6.3: An example of OS process management.

Thus, without context switching, the computer would take 39 ms to finish all four processes, with an average completion time of

$$\frac{11 + 25 + 30 + 39}{4} = 26.25 \text{ ms}$$

per process. This is significantly slower than the system with context switching, which took 29 ms total, with an average completion time of 20.975 ms.

It's a bit weird that in adding the expense of context switching, the time taken to finish all the processes actually decreases. A good analogy is a cashier in a grocery store. Suppose the cashier started checking out the next person in line while you were counting up money from your wallet to pay for your groceries. You may find this irritating, because you know that if the cashier gave you total attention, you wouldn't have to wait for the cashier to check out the next person in line. But, overall, this strategy gets people through the line more quickly, since the cashier is not wasting time waiting for customers to count money.

6.3.3 Memory allocation

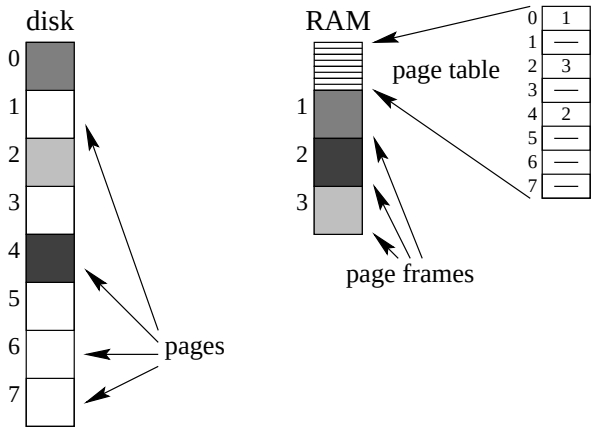
As we saw when we examined the HYMN architecture, the two most central elements to the computer system are the CPU and RAM. In the previous section, we saw how the operating system can manage the CPU to provide for the possibility of multiple processes. Managing memory is just as important an issue.

Swapping Early operating systems used the simple technique of **swapping** to provide for multiple processes. In this system, the computer treats the contents of RAM as part of the process' context, which is swapped along with the register values with each context switch. Of course, the outgoing process's memory must be stored somewhere other than RAM, since the RAM is needed for the incoming process's memory, and so the operating system stashes the data on disk.

This system, though it works, makes for extremely expensive context switches. In such a system, context switching involves both copying the current process's memory from RAM to disk and copying the next process's memory from disk to RAM. Because of the access time of milliseconds for disks, and because processes can have lots of memory, this can add up to a lot of time. Computers that use this simple technique do so only because the simplicity of the CPU makes it the only viable approach.

Paging To avoid the cost of swapping entire processes between disk and RAM, most computer systems today use a system called **virtual memory** (also called **paging**) for allocating memory to processes. In this system, the CPU works with a "virtual address space," which is divided into small pieces called **pages**, of typically one to four kilobytes. The CPU uses a **page table**, which says for each page whether the page is located in RAM and, if so, the address where it starts.

For example, suppose we have a system with four kilobytes of RAM, and we want eight kilobytes of virtual memory.



The system would allocate eight kilobytes on disk to store the pages, and it would divide memory into a page table and three separate **page frames** in RAM that can each potentially hold a single page of memory. In this example, pages 0, 2, and 4 are located in page frames 1, 3, and 2 of RAM. You can see in the page table (located in the first kilobyte of RAM) that it says that page 0 is in frame 1, page 1 is not in any frame, page 2 is in frame 3, and so on.

When a program asks to load memory from an address, the CPU determines which page contains the address, and the CPU refers to the page table (in RAM) to determine whether that page is in RAM. If so, then the page table also says which page frame contains the memory, and the CPU can look within that frame to find the data requested by the program. If the page table indicates that the page is not in RAM, then the CPU generates a **page fault**, which is a signal for the the operating system to load the page into some page frame. The operating system will load the page and update the page table to reflect that the requested page is in the frame and that the page previously in that frame is no longer in RAM.

The advantage of virtual memory is that it only needs to keep the memory that is currently useful in RAM at any time. Processes frequently request large amounts of memory, but they use the memory unfrequently; for example, many Web browsers can play sounds from a Web page, and the code to play these sounds takes up some of the Web browser's memory, but this code lies unused when the user is viewing pages with no accompanying sound. With swapping, this unused code would be copied from disk each time the Web browser is swapped into memory, even though that memory may never be used before the next context switch; with paging, it would only be loaded when it is needed.

Another major advantage is that virtual memory dramatically reduces the need to worry about the amount of RAM in a computer. The computer will not refuse to run processes just because the computer doesn't have enough RAM to fit it into memory; the operating system only needs to be able to fit the process into the virtual memory space, which is vast enough to be essentially infinite.

This does not mean, however, that RAM is irrelevant. If you have too little RAM to store the pages that are frequently needed by the current processes, then the computer will generate frequent page faults. People call such heavy swapping of pages **page thrashing**; when it is happening, you can often hear the noise of the hard drive being accessed continuously even though nobody is opening or closing files, and you will feel the system going dramatically slower as it repeatedly retrieves data from disk that ought to be in RAM. Page thrashing indicates that the computer would be going much faster if it had more RAM, or if the current processes needed less memory.

Chapter 7

Artificial intelligence

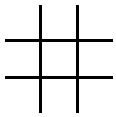
Although artificial intelligence research dates from the dawn of computer science, its goals are so ambitious that it still has far to go. We begin this chapter by exploring the techniques lying behind computer game players. Then we will examine philosophical thought regarding artificial intelligence, and we'll look at some attempts at writing programs that learn which are inspired by the biology of the human brain.

7.1 Playing games

The motivation behind game-playing research is much more serious than it sounds. The primary goal is to have computers adapt and plan, so that they can handle serious tasks like driving a car or managing a production line. Game-playing as a topic of study came about because it was fun, manageable, but somewhat beyond current technology. For similar reasons, some robotics researchers today concentrate on creating robots to juggle — not because juggling is a useful task, but because it requires dexterity and quick thinking that robots need but currently lack.

Classical game-playing techniques work for a variety of games with certain common characteristics. We assume that the game involves two players alternating turns. We assume that both players always know everything about the current state of the game. (This is not true for many card games, for example, because a player does not know the other's hand.) And we assume that the number of moves on each turn is limited. These restrictions encompass many games, including tic-tac-toe, Connect-4, Othello, checkers, chess, and go. Except for go, the techniques covered in this chapter work well for all of the games just listed.

In this chapter we look at the simplest of these, tic-tac-toe. In case your childhood somehow lacked tic-tac-toe, let us review the rules. We start with a 3×3 board, all blank.



It is X's turn first, and X can place his mark in any of the nine blanks. Then O places her mark in one of the eight remaining blanks. In response X has seven choices. In this way the players alternate turns until one of the players has three marks along a horizontal, vertical, or diagonal line (thus winning the game), or until the board becomes filled (this is a tie if neither player has won).

One approach to writing a tic-tac-toe program is to simply enumerate the situations that may occur and what the computer should do in each case. For example, if the computer is O, and X's first move is in a corner, then O should play in the center. If X's first move is in the center, O should play in a corner. And so on. This approach suffers from two major problems. First, while such a list is feasible for a simple game like tic-tac-toe, it is not for more complex games, which have too many possibilities to each be individually

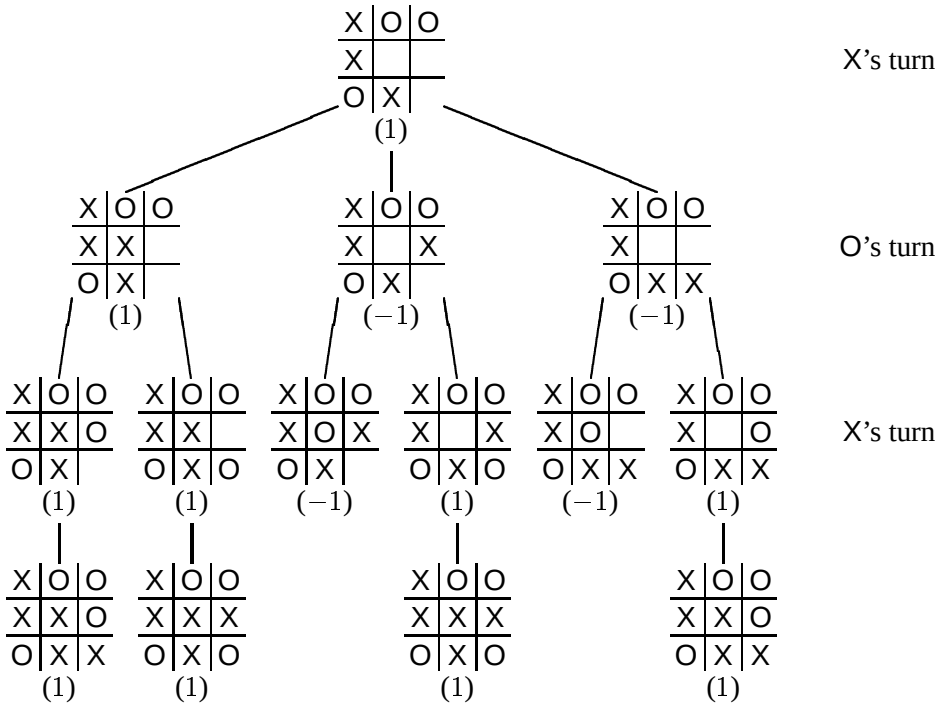
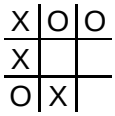


Figure 7.1: Evaluating a board.

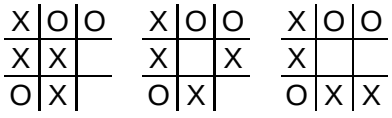
considered by a human. Just as serious, a program playing according to a programmer-provided list will never play any better than the programmer; it's hard to see how such an approach demonstrates intelligence.

7.1.1 Game tree search

A more general approach to is have the computer determine how to move by evaluating choices on its own. Say the current board is



and the computer, playing X, must choose a move. To do this, the computer can consider each of the three possible next boards and consider which is most appealing.



To determine which board is best, the computer can evaluate each one by examining possible moves from it. And to evaluate these resulting boards, the computer can consider the possible moves from them. Essentially, the computer explores all possible futures, which we can picture with a diagram called a **game tree**, as in Figure 7.1.

The parenthesized numbers in Figure 7.1 indicate the "value" determined for each board: We use 0 for a tie, 1 for a guaranteed win for X, and -1 for a guaranteed win for O. At the bottom, when a final board is reached, the value of the board is the outcome for that board: In the figure, the bottom left board is 1 because

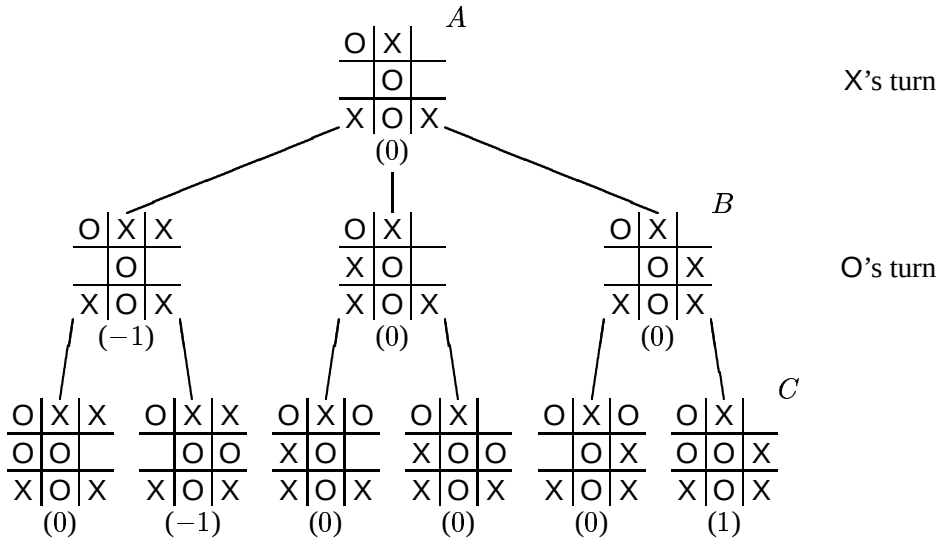


Figure 7.2: Using heuristics to evaluate a board.

