*3.4 BINARY ARITHMETIC

Because numbers are stored in a computer's memory in binary, computers must also perform arithmetic in binary. Binary addition is actually much easier than decimal addition since there are only three basic binary addition facts: `0 + 0 = 0`, `0 + 1 = 1 + 0 = 1`, and `1 + 1 = 10`. (This is `10` in binary, which has the value 2 in decimal.) With these basic facts, we can add any two arbitrarily long binary numbers using the same right-to-left algorithm that we all learned in elementary school.

For example, let's add the binary numbers `1110` and `0111`. Starting on the right, we add the last column: `0 + 1 = 1`.

```
     1  1  1  0
  +  0  1  1  1
  -------------
              1
```

In the next column, we have `1 + 1 = 10`. Since the answer contains more than one bit, we carry the `1`.

```
        1
     1  1  1  0
  +  0  1  1  1
  -------------
           0  1
```

In the next column, we have `1 + 1 = 10` again, but with a carry bit as well. Adding in the carry, we have `10 + 1 = 11` (or 2 + 1 = 3 in decimal). So the answer for the column is `1`, with a carry of `1`.

```
     1  1
     1  1  1  0
  +  0  1  1  1
  -------------
        1  0  1
```

Finally, in the leftmost column, with the carry, we have `1 + 0 + 1 = 10`. We write the `0` and carry the `1`, and we are done.

```
  1  1  1
     1  1  1  0
  +  0  1  1  1
  -------------
  1  0  1  0  1
```

We can easily check our work by converting everything to decimal. The top number in decimal is 8 + 4 + 2 + 0 = 14 and the bottom number in decimal is 0 + 4 + 2 + 1 = 7. Our answer in decimal is 16 + 4 + 1 = 21. Sure enough, 14 + 7 = 21.

More limited precision

Although Python integers can store arbitrarily large values, this is not true at the machine language level. Python integers are another abstraction built atop the native capabilities of the computer. At the machine language level (and in most other programming languages), every integer is stored in a fixed amount of memory, usually four bytes (32 bits). This is another example of *limited precision*.

We can illustrate this by revisiting the previous problem, but assuming that we only have four bits in which to store each integer. When we add the four-bit integers `1110` and `0111`, we arrived at a sum, `10101`, that requires five bits to be represented. When a computer encounters this situation, it simply discards the leftmost bit. In our example, this would result in an incorrect answer of `0101`, which is 5 in decimal. Fortunately, there are ways to detect when this happens, which we leave to you to discover as an exercise.

Negative integers

We assumed above that the integers we were adding were positive, or, in programming terminology, *unsigned*. But of course computers must also be able to handle arithmetic with *signed* integers, both positive and negative.

Everything, even a negative sign, must be stored in a computer in binary. One option for representing negative integers is to simply reserve one bit in a number to represent the sign, say 0 for positive and 1 for negative. For example, if we store every number with eight bits and reserve the first (leftmost) bit for the sign, then `00110011` would represent 51 and `10110011` would represent −51. This approach is known as *sign and magnitude* notation. The problem with this approach is that the computer then has to detect whether a number is negative and handle it specially when doing arithmetic.

For example, suppose we wanted to add −51 and 102 in sign and magnitude notation. In this notation, −51 is `10110011` and 102 is `01100110`. First, we notice that `10110011` is negative because it has `1` as its leftmost bit and `01100110` is positive because it has `0` as its leftmost bit. So we need to *subtract* positive 51 from 102:

$$\begin{array}{rccccccccl}
 & & 0 & 10 & 10 & & 0 & 10 & 10 & \\
 & 0 & \cancel{1} & \cancel{1} & \cancel{0} & 0 & \cancel{1} & \cancel{1} & \cancel{0} & \longleftarrow 102 \\
- & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \longleftarrow 51 \\
\hline
 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \longleftarrow 51
\end{array}$$

Borrowing in binary works the same way as in decimal, except that we borrow a 2 (`10` in binary) instead of a 10. Finally, we leave the sign of the result as positive because the largest operand was positive.

To avoid these complications, computers use a clever representation called *two's complement notation*. Integers stored in two's complement notation can be added directly, regardless of their sign. The leftmost bit is also the sign bit in two's complement notation, and positive numbers are stored in the normal way, with leading zeros if necessary to fill out the number of bits allocated to an integer. To convert a positive number to its negative equivalent, we *invert every bit to the left of the rightmost* `1`. For example, since 51 is represented in eight bits as `00110011`, −51 is represented as `11001101`.

