Chapter 11

Logic and Computers

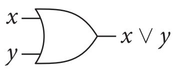

Figure 11.1. "Or" gate.

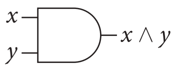

Figure 11.2. "And" gate.

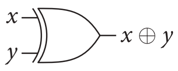

Figure 11.3. "Exclusive or" gate.

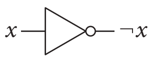

Figure 11.4. "Inverter," or "not" gate. The output is commonly written $\bar{x}$ instead of $\neg x$.

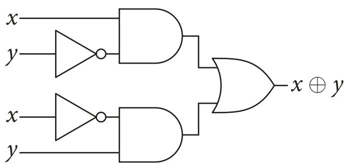

Figure 11.5. Constructing a circuit to compute "exclusive or" using only "and," "or," and "not" gates.

Computers are machines for logical calculations. We may think they are calculating on numbers, but those numbers are represented as bits. Arithmetic operations are logical operations underneath.

In fact, computers are *built* out of logic. *Everything* in a computer is represented as patterns of `0`s and `1`s, which can be regarded as just other names for the two truth values of propositional logic, "true" and "false." Formulas of propositional logic derive their truth values from the truth values of their propositional variables. In the same way, computers manipulate the bits `0` and `1` (page 32) to produce more bits. Computers manufacture bits, using bits as the raw material. For this reason, the design of the physical stuff that produces bits from bits is called a computer's "logic."

The conceptually simplest pieces of computer logic are *gates*. A gate produces a single bit of output from one or two bits of input (or sometimes more). Gates correspond to the operators of propositional logic, such as $\vee$, $\wedge$, $\oplus$, and $\neg$. Pictures like those in Figures 11.1–11.4 should be interpreted as having direction: `1`s and `0`s enter on the "wires" on the left, and a `1` or a `0` comes out on the right.

These gates can be chained together in various ways to produce more complex results using only the simplest of components; such a computing device is called a *circuit*. But in fact, not all the illustrated gates are typically available to a circuit designer. For example, as shown on page 94, we can compute the "exclusive or" using just the "and," "or," and "not" gates, since

$$x \oplus y \equiv (x \wedge \neg y) \vee (\neg x \wedge y).$$

So a logic circuit to compute $x \oplus y$ can be read directly from that formula, as shown in Figure 11.5.

It is customary to show only one source for each input, and to allow branching "wires" when the same input is used in more than one place. When it is necessary to draw wires crossing even though they are not

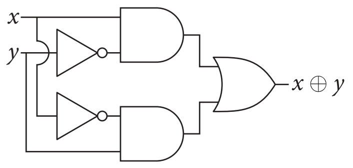

Figure 11.6. The same circuit, shown with one wire crossing over another.

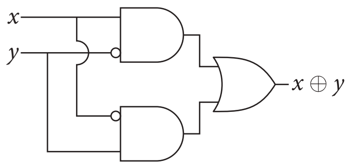

Figure 11.7. This is the same circuit, with negated inputs shown as "bubbles" (small circles) at inputs to the "and" gates.

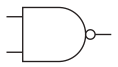

Figure 11.8. "Nand" gate. Following the convention illustrated in Figure 11.7, the logical negation of the output of a gate is shown by adding a "bubble" at the output.

connected, the crossing is shown as a semicircle as in Figure 11.6, as though one wire were hopping over the other.

Also, negated inputs are often shown as a "bubble" at the gate, rather than as a separate negation gate. So the circuit of Figure 11.7 is functionally the same as those of Figures 11.5 and 11.6.

The "and," "or," "not," and "exclusive or" gates illustrated here correspond directly to the various logical operators introduced in Chapter 9. One very useful gate corresponds to the "not-and," also known as *nand* operator:

$$p \mid q \equiv \neg(p \wedge q).$$

This may seem to be a peculiar operator, but it has a lovely property for building logic circuits: any of the conventional operators can be expressed using it! (The "nor" operator of Problem 11.5 has the same property.) For example:

$$\begin{aligned} \neg p &\equiv p \mid p \\ p \wedge q &\equiv \neg(p \mid q) \equiv (p \mid q) \mid (p \mid q). \end{aligned}$$

The | operator is known as the *Sheffer stroke*, and is sometimes written as $\uparrow$. We leave it as an exercise (Problem 11.6) to express $p \vee q$ using just the | operator. The corresponding logic gate is shown in Figure 11.8. If we think in terms of mass production, nand gates are ideal—if you can produce them cheaply, you don't need to produce anything else. Whatever you are trying to accomplish, you "just" have to figure out how to connect them together.

