# Computer representation of numbers

Computers store numbers in a variety of ways, but all use base-2 arithmetic, rather than our base 10. Almost all computers bits in multiples of 8 to store numbers; one units of 8 bits is called a **byte**. There are two main ways to represent numbers: integer and floating point.

## Integer representation

Suppose we consider positive integers only. What range of values can we represent with a single byte?

| | |
|---|---|
| 0000 0000 | equals 0 |
| 0000 0001 | 1 |
| 0000 0010 | 2 |
| 0000 0011 | 3 |
| ... | |
| 1111 1110 | 254 |
| 1111 1111 | 255 |

The range is 0 to 255. In general, the range of unsigned integers stored in N bits is

$$0 \quad \text{to} \quad 2^N - 1$$

What happens if you try to add numbers to the highest possible value?

1111 1111 = 255 + 1 = 0000 0000 = 0
1111 1111 = 255 + 2 = 0000 0001 = 1

Whoops. The value roll over and start again from zero. Likewise, if you subtract from a small number, you find rollunder:

0000 0000 = 0 - 1 = 1111 1111 = 255
0000 0000 = 0 - 2 = 1111 1110 = 254

So, if you use integer storage, you must watch out for rollover/under.

*Signed integers*: Suppose you want to keep track of positive and negative values. You must devote one of the bits to the sign, using, for example, 0 to mean "positive" and 1 to mean "negative". That would leave only 7 of the 8 bits for numerical values. If the highest-order (leftmost) bit is used for the sign, then we have

```
    0000 0000 =   0

    0000 0001 =  +1
    1000 0001 =  -1

    0000 0010 =  +2
    1000 0010 =  -2
```

The range of values has changed from

```
    0      to     255       unsigned
  -127     to    +127       signed
```

Yes, this simple scheme wastes two representations for zero (since +0 and -0 are the same). Real computer schemes use a slightly different arrangement which doesn't waste two values on zero, but instead gives a slightly larger range of -128 to +127.

*Challenge:*

You are in charge of the US Census for 2000. You must design computers to keep track of people. How many bytes should you devote to the population count?

---

**Floating point representations**

Now, what happens if we want to use a computer to simulate a real, physical situation?

*Example 1:* We drop a ball from a height of 1 meter above the ground, and want to calculate its height and velocity as a function of time. Using the equations of 1-D kinematics, we can figure out that the ball will take about 0.452 seconds to reach the ground, and will be travelling at a speed of about 4.4 meters per second when it gets there.

If we use integers, and units of meters and seconds, then we have a big problem: all the action takes place between t = 0 and t = 1 second. Ooops. And the height of the ball can be stated only as h = 1 m and h = 0 m. That's a pretty poor way to simulate a real ball.

There is one thing we can do: rescale the units so that they are better matched to the problem. For example, we could change

| | | | |
|---|---|---|---|
| time | from seconds | to milliseconds | 0 msec --> 452 msec |
| height | from meters | to millimeters | 0 mm --> 1000 mm |

With these units, we can do a decent job of simulating a smooth drop, even with integers.

*Example 2:* We want to calculate the motion of the Earth and the Moon, due to the gravitational forces they exert on each other. We know that the equation for gravitational force is

$$F = G * \frac{M * m}{R^2}$$

where

G = 6.67 x 10^(-11) N*m^2/kg*^2
M = 5.98 x 10^(24) kg
m = 7 x 10^(22) kg
R = 3.82 x 10^8 m

Yikes! It's not so easy to rescale all the units by the same factor, since they are a mix of VERY big and VERY small.

