
COMP110: Principles of Computing

# 11: Numerical Representations

# Worksheets

- Worksheet 7: due **this Wednesday**
- Worksheet 8: due **next Wednesday**

# 2’s Complement

# Modular arithmetic


- Arithmetic **modulo** $N$
- Numbers "wrap around" between 0 and $N-1$
- E.g. modulo 16:
  - $14 + 7 = 5$
  - $4 - 7 = 13$

# Modulo operator

- Present in many programming languages (including C++, C#, Python) as `%`
- `a % b` gives the **remainder** of `a` divided by `b`
- E.g. `21 % 16` gives 5
- Useful for wrapping around e.g. loop indexes or screen coordinates

# 2's complement

- How can we represent negative numbers in binary?
- Represent them modulo $2^n$ (for $n$ bits)
- I.e. represent $-a$ as $2^n - a$
- Instead of an $n$-bit number ranging from 0 to $2^n - 1$, it ranges from $-2^{n-1}$ to $+2^{n-1} - 1$
- E.g. 16-bit number ranges from $-32768$ to $+32767$
- Note that the left-most bit can be interpreted as a **sign** bit: 1 if negative, 0 if positive or zero

# Converting to 2's complement

- Convert the absolute value to binary
- Invert all the bits (i.e. change $0 \leftrightarrow 1$)
- Add 1
- (This is equivalent to subtracting the number from $2^n$... why?)
- This is also the process for converting back from 2's complement, i.e. doing it twice should give the original number

# Why 2's complement?

- Allows all addition and subtraction to be carried out modulo $2^n$ without caring whether numbers are positive or negative
- In fact, subtraction can just be done as addition
- I.e. $a - b$ is the same as $a + (-b)$, where $a$ and $-b$ are just $n$-bit numbers

# Scientific notation

# Integer powers

Let $a \neq 0$ be a real number, and let $b > 0$ be an integer

$$a^b = \underbrace{a \times a \times \cdots \times a}_{b \text{ times}}$$

$$a^0 = 1$$

$$a^{-b} = \frac{1}{\underbrace{a \times a \times \cdots \times a}_{b \text{ times}}}$$

# Powers of 10

$$10^6 = 1\underbrace{000\,000}_{\text{6 zeroes}}$$

$$10^1 = 10$$

$$10^0 = 1$$

$$10^{-1} = 0.1$$

$$10^{-6} = 0.\underbrace{000\,00}_{\text{5 zeroes}}1$$

# Multiplying by powers of 10

Multiplying by $10^n$ is the same as moving the decimal point $n$ places to the **right** (adding zeroes if necessary)

$$3.14159 \times 10^2 = 314.159$$

$$27 \times 10^3 = 27\,000$$

Similarly if $n$ is negative, the decimal point moves to the **left**

$$123.45 \times 10^{-2} = 1.2345$$

# Scientific notation

- A way of writing **very large** and **very small** numbers
- $a \times 10^b$, where
  - $a$ ($1 \leq |a| < 10$) is the **mantissa**
  - ($a$ is a positive or negative number with a single non-zero digit before the decimal point)
  - $b$ (an integer) is the **exponent**
- E.g. 1 light year = $9.461 \times 10^{15}$ metres
- E.g. Planck's constant = $6.626 \times 10^{-34}$ joules
- Socrative `FALCOMPED`

# Scientific notation in code

Instead of writing $\boxed{\times 10}$, write `e` (no spaces)

```
double lightYear = 9.461e15;
double plancksConstant = 6.626e-34;
```

# Floating point numbers

# Floating point numbers

- Similar to scientific notation, but **base 2** (binary)
- $a \times 10^b$, where
  - $a$ $(1 \leq |a| < 2)$ is the **mantissa**
  - ($a$ is a positive or negative number with a single 1 before the "decimal point")
  - $b$ (an integer) is the **exponent**

# Storing the mantissa

- A positive or negative number with a single 1 before the “decimal point”
- Sign is stored as a single bit: $0 = +$, $1 = -$
- We know there’s a 1 before the point so no need to store it — just store the binary digits after the point

# Storing the exponent

- An integer — can be positive, negative or zero
- NB we don't use 2's complement to store it!
- Instead it is stored in binary as a positive integer with a **bias** added on
- (This is so that exponents can be efficiently compared (less/greater than) — 2's complement would be less efficient for this)

