Representing Numbers in a Computer (Continued)

Introduction to Computer Science

Module Code: 4CC509

Overview

- Last time we looked at how unsigned integers, signed integers and fixed-point numbers are stored.
- This week, we will look at an alternative (and more common) mechanism for storing decimal numbers.

Floating Point

- We can represent any decimal number using a *mantissa* multiplied by 10 to the power of an *exponent*

  i.e.

$$\text{mantissa} * 10^{\text{exponent}}$$

Introduction to Computer Science 4CC509

Examples

426527 can be written as $4.26527 * 10^5$

42652.7 can be written as $4.26527 * 10^4$

4265.27 can be written as $4.26527 * 10^3$

426.527 can be written as $4.26527 * 10^2$

42.6527 can be written as $4.26527 * 10^1$

4.26527 can be written as $4.26527 * 10^0$

Introduction to Computer Science 4CC509

Examples

0.426527	can be written as $4.26527 * 10^{-1}$
0.0426527	can be written as $4.26527 * 10^{-2}$
0.00426527	can be written as $4.26527 * 10^{-3}$
0.000426527	can be written as $4.26527 * 10^{-4}$
0.0000426527	can be written as $4.26527 * 10^{-5}$
0.00000426527	can be written as $4.26527 * 10^{-6}$

Introduction to Computer Science 4CC509

Floating Point

- A positive exponent shifts the decimal point to the right
- A negative exponent shifts the decimal point to the left

Introduction to Computer Science 4CC509

Floating Point in Binary

- Floating point in binary works exactly the same as for decimal, except that we store numbers in the form:

$$\text{mantissa} * 2^{\text{exponent}}$$

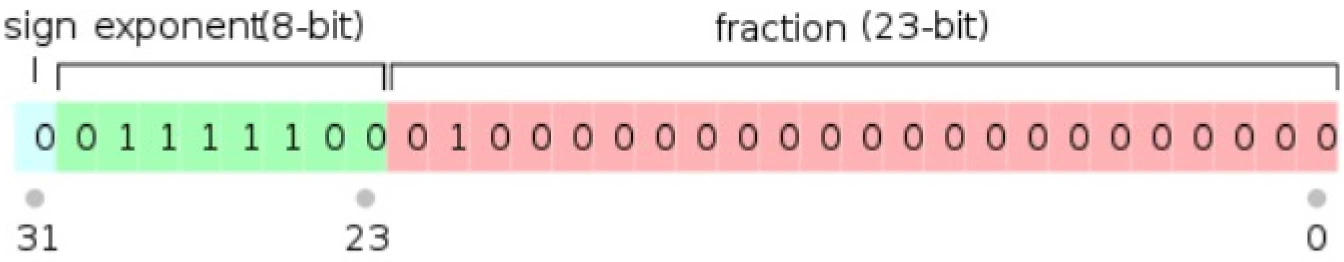
Introduction to Computer Science 4CC509

IEEE 754

- IEEE 754 is an industry standard for representing floating-point numbers in computers.
- Officially adopted in 1985 and superseded in 2008 by IEEE 754-2008.
- The most widely used format for floating-point computation
- IEEE 754 defines both single and double precision representations (stored as 32-bit or 64-bit values).
- For those of you doing Programming 1, in C# you would represent these using the types *float* or *double*.

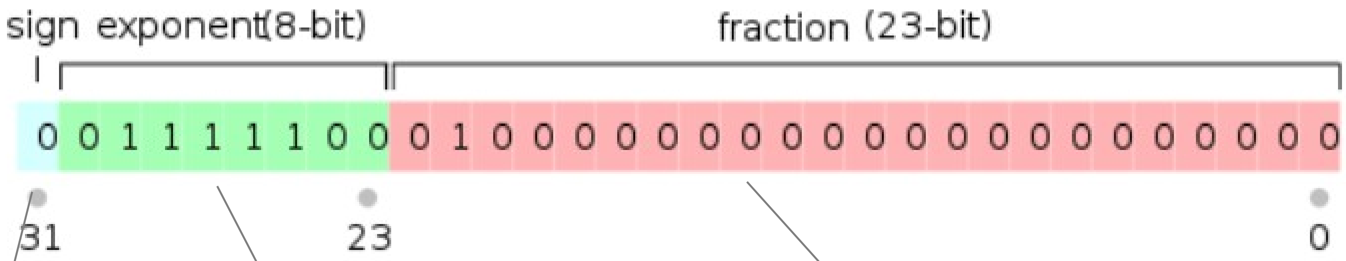
IEE 754

- Single-precision floating-point numbers in IEEE 754 format consist of three fields:
  - a sign bit,
  - a biased exponent (8 bits). The bias is 127.
  - a fraction (23 bits) and an implied ‘1’ before the decimal point (the mantissa).



Source: http://en.wikipedia.org/wiki/IEEE_754-1985

Example



Fractional part = 1.01 (the 1 to the left of the decimal point is implied)

Sign bit = Positive

Exponent = 124 – 127 = -3

=> Number in binary = 1.01×2^{-3}

=> Number in binary = 0.00101

=> Number in decimal = 0.15625

Notes – The Mantissa

- Because of the implied ‘1’ before the decimal point, the number represented by the fractional part f is actually 1 + f.
- This has the additional side effect of giving us a little bit more precision since we now have 24 bits in the mantissa, but only need 23 bits to represent it.

