4: Physical representations of numbers

Summary:

Computers are devices that perform operations on numbers. A computer is also supposed to be a real-world physical machine, whereas a number is something abstract. We will bridge that gap in this chapter by introducing binary representations of numbers.

Recall our two organizing questions: How do we program a computer? How do we build a computer? We've now addressed the first question at a high level: We can program a computer using the nice human-adapted programming language of Python and let the compiler take care of the dirty work of actually telling the machine the corresponding machine-friendly operations.

In the next chapter, we'll start to address the question of what those machine-friendly operations are. In other words, "How do we really program a computer?" However, as we descend closer to the level of the computer itself, we start to run into questions related to the construction of computers. (In other words, the abstraction of a computer leaks!)

It will be profitable to plateau on our journey down the tower of abstractions and take a chapter to address one of these in particular. Specifically, consider the following conundrum: A computer is supposed to be a physical machine, but it is also supposed to be able to add and subtract numbers. Numbers are abstract, whereas the computer is physical, so when we say that the computer "adds" two numbers, we must mean instead that it takes two physical things--maybe pieces of paper with holes in them, or electrons flowing through wires, or perhaps something else entirely--that represent the two numbers it should add, and it outputs somehow into a third physical thing (paper, electrons, etc.) that represents the sum.

This will affect our understanding of how to program computers at a low-level: Low-level arithmetic operations will act on numbers, but they will be constrained and influenced by how the computer actually represents the numbers that it is adding and subtracting. So we take a detour to study the following question: How can we represent numbers and operations on numbers physically?

4.1: Chapter organization

This chapter introduces several systems for representing various different classes of numbers. Each system is introduced with a section describing it conceptually, as well as the associated terminology and the procedures for working with it. Following these introductions, there will be sections containing worked examples of manipulating numbers within that system.

4.2: Unary representation of numbers: A bouncy-ball-based computer

Summary:

Before we get to binary--the main goal of this chapter--we present the simplest possible way of realizing numbers as physical things: To represent the number 15, say, simply take 15 identical objects. This is called a "unary" representation, and it could be used to build a computer rather easily. This computer would be fundamentally limited in the size of the numbers it could deal with, however.

Definitions: unary

A very simple way of representing numbers physically is to have a large supply of identical small objects--say bouncy balls, for example--and then to represent the operation of storing a given number in a specific slot by taking that number of bouncy balls and physically placing them in a slot with the specified name. So, for example, a machine might see the program "x = 2" and physically take an empty container, write "x" on it, and then fill it with 2 bouncy balls.

Adding numbers is also quite easy: For our computer to be able to perform e.g.,

x = x + 5 it needs to have a contraption that, when it sees this operation, finds the container marked "x", and places 5 more bouncy balls into it.

For the purpose of writing down these representations, we will use the digit "1" to stand in for a single bouncy ball. For instance, we can write the bouncy-ball representation of 9 as:

```
111111111
```

And the addition of 9 and 4 would be written:

```
111111111 + 1111 = 1111111111111
```

This way of representing numbers is called **unary**, so-called because the 'un-' prefix indicates that we only need a single digit (1) to write down the representation of any number (just as the usual system of writing numbers is called "decimal", the prefix "deci-" referring to the use of 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Unary is conceptually very convenient for machine-building purposes--arithmetic operations are trivial to describe--but it becomes terribly unwieldy when dealing with numbers that are at all large. For example, the number that is so compactly written in decimal notation as 134 would be written in unary as:

```
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111
```

If we tried to run our factorial program on a unary-based computer, we could easily need more bouncy balls than there are atoms in the universe! So unary might be nice for a simple, slow computer, but not so much for the powerful and compact computers we are used to.

4.2.1: Worked examples

As we've seen, representing numbers in unary is easy: Just write that number of 1s. So 17 is

"11111111111111111". In the other direction, decoding a unary representation into a number is equally easy: Just count the 1s in the representation. So "11111111111111111111111111" is the representation of 25.

But remember that the point of this representation was to be able to build a machine that stores and adds numbers for us. We've talked a little about how this can be done, but we'll take this opportunity to flesh out some of those ideas a little.

We can imagine the machine consisting of three basic parts (as a call-out to the next chapter, these parts could be called the machine's "architecture"):

- 20 trays for storing bouncy balls, numbered 0-19.
- Two inputs, labeled input "A" and input "B", for feeding in bouncy balls.
- Three buttons labeled:
  1. Store input B into tray A
  2. Add input B to stored value in tray A
  3. Add tray B to tray A
- A very large reservoir of bouncy balls for use in computations.

At any point, you can place however many bouncy balls into the two inputs. We just need to specify what happens when you press each of the buttons. You should stop and consider what might be reasonable, and even work through to whatever desired level of detail how to construct a machine that behaves in the way described.

What happens when you press "Store input B into tray A"?

When we press "Store input B into tray A", the machine looks at the number of bouncy balls in input A, finds the corresponding tray (for example, if we place 4 bouncy balls into input A, it finds tray number 4). It then empties this tray of any balls it may have had previously (placing these into the reservoir), and dumps all the balls placed in input "B" into that tray.

So if we place 5 balls in input "A", 10 in input "B", and press "Store", tray 5 will be emptied into the reservoir and then the 10 balls in "B" will be dumped into the newly empty tray 5.

What happens when you press "Add input B to stored value in tray A"?

When we press "Add input B to stored value in tray A", the balls in input "A" should select which stored container to use, and the balls in input "B" should be dumped into that container, exactly as in the case of the "Store" button, except without emptying it first.

So if we place 19 balls in "A" and 6 in "B", then the machine will place the 6 balls from input "B" into container number 19, adding to however many were in that container to begin with.

What happens when you press "Add tray B to tray A"?

In this case, the two inputs both represent containers: The number of balls in input "A" tells is which container to add balls to, and the number of balls in input "B" tells us which container to add them from. Except we don't want to remove those balls from the second container--we just want to add the same number to the first container.

So if we place 19 balls in "A" and 6 in "B", then the machine will look at containers 19 and 6. Count how many balls are in container 6, and take the same number of balls from the reservoir, and add that many balls to container 19.

4.3: Binary representation of non-negative integers

Summary:

Unary was convenient because it had only one digit, so we could represent it easily with physical objects. It was bad at representing large numbers compactly, however. Decimal, on the other hand, is good at representing numbers compactly, but it is sort of complicated--having 10 different possible digits. Binary is a happy medium--a representation that is simple, having only two different digits, but also compact.

