DPST1092 24T3 — Floating Point Numbers

https://www.cse.unsw.edu.au/~dp1092/24T3/

Fractions in different Bases

The decimal fraction 0.75 means

- $7*10^{-1} + 5*10^{-2} = 0.7 + 0.05 = 0.75$
- or equivalently $75/10^2 = 75/100 = 0.75$

Similary 0b0.11 means

- $1*2^{-1} + 1*2^{-2} = 0.5 + 0.25 = 0.75$
- or equivalently $3/2^2 = 3/4 = 0.75$

Similarly 0x0.C means

- $12*16^{-1} = 0.75$
- or equivalently $12/16^1 = 3/4 = 0.75$

Note: We call the `.` a radix point rather than a decimal point when we are dealing with other bases.

Fractions in different Bases

If we want to convert 0.75 in decimal to binary, it may be easy to look and and realise we need 0.5 + 0.25 which gives us 0b0.11. Sometimes it is not that easy and we need a systematic approach.

The algorithm to convert a decimal fraction to another base is:

- take the fractional component and multiply by the base
- the whole number becomes the next digit to the right of the radix point in our fraction.
- We now disregard the whole number part of the previous result and repeat this process until the fractional part becomes exhausted or we have sufficient digits (this process is not guaranteed to terminate).

Fractions in different Bases

For example if we want to convert 0.3125 to base 2

- 0.3125 * 2 = **0**.625
- 0.625 * 2 = **1**.25
- 0.25 * 2 = **0**.5
- 0.5 * 2 = **1**.0

Therefore 0.3125 = 0b0.0101

Exercise 2: Fractions: Decimal → Binary

Convert the following decimal values into binary

- 12.625
- 0.1

Floating Point Numbers

Floating point numbers model a (tiny) subset of real numbers

- many real values don't have exact representation (e.g. 1/3)
- numbers close to zero have higher precision (more accurate)

C has two floating point types

- **`float`** ... typically 32-bit quantity (lower precision, narrower range)
- **`double`** ... typically 64-bit quantity (higher precision, wider range)

Literal floating point values: `3.14159`, `1.0/3`, `1.0e-9`

```
printf("%10.4lf", (double)2.718281828459);
displays     2.7183
printf("%20.20lf", (double)4.0/7);
displays 0.57142857142857139685
```

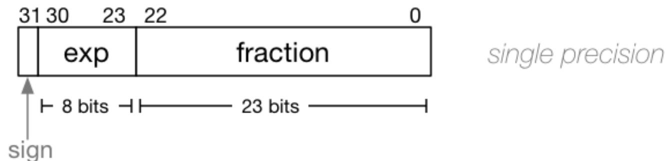
Floating Point Numbers

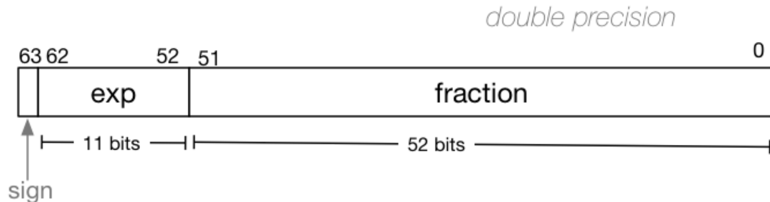
IEEE 754 standard ...

- scientific notation with *fraction* F and *exponent* E
- numbers have form $F \times 2^E$, where both F and E can be -ve
- `INFINITY` = representation for ∞ and -∞ (e.g. 1.0/0)
- NAN = representation for invalid value (e.g. sqrt(-1.0))

Floating Point Numbers

IEEE 754 standard internal structure of floating point values





More complex representation than `int` because *1.dddd e dd*

Floating Point Numbers

Example of normalising the fraction part in binary:

- 1010.1011 is normalized as 1.0101011×2^{011}
- $1010.1011 = 10 + 11/16 = 10.6875$
- $1.0101011 \times 2^{011} = (1 + 43/128) \times 2^3 = 1.3359375 \times 8 = 10.6875$

The normalised fraction part always has `1` before the decimal point.

Example of determining the exponent in binary:

- assume an 8-bit exponent, then bias $B = 2^{8-1}-1 = 127$
- valid bit patterns for exponent `00000001` .. `11111110` (1..254)
- exponent values -126 .. 127

Floating Point Numbers

Example (single-precision):

$150.75 = 10010110.11$

```
// normalise fraction, compute exponent
```

$= 1.001011011 \times 2^7$

```
// sign bit = 0
// exponent = 10000110
// fraction = 00101101100000000000000
= 01000011000101101100000000000000
```

Note: B=127, e=2^7, so exponent = 127+7 = 134 = `10000110`

Floating Point Numbers

Convert the decimal numbers 1 to a floating point number in IEEE 754 single-precision format.

Convert the following floating point numbers to decimal.

Assume that they are in IEEE 754 single-precision format.

```
0 10000000 11000000000000000000000
```

```
1 01111110 10000000000000000000000
```

You can try out more examples with this Floating Point Calculator

Floating Point Numbers

Special cases:

- If every bit (except the sign bit) is 0 then we have the number `0`. This means we can have positive and negative 0.
- If every bit of the exponent is 1 and the fraction is 0 then we have `infinity` (positive or negative)
- If every bit of the exponent is 1 and the fraction is not 0 then we have `NaN` (not a number).
- **Underflow**: If the exponent has minimum value (all zero), special rules for denormalized values are followed. The exponent value is set to 2^{-126} and the "invisible" leading bit for the fraction part is no longer used.