X has completed the diagonal. For other boards, the value is the best of the choices for the current player. For the top board, we have three choices: a win for X, a win for O, or a win for O. It is X's turn, so X would choose the win for X; hence the board's value is 1, and X should move in the board's center.

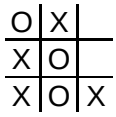
Evaluating such a tree is called the **minimax search** algorithm, since X chooses the maximum of its childrens' values and O chooses the minimum.

7.1.2 Heuristics

The problem with minimax search is that it takes a lot of time. Tic-tac-toe games, which last at most 9 moves, have manageable game trees. But a chess game may last more than 50 moves; the game tree is well beyond the total computing capacity of the world.

The solution is simple. We search only to a certain depth of the tree. When we see a board at the depth that is not in a final state, we apply a **heuristic function** to estimate the board's value. The heuristic function is a function written by the programmer that tells roughly how good the board is.

In tic-tac-toe, a simple heuristic function may calculate the difference of the number of possible wins for X and the number of possible wins for O, where a possible win is a row, column, or diagonal with none of the opponent's pieces. The board



has one possible win for X (the right column) and no possible wins for O; its heuristic value would be 1. We should also make the value of guaranteed wins more extreme (10^6 and -10^6, say) to indicate how sure we are of them.

With such a heuristic function defined, we can evaluate a board by going to a certain depth and using the heuristic function to evaluate the boards at the bottom depth that are not final. We use the same minimax procedure for boards above the maximum depth. Figure 7.2 illustrates an example going to a depth of 2. In this example, X would decide for either the second or third choices.

7.1.3 Alpha-beta search

Heuristics coupled with fixed-depth searching allow reasonably good game-playing programs. To improve performance, however, we can look for any unnecessary computation. One particularly interesting enhancement, which most high-quality game-playing programs use, is called **alpha-beta search**.* This technique allows the computer to skip over some boards in its computation without sacrificing the correctness of its result. That is, we can observe that some of the game tree's results are irrelevant before we reach it, and this can allow us to skip over those portions. This reduced computational cost allows a game-playing program to search to a greater depth.

Figure 7.2 provides an example where this applies. Call the right-most board in the bottom level C, its parent B, and the top of the tree A. Notice that, no matter what the value of C is, the value of B will be at most 0, since O will choose the minimum of its children's values and B already knows that the first choice gives 0. Since at A X already knows it can guarantee 0 by choosing the middle route, the exact value of B does not matter. Through this reasoning, then, we can avoid evaluating C.

In this case we would avoid evaluating a single board — not so impressive. But the reasoning can help tremendously for larger games, almost doubling the depth that can be handled within the time limit.

7.1.4 Summary

Alpha-beta search is very close to what the best game programs use. The programs do, however, have some additional enhancements. For example, a good chess program will have a large list describing specific moves and responses for the beginning of the game. It may also vary the search depth based on how good the board looks, rather than going to a fixed depth. But aside from such minor enhancements, the technique is not much more sophisticated than alpha-beta search. The primary difference between programs is in the sophistication of the heuristic function.

Unfortunately, although these techniques have proven very effective for playing games, they do not generalize well to other planning tasks, where the world is much larger than a few pieces on a board and actions sometimes fail to produce the desired result. (Juggling is an example: You can't predict exactly where something tossed into the air will land, because the effects of rotation and wind currents are too complex.) These real-world problems are much harder. Researchers are currently addressing them, but a long time will pass before they might develop solutions to such problems. Game-playing is just a first step.

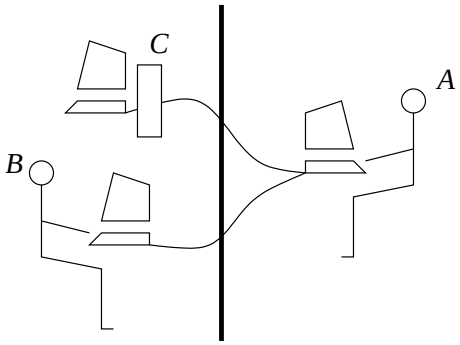
7.2 Nature of intelligence

Philosophically, the game playing techniques are not very satisfying. Can one really say that a computer using exhaustive search is displaying any intelligence? While major chess computers search through millions of boards for each play, a human grandmaster searches through merely hundreds of moves and still performs as well. One cannot accurately say that a computer is actually reasoning as a human does.

7.2.1 Turing test

Alan Turing, a British mathematician working with the first computers back in the 1940's, struggled with this question of what constitutes artificial intellegence. Eventually he proposed the following way of testing whether an entity was intelligent.

*The name *alpha-beta search* is purely historical. In early descriptions of the algorithm, these two Greek letters were important variables.



To see if a computer (C) is intelligent, we place it and a human (B) behind a screen, each connected via a communication wire to a human tester (A) in front. A asks questions of B and C in an attempt to determine which is the human and which is the computer. If A can't reliably tell which of B and C is a human, then C must be intelligent.

This is called the **Turing test**. Many accept this aim as the ultimate AI goal.

When the conversation is restricted to the domain of game playing, computers appear close to passing the Turing test when restricted to games alone. After playing a historic match with a chess computer in 1996, world chess champion Garry Kasparov said of his opponent, "I could feel — I could smell — a new kind of intelligence across the table." Although he won the series then, he lost to an improved version the next year. Yet computers have not completed even this reduced version of the Turing test: Kasparov maintains that the computer has a distinctive style of play, and this indicates that the champion computer would not pass the Turing test, if only because it plays too well.

The general Turing test, though, is a much more difficult goal. It's not something that we're likely to reach soon, but it is something that indicates how we know when we're there.

7.2.2 Searle's Chinese Room experiment

Some people disagree that the Turing test is a good way to evaluate artificial intelligence. It's somewhat irritating that the Turing test is so output-oriented, they say: That computer could be doing *anything*, and we'd be saying that it is intelligent.

Such is the stance taken by philosopher John Searle in 1980. Searle proposed the following thought experiment called the **Chinese room experiment** to illustrate his stance: Suppose that everybody communicates via Chinese, and that the human behind the screen (B) doesn't know any Chinese. In principle, B can still appear intelligent, simply by having a vast phrasebook listing each possible input with some English instructions of how to respond, including a corresponding Chinese-symbol output. The book could omit a translation of what the Chinese means, so that B doesn't understand what is going on. Even so, if the phrasebook is vast enough, then B will appear intelligent to A. But if B has no idea of what is happening, Searle asks, can we say that B is behaving intelligently? (The practicality of such a phrasebook is beside the point. Searle is trying to illustrate the test's shortcoming in principle.)

Searle is saying that the Turing test is flawed — intelligence cannot be defined as simply *appearing* to be intelligent, however convenient that may be to a scientist. To be intelligent, something must actually work intelligently. We cannot define intelligence functionally; the method also matters.

Searle's argument is not universally accepted, but it stands as a credible argument against the Turing test.

7.2.3 Symbolic versus connectionist AI

Searle's problems with the Turing test bears some similarity to a long-standing debate within the artificial intelligence community, a split between those advocating *symbolic AI* and those advocating *connectionist AI*. The symbolic AI camp contends that the best way toward intelligence is to achieve behavior that appears intelligent, by any means possible. And the easiest programs to write are those that manipulate symbols (and thus they take the name *symbolists*). The minimax search technique for game playing is a symbolist's technique: It is a no-holds-barred approach to playing games.

Connectionists assert that this technique is flawed — although you may succeed on some simple problems, they say, such a program will never exceed the specific algorithms plugged into it. The program will always be brittle, breaking as soon as we move away from the restricted problem that the program was designed to solve. Instead, connectionists argue, our work on AI should focus on programs that resemble how the human brain works.

One of the arguments of connectionists is that the human brain does not resemble symbolic AI at all, so it's difficult to see how symbolic programs are solid steps toward intelligence. They might point to studies of human chess grandmasters, who can play dozens of simultaneous timed games with many different people, winning all of the games. Obviously, though beginners might play by searching through a variety of possible moves, human chess mastery involves something different than becoming more efficient at searching through moves. When we work on the minimax search technique, which relies solely on evaluating vast numbers of possible moves, we're chasing up the wrong tree.

Let's review how the brain works. Researchers don't understand it entirely, but they've done enough experimentation to understand the simplest pieces, which are simple cells called **neurons**. Each neuron has several *dendrites*, connected to other neurons via connections called *synapses*. Other neurons can send electrochemical signals through the synapses through the dendrites. Occasionally, the signals may become so intense that the neuron becomes excited and sends its own electrochemical signals down its *axon*, which are relayed through synapses to the dendrites of other neurons.

The connectionists' idea is to simulate the human brain within the computer. (They are called *connectionists* because the systems they develop rely on the *connections* between "neurons.") Since the human brain is a mechanical system, they argue, this plan can only result in success. Symbolists don't disagree with them; they simply feel that this is the difficult road to AI, with little room for intermediate success along the way.

Incidentally, Searle buys into none of this. He certainly does not agree with the symbolists but neither does he accept the connectionists' position. In fact, Searle argues (outside his Chinese room experiment) that AI is impossible. Other philosophers, too, counter that AI researchers have no chance of success. There are a variety of arguments that various philosophers propose for AI's impossibility. Some arguments are based on the assertion that AI requires a materialist view of humanity, where human behavior is understood entirely as a physical phenomenon. Philosophers who reject this materialist view (believing instead in a soul-like entity that affects humans' behavior) thus often reject the possibility of true artificial intelligence. AI advocates tend to have a materialist view of humanity, discounting the possibility that humans may have some nonmaterial being.

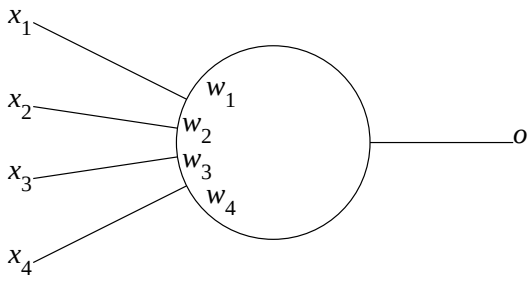
There are also some philosophers who accept the materialist view, but they still argue against the possibility of artificial intelligence. For example, a philosopher might argue that computers can't simulate reality perfectly — simulating quantum mechanics perfectly, for example, is seemingly impossible for a computer, but conceivably the human brain's behavior may depend on the intricacies of quantum mechanics.

7.3 Neural networks

To get a better idea of what connectionist AI is about, we'll look at the **perceptron**, a specific learning device whose behavior is inspired by neurons, and we'll glance at neural networks.

7.3.1 Perceptrons

You can think of a perceptron as looking like the following.



The perceptron takes a set of inputs similar to a neuron's dendrites and it uses its "thoughts" (represented by a weight for each individual dendrite) to generate an output sent along its axon. The pictured perceptron takes four inputs, x_1, x_2, x_3, and x_4, and uses four weights w_1, w_2, w_3, and w_4 to generate its output o. The inputs and outputs will each be either -1 and 1. You can think of -1 representing a FALSE value and 1 representing a TRUE value.

How does a perceptron compute its output? It finds the weighted sum of the inputs

$$w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 \ ,$$

and it outputs $o = 1$ if this sum turns out to be positive, and $o = -1$ otherwise.

For example, suppose that our perceptron's job is to predict in the morning whether somebody will raise a flag on the flagpole during the day. We might have four inputs represent answers to various questions.

1. Is it raining?
2. Is the temperature below freezing?
3. Is it a national holiday?
4. Is school in session?

If the answers to these questions are no, yes, no, and yes, then we would represent these answers to the perceptron with $\langle -1, 1, -1, 1\rangle$. Suppose the current weights within the perceptron are $\langle -0.5, 1, 0, 0.5\rangle$. Then the perceptron would compute

$$(-0.5) \cdot (-1) + 1 \cdot 1 + 0 \cdot (-1) + 0.5 \cdot 1 = 2 \ .$$

Since this is positive, the perceptron would output 1, predicting that the flag will be raised today.

If somebody does raise the flag, then the perceptron was correct. When the perceptron is correct, it sees no need to change its weights. But when it is wrong, the perceptron updates its weights according to the following rules.

