5 Number Representations

In this lecture, we will cover two concept that are important to understand how numbers are represented on a computer. First, we will talk about the binary system. Second, we will cover the finite precision and range of floating point numbers.

5.1 Binary system

Information on a computer is stored in two-state devices such as a flip-flop made of two transistors. Each of these flip-flop devices can either be turned on or off. This is the smallest amount of information a computer can store. We call this unit of information a *bit*. It can be either be 0 or 1. As humans, we normally represent numbers by writing their digits in base 10; after all, we have 10 fingers. However, because computers work with transistors that are either on or off, and do not have ten different levels of *on*, it makes sense to describe numbers in base 2 on a computer.

Let's go over a few examples of converting numbers from decimal to binary and back. You don't need to become an expert in this, but you should be familiar with the issues this conversion can create.

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
10	1010
100	1100100
0.5	0.1
0.25	0.01
0.75	0.11
0.1	0.00011001100110011...
0.33333...	0.101010101...

Code 25: Example of binary/decimal conversion.

Let's discuss the last two examples in more detail. Thise illustrate an important fact: some fractions can be represented exactly in base 10 but become an infinite series in base 2. This brings us to the issue of finite precision.

5.2 Bits and bytes

On a computer there is only a certain amount of space available. In fact, the same is true if you write numbers on a piece of paper. We want to make best use of the available space which is why there are different ways of storing a number on a computer (and on paper). We choose the way that makes most sense for the task we are trying to accomplish.

As you can see in the above example, some of the number cannot be written exactly on any finite piece of memory (or paper). One example is the number 1/3. We had to come up with a completely new way of writing a number, namely as a fraction, to be able to write it down. Some numbers are representable exactly in base 10, but not in base 2, one example is the number $1/10 = 0.1$. Note that computers do (in general) not understand what a fraction is. They use a representation called floating point (see below) as the representation for everything. There are some exceptions to this rule. First of all, you could write a program that understands fractions. Second, there are mathematical packages such as Maple and Mathematica that already come with this functionality. We will not cover those in this lecture, but it is good to keep in mind that solutions already exists if you every work on a problem that makes heavy use of fractions and you want to perform a very accurate calculation.

All types of information on a computer such as numbers, texts, sounds and videos are stored as a collection of bits. A bit is a very small amount of information, so often 8 bits are grouped together and called a byte. The following figure shows other commonly used names for a certain number of bits. You've probably heard *bit* and *byte* before, but maybe not *word* or *nibble*.

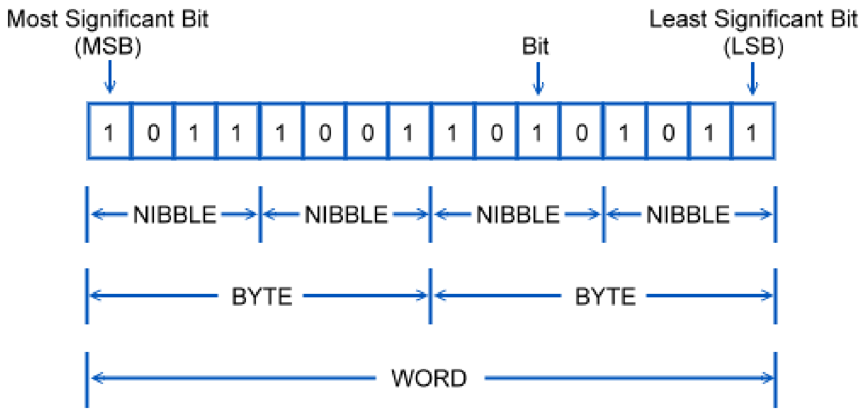

Figure 5: Bits and bytes. Source: `http://www.pcldev.com`.

Most importantly for us, each number on a computer is represented by a finite number of bits (or bytes). This has dramatic consequences for the algorithms we will implement. For that reason, we will discuss in detail how numbers are stored. There are many conceptually different ways to store numbers. We'll talk about the two most commonly used ones: integers and floating point numbers next.

5.3 Integers

Integers are one of the most fundamental datatypes in computer science. Typically an integer consists of at least two bytes on a modern computer. With two bytes (16 bits) we can store numbers from 0 (including) up to

$$2^{16} - 1 = 65535.$$

However, often we also want to be able to store negative integers. We need to reserve one bit for the sign of the integer, thus leaving us 15 bits for the magnitude. The range of a signed two-byte integer is therefore

$$-2^{15} = -32768 \ldots 32767 = 2^{15} - 1.$$

Let's look at what this finite range of possible integers means in practice. If we start with an integer set to 0 and add 1 many times, eventually we will reach the maximum. When this happens, we will get a negative number! In python you can find out what the maximum integer is that it can represent by using the `sys` module. However, python is a high level language and actually pretty smart. It automatically increases the number of bytes for the integer if you try to add 1 to the maximum allowed value for the integer. In that case, the integers becomes what is called a *long integer*, or just *long*. In many other programming languages, this is not done automatically for you and you need to be extremely careful about having integers which are larger then the maximum allowed size.

