9 Floating-point numbers

Floating-point numbers come in (at least) two different lengths: single and double precision, 32 and 64 bits. Single-precision (`float`) are used only where memory is scarce and are not used in this module, except to illustrate how floating-point numbers are encoded.

9.1 Fractions in different bases

A (nonnegative) 'radix-point' number in base R (for want of a better term) is a sequence

$$a_{k-1} \ldots a_0 \cdot d_1 d_2 \ldots$$

and its value is

$$\sum_i a_i R^i + \sum_j \frac{d_j}{R^j}$$

its *integer part* plus its *fractional part.*

To calculate the digits in the fractional part, here is a concrete example.

$$\frac{7}{13} = .d_1 d_2 \ldots$$

in decimal.

Multiply by 10:

$$\frac{70}{13} = d_1.d_2 \ldots$$

so d_1 is the integer part of 70/13.

$$d_1 = 5$$

Subtract 5 from each side and you get

$$\frac{5}{13} = .d_2 d_3 \ldots$$

Multiply by 10

$$\frac{50}{13} = d_2.d_3 \ldots$$

so $d_2 = 3$. Subtract from both sides

$$\frac{11}{13} = .d_3 d_4 \ldots$$

Multiply by 10

$$\frac{110}{13} = d_3.d_4 \ldots$$

and $d_3 = 110/13 = 8$. Subtract 8 from both sides.

$$\frac{6}{13} = .d_4 \ldots$$

Multiply by 10

$$\frac{60}{13} = d_4.d_5\ldots$$

so $d_4 = 4$.Subtract 4 from both sides

$$\frac{8}{13} = .d_5d_6\ldots$$

Multiply by 10

$$\frac{80}{13} = d_5.d_6\ldots$$

So $d_5 = 6$. Subtract 6 from both sides

$$\frac{2}{13} = .d_6d_7\ldots$$

Multiply by 10.

$$\frac{20}{13} = d_6.d_7\ldots$$

So $d_6 = 1$. Subtract 1 from both sides.

$$\frac{7}{13} = .d_7\ldots$$

Now we have a recurrence and

$$\frac{7}{13} = .d_1\ldots d_6d_1\ldots d_6\ldots$$

This can be checked.

$$\frac{538461}{10^{-6} + 10^{-12}\ldots} = \frac{538461}{999999}.$$

Multiply the numerator by 13 and the denominator by 7. They both equal 6999993.

The procedure in general is as follows. To convert a number x, where $0 \leq x < 1$, to a radix R radix-point number,

- Let $x_0 = x$.
- Let $x_1 = Rx_0$, $d_1 = \lfloor x_1/R \rfloor$ (integer part).
- Let $x_2 = R(x_1 - d_1)$, $d_2 = \lfloor x_2/R \rfloor$
- Let $x_{i+1} = R(x_i - d_i)$, $d_{i+1} = \lfloor x_{i+1}/R \rfloor$ and so on.

Tabulating the calculation

x_i	70/13	50/13	110/13	60/13	80/13	20/13
d_i	5	3	8	4	6	1

Example Calculate 7/13 as a ‘binary-point’ number. Just tabulating the calculation.

x_i	14/13	2/13	4/13	8/13	16/13	6/13	12/13	24/13	22/13	18/13	10/13	20/13
d_i	1	0	0	0	1	0	0	1	1	1	0	1

Since the next value of x_i would be 14/13, this recurs every 12 bits. Summing a power series, this evaluates to 2205/4095, which multiplied by 13/7 gives 1.

Example Calculate 7/13 as a 'hex-point' number.

x_i	112/13	128/13	176/13	112/13 ...
d_i	8	9	13	8 ...

and

$$7/13 = .89d89d\ldots$$

9.2 Scientific notation

There is a so-called scientific notation for decimal numbers used on calculators. This notation can be used in `printf` with the `%e` format item.

```
%
% cat scien.c
#include <stdio.h>

int main()
{
  double x = 12345.6789, y = 9876.54321;
  printf("%%f and %%e notations\n");
  printf("%f %e\n", x,x);
  printf("%f %e\n", y,y);
}
% gcc scien.c
% a.out
%f and %e notations
12345.678900 1.234568e+04
9876.543210 9.876543e+03
%
```

In

$$1.234568\text{e+}04 \equiv 1.234568 \times 10^4$$

There is a *sign* (omitted; positive), a *mantissa* 1.234568 and *exponent* 4.

The mantissa is ≥ 1 and < 10. All nonzero numbers can be represented this way. Zero is an exception.

Notice that the mantissa is rounded to 6 decimal places.

9.3 Floating-point single precision

Scientific notation will not be used in this module, except as an introduction to the floating-point encoding.

There is some distinction between 'mantissa' and 'significand' but we'll always refer to mantissa.