- If the correct answer is 1, and the perceptron predicts -1, the weights update according to the formula

$$w_i \leftarrow w_i + r \cdot x_i \ .$$

- If the correct answer is -1, and the perceptron predicts 1, the weights update according to the formula

$$w_i \leftarrow w_i - r \cdot x_i \, .$$

In these formulas, r represents the **learning rate**. How big this number is affects how quickly the perceptron adapts to inputs. You do not want the number too large, because otherwise the perceptron will fluctuate wildly. We'll use 0.1 for r.

Suppose we wake up the the next day and observe that it is not a national holiday, school is not in session, and it is raining and above freezing. The perceptron would compute

$$(-0.5) \cdot 1 + 1 \cdot (-1) + 0 \cdot (-1) + 0.5 \cdot (-1) = -2 \, .$$

Thus, it would output -1, predicting that the flag will not be raised.

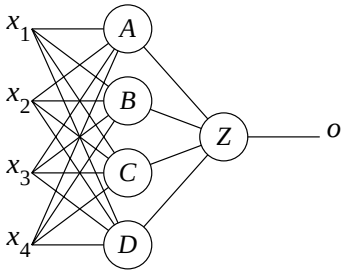
But when we actually look, we may find that the flag is up. So now the perceptron will have to adapt its weights, and they will become $\langle -0.4, 0.9, -0.1, 0.4 \rangle$. Notice that these new weights mean that the perceptron has improved based on what it has just seen: If it sees the same situation again, then it will compute

$$(-0.4) \cdot 1 + 0.9 \cdot (-1) + (-0.1) \cdot (-1) + 0.4 \cdot (-1) = -1.6,$$

whereas before it computed -2. This is closer to being positive, and if the perceptron sees the same situation several times in a row, it would keep getting closer, until it eventually got the answer right.

7.3.2 Networks

A single perceptron can't understand much on its own. The problem is that its prediction can only depend on a particular way of combining its inputs (a *linear combination*), and usually things are more complicated than that. the hope of connectionists is that we can arrange perceptrons in a form of network where the axons of some perceptrons connect to the dendrites of others.



The inputs (x_1 through x_4) are fed to a variety of perceptrons (A through D), and each of these perceptrons say something about the inputs. These perceptrons' outputs go into another perceptron (Z), which combines them into a single output for the entire network. The idea is that the intermediate perceptrons (A through D) might end up computing useful combinations of facts about the inputs. For example, it might be that perceptron C would learn to predict whether freezing rain is likely (that is, if it is raining and below freezing), while B might learn to predict whether the person in charge feels like raising the flag is particularly worthwhile (school is in session and it is a national holiday). The final perceptron (Z) can then combine these useful facts into a more sophisticated concept than possible with a single perceptron. (With just one layer of hidden perceptrons separating the inputs from the final perceptron, this example network is still pretty simple. Neural networks can use more complex designs, but researchers tend to concentrate more on this simple design.)

The difficult part of a neural network is learning. Suppose the network predicts wrongly. Then we are faced with the problem of which perceptron is to blame. We don't necessarily want to penalize all

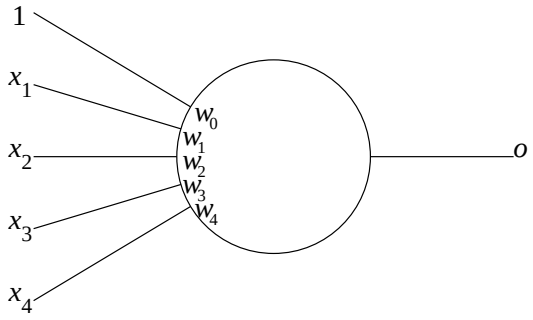
perceptrons, because some of them probably did the right thing. The perceptrons that should adapt are those that made a mistake in their output, but determining which perceptrons erred is difficult. Researchers have come up with an approach to determining this, but it's too complicated for us to examine here.

7.3.3 Computational power

One might validly wonder: How complicated a concept can a neural network represent? After all, a single perceptron, as it is defined here, is very limited in how it can combine inputs into an output. How much additional power can a whole network represent?

It's not too difficult to demonstrate that a neural network can compute anything a logic circuit can, if it simply learns the proper combination of weights. The argument is relatively simple: We simply observe that there is a setting of weights for a perceptron that makes it behave like an AND gate, and similarly a setting corresponding to an OR gate, and another corresponding to a NOT gate. It follows, then, that if you give me a circuit of AND, OR, and NOT gates, then I could give you a neural network that can represent the same concept.

For example, suppose that we have a perceptron that takes four inputs, x_1, x_2, x_3, and x_4, and we want it to compute the function x_1x_3 (the AND of x_1 and x_3). For this to work, we need our network to include an input that is always 1 to go into each perceptron; thus, we'll add another input x_0, which is always 1.



To make the perceptron compute this combination of inputs, we simply configure the weights appropriately. For example, we might choose $w_0 = -0.5$, $w_1 = 0.5$, $w_2 = 0$, $w_3 = 0.5$, and $w_4 = 0$. To verify that this works, we tabulate how it behaves for the four possible variations on x_1 and x_3 and observe that it matches the AND gate's truth table.

x_1	x_3	computation	output
-1	-1	$(-0.5) \cdot 1 + 0.5 \cdot (-1) + 0 \cdot ? + 0.5 \cdot (-1) + 0 \cdot ? = -1.5$	-1
-1	1	$(-0.5) \cdot 1 + 0.5 \cdot (-1) + 0 \cdot ? + 0.5 \cdot 1 + 0 \cdot ? = -0.5$	-1
1	-1	$(-0.5) \cdot 1 + 0.5 \cdot 1 + 0 \cdot ? + 0.5 \cdot (-1) + 0 \cdot ? = -0.5$	-1
1	1	$(-0.5) \cdot 1 + 0.5 \cdot 1 + 0 \cdot ? + 0.5 \cdot 1 + 0 \cdot ? = 0.5$	1

Because we can do this similarly for OR and NOT gates, a neural network (where a constant 1 input goes to each perceptron) can end up learning anything that can be represented by replacing the individual perceptrons with AND, OR, and NOT gates instead.

7.3.4 Case study: TD-Gammon

Classical game playing techniques work well for most two-player games where no information is hidden. But for a handful of such games, the variety of possible moves for each turn is so large that game tree techniques break down. Among these games is backgammon. (The rules to backgammon aren't important

to this discussion. Compared to other classical games, backgammon's most unusual feature is that a player rolls a pair of dice each turn, and the outcome of the roll determines the moves available.)

Researchers have put a lot of effort into backgammon using techniques based on minimax search. They haven't had much success with these techniques, though: The programs played at the level of human masters, but they weren't at the championship level. In 1991, a researcher named Tesauro finally made a breakthrough with his program, TD-Gammon, which used a radical approach based on neural networks.

TD-Gammon incorporates a neural network that takes a variety of inputs representing some state of the board and outputs a number saying how good the board is. When it is TD-Gammon's turn, it asks its neural network about the quality of the board after each possible move. Then TD-Gammon chooses the move that gives the largest value.

Game playing presents new challenges to learning because of the delayed feedback: A player may make a bad move, but the fact that it is bad would not be obvious until several moves later, when the player loses. When the player loses, then the learner is faced with the challenge of determining which of the moves is at fault. This blame-assignment problem is similar, but at a larger scale, than the blame-assignment problem faced with the problem of blaming individual perceptrons for a network's overall prediction. Again, computer scientists have a complex solution to this (called *temporal difference* learning — hence the TD in TD-Gammon's name). Tesauro generated TD-Gammon's neural network by using temporal difference learning to train the network as it played itself through 1.5 million games.

After this, Tesauro stopped training the network and began testing it against people. He found that TD-Gammon could easily beat any previous computer backgammon players, and it could even beat human backgammon players. Since then, he has trained the network more and added a small minimax search element to its move computation; the resulting version is competitive with world champions, much better than was possible with only symbolic techniques.

The success of TD-Gammon, which appears to play games well without resorting to analyzing millions of boards, is a welcome relief to those who are skeptical of the usefulness of symbolic techniques for artificial intelligence. Researchers have tried to duplicate its success for other games (such as chess). These efforts have not reached world champion levels, but most have learned to play their games at a competition level.

Chapter 8

Language and computation

In the 1950's, the American linguist Noam Chomsky considered the following question: What is the structure of language? His answer turns out to form much of the foundation of computer science. In this chapter, we examine a part of this work.

8.1 Defining language

One of the first steps to exploring linguistic structure, Chomsky decided, is to define our terms. Chomsky chose a mathematical definition: We define a **language** as a set of sequences chosen from some set of **atoms**. For the English language, the set of atoms would be a dictionary of English words, and the language would include such word sequences as the following.

this sequence contains five atoms
lend me your ears
what does this have to do with computer science

Though the language of all possible English sentences is quite large, it certainly does not include all possible sequences of atoms in our set. For example, the sequence

rise orchid door love blue

would not be in our language. Each sequence in the language is called a **sentence** of the language.

This definition of *language* is mathematical, akin to defining a *circle* as the set of points equidistant from a single point. Notice that it is general enough to allow for nontraditional languages, too, such as the following three "languages."

- Our atoms could be the letters *a* and *b*, and our language could be the set of words formed using the same number of *a*'s as *b*'s. Sentences in this language include *ba*, *abab*, and *abbaab*.
- Our atoms could be the decimal digits, and our language could be the digit sequences which represent multiples of 5. Sentences in this language include *5*, *115*, and *10000000*.
- Our atoms could be the decimal digits, and our language could be the digit sequences which represent prime numbers. Sentences in this language include *31*, *101*, and *131071*.

In analyzing linguistic structure, Chomsky came up with a variety of systems for describing a language, which have proven especially important to computer science. In this chapter, we'll study two of these systems: *context-free grammars* and *regular expressions*.

8.2 Context-free languages

One of the most important classes of language identified by Chomsky is the *context-free language*. (Chomsky called them *phrase structure grammars*, but computer scientists prefer their own term.)

8.2.1 Grammars

A **context-free grammar** is a formal definition of a language defined by some **rules**, each specifying how a single **symbol** can be replaced by one of a selection of sequences of atoms and symbols.*

We'll represent symbols using boldface letters and the atoms using italics. The following example rule involves a single symbol **S** and a single atom *a*.

$$\mathbf{S} \quad \rightarrow \quad a\ \mathbf{S}\ a \mid a$$

The arrow "$\rightarrow$" and vertical bar "|" are for representing rules. Each rule has one arrow ("$\rightarrow$"). On its left side is the single symbol which can be replaced. The sequences with which the symbol can be replaced are listed on the arrow's right side, with vertical bars ("|") separating the sequences. We can use this example rule to perform the following replacements.

$\mathbf{S} \Rightarrow a\ \mathbf{S}\ a$	replace **S** with the first alternative, $a\ \mathbf{S}\ a$.
$a\ \mathbf{S}\ a \Rightarrow a\ a\ \mathbf{S}\ a\ a$	replace **S** with the first alternative, $a\ \mathbf{S}\ a$.
$a\ a\ \mathbf{S}\ a\ a \Rightarrow a\ a\ a\ a\ a$	replace **S** with the second alternative, *a*.

A **derivation** is a sequence of steps, beginning from the symbol **S** and ending in a sequence of only atoms. Each step involves the application of a single rule from the grammar.

$$\begin{aligned} \mathbf{S} &\Rightarrow a\ \mathbf{S}\ a \\ &\Rightarrow a\ a\ \mathbf{S}\ a\ a \\ &\Rightarrow a\ a\ a\ a\ a \end{aligned}$$

In each step, we have taken a symbol from the preceding line and replaced it with a sequence chosen from the choices given in the grammar.

A little bit of thought will reveal that this grammar (consisting of just the one rule $\mathbf{S} \rightarrow a\ \mathbf{S}\ a \mid a$) allows us to derive any sentence consisting of an odd number of *a*'s. We would say that the grammar *describes* the language of strings with an odd number of *a*'s.

Let's look at a more complex example, which describes a small subset of the English language.

$$\begin{aligned} \mathbf{S} &\rightarrow \mathbf{NP\ VP} \\ \mathbf{NP} &\rightarrow \mathbf{A\ N} \mid \mathbf{PN} \\ \mathbf{VP} &\rightarrow \mathbf{V} \mid \mathbf{V\ NP} \\ \mathbf{A} &\rightarrow \textit{a} \mid \textit{the} \\ \mathbf{N} &\rightarrow \textit{cat} \mid \textit{student} \mid \textit{moon} \\ \mathbf{PN} &\rightarrow \textit{Spot} \mid \textit{Carl} \\ \mathbf{V} &\rightarrow \textit{sees} \mid \textit{knows} \end{aligned}$$

*The term *context-free* refers to the fact that each rule describes how the symbol can be replaced, regardless of what surrounds the symbol (its *context*). This contrasts it with the broader category of *context-sensitive* grammars, in which replacement rules can be written that only apply in certain contexts.

