Floating Point Representation

CS3220 - Summer 2008

Jonathan Kaldor

Floating Point Numbers

- Infinite supply of real numbers
  - Requires infinite space to represent certain numbers
- We need to be able to represent numbers in a finite (preferably fixed) amount of space

Floating Point Numbers

- What do we need to consider when choosing our format?
  - Space: using space efficiently
  - Ease of use: representation should be easy to work with for adding/subtracting/multiplying/comparisons

Floating Point Numbers

- What do we need to consider when choosing our format?
  - Limits: want to represent very large numbers, very small numbers
  - Precision: want to be able to represent neighboring but unequal numbers accurately

Precision, Precisely

- Two ways of looking at precision:
- Let x and y be two adjacent numbers in our representation, with x > y
  - Absolute precision: |x - y|
    - a.k.a. the epsilon of the number y
  - Relative precision: |x - y|/|x|

What is a Floating Point Number?

- Examples: 7.423×10^{3}, 5.213×10^{-2}, etc
- Floating point because the decimal moves around as exponent changes
- Finite length real numbers expressed in scientific notation
  - Only need to store significant digits

What is a Fixed Point Number?

- Compare to fixed point: constant number of digits to left and right of decimal point
- Examples: 150000.000000 and 000000.000015
- Fixed absolute precision, but relative precision can vary (very poor relative precision for small numbers)

Floating Point Systems

- β: Base or radix of system (usually either 2 or 10)
- p: precision of system (number of significant digits available)
- [L,U]: lower and upper bounds of exponent

Floating Point Systems

- Given β, p, [L,U], a floating point number is:
  $\pm(d_0 + d_1/\beta + d_2/\beta^2 + \ldots + d_{p-1}/\beta^{p-1}) \times \beta^e$

  mantissa

where $0 \leq d_i \leq \beta-1$

$L \leq e \leq U$

Example Floating Point System

- Let $\beta = 10$, $p = 4$, $L = -99$, $U = 99$. Then numbers look like

4.560×10^{03} $\quad$ -5.132×10^{-26}

This is a convenient system for exploring properties of floating point systems in general

Example Numbers

- What are the largest and smallest numbers in absolute value that we can represent in this system?

  9.999×10^{99}
  0.001×10^{-99}

- Note: we have shifted to denormalized numbers (first digit can be zero)

Example Numbers

- Lets write zero:
  0.000×10^{0} ... or 0.000×10^{1},
  ... or 0.000×10^{10}
  ... or 0.000×10^{x}
- No longer have a unique representation for zero

Denormalization

- In fact, we no longer have a unique representation for *many* of our numbers

  4.620×10^2
  0.462×10^3

- These are both the same number... almost
- We have lost information in the second representation, however

Normalized Numbers

- Usually best to require first digit in representation be nonzero
  - Requires a special format for zero, now
- Double bonus: in binary (β=2), our normalized mantissa always starts with 1... can avoid writing it down

Some Simple Computations

- $(1.456 \times 10^{03}) + (2.378 \times 10^{01})$

  1.480×10^{03}

- Note: we lose digits
- Note also: Answer depends on rounding strategy

Rounding Strategies

- Easiest method: chop off excess (a.k.a. round to zero)
- More precise method: round to nearest number. In case of a tie, round to nearest number with *even* last digit
- Examples: 2.449, 2.450, 2.451, 2.550, 2.551, rounded to 2 digits

Some Simple Computations

- $(1.000 \times 10^{60}) \times (1.000 \times 10^{60})$

  Number not representable in our system (exponent 120 too large)

- Denoted as ‘overflow’

Some Simple Computations

- $(1.000 \times 10^{-60}) \times (1.000 \times 10^{-60})$

  Number not representable in our system
  (exponent -120 too small)

- Denoted as ‘underflow’

Some Simple Computations

- $1.432 \times 10^2 - 1.431 \times 10^2$
- Answer is $0.001 \times 10^2 = 1.000 \times 10^{-1}$
- However, we have lost almost all precision in the answer
  - Example of *catastrophic cancellation*

Catastrophic Cancellation

- In general, adding/subtracting two nearly-similar quantities is unadvisable. Avoiding it can be important for accuracy
- Consider the familiar quadratic formula...