Definitions: bit, unsigned, significant, LSB, MSB, width, byte, word, nibble

A representation that is still simple enough to be suitable for machines but also compact enough to represent large numbers in less space is binary. In binary, as the prefix 'bi-' suggests, we use two digits: 0 and 1.

To understand precisely how it works, we can think back to the decimal representation that we're used to: When we write the number 134, all we've done is to put the digits 1, 3, and 4 next to each other, and somehow this can denotes a relatively large number--134. It does so by the notion of place-value: The right-most digit--4--represents how many 1s we have. The next digit represents how many 10s we have. The next, how many 100s. And so on. So 134 means: four 1s, three 10s, and one 100, which communicates the desired quantity:

How many...	100s?	10s?	1s?
Decimal digits	1	3	4

We used 1, 10, and 100 as the place values because these are powers of 10. Why we use powers of anything at all is an interesting mathematical idea that we will leave aside for now. Why we use powers of 10 specifically had to do with the fact that we were using 10 possible digits--0-9 (which in turn had to do with the fact that most humans have 10 fingers to use as a counting aid).

Binary, by contrast, will only have two digits--0 and 1. (Binary digits are also called **bits**.) And because we have two possible digits, we will use the powers of 2 for our place values: 1, 2, 4, 8, 16, 32, 64, 128, 256, 1024, etc. Thus the number in binary 11001 represents one 1, no 2s, no 4s, one 8, and one 16, for a total of 25.

Similarly, the number that we wrote in decimal as 134 can be written in binary by breaking it up into powers of 2: To make 134 we need one 128, one 4, and one 2. So:

How many...	128s?	64s?	32s?	16s?	8s?	4s?	2s?	1s?

Binary digits	1	0	0	0	0	1	1	0

Put more mathematically, if we have a binary number with bits a_n, ..., a_1, a_0, where a_i is the i-th digit from the right (so each a_i is either a 0 or a 1), then we have a formula for the number this represents:

$$a_n...a_1a_0 = a_n * 2^n + ... + a_1 * 2^1 + a_0 * 2^0$$

Since 134 = 128 + 4 + 2 = $2^7 + 2^2 + 2^1 = 1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$, the binary representation of 134 is 10000110.

Since we're talking about non-negative integers, that is, integers where there is no concern about sign--whether it is positive or negative--this is sometimes called the representation of an **unsigned integer**.

As in decimal, the first bits in a binary number have a much greater effect on the value: The difference between 1561 and 1562 isn't that large, but the difference between 1561 and 2561 is very large in comparison. For this reason it is common to call the first bits of a binary number more **significant**. Using this terminology, the last two bits would be called the two "least significant bits". Especially common is calling the last bit the **least significant bit** or **LSB** and the first bit, the **most significant bit** or **MSB**.

With 8 decimal digits, the largest number we can represent is 99999999--i.e., one less than the next place value of 100000000. With 8 binary digits, the largest number we can represent is 11111111, or 255--one less than the next place value of 256. In general, if we have an *n*-bit binary number, it can be a number anywhere from 0 through 2^n-1.

To build a computer based on binary, we need a way to represent bits physically. This is also not too hard. If we wish to build our computer out of bouncy balls, then instead of just having a tray that can hold however many bouncy balls, we can have a tray subdivided into cells, and each cell can hold at most one bouncy ball. So to store 8 bits, we need a container with 8 cells, and to store the value of 134, instead of having 134 bouncy balls, we only need 3:

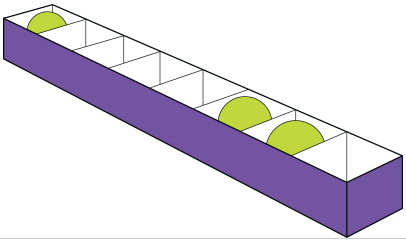

Figure 4.3.1:

In practice, computers are electrical devices, and bits are stored using cells that can either store electrical charge or not, where the presence of electrical charge stands for the bit being a 1, and the absence of charge stands for it being 0. We can then create an 8-bit storage container by taking 8 of these cells as we did with the bouncy balls.

However, if our computer's memory containers have only a fixed number of bits--in this example, 8--then this means we can only store numbers that can be represented with 8 bits. As the largest 8-digit binary number is 11111111, or 255 in decimal, this means 8-bit memories can only store numbers 0-255. Of course, we could do better by using more bits--modern processors often have 64-bit memories, which can store numbers up to 2^{64}-1, or around 16 quintillion. But in general, the computer's internal storage will be limited in this way to storing only numbers of a fixed width, and then it is up to programmers to concoct ways to use these limited memories to store larger numbers if they are needed.

The number of bits that an individual memory slot can store is called its **width**. A few particular widths of memories come up frequently, and so have special terminology associated with them:

- A memory slot that can store an 8-bit binary number is called a **byte**.
- A memory slot that can store a 16-bit binary number is called a **word**.
- A memory slot that can store a 4-bit binary number is called a **nibble**.

4.3.1: Worked examples

Decoding binary representations:

What number does the binary 10100101 represent?

Since the place values are the powers of 2, with 2^0 being on the right and increasing from right to left, and since the digits tell us how many of the corresponding place value to include, we can make a table:

Place value	128	64	32	16	8	4	2	1
Digits	1	0	1	0	0	1	0	1

So this corresponds to 128 + 32 + 4 + 1 = 165.

Alternatively, we can use the formula:

$$a_n...a_1a_0 = a_n * 2^n + ... + a_1 * 2^1 + a_0 * 2^0$$

To get the answer of

$$1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 128 + 32 + 4 + 1 = 165$$

Representing a number in binary:

Write down the binary representation of the number 71

Since each binary digit corresponds to whether or not to include the corresponding power of 2, all we have to do is work out which powers of 2 are needed to make up the given number. We can start by noticing that all powers of 2 such as $2^7 = 128$ and above are too big and stand no chance of being included.

So we can start with $2^6 = 64$. If we include that, then to make 71, we only need further powers of 2 to make 7 more (as 71 - 64 = 7).

To make this 7, we may therefore skip $2^5 = 32$, and $2^4 = 16$ and $2^3 = 8$, and the next power of 2 that is small enough to be included is $2^2 = 4$. So now we've included 64 and 4, for a total of 68. To make the desired total of 71, we need 3 more.

To make this 3, we start by including the next power of 2: $2^1 = 2$. Having included 64, 4, and 2, we have a total of 70, which is just 1 shy of the desired 71.