❈

Binary notation is the name for this way that numbers are represented for processing by computer, using just `0`s and `1`s. Binary is the base-2 number system. The decimal or base-10 system we customarily use has the ten digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9; the binary system is similar, but has only the two binary digits `0` and `1`.

The number zero is represented in binary as `0` or a string of `0`s, for example `00000000`. Counting consists of repeatedly adding 1, subject to the basic rules of binary arithmetic:

$$\begin{aligned} \mathtt{0}+\mathtt{0} &= \mathtt{0} \\ \mathtt{0}+\mathtt{1} &= \mathtt{1} \\ \mathtt{1}+\mathtt{0} &= \mathtt{1} \\ \mathtt{1}+\mathtt{1} &= \mathtt{0}, \text{ carry } \mathtt{1}. \end{aligned} \tag{11.1}$$

That is, adding `1` to `1` in binary has the same effect as adding 1 to 9 in decimal; the result is `0`, but a `1` is carried into the next position to the left. For example, adding `1` to `10111` produces the result `11000`, as shown below, with three carries.

$$\begin{array}{rccccc} & & 1 & 1 & 1 & \\ & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{1} \\ + & & & & & \mathtt{1} \\ \hline & \mathtt{1} & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{0} \end{array}$$

In exactly the same way, we can add two arbitrary numbers together by manipulating the bits of their binary representations. The only rule we need in addition to (11.1) is how to add `1` + `1` plus a carry-in of `1`. The answer is that the sum is `1` and there is a carry of `1` into the column to the left. So let's work one more example.

$$\begin{array}{rcccccc} & & 1 & 1 & 1 & 1 & \\ & & \mathtt{1} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{1} \\ + & & \mathtt{0} & \mathtt{1} & \mathtt{1} & \mathtt{1} & \mathtt{1} \\ \hline & \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} \end{array} \tag{11.2}$$

To interpret a number like `100100` without counting up, use the fact that each column represents a power of two, starting with the rightmost column, which represents 2^0, and adding one to the exponent as we move right to left from column to column. To get the numerical value of a bit string, add up the powers of two corresponding to the columns containing the `1` bits. For example, `100100` represents $2^5 + 2^2$:

$$\begin{array}{cccccc} 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \mathtt{1} & \mathtt{0} & \mathtt{0} & \mathtt{1} & \mathtt{0} & \mathtt{0} \\ \hline 2^5 & 0 & 0 & 2^2 & 0 & 0 \end{array}$$

Since $2^5 + 2^2 = 32 + 4 = 36$, that is the value of the binary numeral[1] `100100`. And (11.2) represents the sum of $2^4 + 2^2 + 2^0$ and $2^3 + 2^2 + 2^1 + 2^0$; that is, 21 + 15—which is indeed 36.

[1]A *numeral* is a name for a number. So we refer here to the binary numeral `100100` to emphasize that we are referring to a string of six bits, which represents the number 36.

Notice that this interpretation is the same one we use in the decimal system, except that in binary we use powers of 2 rather than powers of 10. For example, the numeral `100100` interpreted in decimal represents $10^5 + 10^2$.

The bit in the rightmost position—which represents 2^0 in binary—is known as the *low-order* bit. Similarly, the leftmost bit position is the *high-order* bit. Higher-order bits contribute more to the value of a numeral than lower-order bits, in exactly the way that in the decimal numeral 379, the 3 adds 300 to the value but the 9 adds only 9.

Armed with an understanding of logic gates and of binary arithmetic, how would we design hardware to do computer arithmetic? The simplest operation is the addition of two bits, as shown in (11.1). These equations really create *two* output bits from two input bits, since adding two bits produces both a sum and a carry bit. In the first three cases the carry bit wasn't mentioned—because the carry bit is 1 only when both input bits are 1. The sum, on the other hand, is 1 only when one of the inputs is 1 and the other is 0—in other words, just in case the two input bits are different. So the sum output is the logical "exclusive or" of the input bits and the carry bit is the logical "and" of the input bits. So the device in Figure 11.9, called a *half adder*, transforms input bits x and y into their sum s and their carry bit c.