Some Simple Computations

- $1.000 \times 10^{03} - 1.000 \times 10^{03} + 1.000 \times 10^{-04}$
- Answer depends on order of operations
  - In real numbers, addition and subtraction are associative
  - In floating point numbers, addition and subtraction are NOT associative

Unexpected Results

- Because of its peculiarities, some results in floating point are unexpected
- Take $\sum 1/n$ as $n \rightarrow \infty$
  - Unbounded in real numbers
  - Finite in floating point (why?)

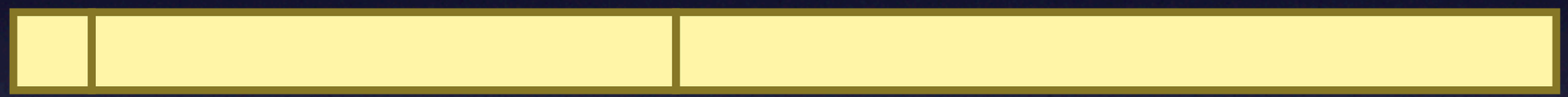
Precision

- As before, epsilon of a floating point number x is defined as the absolute precision of x and the next number in the floating point representation.
  - distance to next representable number
  - note: x is first converted to FP number
- Relative precision: depends on p (mostly)
  - Exception: denormalized numbers

IEEE-754

- Defines four floating point types (with β=2) but we're only interested in two of them
- Single precision: 32 bits of storage, with p = 24, L = -126, U = 127
- Double precision: 64 bits of storage, with p = 53, L = -1022, U = 1023
- One bit for sign

IEEE-754



Sign | Exponent | Mantissa

8 or 11 bits | 23 or 52 bits

IEE-754

- Single precision:
  Largest value: ~3.4028234×10^{38}
  Smallest value: ~$1.4012985 \times 10^{-45}$ (2^{-149})
- Double precision
  Largest value: ~$1.7976931 \times 10^{308}$
  Smallest value: ~2.225×10^{-307} (2^{-1074})

Mantissa

- Recall in $\beta=2$, our mantissa looks like 1.01011101 (we normalize it so that the first digit is always nonzero)
  - First digit is then always 1... so we don't need to store it
  - Gain an extra digit of precision (look back and see definition of p compared to actual bit storage)

Exponent

- Rather than have sign bit for exponent, or represent it in 2s complement, we *bias* the exponent by adding 127 (single) or 1023 (double) to the actual exponent
- i.e. if number is 1.01111×2^{10}, in single precision the exponent stored is 137

Exponent

- Why do we bias the exponent?
  - Makes comparisons between floating point numbers easy: can do bitwise comparison

Example

- What is 29.34375 in single precision?

Zero

- Normalizing mantissa creates a problem for storing 0. To get around this, we reserve the smallest exponent (-127, which when biased is 0) to represent denormalized numbers (implicit digit is 0 instead of 1)
- Exponent 0 is otherwise considered like exponent 1 (in single precision, both are 2^{-126})

Denormalized Numbers

- Thus, zero is the number consisting of all zeros in exponent and mantissa fields (can be signed)
- Nonzero mantissa: denormalized numbers
  - Allows us to express numbers smaller than expected range, at reduced precision
  - “Graceful” underflow

Special Numbers

- Similarly, the maximum exponent (exponent field of all 1's) is reserved for special numbers
  - If mantissa is all zero, then number is either $+\infty$ or $-\infty$
  - If mantissa is nonzero, then number is NaN (Not A Number)

Special Numbers

- If x is a finite number

  ∞ ± x = ∞

  -∞ ± x = -∞

  ±x / 0 = ±∞ (x != 0)

  ±∞ / 0 = ±∞

  ±x / ±∞ = ±0

  ±0 / ±0 = NaN

  ±∞ / ±∞ = NaN

- Any computation with NaN → NaN

Overflow and Underflow

- If computation underflows, result is 0 of appropriate sign
- If computation overflows, result is ∞ of appropriate sign
- Can be an issue, but catastrophic cancellation / precision issues usually far more important

Example of System with Floating Point Error

- (Demo, also part of HW3)