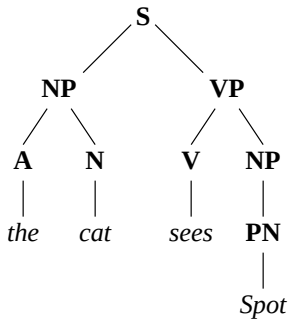
This context-free language consists of many rules and symbols. The symbols — **S**, **NP**, **VP**, **A**, **N**, **PN**, and **V** — stand, respectively for *sentence*, *noun phrase*, *verb phrase*, *article*, *noun*, *proper noun*, and *verb*.

We can derive the sentence, "the cat sees Spot" using this context-free grammar.

$$
\begin{aligned}
\mathbf{S} &\Rightarrow \mathbf{NP\ VP} \\
&\Rightarrow \mathbf{A\ N\ VP} \\
&\Rightarrow \mathbf{A\ N\ V\ NP} \\
&\Rightarrow \textit{the}\ \mathbf{N\ V\ NP} \\
&\Rightarrow \textit{the}\ \mathbf{N}\ \textit{sees}\ \mathbf{NP} \\
&\Rightarrow \textit{the cat sees}\ \mathbf{NP} \\
&\Rightarrow \textit{the cat sees}\ \mathbf{PN} \\
&\Rightarrow \textit{the cat sees Spot}
\end{aligned}
$$

(When you perform a derivation, it's not important which symbol you choose to replace in each step. Any order of replacements is fine.)

In many cases, it's more convenient to represent the steps leading to a sentence described by the grammar using a diagram called a **parse tree**.



A parse tree has the starting symbol **S** at its root. Every node in the tree is either a symbol or an atom. Each symbol node has children representing a sequence of items that can be derived from that symbol. Each atom node has no children. To read the sentence described by the tree, we read through the atoms left to right along the tree's bottom fringe.

Other sentences included in this grammar include the following.

a cat knows
Spot sees the student
the moon knows Carl

Proving that each of these are described by the grammar is relatively easy: You just have to write a derivation or draw a parse tree. It's more difficult to argue that the following are not described by the grammar.

cat sees moon
Carl the student knows

8.2.2 Context-free languages

A **context-free language** is one that can be described by a context-free grammar. For example, we would say that the language of decimal representations of multiples of 5 is context-free, as it can be represented by

the following grammar.

$$\begin{array}{lcl} \mathbf{S} & \rightarrow & \mathbf{N}\,0 \mid \mathbf{N}\,5 \\ \mathbf{N} & \rightarrow & \mathbf{D}\,\mathbf{N} \mid \varepsilon \\ \mathbf{D} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

(We use ε to represent the empty sequence — this is just to make it clear that the space is intentionally blank.)

Often the fact that a language is context-free isn't immediately obvious. For example, consider the language of strings of *a*'s and *b*'s containing the same number of each letter. Is this language context-free? The way to prove it is to demonstrate a context-free grammar describing the language. Thinking of a grammar for this language isn't so easy. Here, though, is a solution.

$$\mathbf{S} \rightarrow a\,\mathbf{S}\,b\,\mathbf{S} \mid b\,\mathbf{S}\,a\,\mathbf{S} \mid \varepsilon$$

To get a feel for how this grammar works, we look at we can derive the string *aabbba* using this grammar.

$$\begin{array}{lcl} \mathbf{S} & \Rightarrow & a\,\mathbf{S}\,b\,\mathbf{S} \\ & \Rightarrow & a\,a\,\mathbf{S}\,b\,b\,\mathbf{S} \\ & \Rightarrow & a\,a\,b\,b\,\mathbf{S} \\ & \Rightarrow & a\,a\,b\,b\,b\,\mathbf{S}\,\mathbf{a} \\ & \Rightarrow & a\,a\,b\,b\,b\,a \end{array}$$

A single example isn't enough to convince ourselves that this grammar works, however: We need an argument that our grammar describes the language. In this case, the argument proceeds by noting that if the string begins with an *a* then there must be some *b* so that there are the same number of *a*'s and *b*'s between the two and there are the same number of *a*'s and *b*'s following the *b*. A similar argument applies if the string begins with a *b*. We'll skip over the details of this argument.

Now consider the language of strings of *a*'s, *b*'s, and *c*'s, with the same number of each. Is this context-free? To test this, we can look for a context-free grammar describing the language. In this case, however, we won't be successful. There is a mathematical proof that no context-free grammar exists for this language, but we won't look at it here.

8.2.3 Practical languages

We can now apply our understanding of context-free languages to the complex languages that people use.

Natural languages Chomsky and other linguists are interested in human languages, so the question they want to answer is: Are human languages context-free? Chomsky and most of his fellow American linguists naturally turned to studying English. They have written thousands of rules in an attempt to describe English with a context-free grammar, but no grammar has completely described English yet. Frustrated with this difficulty, they have also tried to look for a proof that it is impossible, with no success there, either.

Other languages, however, have yielded more success. For example, researcehrs have discovered that some dialects of Swiss-German are not context-free. In these dialects, speakers say sentence like

Claudia watched Helmut let Eva help Hans make Ulrike work.

with the following word order instead. (Of course, they use Swiss-German words instead!)

Claudia Helmut Eva Hans Ulrike watched let help make work.

The verbs in this sequence are in the same order as the nouns to which they apply. Swiss-German includes verb inflections, and each verb inflection must match with its corresponding noun, just as English requires that a verb must be a singular verb if its subject is singular.

To prove that this system isn't context-free, researchers rely on its similarity to an artificial language of strings with *a*'s and *b*'s of the form $X\,Y$, where X and Y are identical. There is a mathematical proof that this artificial language is not context-free, and the proof extends to Swiss-German also.

There are very few languages that researchers know are not context-free, but their existence demonstrates that the human brain can invent and handle such complex languages. This fact, coupled with the tremendous difficulty of accommodating all of the rules of English into a single context-free grammar, leads many researchers to believe that English is not context-free either.

Programming languages On the other hand, programming language designers intentionally design their languages so that programmers can write compilers to interpret the language. One consequence is that programming languages tend to be context-free (or very close to it). Indeed, compilers are usually developed based on the context-free grammar for their language.

As an example, the following is a grammar for a small subset of Java.

S	$\rightarrow$	**Type** *main (* **Type** *Ident)* *{* **Stmts** *}*
Stmts	$\rightarrow$	**Stmt Stmts** $\mid \varepsilon$
Stmt	$\rightarrow$	**Type Ident** = **Expr** *;* $\mid$ **Expr** *;* $\mid$ *while (* **Expr** *)* **Stmt** $\mid$ *{* **Stmts** *}*
Expr	$\rightarrow$	**Expr** $+$ **Expr** $\mid$ **Expr** $-$ **Expr** $\mid$ **Ident** $>$ **Expr** $\mid$ **Ident** $=$ **Expr** $\mid$ **Ident** $\mid$ **Num**
Type	$\rightarrow$	*String* $\mid$ *int* $\mid$ *void* $\mid$ **Type** *[]*
Ident	$\rightarrow$	*x* $\mid$ *y* $\mid$ *z* $\mid$ *main*
Num	$\rightarrow$	*0* $\mid$ *1* $\mid$ *2* $\mid$ *3*

Consider the following Java fragment.

```
void main(String[] args) {
    int y = 1;
    int x = 4;
    while(x > 0) {
        x = x - 1;
        y = y + y;
    }
}
```

This fragment is a sentence generated by our grammar, as illustrated by the parse tree of Figure 8.1. Of course, this grammar covers a small piece of Java. The grammar for all of Java has hundreds of rules. But such a grammar has been written, and the compiler uses this grammar to break each program into its component parts.

Syntax and semantics Throughout this discussion, the distinction between **syntax** and **semantics** is important. *Syntax* refers to the structure of a language, while *semantics* refers to the language's meaning. Context-free grammars only describe a language's syntax. Semantics is another story entirely.

The line separating syntax and semantics is somewhat fuzzy with natural languages. For example, English has a rule that the gender of a pronoun should match the gender of its antecedent. It would be improper for me to say, "Alice watches himself," assuming Alice is female. Most linguists would argue that this is a matter of semantics, as it requires knowledge of the meaning of *Alice*. They would argue, however,

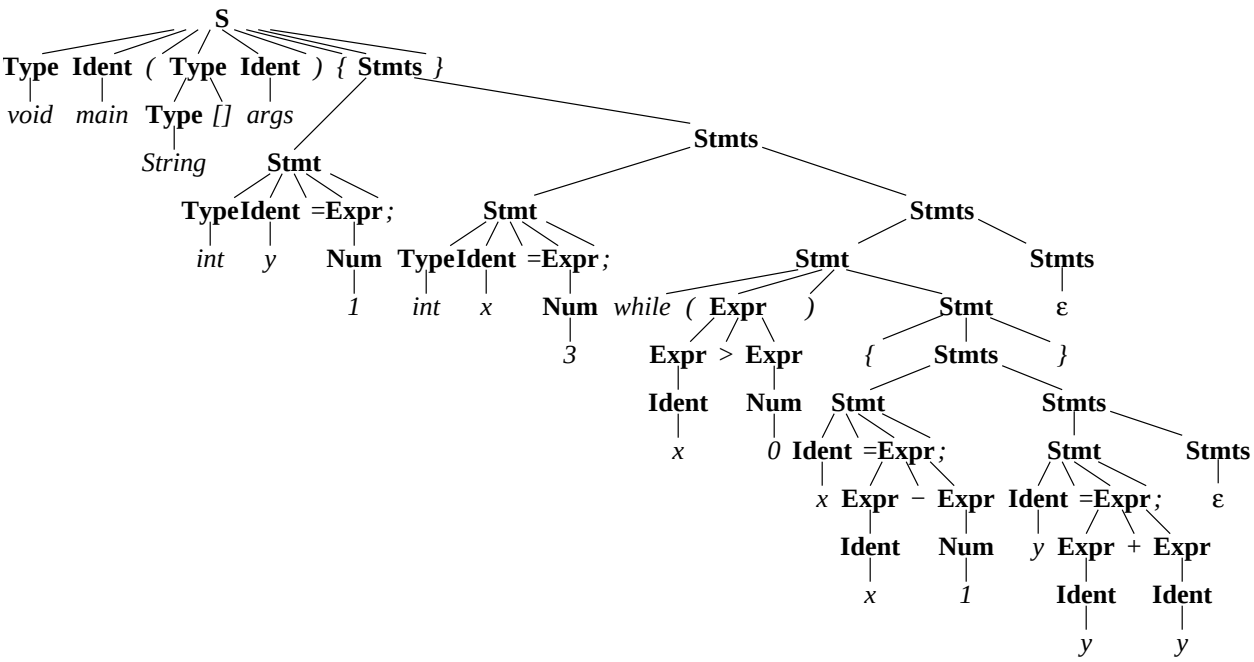


Figure 8.1: Parse tree representing a Java program.

that the issue of subject-verb agreement is a syntactic issue. (An example where the subject and verb do *not* agree is "The hobbits watches me": Since *hobbits* is plural, the verb should be *watch*.)

For programming languages, people generally categorize issues surrounding the generation of a parse tree as syntactic, while others are semantic. The rule that each variable declaration must end in a semicolon, however, is a syntactic rule.

The process of taking a program and discerning its structure is called **parsing**. Thus, you will sometimes see a compiler complain about a "parse error." For compilers built around a context-free grammar, this indicates that your program doesn't fit into the language described by its grammar.

8.3 Regular languages

Chomsky identified another class of languages that has also proven useful in the context of computer science: the class of *regular languages*.

8.3.1 Regular expressions

A **regular expression** is a succinct representation of a language where we use an expression built up of atoms and operators. The simplest regular expression contains a single atom, which represents a language consisting of a string holding that atom only. For example, the regular expression **a** represents the language $\{a\}$. (This book uses a boldface typewriter font to distinguish regular expressions from individual sentences in the languages they represent.)

