Lecture 4: IEEE Floating Point Arithematic

In this lecture, we introduce the IEEE Floating Point number format. Before we begin, we define a function `printbits` that print the bits of floating point numbers in colour, based on whether its the sign bit, exponent bits, or significand bits:

In [1]:

```
printred(x)=print("\x1b[31m"*x*"\x1b[0m")
printgreen(x)=print("\x1b[32m"*x*"\x1b[0m")
printblue(x)=print("\x1b[34m"*x*"\x1b[0m")

function printbits(x::Float16)
       bts=bits(x)
       printred(bts[1:1])
       printgreen(bts[2:7])
       printblue(bts[8:end])
end

function printbits(x::Float32)
   bts=bits(x)
    printred(bts[1:1])
    printgreen(bts[2:2+8-1])
    printblue(bts[2+8:end])
end

function printbits(x::Float64)
   bts=bits(x)
    printred(bts[1:1])
    printgreen(bts[2:2+11-1])
    printblue(bts[2+11:end])
end
```

Out[1]:

```
printbits (generic function with 3 methods)
```

`Float64` is a type representing real numbers using 64 bits, that is also known as double precision. We can create `Float64`s by including a decimal point when writing the number: 1.0 is a `Float64` while 1 is an `Int64/Int32`. We use `printbits` to see what the bits of a `Float64` for a few numbers are.

First, let's check an integer. The format is very different from `Int64/Int32`:

In [2]:

```
printbits(1.0)
```

```
0011111111110000000000000000000000000000000000000000000000000000
```

Even though 1.3 is representable with only two base-10 digits, it requires an infinite number of base-2 digits, which is cut off:

In [3]:

```
printbits(1.3)
```

```
0011111111110100110011001100110011001100110011001100110011001101
```

`Float32` is another type representing real numbers using 32 bits, that is known as single precision. `Float64` is now the default format for scientific computing (on the *Floating Point Unit*, FPU). `Float32` is the default format for graphics (on the *Graphics Processing Unit*, GPU), as the difference between 32 bits and 64 bits is indistinguishable to the eye.

In [5]:

```
printbits(Float32(1.3))
```

```
00111111101001100110011001100110
```

We will now explain the interpretation of this format.

In lectures we worked out the base-2 expansion of 1/3:

$$\frac{1}{3} = (0.0101010101010\ldots)_2$$

This representation is simply code for the infinite sum

$$0 + \frac{0}{2} + \frac{1}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{0}{2^5} + \frac{1}{2^6} + \cdots$$

We can check this on a computer, however, we are only allowed to do a finite number of computations in practice:

In [105]:

```
1/2^2+1/2^4+1/2^6+1/2^8+1/2^10  # approximates 1/3
```

Out[105]:

```
0.3330078125
```

Floats are stored in the format

$$x = \pm 2^{q-S} \times (1.b_1 b_2 b_3 \ldots b_P)_2$$

where S and P are fixed constants that depend on the type, q is an unsigned integer of a fixed number bits, and $b_1 b_2 \ldots b_P$ are binary digits, stored as P bits.

In the case of `Float64`, $S = 1023$, $P = 52$, and q is stored with 11 bits.

Let's do an example:

In [23]:

```
printbits(100+1/3)
```

```
0100000001011001000101010101010101010101010101010101010101010101
```

The red bit tells us that the number is positive. The green bits tell us q:

In [106]:

```
q=parse(Int,"10000000101",2)
```

Out[106]:

```
1029
```

This tells us that the exponent is $1029 - 1023 = 6$, which we can check using the `exponent` command:

In [107]:

```
exponent(100+1/3)
```

Out[107]:

```
6
```

The remaining blue bits tell us the significand, therefore

$$100 + 1/3 = 2^6 * (1 + \frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^8} + \frac{1}{2^{10}} + \frac{1}{2^{12}} + \cdots)$$

Let's check if that works out:

In [112]:

```
2^6*(1+1/2+1/2^4+1/2^8+1/2^10+1/2^12+1/2^14+1/2^16)
```

Out[112]:

```
100.3330078125
```

Subnormal numbers

Whenever $q = 0$, this is called a subnormal number, so does not follow the same interpretation of the bits. Instead, if $q = 0$ the number is represented as

$$x = \pm 2^{1-S} * (0.b_0 b_1 b_2 \ldots b_P)_2$$

The simplest example is 0.0, which has $q = 0$ and all bits $b_k = 0$:

In [113]:

```
bits(0.0)
```

Out[113]:

```
"0000000000000000000000000000000000000000000000000000000000000000"
```