To make up for this missing 1, we include the last power of 2: $2^0 = 1$. So the powers of 2 we include are 64, 4, 2, and 1.

What we've done is written 71 as $2^6+2^2+2^1+2^0$. Or, more suggestively, this same thing can be written as:

$$71 = 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$$

Referring back to our formula, this means that the binary representation for 71 is "1000111".

To summarize the work we did here: We started with 71. We then proceeded down the powers of 2: 64, 32, 16, 8, 4, 2, 1, and noted if we could take one away from what remained. If we could take away that power of 2, we did so and wrote a 1 in the binary representation. Otherwise, we wrote a 0. To wit:

- 71 remains. First small enough power of 2 is 64. This can be taken away, so add a 1 to the representation. Take it away and have 7 remaining. Representation so far: **1**.
- 7 remains. Next power of 2 is 32. This cannot be taken away, so add a 0 to the representation. Representation so far: **10**.
- 7 remains. Next power of 2 is 16. This cannot be taken away, so add a 0 to the representation. Representation so far: **100**.
- 7 remains. Next power of 2 is 8. This cannot be taken away, so add a 0 to the representation. Representation so far: **1000**.
- 7 remains. Next power of 2 is 4. This can be taken away, so add a 1 to the representation. Take it away and 3 remains. Representation so far: **10001**.
- 3 remains. Next power of 2 is 2. This can be taken away, so add a 1 to the representation. Take it away and 1 remains. Representation so far: **100011**.
- 1 remains. Next power of 2 is 1. This can be taken away, so add a 1 to the representation. Take it away and 0 remains. Representation so far: **1000111**.
- 0 remains. We're done! Final representation: **1000111**.

It will be convenient to summarize this even more compactly in tabular form:

What's left?	71	7	7	7	7	3	1
Place value	64	32	16	8	4	2	1
Place value < What's left?	Yes	No	No	No	Yes	Yes	Yes
Digit	1	0	0	0	1	1	1

4.4: Binary representation of negative integers

Summary:

Our main storage mechanism in a computer will be devices that can store sequences of bits. Most often, these bits will be understood to represent non-negative integers according to the above scheme. However, it will be convenient to represent other types of numbers using sequences of bits as well. In this section, we describe how to store general integers that can be either positive or negative.

The most common way of representing negative numbers is called two's complement, and consists of taking the same place-values as above except that the largest one is negative.

Definitions: sign bit, twos complement

There are two common ways of representing arbitrary integers as sequences of bits: With a sign bit, and with two's complement.

The **sign bit** method is as follows: Suppose we have an 8-bit memory and we want to treat it as an integer with sign bit. Then we treat the 7 least significant bits as usual, but the 8th bit, instead of giving it its usual place value of 128, we simply say that if it is 0, the number is positive, and if it is 1, the number is negative.

Thus, under this scheme, 00000001 represents the number 1, and 10000001 represents -1. The largest number we can represent is 01111111, i.e., 127, and the lowest is 11111111, or -127. This should immediately raise our suspicious, because there are only 255 numbers between -127 and 127 inclusive, but there are 256 different sequences of 8 bits. That means that under this scheme, there are two sequences of 8 bits that represent the same number! Stop for a minute and think about what these are.

Click to toggle answer
Under this scheme 00000000 represents the number 0, but so too does 10000000.

Other issues with this method include the fact that the method for adding them is rather complex, both in terms of how to explain it and how to implement it with circuitry, but we'll leave it be for now and instead discuss the more common method employed in the wild:

The preferred method for representing arbitrary integers in binary is called **two's complement**, in which the place values are all the same except the most significant bit's place value is the negative of what it otherwise would be. In the case of 8-bit integers, this means that the MSB has a place value of -128. Explicitly, this means that if the MSB is 0 then the number has the same meaning as in the non-negative case, and i the MSB is 1, then you determine its value by looking at the other bits, computing what they represent as a non-negative integer, and then adding -128.

For example, the numbers 00000000 (0), 00000001 (1), through 01111111 (127) are all the same as before, but 10000000 then represents -128, and 10000001 is -127, all the way until -11111111, which is -1. Thus we can represent any integer between -128 an 127 inclusive--or 256 different values, so there are no repetitions.

This scheme may look odd on paper, but on top of having no redundancy, it will turn out that the same procedure we'll use to add non-negative binary numbers can also be used to add two's complement binary integers as well.

4.4.1: Worked examples

Listing two's complement binary representations:

List all 5-bit two's complement integers

Binary	Integer
00000	0
00001	1
00010	2
00011	3
00100	4
00101	5
00110	6
00111	7
01000	8
01001	9
01010	10
01011	11
01100	12
01101	13
01110	14
01111	15
10000	-16
10001	-15
10010	-14
10011	-13
10100	-12
10101	-11
10110	-10
10111	-9
11000	-8
11001	-7
11010	-6
11011	-5
11100	-4
11101	-3
11110	-2

11111	-1

Decoding two's complement binary representations:

What integer does the 8-bit two's complement 10100101 represent?

The place values for two's complement are the same as for non-negatives--the powers of 2--with the exception that the highest place value is negative. So the place values and corresponding digits in the given representation are:

Place value	-128	64	32	16	8	4	2	1
Digit	1	0	1	0	0	1	0	1

This means that the number represented is -128 + 32 + 4 + 1 = -91.

Representing a number using two's complement:

Write the 8-bit two's complement representation of the number -56.

As with the non-negatives, we have the powers of 2 as our place values, and we have to build -56 out of them. The difference is that the highest place-value--since we're looking at 8-bit representations, that's 128--is actually negative. So the place values out of which we must build -56 are: -128, 64, 32, 16, 8, 4, 2, 1. We can do this using the table as before:

What's left?	-56	72	8	8	8	0	0	0
Place value	-128	64	32	16	8	4	2	1
Place value < What's left?	Yes	Yes	No	No	Yes	No	No	No
Digit	1	1	0	0	1	0	0	0

So the 8-bit two's complement representation of -56 is 11001000.

Representing a number using a different size two's complement representation:

Write the 7-bit two's complement representation of the number -56.

Now, our place-values are -64, 32, 16, 8, 4, 2, 1. Otherwise, we proceed as before:

What's left?	-56	8	8	8	0	0	0
Place value	-64	32	16	8	4	2	1
Place value <= What's left?	Yes	No	No	Yes	No	No	No
Digit	1	0	0	1	0	0	0

So the 7-bit two's complement representation of -56 is 1001000.

Write the 16-bit two's complement representation of the number -56.