To illustrate how addition works in two's complement notation, let's once again add −51 and 102:

$$\begin{array}{cccccccccl}
1 & 1 & & & 1 & 1 & & & & \\
 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & \longleftarrow -51 \\
+ & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & \longleftarrow 102 \\
\hline
\cancel{1} & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \longleftarrow 51
\end{array}$$

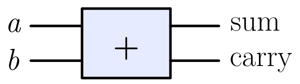
As a final step in the addition algorithm, we always disregard an extra carry bit. So, indeed, in two's complement, $-51 + 102 = 51$.

Note that it is still possible to get an incorrect answer in two's complement if the answer does not fit in the given number of bits. Some exercises below prompt you to investigate this further.

Designing an adder

Let's look at how binary addition is actually performed using only **and**, **or**, and **not** operations.

An *adder* takes two single bit inputs and outputs a two bit answer. We will name the rightmost bit in the answer the "sum" and the leftmost bit the "carry." So we want our abstract adder to look this:



The two single bit inputs enter on the left side, and the two outputs exit on the right side. Our goal is to replace the inside of this "black box" with an actual logic circuit that computes the two outputs from the two inputs.

The first step is to design a truth table that represents what the values of sum and carry should be for all of the possible input values:

a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Notice that the value of carry is 0, except for when a and b are both 1, i.e., when we are computing $1 + 1$. Also, notice that, listed in this order (carry, sum), the two output bits can also be interpreted as a two bit sum: `0 + 0 = 00`, `0 + 1 = 1`, `1 + 0 = 1`, and `1 + 1 = 10`. (As in decimal, a leading `0` contributes nothing to the value of a number.)

Next, we need to create an equivalent Boolean expression for each of the two outputs in this truth table. We will start with the sum column. To convert this column to a

Boolean expression, we look at the rows in which the output is 1. In this case, these are the second and third rows. The second row says that we want sum to be 1 when a is 0 *and* b is 1. The *and* in this sentence is important; for an **and** expression to be 1, both inputs must be 1. But, in this case, a is 0 so we need to flip it with **not** a. The b input is already 1, so we can leave it alone. Putting these two halves together, we have **not** a **and** b. Now the third row says that we want sum to be 1 when a is 1 *and* b is 0. Similarly, we can convert this to the Boolean expression a **and not** b.

a	b	carry	sum	
0	0	0	0	
0	1	0	1	⟵ **not** a **and** b
1	0	0	1	⟵ a **and not** b
1	1	1	0	

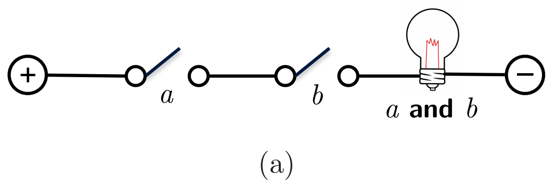
Finally, let's combine these two expressions into one expression for the sum column: taken together, these two rows are saying that sum is 1 if a is 0 and b is 1, *or* if a is 1 and b is 0. In other words, we need at least one of these two cases to be 1 for the sum column to be 1. This is just equivalent to (**not** a **and** b) **or** (a **and not** b). So this is the final Boolean expression for the sum column.

Now look at the carry column. The only row in which the carry bit is 1 says that we want carry to be 1 if a is 1 and b is 1. In other words, this is simply a **and** b. In fact, if you look at the entire carry column, you will notice that this column is the same as in the truth table for a **and** b. So, to compute the carry, we just compute a **and** b.

a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
		↑	↑
		a **and** b	(**not** a **and** b) **or** (a **and not** b)

Implementing an adder

To implement our adder, we need physical devices that implement each of the binary operators. Figure 1(a) shows a simple electrical implementation of an **and** operator. Imagine that electricity is trying to flow from the positive terminal on the left to the negative terminal on the right and, if successful, light up the bulb. The binary inputs, a and b, are each implemented with a simple switch. When the switch is open, it represents a 0, and when the switch is closed, it represents a 1. The light bulb represents the output (off = 0 and on = 1). Notice that the bulb will only light up if both of the switches are closed (i.e., both of the inputs are 1). An **or** operator can be implemented in a similar way, represented in Figure 1(b). In this case, the bulb will light up if at least one of the switches is closed (i.e., if at least one of the inputs is 1).