It's almost always better to give up on integers, and move to a **floating point** representation of numbers. You are familiar with it already: scientific notation is an example:

$$\text{Mass} = 5.98 \times 10^{24} \text{ kg}$$

We can divide this value into three parts: The first part, 5.98, is called the **mantissa.** The second part, 10, is the **base** (we always use base 10 for scientific notation). The last part, 24, is the **exponent.**

What's the advantage of scientific notation? Well, we can mix very small and very large values together easily: we combine the mantissa portions using one sort of arithmetic rule, and the exponent portions using another sort of rule. And this is much more compact than integer representation:

5.98 x 10^(24) vs. 5,980,000,000,000,000,000,000,000

Although we humans usually write numbers as some value between 1 and 10, times ten to some exponent, it turns out that computers usually **normalize** the numbers so that the mantissa lies between 0 and 1:

```
    we write           5.98  x 10^(24)
    computer uses      0.598 x 10^(25)
```

It's no big deal.

There is a drawback to this floating-point representation, though. If we only use a certain number of digits to hold the mantissa -- say, 3 digits, as shown above -- then we end up with a certain amount of **quantization**: we can't represent all the numbers within some range, only a selected few. For example, we can represent these numbers:

```
       5,980    =   5.98 x 10^3
       5,990    =   5.99 x 10^3
```

but not these:

```
       5,981    =   5.981   x 10^3       too many digits in mantissa
       5,985.43 =   5.98543 x 10^3       way too many digits in mantissa
```

---

*Floating point in base 2:*

Humans use base 10 for calculations. But computers use base 2. Let's see how that works out.

The normalized mantissa in a base-2 representation is interpreted like this:

| mantissa bits | first | second | third | fourth | etc. |
|---|---|---|---|---|---|
| hold place for | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | |

So, for example, a 4-bit mantissa of 1011 would mean

$$1011 = 1*2^{-1} + 0*2^{-2} + 1*2^{-3} + 1*2^{-4}$$

$$= 0.5 + 0 + 0.125 + 0.0625$$

= 0.6875

Note the smallest possible value for a normalized mantissa is a one followed by zeroes:

1000 = 0.5

It's kind of wierd to have the smallest possible value be 0.5 (instead of 0.1), but that's what happens when one normalizes in base 2.

Suppose we want to represent the number 4049, and we decide to use

8 bits for the normalized mantissa
4 bits for the exponent

We can write this number in normalized scientific notation:

| | | | | |
|---|---|---|---|---|
| base 10: | 0.4049 | x | 10^(4) | exactly |
| base 2: | 0.98852539 | x | 2^(12) | approximately |
| base 2: | 0.98828125 | x | 2^(12) | to a rougher approximation |

Since we only have 8 bits for the normalized mantissa, we have to cut off the number of digits somewhere. I've chosen the approximation 0.98828125, because we can represent that in 8 bits, as follows:

0.98828125 = 0.5 + 0.25 + 0.125 + 0.0625 + 0.03125 + 0.015625 + 0.00390625

$$= 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + \qquad 2^{-8}$$

$$= 1*2^{-1} + 1*2^{-2} + 1*2^{-3} + 1*2^{-4} + 1*2^{-5} + 1*2^{-6} + 0*2^{-7} + 1*2^{-8}$$

= 1 1 1 1 1 1 0 1

= 1111 1101

The exponent part is easy: we need to express an exponent of 12

12 = 8 + 4

= 1100

So, putting it all together, we can express

```
  0.98828125  x 2^(12)   =     1111 1101            1100
                          normalized mantissa      exponent
```

Gee, 12 bits. That's a lot. Does it actually express the value "4049" accurately? Let's see:

```
    0.98828125 x 2^(12)   =  0.98828125 * 4096  =  4048
```

Hey, that's not exactly right. But it's as close as we can get. If we increase the value of the mantissa by the smallest possible amount,

```
        1111 1101          --->        1111 1111
        0.98828125                   0.99609375
```

then we get

```
    0.99609375 x  2^(12)   =  0.99609375 * 4096  =  4080
```

So, using a floating-point representation with 8 bits for mantissa and 4 bits for exponent, we can't actually encode the number "4049" exactly. The closest we can come is either 4048 (which is pretty good) or 4080 (which is pretty bad).

Source: http://spiff.rit.edu/classes/phys317/lectures/compnum.html