Now, our place-values are -65536, 32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1. Using the same procedure as above (noting that we need -65536 as we are trying to represent a negative number and this is the only negative place value), we come up with an answer of: 1111111111001000

4.5: Binary representation of real numbers

Summary:

There are two schemes for representing fractions in binary: Fixed point (where we simply replace some of the place values with fractional place values) and floating point (which is analogous to "scientific notation" for decimal numbers).

Definitions: fixed point, floating point

To represent real numbers, we can take our cue from decimal representations: What does 12.34 mean? The piece before the point--the 12--is treated as an ordinary decimal number with the usual place values. Then, the digits after the point are given reciprocal place values: 1/10, 1/100, etc. So specifically, this means one 10, two 1s, three 1/10s, and four 1/100s.

If we're instead talking about, say, 8-bit binary numbers, we can declare that the point is always in a fixed place, say right in the middle. So the number 11010110 represents the binary number 1101.0110, that is:

How many...	8s?	4s?	2s?	1s?	1/2s?	1/4s?	1/8s?	1/16s?
Binary digits	1	1	0	1	0	1	1	0

For an end result of 8 + 4 + 1 + 1/4 + 1/8 = 13.375

This scheme is called **fixed point** because the point is at a predetermined and fixed location for all numbers encoded in this scheme

Under this scheme, however, the largest number we can represent is 11111111, or 15.9375.

Even if we extended this scheme to 16 bits, where we still place the point in the middle, the highest number we can represent then becomes 270.9375. Compared with the largest integer value we can store with 16 bits--65535--this is rather small. So another scheme was devised--floating point--inspired by standard scientific notation. To write a number, say 532.9, in scientific notation, you start by putting the point right after the first digit, so in this example, 5.329. That's not the actual number, but if we multiply it by 10 enough times, the decimal point will move back to where it was supposed to be:

5.329×10^2, or commonly, 5.329E2. So rather than having to worry about where the point goes, we have effectively two pieces of information to keep track of: A single fixed point number (5.329), and the exponent, 2. If we wanted a smaller number like .005329, all we have to do is change the exponent to, in this case, -3.

In a **floating point** scheme, we use 16 bits (though there are variants for larger numbers of bits). We use the first bit of these as a sign bit exactly as in the sign bit scheme. The next 5 will represent a non-negative integer that will be called the exponent. (And because we're working in binary, this exponent will tell us the power of 2 to multiply by, rather than a power of 10.) The remaining 10 bits will represent a fixed-point binary number, called the "fraction", with the point just before the first digit. Then the formula is:

$$\text{Number represented} = (\text{sign}) * 2^{\text{exponent} - 15} * (1+\text{fraction})$$

This is all explained best, perhaps, in an example: We will represent the number

1	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0
sign = -1	exponent = 12					fraction = .0011 = .1875									

The first bit is 1, so the number is negative. The next 5 bits, treated as a non-negative integer, represent the number 12, and the fixed-point number represented by the last 10 bits is, in binary, .0011, (.1875 in decimal). So the decimal real number being represented by this binary string is:

$$(-1) * (2^{12-15})(1 + .1875) = -.1484375$$

This scheme being rather complicated means that the procedure for adding or multiplying two such numbers is also quite involved. In practice, computers that deal with real numbers often have a piece of dedicated hardware called a floating point unit (FPU) with circuitry especially for handling these operations.

It is rare to see 16-bit floating point numbers in the wild. More common are 32-bit floating point numbers, which use (as the name suggests) 32 bits instead of 16. They follow a similar formula:

$$\text{32-bit floating point number} = (\text{sign}) * 2^{\text{exponent} - 127} * (1+\text{fraction})$$

and use different numbers of bits representing each of the parts: The sign is still one bit, the exponent is now 8 bits, and the fraction is now the remaining 23 bits.

4.5.1: Worked examples

Decoding a 16-bit floating point representation:

What real number does the word 0100001000000000 represent?

Recall the scheme is: First bit is sign bit; next 5 bits are the exponent; last 10 bits are fixed-point fraction with the point before the first digit. The represented number is then:

$$(\text{sign}) * 2^{(\text{exponent}-15)} * (1 + \text{fraction})$$

The sign bit is 0, so the answer is positive. The next five bits represent the exponent in unsigned form. So we can decode them like any non-negative binary representation:

Place value	16	8	4	2	1
Digit	1	0	0	0	0

So the exponent is 16.

Finally, the base is the fixed-point number .1 with the point after the first digit, i.e:

Place value	1/2	1/4	1/8	1/16	1/32
Digit	1	0	0	0	0

So this represents 1/2 = .5. Thus the answer is $1 * 2^{16-15} * (1+.5) = 3$

Representing a number using floating point:

How to represent the number 7.625 in 16-bit floating point?

Recall the scheme is: First bit is sign bit; next 5 bit are two's complement exponent; last 10 bits are fixed-point base with the point after the first digit. The represented number is then:

$$7.625 = (\text{sign}) * 2^{(\text{exponent}-15)} * (1+\text{fraction})$$

The sign bit is 0, as the number is positive. Now we need to figure out the exponent and the fraction. Because the fraction is a fixed point number, it is a number between 1 and 2. So we are left with solving:

```
7.625 = 2^(exponent-15) * (1+fraction)
```

But 1+fraction is between 1 and 2, so

```
7.625 / 2^(exponent-15)
```

must be between 1 and 2. So we need to find an exponent that will make this true. We quickly find that 7.625/4 will be between 1 and 2, so $2^{(exponent-15)}$ must be 4, i.e., the exponent is 17. Writing this in binary, we get the exponent is 10001.

Finally, this means the 1+fraction must be 7.625/4 = 1.90625, meaning the fraction must be .90625 represented using fixed-point with the point before the first digit.

To represent a number in fixed point, we simply work our way down the place values just as we do for non-negative integers. The place values are 1/2, 1/4, 1/8, 1/16, etc. So we go through these in order, taking off that much if we can (we will not review the meaning of this table here--revisit the examples for non-negative integers if this table makes no sense):

What's left?	0.90625	0.40625	0.15625	0.03125	0.03125	0
Place value	1/2	1/4	1/8	1/16	1/32	1/64
Place value in decimal	0.5	0.25	0.125	0.0625	0.03125	0.015625
Place value <= What's left?	Yes	Yes	Yes	No	Yes	No
Digit	1	1	1	0	1	0

So our fraction is 11101, our exponent is 10001, and our sign bit is 0. Thus the representation is 0100011110100000.

Representing a harder number using 16-bit floating point:

How to represent the number 1/3 in floating point?