Introduction to Computer Science 4CC509

Notes – The Exponent

- The exponent is stored as a biased number, i.e. it is the actual exponent + 127.
- This means that we can store exponents in the range -126 to +127 as numbers in the range 1 to 254 and not have to worry about the sign.
- Storing the biased exponent before the mantissa means we can compare IEEE values as if they were signed integers.

Introduction to Computer Science 4CC509

Converting an IEEE 754-1985 value to Decimal

s	e	f

- The decimal value is given by:

$$(1 - 2s) \times (1 + f) \times 2^{e\text{-}bias}$$

- where:
  - 1 – 2s is 1 or -1 depending on whether the sign bit is 0 or 1
  - bias = 127

Our Example From Last Week

- The particular 4 bytes of memory contain the following values:

10000001	10110000	00000000	00000000

- If we treat the contents of these 4 bytes as a floating point value:

  s = 1

  e = 3 (00000011)

  f = .011

Our Example

- This gives a value of:

  $(1 – 2) \times (1 + .011) \times 2^{3 - 127}$

  $=> \quad -1.011 \times 2^{-124}$

  $=> \quad 6.4652189 \times 10^{-38}$ (rounded)

Introduction to Computer Science 4CC509

Special Values

- The smallest and largest values for the exponent (00000000 and 11111111) are reserved for special values.
  - If the exponent is 0, the number stored represents zero
  - If the exponent is all 1 and the mantissa is all 0, then the value represents infinity (positive or negative depending on the sign bit)
  - If the exponent is all 1 and the mantissa contains a non-zero value, then the value stored is “Not a number” (or NaN).
    - Produced in error situations

Range of Single Precision Floating Point Values

- Just considering positive numbers:
  - The largest possible value (i.e. not a special value) is $(2 – 2^{-23}) \times 2^{127}$
  - The smallest positive non-zero number is 2^{-126}
- By comparison, the largest unsigned integer that can be represented in 32 bits is $2^{32} – 1$ and the smallest non-zero value is 1
- So we can represent a much larger range of numbers using floating point values than we can using integers that take up the same number of bits
- How can this be?

Finiteness

- The problem is that given 32 bits, we can only store 2^{32} numbers (approximately 4 billion), regardless of how we represent them
- Therefore, if we are representing a much larger range of numbers, then it means that there will be gaps in that range, i.e. there are some numbers in that range that we cannot represent
- This causes some real problems when we start doing floating point arithmetic.

Finiteness

- Problems:
  - Not all values in the range can be represented
  - Small roundoff errors can quickly accumulate with multiplication, resulting in big errors
  - Rounding errors can invalidate many basic arithmetic rules such as the associative law

    i.e. $(x + y) + z = x + (y + z)$
- The IEEE 754 standard guarantees that all machines will produce the same results – but those results may not be mathematically correct.

Finiteness – Simple Example using Decimals

- Imagine if 6 digits are used to store the mantissa:
- How is the number 123,456,000,000 stored?

$$1.23456 * 10^{11}$$

- How is the number 123,456,400,000 stored?

$$1.23456 * 10^{11}$$

Introduction to Computer Science 4CC509

Loss of Precision

- Note the loss of precision in the last example. This is because we are only using 6 digits to store the mantissa. This means that very large or very small numbers will lose precision when converted to floating point.
- The more digits we can use to store the mantissa, the less likely we are to lose precision

Introduction to Computer Science 4CC509

Limits of Number Representation

- Similarly, some numbers cannot be represented in the IEEE 754 format:

```
int x = 33554431;
float y = 33554431;
```

- What happens to y?
- Some decimal numbers cannot be finitely represented in binary at all:
    - $0.10_{10} = 0.0001100110011....._2$

Rounding Errors

- You can have problems on arithmetic operations if one value is much smaller than another.
- For example:
  - $(1.5 \times 10^{38}) + 1.0 = 1.5 \times 10^{38}$
  - The 1.0 that has been added has been completely lost

Introduction to Computer Science 4CC509

So why use floating point?

- Speed
  - Most processors these days include floating point processing units that are designed to perform arithmetic operations on floating point values quickly

- Do not use floating point if accuracy is vitally important – e.g. operations involving money. In these cases, it is preferable to use a fixed-point type (in C# you could use the type *decimal*). However, floating point is commonly used for graphics operations where the loss of precision is not usually so critical.

Introduction to Computer Science 4CC509

Summary

- The important thing to take away from these last two lectures is that the *type* of the information stored in a location in memory is important for determining what that information means.

Introduction to Computer Science 4CC509

Summary

- Our 4 bytes of memory:

10000001	10110000	00000000	00000000

- contain:

  2175795200 if we treat is as an unsigned integer

  -2119172096 if we treat is as a signed integer

  -32336.0 if we treat it as a fixed-point decimal number

  6.4652189 x 10^{-38} if we treat it as a floating-point decimal number.

Introduction to Computer Science 4CC509

Summary

- Of course, those 4 bytes of memory might not contain a 32-bit number.
- They could contain part of a 64-bit number, characters from a string or even a machine instruction.
- So how we view what is in memory depends completely on the context of what the piece of memory is used for.

Introduction to Computer Science 4CC509

This Week

- Another formative test has been placed on Course Resources for you to test your knowledge of what has been covered last week and this week
- It can be found in the same location as the slides for the lecture this week.

Introduction to Computer Science 4CC509