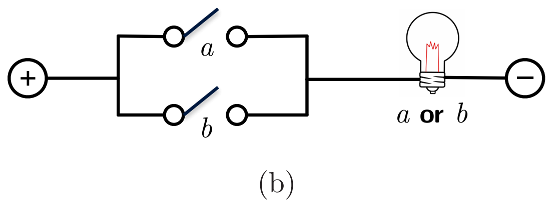


Figure 1 Simple electrical implementations of an (a) **and** and (b) **or** gate.

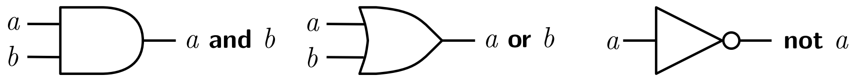
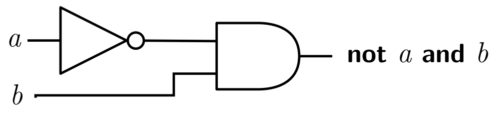


Figure 2 Schematic representations of logic gates.

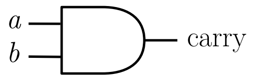
Physical implementations of binary operators are called ***logic gates***. It is interesting to note that, although modern gates are implemented electronically, they can be implemented in other ways as well. Enterprising inventors have implemented hydraulic and pneumatic gates, mechanical gates out of building blocks and sticks, optical gates, and recently, gates made from molecules of DNA.

Logic gates have standard, implementation-independent schematic representations, shown in Figure 2. Using these symbols, it is a straightforward matter to compose gates to create a *logic circuit* that is equivalent to any Boolean expression. For example, the expression **not** a **and** b would look like the following:

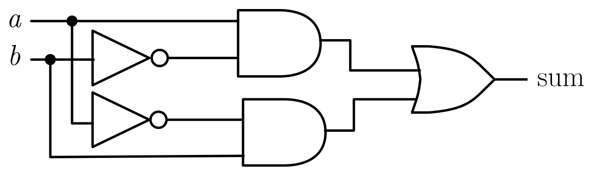


Both inputs a and b enter on the left. Input a enters a **not** gate before the **and** gate, so the top input to the **and** gate is **not** a and the bottom input is simply b. The single output of the circuit on the right leaves the **and** gate with value **not** a **and** b. In this way, logic circuits can be built to an arbitrary level of complexity to perform useful functions.

The circuit for the carry output of our adder is simply an **and** gate:

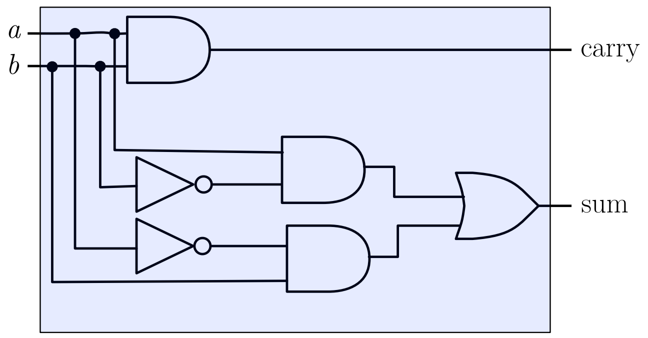


The circuit for the sum output is a bit more complicated:

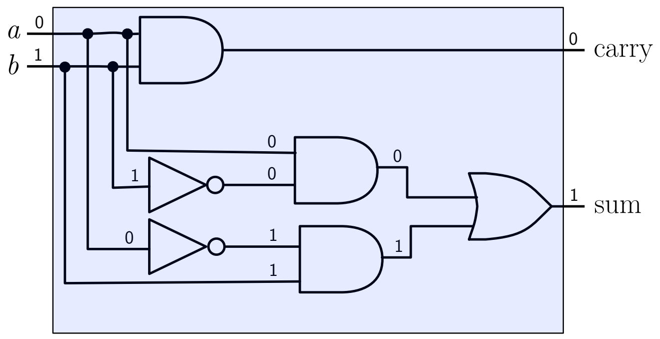


By convention, the solid black circles represent connections between "wires"; if there is no solid black circle at a crossing, this means that one wire is "floating" above the other and they do not touch. In this case, by virtue of the connections, the a input is flowing into both the top **and** gate and the bottom **not** gate, while the b input is flowing into both the top **not** gate and the bottom **and** gate. The top **and** gate outputs the value of a **and not** b and the bottom **and** gate outputs the value of **not** a **and** b. The **or** gate then outputs the result of **or**ing these two values.