As before, the sign bit is 0. 1/3 is not between 1 and 2, so we need to multiply it by 2 until it is: 2 * 1/3 is still not in the range, but 2*2*1/3 = 4/3 is. So 1+fraction will have to be 4/3 and the exponent part will have to be 1/4, i.e. 2^{-2}. That is, $2^{exponent - 15}$ must be 2^{-2}, so exponent must be 13. As a 5-bit non-negative integer, the binary representation of 13 is 01101

Now, 1+fraction is 4/3, so fraction is 1/3. We need to number in fixed point. We go through the same table as before:

Place value	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024
Place value in decimal	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	.00390625	.001953125	.0009765625
What's left?	.333...	.333...	.08333...	.08333...	.0208333...	.0208333...	.005208333...	.005208333...	.00130208333...	.0013020833
Place value <= What's left?	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
Digit	0	1	0	1	0	1	0	1	0	1

But at this point we've got 10 digits already for the fraction, which is all we are allowed. However, we didn't get what was left over all the way down to 0! So the number we are representing isn't actually 1/3, but is simply as close as we can get to it using a 10-bit fraction part.

For this example, our fraction is 0010101010, our exponent is 01101, and our sign bit is 0. Thus the binary representation for the 16-bit floating point number that is as close to 1/3 as we can get is 0110101010101010. But the actual value of this is only .3330078125, rather than 0.3333333...

This might be a little unnerving, however it shouldn't come as too much of a surprise: There are only 16 bits, so there are only 2^{16} = 65536 possible fractions we can represent. However there are infinitely many fractions even just between 0 and 1--for example 1/2, 1/3, 2/3, 1/4, 3/4, 1/5, 2/5, ... Our scheme by definition cannot represent them all if each possible sequence of bits corresponds to one fraction. 1/3 is just the simplest one that cannot be represented exactly by this scheme, but any scheme that uses a fixed number of bits will necessarily have some manifestation of the same issue.

4.6: Hexadecimal

Summary: A third representation beyond unary and binary is hexadecimal. It is even more compact than binary, but if you have a hexadecimal representation it is easy to read off the binary representation (unlike with a decimal representation).

Binary is more compact than unary, but large-ish numbers still have many digits--the number we'd write as 99 becomes "1100011". When looking at a lot of numbers (as we will do when looking up the numbers that comprise, say, a simple image file), it is convenient to have an even more compact representation, but one from which we can still easily deduce the corresponding binary representation (unlike, say, converting from decimal to binary, which was a rather involved procedure).

For example, the number 8888, if represented in binary, would be:

```
10001010111000
```

It gets worse when we are dealing with a large number of bytes. For example, in chapter 6 we will see a long sequence of bytes that collectively represent a simple image. If we write these bytes in binary, then the sequence looks like:

```
01000010 01001101 00111010 00000101
00000000 00000000 00000000 00000000
00000000 00000000 10001010 00000000
00000000 00000000 01111100 00000000
00000000 00000000 00010100 00000000
00000000 00000000 00010100 00000000
00000000 00000000 00000001 00000000
00011000 00000000 00000000 00000000
00000000 00000000 10110000 00000100
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
11111111 00000000 00000000 11111111
00000000 00000000 11111111 00000000
00000000 00000000 00000000 00000000
00000000 11111111 01000010 01000111
01010010 01110011 10000000 11000010
11110101 00101000 01100000 10111000
00011110 00010101 00100000 10000101
```

In fact this is only about a quarter of the file, and it is already quite long and unwieldy. There is a further representation that is both more compact than binary but is very easily converted to binary, called **hexadecimal**. The idea is that we split every binary representation into groups of 4 bits, and then simply translate according to the following table, which includes all 4-bit binary representations along with the numbers they represent in decimal and the new hexadecimal digit that corresponds to those 4-bits:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hexadecimal	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

For example, instead of 10001010111000, we can write in hexadecimal as 22b8 because:

4-bit section	0010	0010	1011	1000
Corresponding hexadecimal digit	2	2	b	8

One feature of this scheme is that because every byte is two 4-bit chunks, any byte can be represented neatly by two hexadecimal digits.

Numbers written in hexadecimal can be visually similar to numbers written in decimal if none of the hexadecimal digits happen to be a-f. For example, the number '33' might be the decimal number thirty-three, or it might be the hexadecimal representation for the number 51. To alleviate this, numbers written in hexadecimal are usually preceded by "0x". So in future, unless context removes this ambiguity, we will write 33 for the decimal number "thirty-three", and 0x33 for the hexadecimal representation (which turns out to represent the number "fifty-one").

While the details do not much emphasis at this point, hexadecimal can also be understood as another representation system with 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 , a, b, c, d, e f, and whose place values are the powers of 16. So 0xff represents the number $f*16^1 + f*16^0$, where f is the digit representing the number 15. So this is 15*16 + 15 = 255. Of course, 0xff is also in binary 11111111, which we already know represents the number 255. So everything works out in the end.

4.6.1: Worked examples

Writing a binary number in hexadecimal:

Write the binary number 1001101 in hexadecimal

We would like to break this into 4-bit chunks, except it is 7 bits long, so this won't work evenly. However, just like in decimal we could redundantly write 0008 as another way of writing the number 8, here we can tack on 0s to the beginning of the representation without changing its value. So 01001101 represents the same value, but is conveniently 8 bits, so can be broken up.

We break this up as 0100 1101, which (consulting the above table) correspond to hex digits 4 and d respectively, so the hex representation is 0x4d.

Reading a hexadecimal representation in binary:

Write the hexadecimal representation 0xf00d in binary

Each digit represents a 4-bit chunk in binary. Consulting the table: 0xf = 1111, 0x0 = 0000, 0x0 = 0000, 0xd = 1101, so 0xf00d = 1111000000001101.

Looking at many hexadecimal numbers:

Convert the above segment of the file into hex

We can play the game from the first exercise on all the bytes in the excerpt above

```
42 4d 3a 05
```

```
00 00 00 00
00 00 8a 00
00 00 7c 00
00 00 14 00
00 00 14 00
00 00 01 00
18 00 00 00
00 00 b0 04
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
ff 00 00 ff
00 00 ff 00
00 00 00 00
00 ff 42 47
52 73 80 c2
f5 28 60 b8
1e 15 20 85
```

This is at least a little more manageable.

4.7: 8 bits--three different meanings?

