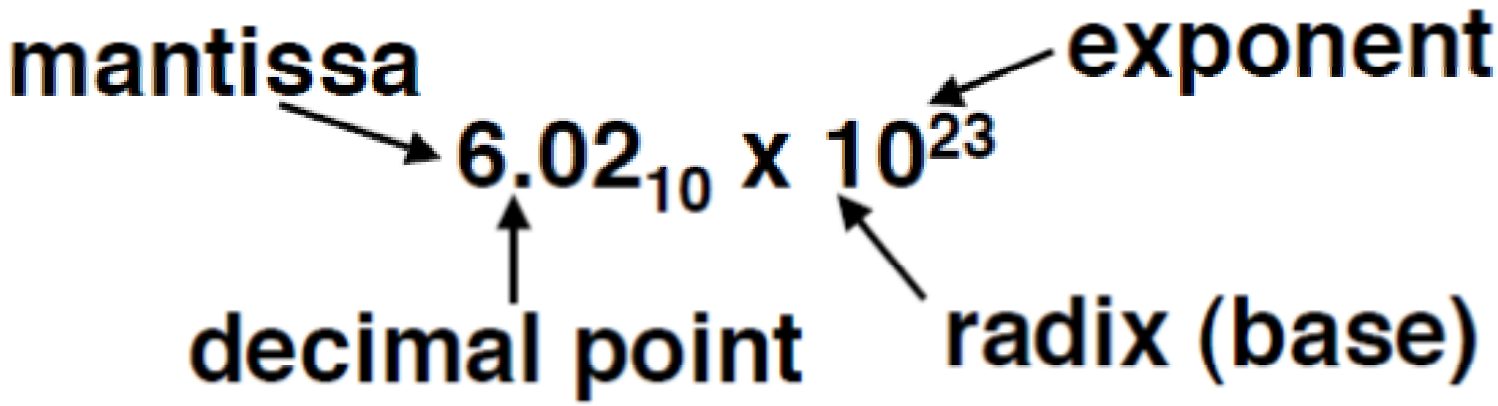
On Floating Point Representation of Real Numbers in Computers

Amjad Ali

Centre for Advanced Studies in Pure and Applied Mathematics
(CASPAM)
Bahauddin Zakariya University (BZU),
Multan 60800, Pakistan.

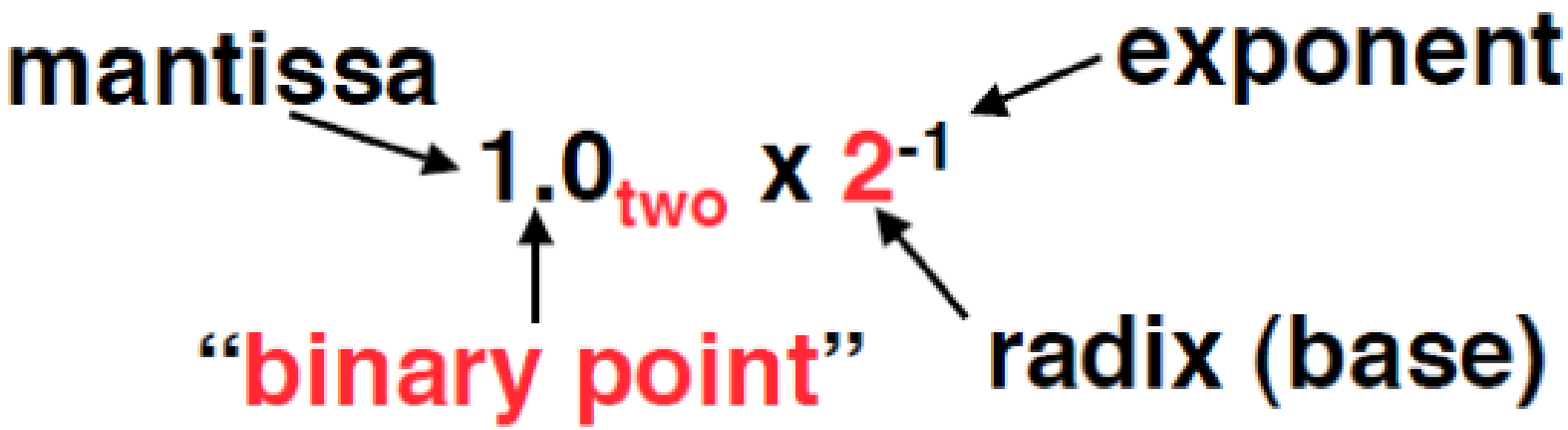
Advanced School on Scientific Software Development (SMR2330 Activity), 20 Feb. to 02 Mar. 2012, ICTP, Italy

Scientific Notation (in Decimal)



- Normalized form: no leadings 0s (exactly one digit to left of decimal point)
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

Scientific Notation (in Binary)



- Computer arithmetic that supports it called floating point, because it represents numbers where the binary point is not fixed, as it is for integers
  - Declare such variable in C as `float`

Exponential Notation of Real Numbers

A real number x is approximated in machine by a rational number of the form:

$$x = (-1)^s \times m \times \beta^{\text{exponent}}$$

Where:

- β is the radix:

  $\beta = 10$ in your calculator and (usually) your head
  $\beta = 2$ in most computers

- $s \in \{0, 1\}$ is a sign bit. $s=0$ specifies a +ve number and $s=1$ specifies a -ve number.
- m is the mantissa, a rational number of n_m digits in radix β, such that

  $$m = d_0 d_1 d_2 \ldots d_{nm-1}$$

- exponent is a signed integer.

n_m specifies the precision of the format.