There are other ways to wire gates together to produce the same outputs from the same inputs, and which way works the best in practice may depend on details of the particular implementing technology. So as we move from adding bits to adding longer binary numerals, it makes sense to represent the half adder as a box with inputs x and y and outputs s and c, hiding the internal details (Figure 11.10). Such a construct, with known inputs and outputs but hidden internal details, is often called a *black box*. If we construct more complicated circuits out of these boxes and then later decide to revise the internals of a box, our circuits will still work the same as long as the functions of the boxes remain the same.

Imagine we want to design a circuit that can compute binary sums such as (11.2). We can do this using several instances of a standard subcircuit. The half adder, as its name suggests, is not quite up to the job: it adds two bits, but when adding numbers with multiple digits we sometimes need three inputs: the original two bits plus a carry bit. A *full adder* takes in three inputs—X, Y, and the carry-in bit C_{in}—and puts out two bits, the sum S and the carry-out C_{out}. Figure 11.11 shows the truth table for computing the two outputs from the three inputs.

One way to construct a full adder is out of two half adders, as shown in Figure 11.12. This circuit produces the sum bit S by adding X and Y and then adding C_{in} to the result; the C_{out} bit is 1 if there is a carry from either the first or the second of these additions.

If we represent the full adder schematically as another black box, as shown in Figure 11.13, we can chain two of them together to produce a proper two-bit adder (Figure 11.14), which adds two two-bit binary numerals x_1x_0 and y_1y_0 to produce a two-bit sum z_1z_0, plus a carry-out bit—which is `1` if the sum, such as `11` + `01`, can't be represented in two bits.

In exactly the same way, any number n of full adders could be chained together to produce an adder for two n-bit numbers. This kind of adder is called a *ripple-carry adder*; it implements in hardware the addition algorithm you were taught in grade school, which we used in (11.2).

But having gotten this far, we are confronted by lots of interesting questions. Does Figure 11.12 represent the most efficient implementation of the truth table in Figure 11.11? What if the "exclusive or" gate used in Figure 11.9 is not available, and we have to make do with some other set of primitives, perhaps just "nor" gates—what is then the simplest full adder? In designing real hardware to be fabricated as an integrated circuit, the number of gates is

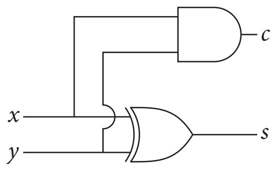

Figure 11.9. A half adder produces sum and carry bits from two input bits.

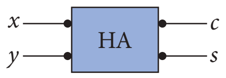

Figure 11.10. A half adder represented as a "black box."

X	Y	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 11.11. Truth table for the design of a full adder.

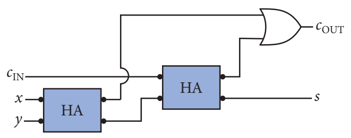

Figure 11.12. A full adder constructed from two half adders.

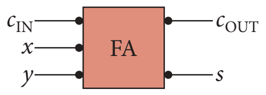

Figure 11.13. Abstract representation of a full adder.

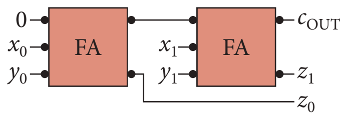

Figure 11.14. Two-bit adder. The addends are x_1x_0 and y_1y_0; x_0 and y_0 are the low-order bits. The carry-in to the first full adder is set to zero, the carry-out of the first adder becomes the carry-in to the second, and the carry-out of the second indicates overflow.

not even the most important metric—the wires actually cost more than the gates. Exploring that issue would take us too far afield, but we could certainly think about the *time* required for a circuit to compute its answer.

We can use *delay* as a simple measure of time. The delay of a circuit is the maximum number of gates between an input and an output. For example, the circuit of Figure 11.6 has delay 3.

A ripple-carry adder produces the output bits one at a time, from low order to high order (just like grade-school arithmetic). This means that a 64-bit adder would be eight times slower than an 8-bit adder—the delay doubles every time the number of input bits is doubled. That is unacceptable, and also unnecessary. By using extra hardware, the speed of such a "long" addition can be greatly increased; we explore this in Problem 11.10. Sometimes at least, it is possible to trade off hardware size against circuit speed.