Above we described several ways of interpreting sequences of bits as binary numbers. These interpretations were not inherent to the bits themselves: The sequence 11000110 would be 198 as an unsigned integer, -70 as an integer with sign bit, -58 as a two's complement integer, 12.375 as a fixed-point number with the point in the middle, and 1.546875 if considered as a fixed-point number with the point after the first digit.

When a computer stores this sequence, it doesn't differentiate between these interpretations until it comes time to, say, add two numbers. For instance, if we wanted to add 2 to this number, then as an unsigned integer, we should get 200, or 11001000. Incidentally, if we add two to this as a two's complement integer, we get the answer -56, which in two's complement is also 11001000. On the other hand, to add two to the fixed-point number with the point in the middle, we get the answer of 14.375, represented as 11100110, whereas we cannot actually add 2 to the fixed-point number with the point after the first digit, as this cannot represent any number larger than 11111111, or 1+127/128, or 1.9921875.

The point is that the procedures for adding numbers vary according to what representation you wish to use. So if you're telling the computer to add two numbers it has stored, you have to know how the numbers are stored and tell it to use the appropriate procedure for adding those two kinds of numbers.

To play around with the different possible interpretations of the same sequence of bits, enter a sequence of bits into the box at the top of this widget. When you click the "[update]" button, the various interpretations will be rendered when they make sense (for example, if you enter a 13-bit number and specify in the second row "8-bit two's complement integer", interpreting a 13-bit representation as an 8-bit integer doesn't make sense and an appropriate error will be displayed).

```
[Binary simulator goes here in website version]
```

4.8: Integer operations on representations

Summary:

We've been talking about ways to interpret sequences of bits, but sequences of bits are just sequences of bits, and a computer doesn't care how you the human are thinking about them. The only time it actually matters how you interpret them is when you go to add or otherwise operate on them.

To add binary integers represented in unsigned or two's complement, simply execute the grade-school "add each digit with carrying" procedure except in binary.

To negate a two's complement integer, simply negate it and then add 1 to the result.

Definitions: concatenation

When it comes time to talk about building computers, we will want to have circuitry that performs arithmetic operations (as these are one of the four things a computer actually does!) And to design such circuitry, it will be useful to have an idea of how one might perform various arithmetic operations on binary representations of numbers by hand. In this section, we'll stick to unsigned and two's complement integers.

Addition: The procedure for adding numbers in binary is actually the same as the procedure you likely learned in grade school, where you add the digits in order from right to left with carrying. For example, 1+0 = 1, and 1+1 = 10.

Negation: You get the two's complement integer 11101100. How to quickly find the two's complement representation of its negative? Easy: First, flip all the bits to their opposite values: 00010011, and then add 1 to that as in the previous paragraph, for an answer of 00010100. As a check, the original representation 11101100 represented -20, and the result of this procedure is indeed 20.

Subtraction: Now that we can negate and add, we can compute x - y for two given two's complement representations by simply negating y and then adding that to x.

Concatenation:

If we have two bytes stored somewhere, then each of these can only hold values 0-255 (if being interpreted as unsigned integers). However, despite the fact that they are stored separately, we can think of these 8-bit quantities as together comprising one big 16-bit quantity. This is called **concatenation**, and is denoted with a ":", as in "34:209" is the concatenation of two bytes: 34, and 209. We can always compute the concatenation of two bytes X and Y by the formula

```
```

```
X:Y = X*256 + Y
```

4.8.1: Worked examples

Adding non-negative integers:

Add the unsigned binary numbers 10011011+11110001

We said that the same "add the digits with carrying" procedure as we learned in grade school applies here too. Let us then begin by remembering this procedure for adding decimal numbers: To add 987 + 155, we would line up the numbers:

	9	8	7
+	1	5	5

Then we would add the right-most digits: 7+5 = 12, so we say that the answer is 2, but we carry the 1:

	9	2^{+1}	7
+	1	5	5
=			2

Then we add the next right-most digits, along with the carried 1: So we have 2+1+5 = 8. This has no carry, so we can proceed normally:

	9	2^{+1}	7
+	1	5	5
=		8	2

Finally, we are left with 9+1, which is 10, or "0, and carry the 1":

	$^{+1}$	9	2^{+1}	7
+		1	5	5
=		0	8	2

This final carrying created a fourth digit to add, which is just 1+0 = 1.

	$^{+1}$	9	2^{+1}	7
+		1	5	5
=	1	0	8	2

So the answer is 1082.

Crucial to this procedure was our ability to quickly add single digits, e.g., 9+1 = 10, 7+5 = 12, and to recognize which of these involved carrying. The way we learn how to do this is to more or less know by heart a giant table of how to add individual digits:

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

In binary, the procedure is exactly the same, except there are only two possible digits, so knowing how to add the individual digits is way easier: 0+0 = 0, 0+1 = 1, 1+0 = 1, and 1+1 = 2, but remember that we are working in binary, so 2 is actually represented by 10. When we're doing addition, this will be "0, and carry the 1".

So let us do the given addition problem: 10011011+11110001. We set it up exactly as with the decimal problem:

	1	0	0	1	1	0	1	1

+	1	1	1	1	0	0	0	1
=								

Adding the right-most digits is 1+1 = 10, i.e., "0 and carry the 1":

	1	0	0	1	1	0	1^{+1}	1
+	1	1	1	1	0	0	0	1
=								1

Then adding the next digits gives 1+1+0 = 10 = "0, carry the 1":

	1	0	0	1	1	0^{+1}	1^{+1}	1
+	1	1	1	1	0	0	0	1
=							0	1

Adding the next digits (with the carry) is 0+1+0 = 1 with no carry:

	1	0	0	1	1	0^{+1}	1^{+1}	1
+	1	1	1	1	0	0	0	1
=						1	0	1

We proceed like this all the way down and get:

	1^{+1}	0^{+1}	0^{+1}	1	1	0^{+1}	1^{+1}	1
+	1	1	1	1	0	0	0	1
=	11	0	0	0	1	1	0	1

For a final answer of 110001101.

If this procedure were happening inside a computer, i.e., with bytes, then this answer wouldn't actually fit in a memory space! What the computer would do in this case is not store the highest bit in the memory, so the answer would be just 10001101, and then elsewhere it would record the most significant bit of the answer.

This is a fundamental fact about using computers--memories always have fixed width. If the memories are bytes, then they can only store numbers 0-255. In this case, we are adding the numbers 155+241, whose answer is much larger than 255 so it doesn't fit in a byte. So the computer would store the least significant 8 bits of the answer: 10001101, or 141, and would store the most significant bit, which didn't fit, in a special location (the "status register", as we shall learn later).