Finally, we can combine these two circuits into one grand adder circuit with two inputs and two outputs, to replace the "black box" adder we began with. The shaded box represents the "black box" that we are replacing.



Notice that the values of both a and b are each now flowing into three different gates initially, and the two outputs are conceptually being computed in parallel. For example, suppose a is 0 and b is 1. The figure below shows how this information flows through the adder to arrive at the final output values.



In this way, the adder computes $0 + 1 = 1$, with a carry of 0.

Exercises

3.4.1* Show how to add the unsigned binary numbers `001001` and `001101`.

3.4.2. Show how to add the unsigned binary numbers `0001010` and `0101101`.

3.4.3* Show how to add the unsigned binary numbers `1001` and `1101`, assuming that all integers must be stored in four bits. Convert the binary values to decimal to determine if you arrived at the correct answer.

3.4.4. Show how to add the unsigned binary numbers `001010` and `101101`, assuming that all integers must be stored in six bits. Convert the binary values to decimal to determine if you arrived at the correct answer.

3.4.5. Suppose you have a computer that stores unsigned integers in a fixed number of bits. If you have the computer add two unsigned integers, how can you tell if the answer is correct (without having access to the correct answer from some other source)? (Refer back to the unsigned addition example in the text.)

3.4.6* Show how to add the two's complement binary numbers `0101` and `1101`, assuming that all integers must be stored in four bits. Convert the binary values to decimal to determine if you arrived at the correct answer.

3.4.7. What is the largest positive integer that can be represented in four bits in two's complement notation? What is the smallest negative number? (Think especially carefully about the second question.)

3.4.8. Show how to add the two's complement binary numbers `1001` and `1101`, assuming that all integers must be stored in four bits. Convert the binary values to decimal to determine if you arrived at the correct answer.

3.4.9. Show how to add the two's complement binary numbers `001010` and `101101`, assuming that all integers must be stored in six bits. Convert the binary values to decimal to determine if you arrived at the correct answer.

3.4.10. Suppose you have a computer that stores two's complement integers in a fixed number of bits. If you have the computer add two two's complement integers, how can you tell if the answer is correct (without having access to the correct answer from some other source)?

3.4.11. Subtraction can be implemented by adding the first operand to the two's complement of the second operand. Using this algorithm, show how to subtract the two's complement binary number `0101` from `1101`. Convert the binary values to decimal to determine if you arrived at the correct answer.

3.4.12. Show how to subtract the two's complement binary number `0011` from `0110`. Convert the binary values to decimal to determine if you arrived at the correct answer.

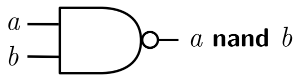
3.4.13. Copy the completed adder circuit, and show, as we did above, how the two outputs (carry and sum) obtain their final values when the input a is 1 and the input b is 0.

3.4.14* Convert the Boolean expression **not** (a **and** b) to a logic circuit.

3.4.15. Convert the Boolean expression **not** a **and not** b to a logic circuit.

3.4.16. The single Boolean operator **nand** (short for "not and") can replace all three traditional Boolean operators. The truth table and logic gate symbol for **nand** are shown below.

a	b	a **nand** b
0	0	1
0	1	1
1	0	1
1	1	0



Show how you can create three logic circuits, using only **nand** gates, each of which is equivalent to one of the **and**, **or**, and **not** gates. (Hints: you can use constant inputs, e.g., inputs that are always 0 or 1, or have both inputs of a single gate be the same.)

Selected Exercise Solutions

3.4.1

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

3.4.3

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

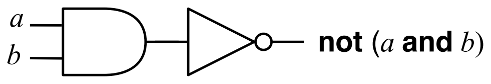
This answer is incorrect since we had to discard the leftmost `1` from the answer.

3.4.6

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

This answer is correct: $5 + -3 = 2$.

3.4.14


a
b
not (a and b)