Chapter Summary

- In computers, all logic is represented as patterns of `0`s and `1`s.
- A *gate*, the simplest piece of computer logic, produces a bit of output by performing an operation (like "or," "and," "exclusive or," or "not") on one or more bits of input.
- A *circuit* is a logical device built out of gates.
- The "nand" operator (denoted by | or $\uparrow$) is convenient for building circuits, because all of the conventional operators can be expressed using only "nand" operators. The "nor" operator (denoted by $\downarrow$) has the same property.
- *Binary notation* is the name for the base-2 number system used by computers, consisting of just `0`s and `1`s (as opposed to decimal notation, our familiar base-10 system).
- In binary, each column represents a power of 2. The right-most bit represents 2^0 and is called the *low-order* bit, while the left-most bit represents the largest value 2^n and is called the *high-order* bit.
- The value of a bit string is the sum of the powers of 2 corresponding to the columns of the bit string that contain the digit `1`.
- A *half adder* is a circuit that takes two input bits, the values to be summed; and returns two output bits, the sum and the carry bit.
- A *full adder* is a circuit that takes three input bits, the values to be summed and a carry-in bit; and returns two output bits, the sum and the carry-out bit. It can be built by combining two half adders.
- A *ripple-carry adder* is a circuit that adds two n-bit numbers. It can be built by combining n full adders.
- One way to measure a circuit's efficiency is its *delay*: the maximum number of gates between an input and an output.

Problems

11.1. (a) Convert the binary numeral `111001` to decimal, showing your work.
(b) Convert the decimal numeral 87 to binary, showing your work.

11.2. *Hexadecimal notation* is a particular way of writing base-16 numerals, using sixteen hexadecimal[2] digits: 0 through 9 with the same meanings they have in decimal, and `A` through `F` to represent decimal 10 through 15 (Figure 11.15). It is easy to translate back and forth between hexadecimal and binary, simply by substituting hexadecimal digits for blocks of four bits or vice versa. For example, since `8` in hex is `1000` in binary and `A` in hex is `1010` in binary, the four-hex-digit numeral `081A` would be written in binary as

`0000 1000 0001 1010`

where we have introduced small spaces between blocks of four bits for readability.

(a) Write the binary numeral `1110000110101111` in hexadecimal and in decimal.
(b) Write decimal 1303 in binary and in hexadecimal.
(c) Write hexadecimal `ABCD` in binary and in decimal.

[2]"Hexadecimal" comes from the Greek for "sixteen."

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

Figure 11.15. The sixteen hexadecimal digits and their corresponding binary and decimal notations.

11.3. We have described the use of sequences of n bits, for example $b_{n-1}\dots b_0$, to represent nonnegative integers in the range from 0 to $2^n - 1$ inclusive, using the convention that this sequence represents the number

$$\sum_{i=0}^{n-1} b_i 2^i.$$

Alternatively, the 2^n sequences of n bits can be used to represent 2^{n-1} negative numbers, zero, and $2^{n-1}-1$ positive integers in the range from -2^{n-1} to $2^{n-1}-1$ inclusive by interpreting $b_{n-1}\dots b_0$ to represent

$$-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i.$$

This is known as *two's complement* notation. Let $n=4$ for the purposes of this problem.

(a) What are the largest and smallest numbers that can be represented?
(b) What is the two's complement numeral for -1?
(c) What number is represented by `1010`?
(d) Show that a two's complement binary numeral represents a negative number if and only if the leftmost bit is 1.
(e) Convert these numbers to two's complement notation, work out their sum using the same rules for addition as developed in this chapter, and show that you get the right answer: $-5+4=-1$.

11.4. A string of n bits can also be used to represent fractions between 0 and 1, including 0 but not including 1, by imagining the "binary point" (like a decimal point) to the left of the leftmost bit. By this convention, the bit string $b_1\dots b_n$

represents the number

$$\sum_{i=1}^{n} b_i \cdot 2^{-i}.$$

For example, if $n = 4$, the string `1000` would represent $\frac{1}{2}$.

(a) Just as 0.9999 represents a number close to but less than 1 in decimal notation, so `1111` represents a number close to 1 in this binary notation. What is the exact value represented by `1111`? In general, what is the value represented by a string of n `1`s?

(b) Work out the sum $\frac{1}{2} + \frac{3}{8} = \frac{7}{8}$ using this notation.

11.5. Let $p \downarrow q$ denote the "nor" operator: $p \downarrow q$ is true if and only if neither p nor q is true. The $\downarrow$ operator is known as *Peirce's arrow*.[3]

[3]Named for the American mathematician, philosopher, and scientist Charles Sanders Peirce (1839–1914).

(a) Write the truth table for $p \downarrow q$.

(b) Using only the $\downarrow$ operator, write a formula equivalent to $\neg p$.

(c) Show that $\vee$ and $\wedge$ can also be expressed using just the $\downarrow$ operator.

