Example 6. (review) Express 0.1 in base 2.

Solution.

- $2\cdot\boxed{0}.1=\boxed{0}.2$
- $2\cdot 0.2=\boxed{0}.4$
- $2\cdot 0.4=\boxed{0}.8$
- $2\cdot 0.8=\boxed{1}.6$
- $2\cdot 0.6=\boxed{1}.2$ and now things repeat...

Hence, $0.1=(0.00011\cdots)_2$ and the final digits 0011 repeat: $0.1=(0.0001100110011\cdots)_2$

Example 7. Express $35/6$ in base 2.

Solution. Note that $35/6=5+5/6$ so that $35/6=(101.\cdots)_2$ with $5/6$ to be accounted for.

- $2\cdot 5/6=\boxed{1}+4/6$
- $2\cdot 4/6=\boxed{1}+2/6$
- $2\cdot 2/6=\boxed{0}+4/6$ and now things repeat...

Hence, $35/6=(101.110\cdots)_2$ and the final two digits 10 repeat: $35/6=(101.110101010\cdots)_2$

Floating-point numbers (and IEEE 754)

Possibilities for binary representations of real numbers:

- fixed-point number: $\pm x.y$ with x and y of a certain number of bits

$\pm x$ is called the integer part, and y the fractional part.

- floating-point number: $\pm 1.x\cdot 2^y$ with x and y of a certain number of bits

$\pm 1.x$ is called the significand (or mantissa), and y the exponent.

In other words, the floating-point representation is "scientific notation in base 2".

Important comment. In order to represent as many numbers as possible using a fixed number of bits, it is crucial that we avoid unnecessarily having different representations for the same number. That is why the exponent y above is chosen so that the significand starts with 1 followed by the "decimal" point. This has the added benefit of not needing to actually store that 1 (rather it is "implied" or "hidden").

IEEE 754 is the most widely used standard for floating-point arithmetic and specifies, most importantly, how many bits to use for significand and exponent.

1985: first version of the standard

IEEE: Institute of Electrical and Electronics Engineers

Used by many hardware FPUs (floating point units) which are part of modern CPUs.

For more details: https://en.wikipedia.org/wiki/IEEE_754

Example 8. $4.5=(100.1)_2=(1.001)_2\cdot 2^2 = +\underbrace{1.001}_{\text{binary}}\cdot 2^2$

Next, we will see exactly how IEEE 754 would store this as bits (32 bits in the case of "single precision").

Armin Straub
straub@southalabama.edu

IEEE 754 offers several choices but the two most common are:

- **single precision**: 32 bit (1 bit for sign, 23 bit for significand, 8 bit for exponent)
- **double precision**: 64 bit (1 bit for sign, 52 bit for significand, 11 bit for exponent)

In each case, 1 bit is used for the sign. Also, recall that the significand is preceded by an implied bit equal to 1.

Comment. IEEE 754 also offers half precision as well as higher precisions but single and double are the most commonly used because this is what older and current CPUs use. Moreover, the base (also called radix) can also be 10 instead of 2.

In IEEE 754, a constant, called a **bias**, is added to the exponent so that all exponents are positive (this avoids using a sign bit for the exponent). Namely, one stores $x + \text{bias}$ where $\text{bias} = 2^7 - 1 = 127$ for single precision ($\text{bias} = 2^{10} - 1$ for double precision).

Example 9. Represent 4.5 as a single precision floating-point number according to IEEE 754.

Solution. $4.5 = (100.1)_2 = (1.001)_2 \cdot 2^2 = \boxed{+}\underbrace{1.\boxed{001}}_{\text{binary}} \cdot 2^2$

The exponent 2 gets stored as $2 + 127 = \boxed{1000{,}0001}$.

Overall, 4.5 is stored as $\boxed{0}\boxed{1000{,}0001}\boxed{0010{,}0000{,}0000{,}0000{,}0000{,}000}$.

Example 10. Represent -0.1 as a single precision floating-point number according to IEEE 754.

Solution. In Example 6, we computed that $0.1 = (0.0001100110011\cdots)_2$.

Hence: $-0.1 = \boxed{-}\underbrace{1.\boxed{1001{,}1001\ldots}}_{\text{binary}} \cdot 2^{-4}$

The exponent -4 gets stored as $-4 + 127 = \boxed{0111{,}1011}$.

Overall, -0.1 is stored as $\boxed{1}\boxed{0111{,}1011}\boxed{1001{,}1001{,}1001{,}\ldots}$.

Caution. Note that we are not able to store -0.1 exactly. Therefore we need to be careful about how to choose the final bit to best approximate -0.1. According to IEEE 754, the final bit should be 1 (rather than 0 which we would get if we simply truncated) so that -0.1 gets stored as $\boxed{1}\boxed{0111{,}1011}\boxed{1001{,}1001{,}1001{,}1001{,}1001{,}101}$.

Note that it certainly makes sense to round up the final 0 to 1 because it is followed by $1\ldots$ (this is similar to us rounding up a final 0 in decimal to 1 if it is followed by $5\ldots$).

Using Python as a basic calculator

Example 11. `Python` We can use Python as a basic calculator. Addition, subtraction, multiplication and division work as we would probably expect:

```
>>> 16*3+1

    49

>>> 3/2

    1.5
```

To compute powers like 2^{64}, we need to use `**` (two asterisks).

```
>>> 2**64

    18446744073709551616
```

Division with remainder of, say, 49 by 3 results in $49 = 16 \cdot 3 + 1$. In Python, we can use the operators // and % to compute the result of the division as well as the remainder:

```
>>> 49 // 3

    16

>>> 49 % 3

    1
```

% is called the **modulo** operator. For instance, we say that 49 modulo 3 equals 1 (and this is often written as $49 \equiv 1 \pmod{3}$).

IMPORTANT. The commands here are entered into an interactive Python interpreter (this is indicated by the >>>). When running a Python script, we need to use `print(16*3+1)` or `print(3/2)` to receive the first two outputs above.