Fixed-point numbers

Example 5. (warmup)

(a) Which number is represented by $(11001)_2$?

(b) Which number is represented by $(11.001)_2$?

(c) Express 5.25 in base 2.

(d) Express 2.625 in base 2. [Note that $2.625 = 5.25/2$.]

Solution.

(a) $(11001)_2 = 1 + 8 + 16 = 25$

(b) $(11.001)_2 = 2^1 + 2^0 + 2^{-3} = 3.125$

Alternatively, $(11.001)_2$ should be $(11001)_2 = 25$ divided by 2^3 (because we move the "decimal" point by three places). Indeed, $(11.001)_2 = 25/2^3 = 3.125$.

Comment. The professional term for "decimal" point would be radix point or, in base 2, binary point (but I have heard neither of these used much in my personal experience).

(c) Note that $5.25 = 2^2 + 2^0 + 2^{-2}$. Hence $5.25 = (101.01)_2$.

(d) Since multiplication (respectively, division) by 2 shifts the digits to the left (respectively, right), we deduce from $5.25 = (101.01)_2$ that $2.625 = (10.101)_2$

Example 6. Express 1.3 in base 2.

Solution. Suppose we want to determine 6 binary digits after the "decimal" point. Note that multiplication by $2^6 = 64$ moves these 6 digits before the "decimal" point.

$2^6 \cdot 1.3 = 83.2$ and $83.2 = (1010011.\cdots)_2$ (fill in the details!).

Hence, shifting the "decimal" point, we find $1.3 = (1.010011\cdots)_2$.

Solution. Alternatively, we can compute one digit at a time by multiplying with 2 each time:

- $\boxed{1}.3$ [Hence, the most significant digit is $\boxed{1}$ with 0.3 still to be accounted for.]
- $2 \cdot 0.3 = \boxed{0}.6$ [Hence, the next digit is $\boxed{0}$ with 0.6 still to be accounted for.]
- $2 \cdot 0.6 = \boxed{1}.2$ [Hence, the next digit is $\boxed{1}$ with 0.2 still to be accounted for.]
- $2 \cdot 0.2 = \boxed{0}.4$ [Hence, the next digit is $\boxed{0}$ with 0.4 still to be accounted for.]
- $2 \cdot 0.4 = \boxed{0}.8$ [Hence, the next digit is $\boxed{0}$ with 0.8 still to be accounted for.]
- $2 \cdot 0.8 = \boxed{1}.6$ [Hence, the next digit is $\boxed{1}$ with 0.6 still to be accounted for.]
- And now things repeat because we started with 0.6 before...

Hence, $1.3 = (1.01001\cdots)_2$ and the final digits 1001 will be repeated forever: $1.3 = (1.0100110011001\cdots)_2$

Comment. As we saw here, fractions with a finite decimal expansion (like $13/10 = 1.3$) do not need to have a finite binary expansion (and typically don't).

Example 7. 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 8. **(extra)** 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

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.

Example 9. $4.5 = 1.125 \cdot 2^2 = +\underbrace{1.001}_{\text{binary}} \cdot 2^2$

Next time, we will see exactly how IEEE 754 (single precision) would store this as 32 bits.

Example 10. $\boxed{\texttt{Python}}$ Python automatically uses (double precision) floats when we enter numbers with a decimal point or as the result of divisions.

```
>>> 1.1

    1.1

>>> 1/3

    0.3333333333333333
```

Comment. Note how we can see (roughly) the 52 bit precision of the double precision floats (there are 15 decimal digits after the 0.3, which translates to about $15 \cdot \log_2 10 \approx 50$ binary digits).

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(1.1)` or `print(1/3)` to receive the above output.

When using Replit, you can either run a script (with code written on the left side) or you can enter single lines of code in an interactive Python interpreter (on the right side; this is also where you see the output from running a script).

For very large (or very small) numbers, scientific notation is often used:

```
>>> 2.0 * 10**80

    2e+80

>>> (1/2)**100

    7.888609052210118e-31
```

The following are two things that are (somewhat) special to Python and are often handled differently in other programming languages. First, integers are not limited in size (often, integers are limited to 64 bits, which can cause issues like overflow when one exceeds the 2^{64} possibilities). This is illustrated by the following (this explains why we wrote 2.0 above):

```
>>> 2 * 10**80

    200000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Second, Python likes to throw errors when a computation runs into an issue (there are nice ways to "catch" these errors in a program and to react accordingly, but that is probably beyond what we will use Python for).

```
>>> 1 / 0

    ZeroDivisionError: division by zero
```

Some other programming languages would instead (silently, without error messages) return special floats representing $+\infty$, $-\infty$ or `NaN` (not-a-number).