A single-precision floating-point number occupies 32 bits. Counted from 'high order' to 'low order' they are

- Sign bit, 1 for negative, 0 for nonnegative.
- 8-bit 'biased' exponent b.
- 23-bit mantissa
$$a_1 \dots a_{23}$$
- **Definition** Except when all bits (except the sign bit) are zero, the number represented is
$$\pm 2^{b-127} \times (1.a_1 \dots a_{23})_2$$
- Note that the 'true' mantissa is at least 1 and less than 2.

The exponent is 'biased' rather than 2s-complement. The reason for this is probably:
Fact. $(00000000)_{16}$ encodes zero.

9.4 Example

Calculate the single-precision encoding of $-5/104$.

First, we need to express it in the form

$$\text{(sign)(power of 2)}M$$

where $1 \leq M < 2$. The sign is negative. To get the correct power of 2, keep multiplying (or in other examples dividing) by 2 until you get a number in this range.

$$5/104 \mapsto 5/52 \mapsto 5/26 \mapsto 5/13 \mapsto 10/13 \mapsto 20/13$$

The last, 20/13, is in the correct range. It required multiplication by 32 to get this.

$$-5/104 = (-1)(2^{-5})(20/13)$$

Note that, here at least,

$$1.a_1a_2\dots$$

encodes

$$1 + a_1/2 + a_2/4 + a_3/8 \dots = 20/13$$

We compute the bits a_i by successive doubling and subtracting 1 where ≥ 1. At the first step, 1 is subtracted and we get $7/13 = a_1/2 + a_2/4 \dots$

```
7/13 double for a_1
14/13 a_1 = 1. Sub 1: 1/13
2/13  a_2 = 0
4/13  a_3 = 0
8/13  a_4 = 0
16/13 a_5 = 1.  Sub 1: 3/13
6/13  a_6 = 0
12/13 a_7 = 0
24/13 a_8 = 1  Sub 1: 11/13
22/13 a_9 = 1  sub 1: 9/13
18/13 a_10 = 1 sub 1 5/13
10/13 a_11 = 0
20/13 a_12 = 1 sub 1 7/13
14/13 a_13 = 1

20/13 = 1.1000100111011
          ------------  recurrent
20/13 = 1.100010011101 100010011101 100010011101 etc
```

This can be checked as a geometric series. The 12-bit recurring block gives

$$1 + 2205 \times \frac{1}{4096}(1 + (1/4096) + (1/4096)^2 \ldots) = 1 + \frac{2205}{4095} = 1 + \frac{7}{13}$$

as expected.

Convert the mantissa to 23 bits. Discarding the '1.' part,

$$100010011101100010011101\ldots$$

Now, the last of these 24 bits is a 1-bit, and the rule is to *round*. In terms of binary arithmetic, this amounts to adding 1 to a 23-bit number

```
100010011101 10001001110
+                      1
100010011101 10001001111
```

(This time there was no carrying. There usually is).

Now the sign and mantissa have been computed. The true exponent is -5. We convert it to the biased exponent by adding 127.

$127 - 5 = 122$. Convert 122 to binary. Or, rather more efficiently, convert it to hex. This is ok since the 8 binary digits fit two hex digits exactly.

```
         7       answer (7a)_16 = (0111 1010)_2
   16 ) 122
        112
         10
```

Putting all together

```
1   sign
0111 1010   biased exponent
100010011101 10001001111 rounded mantissa

1 011 1 101 0 100 0100 1110 1 100 0100 1111
A hex answer is required
1 011 1 101 0 100 0100 1110 1 100 0100 1111
b     d     4     4    e    c     4    f
```

One last wrinkle. On Intel processors these are stored 'little endian' by reversing the *bytes* (not the hex digits within the bytes).

$$4fec44bd$$

9.5 Double precision

This format is the same as single precision, with different numbers, of course.

- Sign bit.
- 11-bit biased exponent. The bias is 1023.
- 52-bit rounded mantissa.

The last example is easily adapted to double precision.

For the biased exponent, convert $1023 - 5 = 1018$ to 11-bit binary. This will fit in 3 hex digits.

```
       63 r 10       3 r 15   3fa
16 ) 1018        16)63      011 1111 1010
      96            48
       58           15
       48
       10

1 011 1111 1010
```

53 bits for mantissa.

```
100010011101 100010011101 100010011101 100010011101 1000  1
                                                          + 1
100010011101 100010011101 100010011101 100010011101 1001
Altogether
1 011 1111 1010
b     f    a
1000 1001 1101 1000 1001 1101
8    9    d    8    9    d
```

```
1000 1001 1101 1000 1001 1101 1001
8    9    d    8    9    d    9

Little endian
d9 89 9d d8 89 9d a8 bf
```

9.6 Ints are little-endian too

Actually `short` and `int` variables are stored little-endian as well. This was not mentioned before because it would make hand-calculation exercises more confusing.