Adding integers:

Add the 8-bit two's complement binary representations 00101011+11110001

We said that the addition procedures for two's complement and for unsigned binary representations are actually exactly the same, so we run through exactly the same procedure as above:

	0^{+1}	0^{+1}	1	0	1	0^{+1}	1^{+1}	1
+	1	1	1	1	0	0	0	1
=	10	0	0	1	1	1	0	0

For a final answer of 100011100.

To consider whether this makes sense, let us think about these binary representations in decimal: 00101011 is 43, and 11110001 is -15. The answer is 100011100, but remember that the only thing that gets stored is the last 8 bits, or 00011100, which represent the number 28. So in fact this is the correct answer for the problem 43+(-15).

Negating integers:

Compute the negative of the 8-bit two's complement representation 00111100

We said that the procedure for negating a two's complement representation was to first flip all the 0s to 1s and 1s to 0s, and then to add 1 to the result. When we flip all the bits, we get 11000011. Then adding 1 to this is

	1	1	0	0	0	0^{+1}	1^{+1}	1
+								1
=	1	1	0	0	0	1	0	0

For a final answer of 11000100.

As a sanity check, 00111100 represents the number 60, and 11000100 represents (also as an 8-bit two's complement representation) -60, exactly as we'd have hoped.

Subtracting integers:

Subtract the two binary representations 10011110 - 00000011

We know how to add signed representations and we know how to negate, so all we should have to do is negate 00000011 and then add that to 10011110. So off we go:

We first flip all the bits: 11111100, and then we add 1:

	1	1	1	1	1	1	0	0
+								1
=	1	1	1	1	1	1	0	1

Which conveniently happens to involve no carries for this particular example.

Next we simply perform the usual addition procedure on 10011110 + 11111101:

	1^{+1}	0^{+1}	0^{+1}	1^{+1}	1^{+1}	1	1	0
+	1	1	1	1	1	1	0	1
=	11	0	0	1	1	0	1	1

So the answer should be the least significant 8 bits of this: 10011011. As a sanity check, the original problem was 158 - 3 and the answer we've come out with is indeed 155. Huzzah!

Concatenating integers:

Compute 34:209

First, we write down the binary representations of these numbers: 34 is represented as 00100010, and 209 as 11010001. The concatenation of these is then formed by just placing them one after the other to form a 16-bit number: 0010001011010001. This represents the number 8913.

As advertised, this is computed successfully by the formula 34*256 + 209, which is indeed 8913.

4.9: Boolean operations

Summary: We sometimes want to think of individual bits as true/false indicators rather than as digits of a larger binary number. When we do this, we can perform logical operations like "and" and "or" on individual bits. These are called "boolean operations".

Definitions: boolean logic, truth table

There is a further way to think about sequences of bits, namely by thinking about each individual bit separately, rather than as some part of a larger number. An individual bit on its own can only either be 0 or 1, of course, so we can and often will think of it as representing a truth value--0 being "false" and 1 being "true". Then a sequence of 8 bits can simply describe the truth of 8 separate assertions; for example 11000100 means that assertions 1, 2, and 6 are true and the rest are false.

This perspective is called **boolean logic**, and it has its own set of operations that we will now discuss.

AND: For instance, if I have information about the truth of two statements--statement A and statement B, I can inquire about the combined statement "A is true and B is also true". If I have two bits, one for each of my statements, then I can perform the AND operation to get the truth value of the combined statement. This operation is generally denoted by the symbol "&", and for two bits x and y, the value of x & y is "true" (i.e., 1) if both x and y are true (i.e., 1), and "false" (0) otherwise.

It is common to see such operations described with a table like so, called a **truth table**:

x	y	x & y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

So, for example, 0 & 1 can be read off from the first column and second row of this table as being 0.

There are four boolean logic operations that we wish to describe here (though there are of course as many operations as there are truth tables you can write down): "and", "or", "xor", and "not". We just discussed "and"

OR: This operation is written with a vertical "pipe" character: |, and corresponds to asking whether it's true that either statement A is true or statement B is true. This combined statement is of course true provided at least one of statement A or statement B is true. This manifests as saying that x | y is 1 if either x is 1 or y is 1, and 0 otherwise. This gives the truth table:

x	y	x | y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

XOR: Closely related to OR, the XOR or "exclusive-or" operation is written with a "caret" character: ^ (not to be confuse with the same character's occasional use to denote exponentiation), and corresponds to asking whether it's true that either statement A is true or statement B is true, *but not both*. This combined statement is true provided exactly one of statement A or statement B is true. This manifests as saying that x ^ y is 1 if x and y have opposite values, and 0 otherwise. This gives

the truth table:

x	y	x ^ y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

NOT: Unlike the previous operations, which operated on two bits, the not operation operates on a single bit and simply flips its value to be the opposite of what it was before. NOT of a bit called x is written ~x, and can be described with a very simple "truth table" thus:

x	~x
0	**1**
1	**0**

Computers rarely work on single bits at a time, however. So in practice, we can extend these operations to acting on whole bytes at a time instead of individual bits by saying that to take two bytes x and y and compute x & y, we simply compute the "and" of the first bits and make that the first bit of the answer. Then compute the AND of the second bits and make that the second bit of the answer, and so on.

There are a few operations that act on sequences of bits while still treating them as boolean values: The "shift left" (written >> 1) and "shift right" (<< 1) operators simply take all the bits and shift them one slot to the right or left (respectively).

So, for example, 11000110 >> 1 = 01100011 and 11000110 << 1 = 10001100.

4.9.1: Relation between boolean and arithmetic operations

Despite being conceptually very different things, there are some relations between boolean operations and arithmetic ones. For example, an unsigned integer represented in binary is even only if its last digit is 0, and odd otherwise. Thus to test whether a number is even or odd, we don't have to do the rather complicated "%" operation we introduced in chapter 2, but may simply "and" it with 00000001. If the result is 1, then the last digit was a 1 and the number was therefore odd, and if the result is 0, then the original number was even.

There are a couple of tricks like this that are sometimes used to optimize code. We leave these here for you to think about and test your understanding on:

- "shift left" on an unsigned binary number will multiply that number by 2.
- For any sequence of bits x, x ^ x = 0 always.
- To negate an integer called x represented with a sign bit, we can do x = x ^ 10000000.

4.9.2: Worked examples

AND:

Compute 01110101 & 11000110.