Larger regular expressions can be built using one of three possible *operators*. Arithmetic has operators like $+$ and $\div$. These operators take two numbers and generate a different number. Similarly, the regular expression operators take two languages (described using smaller regular expressions) and generate a new language.

The union operator The vertical bar ("|") is the simplest operator, which we pronounce *or*. It means to unite two languages together. For example, the regular expression **a|b** is the combination of the two regular expressions **a** $= \{a\}$ and **b** $= \{b\}$, which gives the language $\{a, b\}$.

The catenation operator When we write one regular expression after another, as in AB, it represents the language of all strings composed of a string from A followed by a string from B. To illustrate, we look at some examples using the union operator and the catenation operator.

ab represents any string chosen from **a** $= \{a\}$ followed by any string from **b** $= \{b\}$. There is only one choice from each language, so the only possible result is *ab*. Thus the expression **ab** represents the language $\{ab\}$.

b|ca represents either a string chosen from **b** $= \{b\}$ or a string chosen from **ca** $= \{ca\}$. This union gives us the language $\{b, ca\}$.

This illustrates that the catenation operator has a higher precedence than the union operator, just as multiplication precedes addition in arithmetic. You can use parentheses when you don't like this precedence order.

(b|c)a represents the catenation of any string from **b|c** $= \{b, c\}$ with any string from **a** $= \{a\}$. There are two possibilities for the first choice, and one for the second choice, giving a total of two possibilities, $\{ba, ca\}$.

(b|c)a(b|d) is the catenation of any string chosen from **(b|c)a** $= \{ba, ca\}$ with any string chosen from **b|d** $= \{b, d\}$. This gives the language $\{bab, cab, bad, cad\}$.

The repetition operator Finally, we can create a regular expression by following a smaller regular expression with an asterisk ('*', also called a *star*). This represents any repetition of strings chosen from the preceding regular expression's language. Or, in other words, the expression A***** represents the set of all strings that can be separated into strings from A. Here are some examples.

a* allows us to repeat any number of choices from the language $\{a\}$. Thus, we get the following language.[†]

$$\{\varepsilon, a, aa, aa, aaa, \dots\}$$

ab* represents the catenation of any string chosen from **a** $= \{a\}$ and any string chosen from **b*** $= \{\varepsilon, b, bb, bbb, \dots\}$. This gives us the language

$$\{a, ab, abb, abbb, \dots\} .$$

This example illustrates that the repetition operator precedes the catenation operator (which we've already seen precedes the union operator).[‡] Again, we can use parentheses can designate a different ordering.

[†]Here, as with context-free languages, ε represents an empty sequence.

[‡]If this order of operations confuses you, think of the order of operations in arithmetic if we understand the union operator as being equivalent to addition, the catenation operator as being equivalent to multiplication, and the repetition operator as being equivalent to squaring something. To understand the regular expression **a|bc***, then, we would translate it to the arithmetic expression $a + b \cdot c^2$, which would be evaluated in the order $a + (b \cdot (c^2))$, and so the original regular expression's order of precedence is **a|(b(c*))**.

`a*b*` represents the catenation of any string chosen from **`a*`** $= \{\varepsilon, a, aa, aaa, \ldots\}$. with any string chosen from **`b*`** $= \{\varepsilon, b, bb, bbb, \ldots\}$. This union is

$$\{\varepsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, \ldots\} .$$

In this language, the string *abb* comes from choosing *a* from the first set and *bb* from the second set; the string *aa* in this language comes from catenating *aa* from the first set and the empty string from the second set. The regular expression **`a*b*`** describes the language of all strings consisting of any number of *a*'s followed by any number of *b*'s.

`(a|ab)*` Here, we can repeatedly choose from the language **`a|ab`** $= \{a, ab\}$, giving the language

$$\{\varepsilon, a, ab, aa, aab, aba, abab, aaa, aaab, aaba, \ldots\} .$$

Notice that we don't have to keep choosing the same string: We arrive at *aaba* by choosing *a* first, then *ab*, then *a*. The regular expression describes the language of all strings of *a*'s and *b*'s where each *b* has an *a* directly before it.

8.3.2 Regular languages

A regular language would be any language that can be described using a regular expression. We've already seen several examples of regular languages starting from regular expressions.

The language of decimal representations of multiples of 5 is another language that is obviously regular, since we can express the fact that we need a string of digits ending in a *0* or a *5* with the regular expression

`(0|1|2|3|4|5|6|7|8|9)*(0|5)` .

Sometimes it takes a bit of thinking to determine whether a language is regular. For example, you shouldn't expect the language of binary representations of multiples of three to be regular: A regular expression for it is simply not easy to discover. But, as it happens, the following regular expression does the job, and so the language *is* regular.

`(0|1(01*0)*1)*`

I wouldn't expect you to make sense of this expression, though.

8.3.3 Relationship to context-free languages

Now that we have the ability to classify a language as regular and/or context-free, it's natural for us to ask: How do these language classes relate? We've already seen that the language of multiples of 5 is both regular and context-free, so apparently a language can be classified as both. But is it possible for a language to be context-free but not regular? Can a language be regular but not context-free?

The answer to the first question is yes, some context-free languages are not regular. An example of such a language is the set of strings of *a*'s and *b*'s with the same number of each. We've already seen that this is a context-free language. It's more difficult to prove that it's not regular, and we won't explore the proof now.

To the second question ("Can a language be regular but not context-free?"), the answer is no. This, we will prove.

Theorem 3 *Every regular language is context-free.*

Proof: Every regular language can be described by a regular expression. We'll see how we can build up a context-free grammar that corresponds to any regular expression.

First, take the simplest possible regular expression x, where x is some single atom. Building a context-free grammar for this expression is simple: It consists of the single rule $\mathbf{S}_x \to x$.

Now suppose we have a larger regular expression. There will be some final operator applied in this expression, and this will be a catenation operator, repetition operator, or union operator. We'll see how to construct a grammar for each of these three cases.

The union operator Our regular expression is of the form $A\texttt{|}B$. We can build a context-free grammar for A, beginning from some symbol $\mathbf{S}_A$, and for B, beginning from some symbol $\mathbf{S}_B$. To get our context-free grammar for $A\texttt{|}B$, we combine these two smaller grammars and add a new rule

$$\mathbf{S}_{A\texttt{|}B} \to \mathbf{S}_A \mid \mathbf{S}_B \ .$$

The catenation operator Our regular expression is of the form $A\,B$. We combine the grammar for A, beginning with the symbol $\mathbf{S}_A$, and for B, beginning with the symbol $\mathbf{S}_B$, and we add a new rule

$$\mathbf{S}_{A\,B} \to \mathbf{S}_A\,\mathbf{S}_B \ .$$

The repetition operator If our regular expression is of the form $A\texttt{*}$, we take the grammar for A, beginning with the symbol $\mathbf{S}_A$, and we add the rule

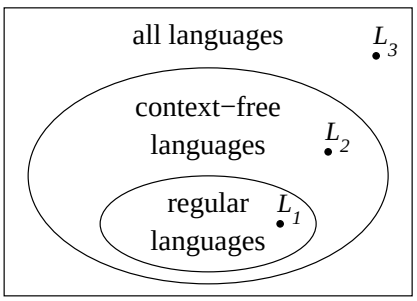
$$\mathbf{S}_{A\texttt{*}} \to \varepsilon \mid \mathbf{S}_A\,\mathbf{S}_{A\texttt{*}}$$

As an example of how this might be applied to a regular expression, suppose we were to take the regular expression `(a|bc)*`. The construction of this proof would build up the following context-free grammar.

$$\begin{aligned}
\mathbf{S}_{\texttt{(a|bc)*}} &\to \varepsilon \mid \mathbf{S}_{\texttt{a|bc}}\mathbf{S}_{\texttt{(a|bc)*}} \\
\mathbf{S}_{\texttt{a|bc}} &\to \mathbf{S}_{\texttt{a}} \mid \mathbf{S}_{\texttt{bc}} \\
\mathbf{S}_{\texttt{bc}} &\to \mathbf{S}_{\texttt{b}}\mathbf{S}_{\texttt{c}} \\
\mathbf{S}_{\texttt{a}} &\to a \\
\mathbf{S}_{\texttt{b}} &\to b \\
\mathbf{S}_{\texttt{c}} &\to c
\end{aligned}$$

Since this grammar was built so that it describes the same language as the regular expression, the original regular language must also be context-free.

Based on this theorem, we can construct a Venn diagram illustrating the relationship between our two language classes, along with examples of languages contained in each region.



L_1**:** the language of strings containing an even number of *a*'s.

L_2**:** the language of strings containing the same number of *a*'s and *b*'s.

L_3**:** the language of strings containing the same number of *a*'s, *b*'s, and *c*'s.

Conclusion

Chomsky's investigation of classes of languages turns out to be surprisingly useful to the computer science discipline. It lays the foundation for three very distinct areas of the field.

- It provides a structure for defining programming languages and building compilers for them.
- It is a starting point for constructing computer systems dealing with human language, such as grammar checkers, speech and handwriting recognition programs, and Web search engines.
- It lays the theoretical foundations for studying computational power.

It is this last, most surprising connection between linguistics and computer science that we explore in the next chapter.

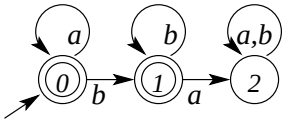
Chapter 9

Computational models

Mathematicians, logicians, and computer scientists are interested in the inherent limits of computation. Is there anything, they ask, that computers *can't* compute? The first step to building such a mathematical understanding of the question is to construct models of computation that are simple enough to allow for formal arguments. In this chapter, we'll look at two such computational models, the finite automaton and the Turing machine.

9.1 Finite automata

The **finite automaton** (also called a *finite state machine*) is a diagram of circles, representing **states**, and arrows, representing **transitions** between states. Here is an example. (The numbers within the circles are not really part of the automaton; they are just for reference.)



Each arrow extending from one state to another represents a transition. There is also one arrow which points into a state from nowhere; this indicates the *initial state*. States represented as double circles are called *accepting states*.

The purpose of the automaton is to process strings to determine whether they are *accepted* or *rejected*. This is not an obvious way of modeling computation — why not, for example, do something explicitly based on arithmetic? But the simplicity and generality of the computation of determining whether a string is accepted makes it appropriate for our mathematical model. (It also allows for drawing parallels to Chomsky's language hierarchy.)

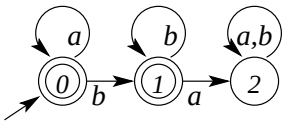
To determine whether the automaton accepts or rejects a string, it goes through the string left-to-right, always occupying a state. We say that it accepts the string if, when it reaches the end, the automaton is in an accepting state (a double circle).

We can diagram an automaton's current situation as follows.

$$\frac{0}{aabb}$$

This represents that the automaton is processing the string *aabb*; the horizontal bar marks the character at which the automaton is currently looking, with the number above representing the current state of the automaton. This example illustrates an automaton in state 0 while looking at the first character of *aabb*.

As an example of an automaton at work, let's step through an example: How does the following automaton work given the string *aabb*?



$\underline{0}$
aabb We start on the first letter, with our current state being state 0, since that's where the arrow with nothing at its beginning points. We look for the transition starting at state 0 labeled with this current letter (*a*). Note that the arrow so labeled loops back to state 0. Thus, when we move to the next letter, we'll remain at state 0.

$\underline{0}$
aabb Now we're on the second letter, which is also an *a*. As we move to the next letter, we take the arrow labeled *a* from our current state 0. This arrow keeps us in state 0.

$\underline{0}$
aabb Now we're looking at a *b* from state 0. As we move to the next letter, we move along the arrow from state 0 labeled *b*, which in this case takes us to state 1.

$\underline{1}$
aabb We now take the arrow labeled *b* from state 1, which keeps us in the same state.

$\underline{1}$
aabb We complete the string in state 1. Since this is an accepting state (as indicated by the double circle), we would say that *aabb* is *accepted* by this automaton.

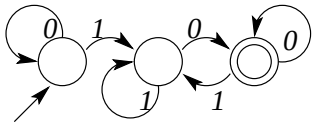
If you look at this particular automaton, you can see that it will be in state 0 as long as it looks at *a*'s at the beginning of the string. Then, when it sees a *b*, it moves to state 1 and remains there as long as it sees *b*'s. It moves to state 2 when it sees an *a*, and it will remain there thereafter. Since state 2 is its only only non-accepting state, then, this automaton will reject only those strings that have a *b* followed by an *a*. Another way of saying this is that this automaton accepts all strings in which all the *a*'s precede all the *b*'s.