An example code for python is given in Code 26. Remember, python does increase the number of bytes automatically. The same code in C, would give you a negative number.

```
>>> import sys
>>> sys.maxint
9223372036854775807
>>> sys.maxint + 1
9223372036854775808L
```

Code 26: Interactive python session testing the maximum size of int.

5.4 Floats

Most numbers in scientific calculations will be represented by something called a *float* or a *floating point number* on the computers that we use. The name comes from the fact that the decimal point will shift (float) and is not fixed. You can consider the integer datatype as a fixed precision datatype (i.e. no information about the digits after the decimal point is stored). In python and almost all other programming languages, the industry standard of representing such a number is IEEE-754. There is a single and double precision version of this standard. We use the *double precision* version. It uses 8 bytes, i.e. 64 bits. On some special hardware, for example GPUs, a datatype with less precision, *single precision* is used. So how do we represent a number x where the decimal point can shift? We describe the number as a combination of three other numbers:

$$x = \text{sign} \times \text{mantissa} \times 2^{\text{exponent}}$$

The sign is just a single bit, either 0 or 1, describing whether the number is positive or negative. The mantissa (sometimes called fraction) consists of 52 bits. These are the digits after the decimal point. The exponent contains 11 bits, which is just an integer. The following figure illustrates how a double floating point number is stored in memory.

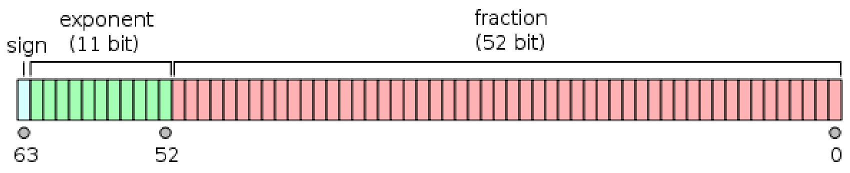

Figure 6: Double floating point number in memory. Source: `http://en.wikipedia.org`.

Putting this all together we can calculate the number x from the sign, mantissa and exponent bits using the following formula:

$$x = (-1)^{\text{sign}} \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i} \right) 2^{e-1023}$$

This formula is crucial to understand how a floating point number is constructed. Floating point numbers have important limitations for both the precision and the range of possible numbers.

Let's look at the precision first. There are only 52 bits available in the mantissa. Thus the binary representation of a number is truncated after 52 binary digits. 52 binary digits correspondsto approximately 16 decimal digits. Any digits after that cannot be represented and are simply ignored. Let's look at 1/10 and 2/10. Both of these numbers cannot represented exactly as a binary fraction (see above). For that reason, the addition of the two numbers will also not exactly give us 3/10 (which also can not be represented exactly in the first place). Have a careful look at the python snippet in Code 27 and make sure you understand why $0.2 + 0.1$ is not the same number as 0.3.

```
>>> from decimal import Decimal
>>> Decimal(0.1)
Decimal('0.10000000000000000555111512312578270211815')
>>> Decimal(0.2)
Decimal('0.20000000000000001110223024625156540423631')
>>> Decimal(0.2+0.1)
Decimal('0.30000000000000004440892098500626161694526')
>>> Decimal(0.3)
Decimal('0.29999999999999998889776975374843459576368')
```

Code 27: Python listing testing the floating point precision. The first line just loads an external module that allows you to print the decimal number very accurately on the screen.

Next, let's look at the finite range. Let's start with the number $x = 1$ and multiply it with the number 2 multiple times. Eventually we will run over the maximum range of what double precision floating point numbers can represent. This happens when we run out of bits in the exponent. Let's see when this occurs in python. A simple test is implemented in Code 29. The code outputs the largest number before we reach infinity. Here, infinite means that the computer has noticed that we ran above the maximum range. It's not actually infinity, just a number larger than what the computer can represent. In python (using the default double precision) we get a value around 10^{307}. Note that python outputs this in *scientific notation* with a syntax involving `..e+..`. Can you explain why we get the number 307 (roughly)?

```
>>> x = 1.
>>> while x<x*2.:
...     x=2.*x
...     print x
...
2.0
4.0
8.0
16.0
[...]
4.49423283716e+307
8.98846567431e+307
inf
```

Code 28: Python listing testing the floating point maximum.