The answer is 01000100, computed as follows:

	0	1	1	1	0	1	0	1
AND	1	1	0	0	0	1	1	0
=	0	1	0	0	0	1	0	0

OR:

Compute 01110101 | 11000110.

The answer is 11110111, computed as follows:

	0	1	1	1	0	1	0	1
OR	1	1	0	0	0	1	1	0
=	1	1	1	1	0	1	1	1

XOR:

Compute 01110101 | 11000110.

The answer is 10110011, computed as follows:

	0	1	1	1	0	1	0	1
XOR	1	1	0	0	0	1	1	0
=	1	0	1	1	0	0	1	1

NOT:

Compute ~01110101.

The answer is 1001010, computed by replacing every 0 with a 1 and every 1 with a 0.

4.10: Exercises

4.1: Convert the following numbers to binary(unsigned) and then hexadecimal:

1. 12
2. 35
3. 67
4. 5
5. 129

4.2: Convert the following unsigned binary numbers to decimal:

1. 11111
2. 11011
3. 01
4. 1111011111
5. 10101010
6. 001101

4.3: Convert these negative decimal numbers to binary(two's complement):

1. -21
2. -12
3. -1
4. -9
5. -45

4.4: In this section we performed the operation y = x + 5 using unary representation (BBs). Intuitively, you may have assumed that we could create the "y" container with 5 BBs and then simply dump the contents of the "x" container into it as well. However, instead, we went through all the trouble of weighing out the same number of BBs. Why is all this necessary? What is wrong with dumping x's contents into y?

4.5: As you have seen in the last few chapters, there are many ways to interpret the same sequence of zeros and ones. For example, the byte 11010111 has different values when represented as unsigned vs. signed. Fill in the representations for this byte 11010111

1. unsigned
2. signed
3. sign bit
4. two's complement

4.6: What is the role of the MSBs in both of the signed representations from this chapter?

4.7: Convert the following hexadecimal number to its floating point value: 7D00 (Note: there is no direct way to do this; you must first translate the hexadecimal number into an intermediate binary number in order to do the floating point conversion.)

4.8: What is the difference between a signed bit representation and a two's complement representation in binary?

4.9: What are advantages of floating point representation over fixed point representation?

4.10: How are binary numbers translated into hexadecimal numbers?

4.11: What are the hexadecimal numbers and what decimal numbers do they correspond to?

4.12: (4.1.1) Why don't we use Unary representation for storing information in computers?

4.13: (4.1.1) What would be the Unary representation of the decimal number 7?

4.14: (4.1.2) Define byte, word, and nibble.

4.15: (4.1.2) What is the MSB in the binary number 1010? What's the LSB?

4.16: (4.1.2) How many non-negative integers can be represented by a nibble? Which is the largest of these integers?

4.17: (4.1.2) Assuming non-negative integer binary representation, which decimal number does 10010 equal to?

4.18: (4.1.3) Why is two's complement preferred over sign-magnitude to denote negative integers?

4.19: (4.1.3) What is the smallest (most negative) integer you can represent by a nibble, when using two's complement representation? What if you use signed-magnitude representation? What about the largest integer in either case? How many integers in total in either case?

4.20: How would you write the following in 2's complement notation: +39, - 39?

4.21: (4.1.4) How does fixed-point representation of real numbers vary from floating-point representation?

4.22: (4.1.4) What decimal number does 101.101 represent? (Note this is a fixed point representation)

4.23: (4.1.4) Assuming Floating point representation mentioned in the book, what decimal number do the following represent : 1,11000,0010000000 and 1,00110,0110000000 (The commas have been added just to make reading the various components of the representation easier)

4.24: Assuming two's complement notation, perform the following:

1. 01000011 + 00100011
2. 11000011 + 00100011
3. 11000011 + 10100011
4. 01100011 + 01100011
5. 01000011 - 00100011
6. 11000011 - 00100011
7. 11000011 - 10100011
8. 00100011 - 11000011

Which of the above would cause an overflow, assuming the memory width is 8 bits.

4.25: Convert the following numbers and perform the functions

1. 3 + 4
2. 5 - 4
3. 9 - 2
4. 13 - 5
5. 4 - 14
6. 4 + 9

4.26: The number 183 can be interpreted as a decimal or hexadecimal number, but not binary. Why is this? Convert the decimal number 183 to hexadecimal, and then convert hexadecimal number 183 to decimal.

4.27: What is the sum 11010011 + 01001010 in unsigned binary? In two's complement binary?

4.28: How are two's complement numbers negated? Which arithmetic operations are easily allowed with these numbers?

4.29: Construct a truth table for a NAND gate

4.30: Construct a truth table for a NOR gate

4.31: This chapter introduces some ways to utilize the binary operations we learned to optimize code. The first trick is to left shift instead of multiplying by 2. This trick is related to the fact that every digit in the binary number represents a different power of 2, and you could make a similar conclusion about left-shifting decimal numbers. Write the binary and decimal values of each number below to see how binary and decimal are related to this left-shift trick:

00000001

00000010

00000100

00001000

00010000

00100000

01000000

10000000

4.32: Now it should be obvious why/how the left-shift trick works. What effect does left shifting have on decimal numbers? How does this relate to the place values of each representation?

4.33: Another trick from this section is that x^x = 0 always, for any sequence x. Using the XOR truth table from the previous section, perform this trick for the following sequences and explain why this is always true.

1. 0
2. 1
3. 0000
4. 1111
5. 01010101

4.34: The last trick from this section is that you can negate an integer called x represented with a sign bit, by changing its value to x ^ 10000000. How does this operation work when x's

sign bit is 0 as opposed to 1? To answer this, compute x ^ 10000000 for the following x values (and convert all signed binary values to decimal when you're done):

1. x = 10101010
2. x = 01010101

4.35: What is 11010011 & 01001010, using the boolean operator "and"?

4.36: What is 01001101 | 01100101, using the boolean operator "or"?

4.37: What is 10100111 ^ 01111001, using the boolean operator "xor"?

4.38: What is ~10011110, using the boolean operator "not"?

4.39: Write the truth tables for AND, OR, and XOR gates (two inputs), and the NOT gate (1 input).

4.40: Which gate gives an output one only when exactly one of the two inputs is 1?

4.41: What would the following result in? (When x appears, assume it is an 8-bit binary number)

1. 10010001>>2
2. 10010001 | 01011001
3. 10010001 & 01011001
4. 10010001 ^ 01011001
5. ~10010001
6. x & x
7. x | x
8. x & ~x
9. x | ~x
10. x ^ x
11. x ^ ~x