# Exponent bias

- E.g. if the bias is 127:

| An exponent of... | ... is stored as |
|---:|---|
| -126 | 00000001 (1) |
| ⋮ | ⋮ |
| -1 | 01111110 (126) |
| 0 | 01111111 (127) |
| 1 | 10000000 (128) |
| ⋮ | ⋮ |
| 127 | 11111110 (254) |

- (Exponents of 00000000 and 11111111 have special meaning — more on this later)

# IEEE 754 floating point formats

| Type | Sign | Exponent | Mantissa | Total |
|---|---|---|---|---|
| Single precision | 1 bit | 8 bits<br>bias 127 | 23 bits | 32 bits |
| Double precision | 1 bit | 11 bits<br>bias 1023 | 52 bits | 64 bits |
| Extended precision | 1 bit | 15 bits<br>bias 16383 | 64 bits | 80 bits |

# IEEE 754 floating point formats

- C# (and many other languages) have `float` and `double` types for single and double precision respectively
- Literals are interpreted as `float` if they end in `f`, otherwise `double`
  - `3.14f` is a `float`
  - `3.14` is a `double`
- Python's `float` type is double precision as standard
- Extended precision is not usually used in programs, but is used internally on Intel CPUs

# Example

What is the value stored in the following IEEE single-precision floating point number?

`01000000110100000000000000000000`

# Example

$\boxed{0}$ 1000000110100000000000000000000

- Sign bit is 0
- Therefore the number is positive

# Example

$0\ \boxed{10000001}\ 10100000000000000000000$

- Exponent is 10000001
- This is 129 in binary
- Exponent is stored with a bias of 127, therefore the actual exponent is $129 - 127 = 2$

# Example

0 10000001 $\boxed{\texttt{10100000000000000000000}}$

- Mantissa is $101000\ldots$
- Remember we only store the digits after the "decimal point", so the mantissa is actually $1.101000\ldots$
- The exponent is 2, so we move the point 2 places to the right: $110.1000\ldots$
- $4 + 2 + \frac{1}{2} = 6.5$

# Special floating point numbers

- Zero is represented by a mantissa and exponent of all 0s
- An exponent of all 1s is used to represent infinity or NaN
  - `float.PositiveInfinity` or `double.PositiveInfinity`: a number which is greater than every other number
  - `float.NegativeInfinity` or `double.NegativeInfinity`: a number which is less than every other number
  - `float.NaN` or `double.NaN`: "Not A Number" — `<`, `>`, `==` always return `false`
- Can check for these with `float.IsInfinity`, `double.IsNaN`, etc.
- Infinities and NaNs sometimes arise from calculations (e.g. dividing by zero)

FALMOUTH UNIVERSITY

# Numerical precision

# Precision of floating point numbers

- Precision **varies** by **magnitude**
- Numbers near 0 can be stored more accurately than numbers further from 0
- Analogy: in scientific notation with 3 decimal places
  - Around $3.142 \times 10^0$: can represent a difference of 0.001
  - Around $3.142 \times 10^3$: can represent a difference of 1
  - Around $3.142 \times 10^6$: can represent a difference of 1000

# Range of floating point numbers

| Type | Smallest value (closest to 0) | Largest value (furthest from 0) |
|---|---|---|
| Single precision | $\pm 1.175 \times 10^{-38}$ | $\pm 3.403 \times 10^{38}$ |
| Double precision | $\pm 2.225 \times 10^{-308}$ | $\pm 1.798 \times 10^{308}$ |

FALMOUTH UNIVERSITY

# Rounding errors

- Many numbers cannot be represented exactly in IEEE float
  - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$ or $\frac{1}{7} = 0.142857\ldots$
- Decimal: can represent $\frac{a}{b}$ exactly iff $b = 2^m 5^n$
- Binary: can represent $\frac{a}{b}$ exactly iff $b = 2^n$
- In particular, IEEE float can't represent $\frac{1}{10} = 0.1$ exactly!
- This can lead to **rounding errors** with some calculations

# Testing for equality

- Due to rounding errors, using `==` or `!=` with floating point numbers is almost always a bad idea
- E.g. in most languages, `0.1 + 0.2 == 0.3` evaluates to **`false`**!
- Better to check for **approximate equality**: calculate the difference between the numbers, and check that it's smaller than some threshold
- E.g. Unity has `Mathf.Approximately` which does exactly this

# Decimal type

- C# has a `decimal` type
- Uses base 10 rather than base 2, so avoids some of the surprises of IEEE float
- ... however not natively supported by the CPU, hence much slower than `float`/`double`