Finite automata are extremely simple devices, which makes them quite handy for mathematical purposes. But they're also powerful enough to solve some moderately interesting problems. Let's look at some other examples of automata that solve particular problems.

Positive multiples of 2 Suppose we wanted an automaton for identifying all binary representations of positive multiples of 2, such as

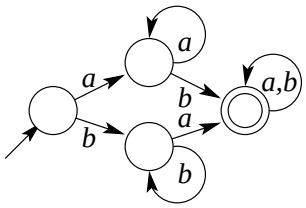
$$10, 100, 110, 100, 1010, 1100, \ldots .$$

Essentially, we want the automaton to accept all strings that have at least one nonzero bit and end in a *0*. The following automaton implements this idea.



In this finite automaton, we will be in the left state until we have found a *1* in the input. Then we will be in the center state whenever the last bit read is a *1* and in the right state when the last bit read is a *0*.

Strings containing both an *a* and a *b* The following automaton identifies strings of *a*'s and *b*'s containing at least one *a* and at least one *b*.



Understanding this automaton is slightly more difficult. To understand it, we can look at why we would be in each state.

- We will be in the left state only at the beginning of the string.
- We will be in the top state when we have seen only *a*'s so far.
- We will be in the bottom state when we have seen only *b*'s so far.
- We will be in the right state when we have seen both an *a* and a *b*.

With this in mind, you can convince yourself that each transition represents what ought to be going on. For example, if you're in the top state (you've seen only *a*'s so far), and then you see a *b*, then you should go to the right state. Based on this reasoning, we would expect an arrow labeled *b* from the top state to the right state, and indeed we see that in the automaton.

9.1.1 Relationship to languages

Each finite automaton accepts some strings and rejects others. This set of strings that it accepts is identical to the concept of *language* we saw in the previous chapter. This provides new opportunities for analysis: In particular, how does the class of languages accepted by finite automata compare to the class of regular languages and the class of context-free languages?

As it turns out, finite automata can accept exactly those languages that are regular. This can be proven as a mathematical theorem, although its proof is complex enough that we won't address it now.

Theorem 4 *The class of regular languages is identical to the class of languages accepted by finite automata. That is, each regular language is accepted by some finite automaton, and each finite automaton accepts a language that is regular.*

This equivalence is somewhat surprising: From a first examination, there isn't any reason to suspect that regular expressions and finite automata can both describe exactly the same languages.

9.1.2 Limitations

The equivalent power of regular expressions and finite automata means that we can't build an automaton to accept any language that can't be described by a regular expression. Earlier, I mentioned that it's impossible to write a regular expression describing the language of strings with the same number of *a*'s as *b*'s. We didn't prove it, though, because proving this with regular expressions is rather difficult. Proving it with finite automata, however, isn't so bad, and in fact the proof is rather interesting, so we'll examine it.

Theorem 5 *No finite automaton accepts the language of strings containing only* a*'s and* b*'s where the number of* a*'s equals the number of* b*'s.*

Proof: The proof proceeds by contradiction. Suppose that somebody gives us an automaton that they purport accepts the desired language; we'll demonstrate a string on which their automaton gives the wrong answer. Let n represent the number of states in this automaton we are given. Now see what state the automaton reaches on a string containing 1 *a*, 2 *a*'s, 3 *a*'s,..., n *a*'s, $n + 1$ *a*'s. We're trying $n + 1$ different strings, and each one ends in one of the n states of the automaton. Thus, at least two of these strings must end in the same state. Let i and j be two different numbers so that i *a*'s and j *a*'s both end in the same state.

Now consider two strings: i *a*'s followed by i *b*'s, and j *a*'s followed by i *b*'s. When we feed each string into the automaton, the automaton will end in the same state, since both strings get to the same state after the *a*'s, and from there the automaton will proceed identically as it goes through the remaining i *b*'s. Thus, the automaton will either accept both strings (if this same ending state is an accepting state) or it will reject both strings (if it is not).

But the first of these strings contains the same number of *a*'s as *b*'s, and the other does not. Thus, whether our automaton accepts both strings, or it rejects both strings, it will be wrong for one of the two strings. Thus our automaton does not accept the language of strings with the same number of *a*'s as *b*'s.

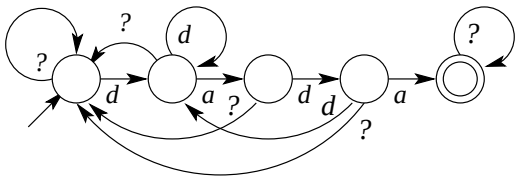
(By the way, this proof is a wonderful instance of the **pigeonhole principle** in mathematics, where we assert that if we have $n + 1$ pigeons to fit into n pigeonholes, some pigeonhole must receive more than one pigeon.)

9.1.3 Applications

It is not too far-fetched that we can regard a computer as a finite automaton: Each setting of bits in its registers and memory would represent a different state. But this would yield a *huge* number of registers. For HYMN, there are 35 bytes of memory and registers, each with 8 bits. A finite automaton representing the CPU, then, would have $2^{35 \cdot 8} = 2^{280}$ states, each representing a certain combination of 0 and 1 values among these bits. That's a massive number — about 2×10^{84}. Thus if we were to build the finite automaton, even with just one atom per state, it would exceed the known universe. (Physicists estimate that the universe contains about 10^{79} atoms.) But the CPU, if built, could easily fit in your fist. The finite automaton's scale is so large that its limitations simply aren't that meaningful.

But this doesn't mean that finite automata don't have their uses. They're useful for specifying very simple circuits. Given a finite automaton, it's not difficult to automatically construct a circuit that implements the automaton.

In software, finite automata are quite useful for searching through text. In fact, most good text editors and word processors have a search function where you can type a regular expression to specify the search. The editor will build the corresponding finite automaton internally and use it to go through the text looking for situations where the automaton gets into an accepting state. Even if the user types a simple string for which to search (not a more complex regular expression), many text editors will build an automaton. For example, if the user asked to search through a document for *dada*, the editor might build the following automaton.



Here the leftmost state represents that the string doesn't look like *dada* is happening any time soon. The next state says that we have matched the first *d* of the word so far. The next state says we have matched the

first two letters. The next state says we have matched the first three. And the last state says we have matched all four — which means that we have completed the string successfully. When the text editor reaches this state, it'll stop and show the user that it has found the string.

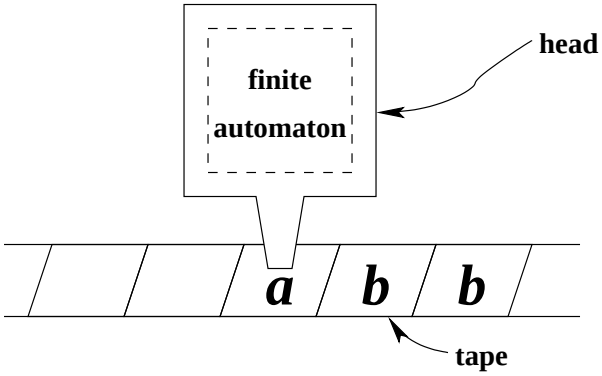
The advantage of this technique of building an automaton is that, as the editor goes through the text, it only has one action to perform for each letter it examines. With a bit of preprocessing for building the automaton, we can step through a large piece of text very efficiently to find a result.

9.2 Turing machines

Alan Turing, a logician working in the 1930's, considered whether one could have a mechanical process for proving mathematical theorems. To address this question, he came up with a model of computing called the *Turing machine* today. Today, the Turing machine is still the most popular model of computation. (We saw Alan Turing before, when we looked at the Turing test, but he was quite a bit older then: He invented the Turing test in 1950, while he invented Turing machines in the mid-1930's.)

9.2.1 Definition

As it computes, a **Turing machine** looks something like the following.



At its heart, a Turing machine is a finite automaton. But the automaton has the capability to use a **tape** — a one-dimensional string of characters on which the machine can read and write. The tape extends infinitely in both directions. At any moment, the Turing machine's **head** is examining a single square on the tape.

Computation in a Turing machine proceeds as follows.

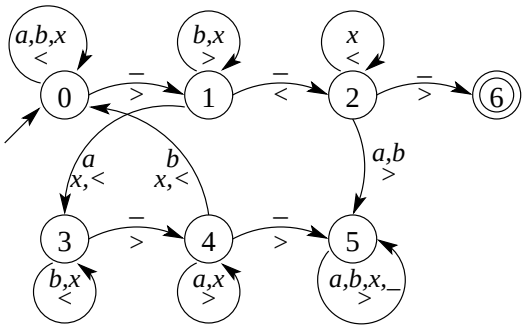
1. The head looks at the character currently under its head, and it determines which transition it will take based on the machine's current state within its finite automaton.
2. The machine may optionally replace the character at the head's current position with a different character.
3. The machine will move its head left or right to an adjacent square.
4. The machine changes its state according to the selected transition.
5. The machine repeats the process for its now-current state and head position.

To see whether a Turing machine accepts a string, we write the string onto the tape, with blanks everywhere else, and we place the head at the first character of the string. We start the Turing machine running. When the Turing machine reahces a state where there is no arrow saying what to do next, we say that the

machine has **halted**. If it halts in an accepting state (a double circle), then the Turing machine has accepted the string initially written on the tape. If, however, it never halts, or if it halts in a non-accepting state, then we say that the Turing machine does not accept the string initially on the tape.

9.2.2 An example

The following picture diagrams the finite automaton that lies within the head of one Turing machine.



In this diagram, the action for each transition is listed below the characters to which the transition applies. The less-than (<) and greater-than (>) symbols represent which way the machine should move on the tape as it goes to the next state. If there is a character preceding this symbol (like the *x* in "*x*,<" on the first down arrow), then this represents the character to write before making a move.

The easiest way to understand the Turing machine is to go through an example of it working. In Figure 9.1, we suppose that we give the string *ba* to the Turing machine illustrated above, and then we start it up. It's worth working through the steps to understand how the Turing machine processes the string. The right-hand column summarizes the current situation for the machine at each step. For example, the fourth row in this column says:

$$\underline{\overset{1}{b}}a$$

This represents that the tape still contains *ba*. The line above the *a* represents that the head is currently pointing at the *a*, and the 1 above the line represents that the Turing machine is currently in the automaton's state 1.

Suppose that the machine starts with the string *abb* instead. Then it goes through the following situations.

$$\underline{\overset{0}{a}}bb \quad \underline{\overset{0}{a}}bb \quad \underline{\overset{1}{a}}bb \quad \underline{\overset{3}{x}}bb \quad \underline{\overset{4}{x}}bb \quad \underline{\overset{4}{x}}bb \quad \underline{\overset{0}{x}}xb \quad \underline{\overset{0}{x}}xb \quad \underline{\overset{1}{x}}xb \quad \underline{\overset{1}{x}}xb \quad \underline{\overset{1}{x}}xb \quad \underline{\overset{1}{x}}xb \quad \underline{\overset{2}{x}}xb \quad \underline{\overset{5}{x}}xb \quad \cdots$$

Once it reaches this point, where it is in state 5 looking at a blank, it will continue going to the right and finding more blanks. The machine will continue reading blanks, never reaching an accepting state, and thus never accepting the string.

This machine, as it happens, accomplishes the task of recognizing strings containing the same number of *a*'s as *b*'s. If you remember, Theorem 4, proved that finite automata cannot compute this language. But Turing machines are slightly more flexible: Unlike finite automata, they have the ability to change the letters on the tape and to move both left and right. This added ability allow Turing machines to compute a wider variety of languages than finite automata, including this language.

To understand how the machine accomplishes the task, we need to understand the purpose of each state in the automaton.

- State 0's job is to move the machine back to the leftmost non-blank character in the string. Once it finds a blank, it moves to the right into state 1.

