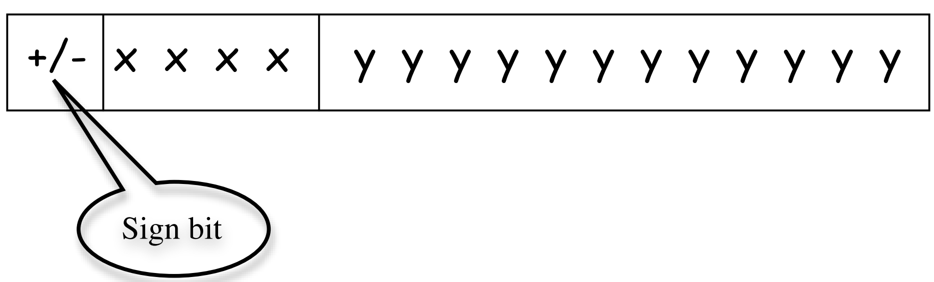
Floating point Representation of Numbers

FP is useful for representing a number in a wide range: **very small** to **very large**. It is widely used in the scientific world. Consider, the following FP representation of a number

Exponent E   significand F (also called ***mantissa***)



In **decimal** it means (+/-) **1**. $yyyyyyyyyyyy \times 10^{xxxx}$

In **binary**, it means (+/-) **1**. $yyyyyyyyyyyy \times 2^{xxxx}$

(The 1 is implied)

IEEE 754 single-precision (32 bits)

s	xxxxxxxx	yyyyyyyyyyyyyyyyyyyyyyy
1	8	23 bits

Single precision

Largest = 1. 1 1 1 … $\times\ 2^{+127} \approx 2 \times 10^{+38}$

Smallest = 1.000 … $\times\ 2^{-128} \approx 1 \times 10^{-38}$

These can be positive and negative, depending on s.

(But there are exceptions too)

IEEE 754 double precision (64 bits)

S	exponent	significand
1	11 bits	52 bits

Largest = 1. 1 1 1… $\times\ 2^{+1023}$

Smallest = 1.000… $\text{X}\ 2^{-1024}$

Overflow and **underflow** in FP

An **overflow** occurs when the number if too large to fit in the frame. An **underflow** occurs when the number is too small to fit in the given frame.

How do we represent zero?

IEEE standards committee solved this by making **zero** a special case: if every bit is zero (the sign bit being irrelevant), then the number is considered zero.

Then how do we represent 1.0?

It should have been 1.0×2^0 (same as 0)! The way out of this is that the interpretation of the exponent bits is not straightforward. The exponent of a single-precision float is **"shift-127" encoded** (**biased representation**), meaning that the actual exponent is (xxxxxxx minus 127). So thankfully, we can get an exponent of zero by storing 127.

Exponent = 11111111 (i.e. 255) means 255-127 = 128

Exponent = 01111111 (i.e. 127) means 127-127 = 0

Exponent = 00000001 (i.e. 1) means 1-127 = -126

The consequence of shift-127

Exponent = 00000000 (reserved for 0) can no more be used to represent the smallest number. We forego something at the low extreme of the spectrum of representable magnitudes, which should be 2^{-127}

It seems wise, to give up the smallest exponent instead of giving up the ability to represent 1 or zero!

More special cases

Zero is not the only "special case" float. There are also representations for positive and negative infinity, and for a not-a-number (NaN) value, for results that do not make sense (for example, non-real numbers, or the result of an operation like infinity times zero). How do these work? A number is infinite if every bit of the exponent is 1 (yes, we lose another one), and is NaN if every bit of the exponent is 1 plus any mantissa bits are 1. The sign bit still distinguishes +/-inf and +/-NaN. Here are a few sample floating point representations:

Exponent	**Mantissa**	**Object**
0	0	Zero
0	Nonzero	Denormalized number*
1-254	Anything	+/- FP number
255	0	+ / - infinity
255	Nonzero	NaN

* Any non-zero number that is smaller than the smallest normal number is a denormalized number. The production of a denormal is sometimes called gradual underflow because it allows a calculation to lose precision slowly when the result is small.

Floating Point Addition

Example using decimal

A = 9.999×10^{1}, B = 1.610×10^{-1}, A+B =?

Step 1. Align the smaller exponent with the larger one.

B = $0.0161 \times 10^{1} = 0.016 \times 10^{1}$ (round off)

Step 2. Add significands

$9.999 + 0.016 = 10.015$, so A+B = 10.015×10^{1}

Step 3. Normalize

A+B = 1.0015×10^{2}

Step 4. Round off

A+B = 1.002×10^{2}

Now, try to add 0.5 and –0.4375 in binary.

Floating Point Multiplication

Example using decimal

$A = 1.110 \times 10^{10}$, $B = 9.200 \times 10^{-5}$ $A \times B = ?$

Step 1. Exponent of $A \times B = 10 + (-5) = 5$

Step 2. Multiply significands

$1.110 \times 9.200 = 10.212000$

Step 3. Normalize the product

$10.212 \times 10^5 = 1.0212 \times 10^6$

Step 4. Round off

$A \times B = 1.021 \times 10^6$

Step 5. Decide the sign of $A \times B$ $(+ \times + = +)$

So, $A \times B = + 1.021 \times 10^6$