Imposing d_0 to be non-zero ensures uniqueness of representation (*normalization*).

A floating-point number system is called *normalized* if the leading digit d_0 is always nonzero, unless the number represented is zero

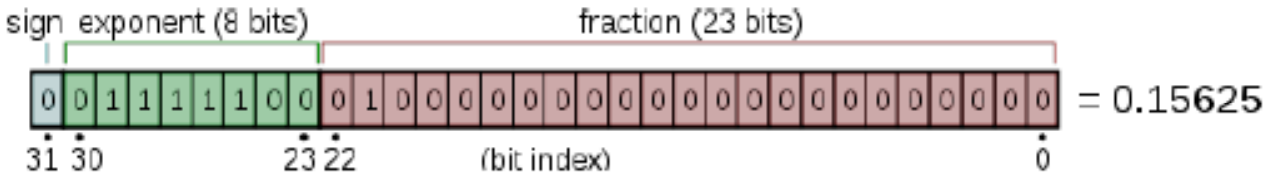
Why Normalize

- The representation of each number is unique
- Maximize precision because no digits wasted on leading zeros
- Gain one extra bit of precision for a given field width in the Binary system ($\beta = 2$) because the leading bit is always 1 and need not be stored

Floating-point representation and bits

- We can represent floating-point numbers with three binary fields: a sign bit s, an exponent field e, and a fraction field f.

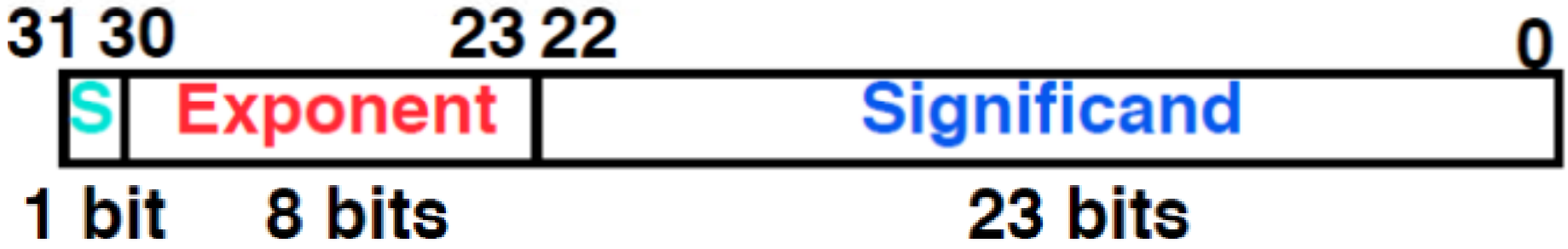
- The IEEE 754 standard defines several different precisions.
  - Single precision numbers include an 8-bit exponent field and a 23-bit fraction, for a total of 32 bits.



  - Double precision numbers have an 11-bit exponent field and a 52-bit fraction, for a total of 64 bits.

Floating Point Representation (1/2)

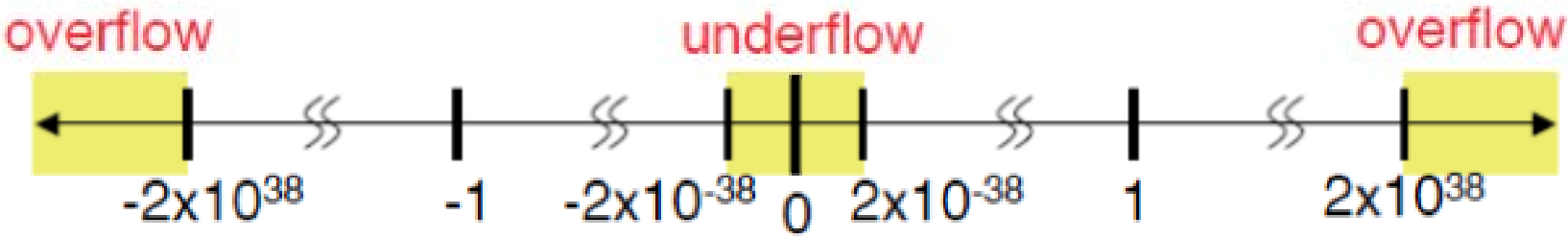
- Normal format: $+1.xxxxxxxxxxx_{two} * 2^{yyyy_{two}}$
- Multiple of Word Size (32 bits)