The machine starts in the initial state. The letter at which it starts is arbitrary.	$\underline{0}\ ba$
The machine takes the transition from the current state (0) labeled with the letter currently under the head (*b*). In this case, the transition loops back to state 0 and says "<," so the machine is now in state 0 looking at the letter to the left of the *b*, which is a blank.	$\underline{0}\ ba$
The machine takes the transition labeled with a blank from the current state, which brings it into state 1. Since the transition says ">," the machine moves it head to the right.	$\underline{1}\ ba$
The transition from state 1 labeled *b* goes to state 1 and says ">."	$b\underline{1}\ a$
The transition from state 1 labeled *a* goes to state 3 and says "*x*,<." The machine replaces the *a* with an *x* before moving left.	$\underline{3}\ bx$
The transition from state 3 labeled *b* goes to state 3 and says "<."	$\underline{3}\ bx$
The transition from state 3 labeled with a blank goes to state 4 and says ">."	$\underline{4}\ bx$
The transition from state 4 labeled *b* goes to state 0 and says "*x*,<."	$\underline{0}\ xx$
The transition from state 0 labeled with a blank goes to state 1 and says ">."	$\underline{1}\ xx$
The transition from state 1 labeled *x* goes to state 1 and says ">."	$\underline{1}\ xx$
The transition from state 1 labeled *x* goes to state 1 and says ">."	$\underline{1}\ xx$
The transition from state 1 labeled with a blank goes to state 2 and says "<."	$\underline{2}\ xx$
The transition from state 2 labeled *x* goes to state 2 and says "<."	$\underline{2}\ xx$
The transition from state 2 labeled *x* goes to state 2 and says "<."	$\underline{2}\ xx$
The transition from state 2 labeled with a blank goes to state 6 and says ">." Since state 6 is an accepting state, the machine halts, accepting the initial string *ba*.	$\underline{6}\ xx$

Figure 9.1: The example Turing machine processing *ba*.

- State 1 is to go through the string (to the right) searching for an *a*. Once it finds it, it marks the *a* out with an *x* and enters state 3. If it can't find an *a*, it enters state 2.

- We reach state 2 if we were looking for an *a* in state 1 and couldn't find any. Since there are no *a*'s in the string, we hope that there are no *b*'s left. State 2 goes back through the string to verify that the string contains only x's. If it finds none, and so reaches a blank, it enters state 6, the accepting state. It it finds a *b*, we enter state 5, which simply loops infinitely to reject the string.

- When we reach state 3, we have just marked out an *a*. State 3 resets the machine back to the leftmost character by repeatedly moving left until we hit a blank, whereupon we move right into state 4.

- In state 4, we go through the string looking for a *b* to match up with the *a* we marked out when we moved from state 1 to 2. We go through the string, passing any *a*'s or *x*'s we find. When we reach a *b*, we mark it out and go back to state 0 to look for another *a*-*b* pair. If we get all the way through the string without reaching a *b*, we want to reject the string because it has no *b* corresponding to the *a* we marked out, and hence we enter state 5.

9.2.3 Another example

Now let's suppose that we want a Turing machine that determines whether a string begins with some number of *a*'s, followed by the same number of *b*'s. For example, the language we want to handle icnludes the strings *ab*, *aabb*, and *aaabbb*, but not *abab* or *abba*.

The first thing we need, as we consider how to build a Turing machine, is a strategy for how we might possibly do this with a Turing machine. In this case, we know the string should start with an *a*. It would make sense to delete it and immediately go to the other end to delete the matching *b*. Then we can come back to the beginning again and repeat the process. This would slowly whittle down the string to nothing. If we get to nothing with no problems, then the string must fit the desired description.

For example, if we begin with the string *aaabbb*, the process would work as follows.

aaabbb	We begin here.
aabb	We remove the first *a* and the last *b*
ab	We remove the first *a* and the last *b*
	We remove the first *a* and the last *b*

Since we end up with nothing, the original string must be good.

To convert this into a finite automaton, we decide on our states. Each state will be responsible for handling a particular task.

1. Begin at the string's beginning and remove the first *a*.

2. Move to the end of the string.

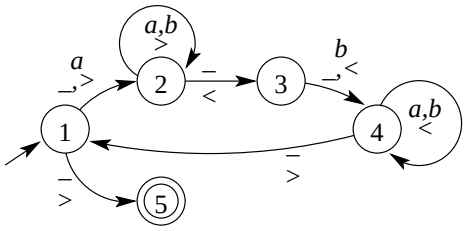
3. Delete the *b* at the end of the string.

4. Move back to the beginning of the string, returning to state 1.

5. If the string has been whittled away, accept the string.

Designing these states becomes more straightforward with practice.

With this description in hand, we can build our Turing machine.



Now we need to test it.

First we'll try a string the machine ought to accept: *aabb*.

The machine starts here.	$\underline{\overset{1}{a}}abb$
The machine erases the first *a* and goes to state 2.	$\underline{\overset{2}{a}}bb$
⋮	
The machine keeps going right as long as it's looking at *a*'s and *b*'s.	$abb\underline{\overset{2}{\ }}$
The machine goes to the left and moves into state 3. It has now reached the end of the string.	$ab\underline{\overset{3}{b}}$
The machine deletes the final *b* and moves left and enters state 4.	$a\underline{\overset{4}{b}}$
⋮	
The machine keeps going left as long as it's looking at *a*'s and *b*'s.	$\underline{\overset{4}{a}}b$
The machine sees a blank from state 4, so it moves to the right and enters state 1.	$\underline{\overset{1}{a}}b$
The machine replaces the *a* with a blank and goes right.	$\underline{\overset{2}{b}}$
The machine goes to the right.	$\underline{\overset{2}{b}}$
The machine goes to the left.	$\underline{\overset{3}{b}}$
The machine replaces the *b* with a blank and goes left.	$\underline{\overset{4}{\ }}$
The machine goes right and enters state 1.	$\underline{\overset{1}{\ }}$
The machine goes right and enters state 5.	$\underline{\overset{5}{\ }}$

Since at this point, the machine is in state 5 looking at a blank, and there's nothing to do in state 5 for a blank, the machine stops. Because state 5 is an accepting state, we say that the machine accepted the input *aabb*.

The process should reject a string that begin with more *a*'s than there are *b*'s at the end. Let's see what happens for *aaabb*.

aaabb	We begin here.
aab	We remove the first *a* and the last *b*
a	We remove the first *a* and the last *b*

Now, we can remove the first *a*, putting us in state 2 of the machine, and state 2 will move to the first space following the string, and then go back one and enter state 3. When it reaches state 3, though, there's no place

to go: The head is looking at an empty square (in fact, the whole tape is empty), and there's no transition from state 3 saying what to do. So the machine stops. Since it stops in state 3, and state 3 isn't an accepting state, the machine has rejected the string.

If there are fewer *a*'s at the beginning than *b*'s at the end, the process should reject it, too.

aabbb	We begin here.
abb	We remove the first *a* and the last *b*
b	We remove the first *a* and the last *b*

When we go to remove the first *a* now, we'll be in state 1, and the head will be looking at a *b*. There's no arrow from this state saying what to do in state 1 for a *b*, and so the machine stops. State 1 isn't an accepting state, so the machine rejects the string.

Since the machine has passed all our tests, it seems that it works correctly.

9.2.4 Church-Turing thesis

Turing proposed the following thesis, which is called the **Church-Turing thesis**. (Alonzo Church gets credit for this too, because he independently came up with the same idea. His version uses a different, less accessible model that turns out to be equivalent to Turing machines.)

> Every effectively computable language can be accepted by some Turing machine.

Turing is saying here that his computational model is as powerful as any other mechanical computational model.

This isn't the sort of thing that can be proven mathematically, because "effectively computable" is not specific enough to be proven. But, over the years, some strong evidence has piled up that it is true: People have thought of many other models of computation, and invariably they have found that Turing machines are just as powerful.

To prove that one computational model is as powerful as a Turing machine, we use a proof technique called a **reduction**, where we take an arbitrary machine from our model and demonstrate how to construct a Turing machine that accepts the same language.

For example, suppose we wanted to show that Turing machines can compute everything that HYMN can compute. To do this, we first need to establish that a system for HYMN programs to describe language. Let's imagine that we translate each atom of a language into a separate number, and then we can give a sentence to a HYMN program by typing in the numbers representing the atoms of the sentence, followed by -1. For example, if we're dealing with a language of *a*'s and *b*'s, then we can assign 0 to *a* and 1 to *b*, and then to query a program whether, say, *aabb* is in the language, we can type 0, then 0, then 1, then 1, then -1. We'l suppose that the HYMN program is supposed to respond either with 0 (representing *no*) or 1 (representing *yes*).

Theorem 6 *Turing machines are as powerful as HYMN programs.*

Proof: Suppose you give me a program P for HYMN. I'll show you how I can construct a Turing machine that accepts the same language as your program. Based on this construction, we can conclude that the Turing machines are as powerful as HYMN programs.

Basically, we'll just build a Turing machine that can simulate the HYMN system. When it starts, our Turing machine will have the string in question on the tape. The first thing our machine will do will be to write the following just after the string's end.

$$;temps;regs;mem$$

These additional characters represent "storage space" for representing a HYMN computer as it executes the program. The $temps$ portion is a sequence of three 8-bit temporary spaces (initially all 0); $regs$ includes the three 8-bit register values; and mem holds the machine language representation of P (which, at 32 bytes long, and with a comma before each byte of memory, would take 32×9 places on the tape). We'll call the three temporary spaces T_0, T_1, and T_2.

Building a Turing machine to move around on the tape is complicated. We'll just look at the fetch cycle to see what happens then. The machine will move through the tape to the portion of $regs$ representing PC and copy this into T_0, placing a "caret" before the first location in the memory. Placing a "caret" involves replacing the comma before the byte with a different character (such as "^") temporarily. Then the machine will successively decrement the number in T_0 and move the caret forward 9 bits, until T_0 holds 0. Then it copies the 8 bits following the caret (which is the current instruction) into the portion of $regs$ representing IR. The net result of all this computation is that the current instruction to execute has been copied into the IR region of the tape.

Having our Turing machine simulate the execute cycle is much more complicated because of the variety of instructions. But they, too, can be done in principle. We'll consider just one, special instruction: When it gets to executing a "`STORE 31`" instruction (for displaying the AC), the Turing machine should go to the AC region of the tape and determine whether that region contains a 0 or a 1. The HYMN program was supposed to print a 0 if it rejected the string and a 1 if it accepted the string. Thus, the Turing machine will likewise enter an accepting state if the AC region of the tape contains a 1, and if that region contains a 0, the machine will enter a non-accepting state with no arrows going out (and so the machine would reject the initial string).

The final Turing machine will have thousands of states, but the the size isn't important: As long as we can build a fixed-size Turing machine corresponding to any given HYMN program, we can conclude that Turing machines can do anything that HYMN programs can do.

It's very important to realize here that we're saying nothing about efficiency. HYMN would run much faster, largely because it can skip around within memory. Here, we're only concerned with what the machines can perform, not how fast they can perform them. And Turing machines can perform anything that the HYMN can.

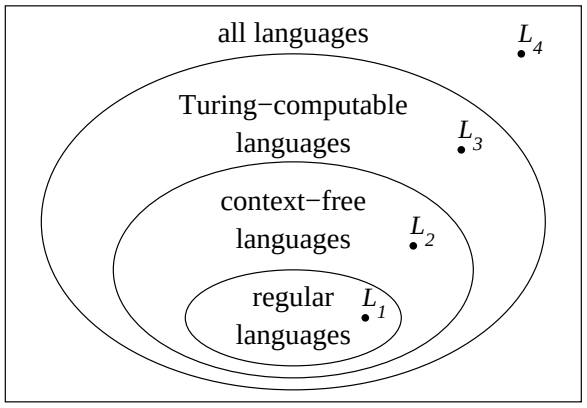
What about the human brain? How does it relate to the Church-Turing thesis? A person who accepts the Church-Turing thesis (as many people do) and who believes that AI is possible must agree that the human brain can accomplish no more than a Turing machine can — that the human brain is essentially a very fancy Turing machine.

9.3 Extent of computability

As it happens — though we will not examine a proof here — every context-free language is "Turing-computable." That is, for any context-free grammar, there is some Turing machine that accepts the same language as that described by the grammar.

On the other hand, there are some Turing-computable languages that are not context-free. An example of this is the language of strings of the form XY, where X and Y are identical. This language includes such strings as *aabaab*, *aa*, and *bbbabbba*. This language is not context-free, but it isn't difficult to build a Turing machine to accept the language. Thus, though the set of context-free languages is a subset of Turing-computable languages, the two sets are not identical.

With this in mind, we can extend the Venn diagram from Section 8.3.3.