11.6. Write a formula involving only the | operator that is equivalent to $p \vee q$.

11.7. Write the logical formulas for the values of Z_0, Z_1, and C_{out} of the two bit adder of Figure 11.14, in terms of the inputs X_0, Y_0, X_1, and Y_1.

11.8. Consider the boolean variables a, b, c each representing a single bit. Write two propositional formulas: one for the result of the boolean subtraction $a - b$, and another for the "borrow bit" c, indicating whether a bit must be borrowed from an imaginary bit to the left of a (the case when $a = 0$ and $b = 1$) to ensure that the result of the subtraction is always positive. Start by creating a truth table for $a - b$ and the borrow bit c. The borrow bit c is always `1` initially, and is set to `0` only if borrowing was necessary.

11.9. A common way of displaying numerals digitally is to use a pattern of up to seven straight line segments (see Figure 11.16). By turning various segments on and off, it is possible to form recognizable representations of the digits 0 through 9. In the figure, the seven strokes have been labeled with the letters A through G. The numeral 8 has all seven strokes on; 1 has C and F on and the others off; and 2 has A, C, D, E, and G on and the others off.

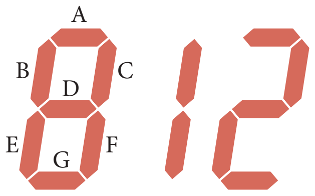

Figure 11.16. Digital numerals, composed of line segments A through G that can be turned on and off.

(a) Write out the strokes that should be turned on to represent each of the remaining digits.

(b) Write a truth table with five columns, showing when the A segment should be turned on given the binary representation of a number in the range 0 through 9. The first four columns are the bits of the digit to be displayed, and the last column is 1 if and only if stroke A is on in the representation of that digit. For example, if the values in the first four columns are `0010`, the binary representation of 2, the last column should be 1 since the A segment is on when 2 is displayed, as illustrated.

(c) Write a DNF formula for A based on the truth table from part (b).

(d) Draw a logic circuit that implements the formula from part (c). (A full design for this kind of display would need six other circuits, one for each of the other segments.)

11.10. Your task is to design a four-bit adder; that is, a circuit to compute the four-bit sum $Z_3Z_2Z_1Z_0$ of two four-bit inputs $X_3X_2X_1X_0$ and $Y_3Y_2Y_1Y_0$. The circuit also produces a carry-out bit C.

(a) Draw the two-bit adder of Figure 11.14 as a box with five input bits and three output bits (in the figure, the carry-in bit is set to 0). What is its delay, assuming that the half adder and full adder are implemented as shown in Figures 11.9 and 11.12 and each gate has delay 1? Show how to chain two of these boxes together to create a four-bit adder. What is the delay of your four-bit adder?

(b) The solution of part (a) has the disadvantage that the delay keeps doubling every time the number of input bits is doubled. An alternative solution uses more hardware to reduce the delay. It uses three two-bit adders, instead of two, to compute simultaneously

(A) the sum of the low-order bits (X_1X_0 and Y_1Y_0);

(B) the sum of the high-order bits (X_3X_2 and Y_3Y_2) on the assumption that the carry-in is `0`;

(C) the sum of the high-order bits on the assumption that the carry-in is `1`.

It then uses a few more gates to output either the output of (B) or the output of (C), depending on the value of (A). Diagram this solution, and compute its delay.

(c) If the method of parts (a) and (b) were generalized to construct 2^k-bit adders, what would be the delay of the circuits?

11.11. This problem further explores the Thue sequence (page 33).

(a) Prove that t_n, the n^{th} bit in the Thue sequence, is the exclusive or of the bits of the binary notation for n. (The exclusive or of a single bit is the bit itself.)

(b) Show that for every $n \geq 0$, $t_{2n} = t_n$, and t_{2n+1} is the complement of t_n.

11.12. A truth function $f : \{0, 1\}^k \to \{0, 1\}$ is said to be *monotone* if

$$f(x_1, \ldots, x_k) \leq f(y_1, \ldots, y_k) \text{ whenever } x_i \leq y_i \text{ for each } i = 1, \ldots, k.$$

That is, changing any argument from `0` to `1` cannot change the value from `1` to `0`. Show that a truth function is monotone if and only if it can be expressed by a propositional formula using only $\vee$ and $\wedge$ (and without $\neg$). Equivalently, it can be computed by a circuit with only "and" and "or" gates.