The smallest normal number is $q = 0$ and b_k all zero. For a given floating point type, it can be found using `realmin`:

In [114]:

```
mn=realmin(Float64)
```

Out[114]:

```
2.2250738585072014e-308
```

In [115]:

```
2.0^(1-1023)
```

Out[115]:

```
2.2250738585072014e-308
```

In [34]:

```
printbits(mn)
```

```
0000000000010000000000000000000000000000000000000000000000000000
```

If we divide by two, we get a subnormal number:

In [35]:

```
printbits(mn/2)
```

```
0000000000001000000000000000000000000000000000000000000000000000
```

In [36]:

```
printbits(mn/4)
```

```
0000000000001000000000000000000000000000000000000000000000000000
```

We have both 0.0 and -0.0:

In [44]:

```
printbits(0.0)
```

```
0000000000000000000000000000000000000000000000000000000000000000
```

In [43]:

```
printbits(-0.0)
```

```
1000000000000000000000000000000000000000000000000000000000000000
```

Special numbers

Whenever the bits of q are all 1, that is, for `Float64` $q = 2^{11} - 1 = 2047 = (11111111111)_2$, the number is treated differently. If all $b_k = 0$, then the number is interpreted as either $\pm\infty$, called `Inf`:

In [119]:

```
1.0/0.0
```

Out[119]:

```
Inf
```

In [120]:

```
printbits(Inf)
```

```
0111111111110000000000000000000000000000000000000000000000000000
```

Another special type is `NaN`, which represents *not a number*. For example, `0/0` is not defined, so returns `NaN`:

In [122]:

```
0/0
```

Out[122]:

```
NaN
```

`NaN` is stored with $q = (11111111111)_2$ and at least one of the $b_k = 1$:

In [123]:

```
printbits(NaN)
```

```
0111111111111000000000000000000000000000000000000000000000000000
```

What happens if we change some other b_k to be nonzero?

In [126]:

```
i=parse(UInt64,
"1111111111110000000000000000000010000001000000000010000000000000",2)
reinterpret(Float64,i)
```

Out[126]:

```
NaN
```

Thus, there are more than one `NaN`s on a computer. How many are there?

Arithmetic works differently on `Inf` and `NaN`:

In [127]:

```
Inf*0      # NaN
Inf+5      # Inf
(-1)*Inf   # -Inf
1/Inf      # 0
1/(-Inf)   # -0
Inf-Inf    # NaN


Inf==Inf   # true
Inf==-Inf  # false
```

Out[127]:

```
false
```

In [80]:

```
NaN*0      # NaN
NaN+5
1/NaN

NaN==NaN    # false
NaN!=NaN    #true
```

Out[80]:

```
true
```

Let's figure out the format for Float32. We can use the fact that realmin(Float64) has $q = 1$ to determine what S should be:

In [128]:

```
S_64=1-exponent(realmin(Float64))
```

Out[128]:

```
1023
```

In [129]:

```
S_32=1-exponent(realmin(Float32))
```

Out[129]:

```
127
```

In [86]:

```
printbits(Float32(1.0))
```

```
00111111100000000000000000000000
```

So for Float32, we have S = 127, P = 23, and q uses 8 bits.

Rounding

There are three rounding strategies: round up/down/ towards zero/ nearest integer.

The default is round to nearest.

Let's try taking a `Float64`, and round it to a `Float32`.

In [90]:

```
x=1.3
printbits(1.3)  # 64 bits
```

```
0011111111110100110011001100110011001100110011001100110011001101
```

In [91]:

```
printbits(Float32(1.3))  # 32 bits
```

```
00111111101001100110011001100110
```

Let's compare the difference in the significands. We can get the bits of the significand as follows:

In [130]:

```
x=1.3
str=bits(1.3)
bts64=str[13:end]  # lets get the bits for the significand.  This uses the
                   # `end` keyword for getting all the characters of a strin
g
                   # up to the last one
```

Out[130]:

```
"0100110011001100110011001100110011001100110011001101"
```

In [131]:

```
x=1.3
str=bits(Float32(1.3))
bts32=str[10:end]  # lets get the bits for the significand.  This uses the
                   # `end` keyword for getting all the characters of a strin
g
                   # up to the last one
```

Out[131]:

```
"01001100110011001100110"
```

In [99]:

```
bts64
```

Out[99]:

```
"0100110011001100110011001100110011001100110011001101"
```

In [100]:

```
bts32
```

Out[100]:

```
"01001100110011001100110"
```

We see from the fact that the last digit is zero that rounding strategy is either round down, round towards zero, or, round to nearest.