L_1: language of strings containing an even number of *a*'s.

L_2: language of strings containing the same number of *a*'s and *b*'s.

L_3: language of strings of the form XY where X and Y are identical.

L_4: ???

We haven't seen an example of a language like L_4 in the diagram, and this omission leaves us wondering whether such a language exists. That is, is Turing's model all-powerful in describing languages? Or are there some languages that are not "Turing-computable"?

We're going to get to that answer in a moment. But first, we'll take a side tour exploring what computer programs cannot do.

9.3.1 Halting problem

Suppose we define the following language.

> The **Java halting language** is the set of strings of the form *prog!input*, where *prog* represents a Java program, and if I run *prog* and type *input*, then *prog* eventually stops (i.e., it doesn't enter an infinite loop).

Now, we can reasonably ask, is there a Java program that can identify strings in this language? That is, our program would read in a string (of the form *prog!input*) from the user and then display either "`yes`" or "`no`" depending on whether that string is in the Java halting language.

As it happens, we can prove that in fact writing a Java program that identifies the Java halting language is impossible. The argument is interesting, and so we'll look at it closely. (There's nothing special about Java in this proof — we could choose any good programming language, and the argument would still apply.)

Theorem 7 *No Java program exists to identify strings in the Java halting language.*

Proof: The proof proceeds by contradiction: Suppose there were such a Java program, which we'll call `Halter`. We'll see how such a program leads to a contradiction.

The program given us would look something like the following.

```
public class Halter {
    public static boolean isInLanguage(String x) {
        ...
    }
}
```

That is, `Halter` contains some code that takes a string input and determines whether or not that string is in the Java halting language.

Suppose we take this program and use it to compose a different program, called `Breaker`.

```
public class Breaker {
    public static boolean isInLanguage(String x) {
        ... // this is taken verbatim from Halter
    }
```

```
    public static void main(String[] args) {
        String prog = readLine(); // read program from user
        if(isInLanguage(prog + "!" + prog)) {
            // if prog!prog is in language, go into infinite loop
            while(true) { }
        } else {
            // if prog!prog isn't in language, exit program
            io.println("done");
        }
    }
}
```

This is a well-defined program, which would compile well. Now consider the following question: What happens if we run `Breaker`, and when it reads a program from the user, we type `Breaker` again?

The call to "`isInLanguage(prog + "!" + prog)`" will return either `true` or `false`. Suppose it returns `true` — that is, it says that *prog!prog* is not in the language. Based on the definition of our language, this response indicates that if we were to run *prog* (that is, `Breaker`) and type *prog*'s code as input, then *prog* would eventually stop. But if you look at the code for `Breaker`, you can see that this isn't what actually happens: We ran `Breaker` and typed `Breaker` as input, and we're supposing that `isInLanguage` returns `true`, and so the program would go into the `if` statement and enter an infinite loop. We can conclude, then, that the `Halter` program would be wrong if its `isInLanguage` method responds with `true`.

So let's suppose that `isInLanguage` returns `false`, a response that means *prog!prog* is in the Java halting language. Based on the definition of this language, this indicates that if we were to run *prog* (that is, `Breaker`) and type *prog*'s code as input, then *prog* would not stop. But when we look at `Breaker`'s code to see what happens when `isInLanguage` returns `false`, we see that what will actually happen is that the program prints, "done," and it promptly stops. Thus, we can conclude that the `Halter` program would be wrong if its `isInLanguage` method responds with `false`.

We've trapped `Halter` into a quandary: We were able to work it into a situation where whatever it says — *yes* or *no* — will be wrong. Thus, we can conclude that any program for the Java halting problem will be wrong in at least some circumstances.

The fact that we can't solve the halting problem has important implications: One consequence is that there is no way that an operating system could be written to reliably terminate a program when it enters an infinite loop. Such a feature would be nice, because the system could guarantee that the system would never freeze. But such a feature would imply a solution to the halting problem, which we've just proven is impossible to solve.

9.3.2 Turing machine impossibility

So we know that there are some things that computers can't do. But this doesn't immediately imply anything about Turing machines. We *have* seen that Turing machines can do anything that computers can do (Theorem 6), but this doesn't apply the other way: Based on what we've seen so far, Turing machines might be able to do some things that regular computers can't. Maybe Turing mahcines could identify the Java halting language.

In fact, in the 1936 paper describing his machines, Turing demonstrates that there are some languages that Turing machines cannot compute. His argument proceeds similarly to the one we just examined for the Java halting language: He gives an example of a particular language and then demonstrates how any Turing machine constructed for that language can be broken.

Before we define the language Turing defined, we first need to observe that it's possible to represent a Turing machine as a string rather than as a diagram of circles and arrows. This string will use the characters used by the machine, plus four additional characters, which we'll name 0, 1, comma, and semicolon. Suppose that the machine has n states; we'll name each state with a different binary number between 0 and $n - 1$. To describe the machine completely, we must represent the initial state, the final states, and all transitions. The format of our string will be as follows.

initial;finals;transitions

This consists of three parts, separated by semicolons.

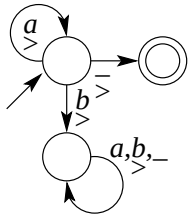
- The *initial* portion is the binary name of the machine's initial state.
- The *final* portion contains the binary names of the accepting states, separated by commas.
- The *transitions* portion contains a list of all transitions, separated by semicolons. Each transition is represented as follows.

  source,read,write,dir,dest

  This has five parts, separated by commas.

  - The name of the state from which the transition comes.
  - A list of the characters to which the transition applies.
  - A list of the characters that should be written to the tape for each possible character read. (If the machine is to leave a character unchanged, *read* and *write* will be identical.)
  - 1 if the machine should go right on the transition, 0 if it says to go left.
  - The name of the state to which the transition points.

For example, suppose we want to describe the following machine as a string.



We'll assign the name 0 to the initial state, 1 to the final state, and 10 to the nonterminating state. Then we can make our string describing this machine.

$$0;1;\overbrace{0,a,a,1,1;0,\_,\_,1,10;0,b,b,1,10;10,ab\_,ab\_,1,10}^{\text{transitions}}$$

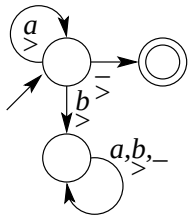
Now we can define the halting problem language.

> The **halting problem language** includes all strings of the form *M!x* where M is a string representation of a Turing machine, and if we write x on a tape and start M, then M does not accept x.

For example, the string

0;1;0,a,a,1,1;0,_,_,1,10;0,b,b,1,10;10,ab_,ab_,1,10!b

is in the language: The part of string before the exclamation point describes the machine

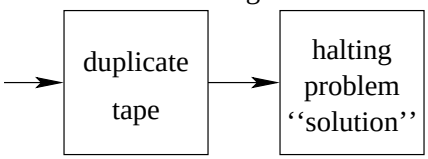


and this machine does not accept *b* as an input (it loops infinitely). However, the following string is *not* in this language, since the same machine accepts *a* as an input.

0;1;0,a,a,1,1;0,_,_,1,10;0,b,b,1,10;10,ab_,ab_,1,10!a

Theorem 8 *No Turing machine exists to solve the halting problem language.*

Proof: The argument proceeds by contradiction. Suppose, he says, somebody gives us such a machine, called M. Then we can construct the following machine and call it A.



(This is only a diagram of the machine. The boxes contain all the arrows and circles needed to define a complete Turing machine.) This machine begins with a "duplicate tape" portion, which is simply a sequence of states that replaces a tape containing the string x with a tape containing the string *x!x*. Once this is done, this machine enters the initial state of the machine proposed as a solution to the halting problem.

This machine A we just built is a normal Turing machine, and so we can represent it as a string. Suppose we put this string representation of A down on the tape, and we start up A to see what happens. Either A will eventually accept this input, or it won't.

- Suppose it accepts this input. This means that M accepted *A!A* as an input. Thus, according to M, A does not accept the string representation of A. But we were supposing that A accepts the string it was given, which was a string representation of A. Hence M must be wrong.
- Suppose it rejects this input. This means that M did not accept *A!A* as an input. Thus, according to M, A must accept the string representation of A. But we were supposing that A does not accept the string it was given, which was a string representation of A. Hence M must be wrong.

Either way, the proposed solution M is wrong, says Turing.

Thus, no matter what Turing machine anybody proposes for the halting problem language, the machine will fail sometimes. It's impossible to build a Turing machine that identifies this language.

Chapter 10

Conclusion

Throughout this course, we have seen a variety of models of computing, both practical and theoretical.

- Logic circuits allow us to design circuits to perform computation, including addition and counting (as we saw in Chapter 4).
- Machine language (Chapter 5) is a simple technique for encoding computation in a sequence of bits.
- Programming languages, such as Java, are systems for describing computation in a format convenient for humans.
- Neural networks (Section 7.3) can represent general computation.
- The Turing machine (Section 9.2) is a simple model of computing that, according to the Church-Turing thesis, is general enough to cover all forms of computing.

To show that one model is as powerful as another, we use a proof technique called **reduction**: If we want to show that A can do everything that B can, we only have to show that any construction within the A system can be translated into the B system. Usually, this involves showing how computations in A can be simulated on some B construction.

Theorem 6 (Section 9.2.4), in which we saw that Turing machines can do anything that HYMN programs can do, illustrated exactly this technique: We showed how we could take a HYMN program and construct a Turing machine to accomplish the same thing. Another example of a reduction was in Section 7.3, where we saw that any logic circuit can be simulated on a neural network.

Some reductions are very practical. A Java compiler, for example, translates a program in Java into machine language. This compiler is a proof that machine language programs can do anything that Java programs can. Similarly, the program *Logisim* (which is written in Java to simulate logic circuits) demonstrates that Java programs can compute anything that logic circuits can.

Figure 10.1 diagrams some of the reductions between various computational models. Each arrow points from one model to another that is just as powerful. You can see several cycles within the picture, and these cycles mean that all the models contained in the cycle are equally powerful. The picture contains several cycles:

$$\text{Java} \to \text{machine language} \to \text{logic circuits} \to \text{Java}$$
$$\text{Java} \to \text{machine language} \to \text{Turing machines} \to \text{Java}$$
$$\text{Java} \to \text{machine language} \to \text{logic circuits} \to \text{neural networks} \to \text{Java}$$

Despite their differences, then, all of these models are equally powerful.

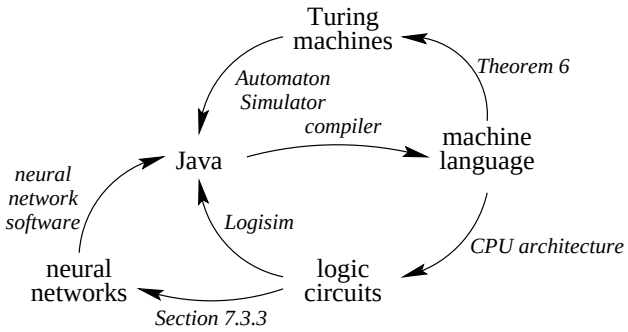


Figure 10.1: Reductions between computational models.

You might wonder: Can we get any more power by having several machines running at the same time? The answer turns out to be *no*. In fact, we saw this when we looked at operating systems in Chapter 6: Modern operating systems end up simulating a computer system with multiple threads of computation, even though the system may in fact have only one CPU. The same technique — where we run each computation for a while, and then perform a context switch into another computation — extends to other models, too.

Through this class, then, we have seen many computational models, developed to serve different purposes. Though they bear little resemblance to each other, they are in fact all computationally equivalent. The Church-Turing thesis asserts that this level of computational power is as much as is possible.

So which model should we use in practice? This is a matter of evaluating the practical properties of the model for the problem we have at hand. If you want to prove something about the extent of computation, the simplicity of the Turing machine makes it ideal. But the Turing machine's simplicity is exactly what makes it horrid for developing large-scale software, for which high-level programming languages are better suited.

The discipline of computer science involves gaining a greater understanding of these models' capabilities, learning how to use them with maximum efficiency, and exploring the effects of computation on humanity. Computer science includes perspectives drawn from many other disciplines, including mathematics, engineering, philosophy, management, sociology, psychology, and many others. The appeal of the subject is how it brings people of all these perspectives into one room, with one purpose, which is at once both interesting and practical: to understand the potential of the concept of *computation*.

Index