5.5 Compensated summation

In the above experiments, we found a potential problem with floating point numbers: the limited precision. Imagine you are working in single precision and want to send a spacecraft to Pluto. Pluto is about $5 \cdot 10^{12}$ m away from Earth. The radius of Pluto is $1.1 \cdot 10^{6}$ m. The ratio of those two number is $2 \cdot 10^{-7}$. The number of bits for the fraction in single floating point precission is 23. Converting this precision to decimal gives $2^{-23} = 10^{-7}$. So using single precision, you can not determine whether your spacecraft will hit Pluto or not. But you can find out if you use double precision.

To solve problems like this, one could simply use a data format with more bits. The buzzword for this is extended (or even arbitrary) precision. The downside to this is that one needs to either build a new computer, which is expensive, or implement it in software, which is slow.

There is another way to solve the problem. This solution only uses floating point numbers, so we don't need to build a new computer and the implementation is very fast. It will be our first complex algorithm that we discuss in this course, so we will go through it very slowly. It is known as compensated summation, or *Kahan summation*.

Let's look at the problem first. Adding two numbers works well only if the numbers are of similar size. Otherwise we get relative large errors.

```
>>> print "%.20f" % (1. + .1)
1.10000000000000008882
>>> print "%.20f" % (1e15 + 0.1)
1000000000000000.12500000000000000000
>>> print "%.20f" % (1e15 + 0.1 - 1e15)
0.12500000000000000000
```

Code 29: Python listing showing the problem using non-compensated summation.

Don't worry about the complicated looking print statement. It's only purpose is to show you the number with 20 decimal digits (instead of 5 which is the default). One can see that a huge relative error arrises when we add large numbers to small numbers. The equation

$$(10^{15} + 0.1) - 10^{15} = 0.1$$

is only correct to within 25%. Suppose you have 10 cents in your bank account. If the bank accidentally deposits 10^{16} dollars into your account, notices the change and then withdraws the money again, you would end up with 12.5 cents. So you would have made a 2.5 cent profit. With compensated summation we can avoid this issue. Here is how the algorithm works.

Let s be the sum of numbers and δ_n a series of number that we want to add to s. We need one additional variable e, also a floating point number. Initially $s = 0$ and we also set $e = 0$. Then, in pseudo code:

```
**for** n=0,1,2,... **do**
    $y = \delta_n - e$
    $t = s + y$
    $e = (t - s) - y$
    $s = t$
**end for**
```

If these were all mathematical statements as you know them, we could simplify all the above to simply get

```
**for** n=0,1,2,... **do**
    $s = s + \delta_n$
**end for**
```

which is of course just the simple summation. Let's look at what happens to the variables in the compensated summation algorithm in detail. We'll use a table and go through every step of the algorithm. We'll add the numbers 100, 0.0005 and 0.0005 and assume we work in a floating point system with 6 decimal places. The right hand side of the table shows the floating point numbers. Note that each floating point number has only a finite precission. This is represented by two gray blocks.

s		.000	000	
e		.000	000	
δ_0	100	.000		
$y = \delta_0 - e$	100	.000		
$t = s + y$	100	.000		
$t - s$	100	.000		
$e = (t - s) - y$		.000	000	
$s = t$	100	.000		

δ_1		.000	500	
$y = \delta_1 - e$		.000	500	
$t = s + y$	100	.000		
$t - s$		.000	000	
$e = (t - s) - y$		-.000	500	000
$s = t$	100	.000		

δ_2		.000	500	
$y = \delta_2 - e$		.001	000	
$t = s + y$	100	.001		
$t - s$		.001	000	
$e = (t - s) - y$		.000	000	
$s = t$	100	.001		

Using only standard summation, the last digits would be lost and the final result would be 100.000. Here, we carry the extra digits along and thus achieve higher precision!

You will have to implement this algorithm in the next assignment. And you will be asked to implement this in the assembler language we introduced in the last lecture. To get you started, we write an algorithm for non-compensated summation.

You may want to use two new assembler commands for this assignment that we have not used before:

Command	Argument 1	Argument 2	Argument 3	Result
`INPUT`	r0			Input value from user and store it in r0
`SUB`	r0	r1	r2	r2 := r0 - r1

With those, a non-compensated summation algorithm in assembler could be written as follows.

```
SET 0 r0
SET −3 r9
INPUT r1
ADD r0 r1 r0
PRINT r0
JUMP r9
```

Code 30: Non-compensated summation in assembler.

To give you some help with the assigment, you may want to use the registers as follows:

```
s = r0
e = r1
d = r2
y = r3
t = r4
t−s = r5
```

End of lecture 3.