- S represents Sign
  Exponent represents y's
  Significand represents x's
- Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}

Floating Point Representation (2/2)

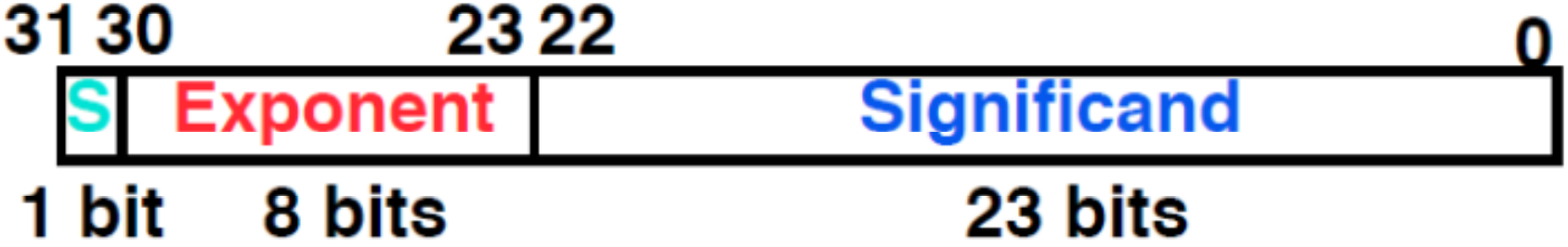
- What if result too large?
  - ($> 2.0 \times 10^{38}$, $< -2.0 \times 10^{38}$)
  - Overflow! ⇒ Exponent larger than represented in 8-bit Exponent field
- What if result too small?
  - (>0 & $< 2.0 \times 10^{-38}$, <0 & $> -2.0 \times 10^{-38}$)
  - Underflow! ⇒ Negative exponent larger than represented in 8-bit Exponent field



- What would help reduce chances of overflow and/or underflow?

IEEE 754 Floating Point Standard (1/3)

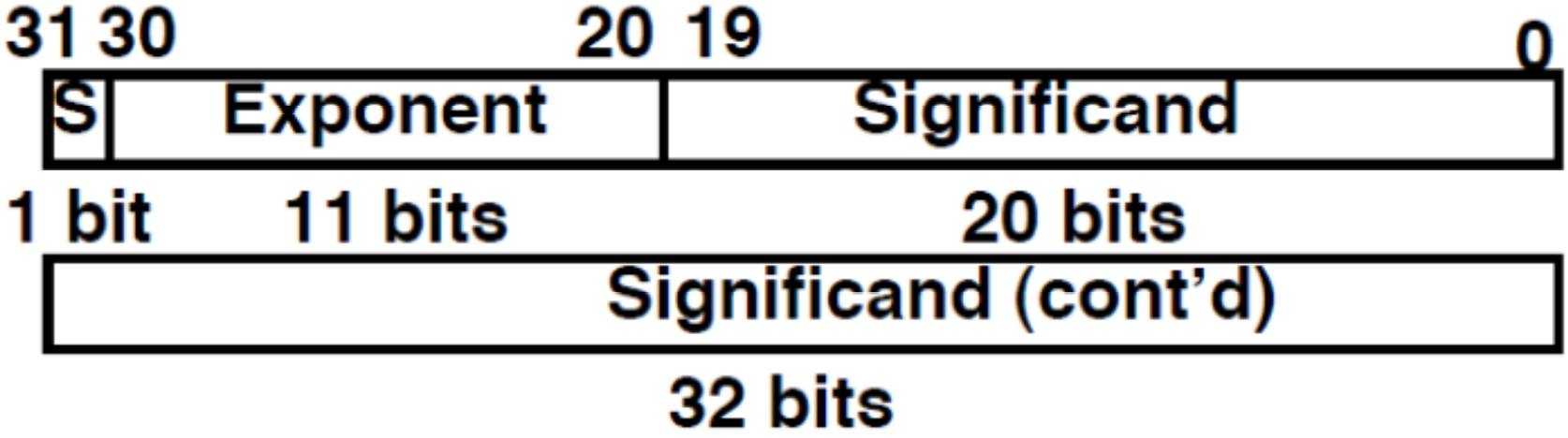
Single Precision (DP similar):



- Sign bit: 1 means negative
  0 means positive
- Significand:
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true: 0 < Significand < 1 (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

Double Precision Fl. Pt. Representation

- Next Multiple of Word Size (64 bits)



- Double Precision (vs. Single Precision)
  - C variable declared as `double`
  - Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}
  - But primary advantage is greater accuracy due to larger significand

QUAD Precision Fl. Pt. Representation

- Next Multiple of Word Size (128 bits)
  - Unbelievable range of numbers
  - Unbelievable precision (accuracy)
- Currently being worked on (IEEE 754r)
  - Current version has 15 exponent bits and 112 significand bits (113 precision bits)
- Oct-Precision?
  - Some have tried, no real traction so far
- Half-Precision?
  - Yep, that's for a short (16 bit)

en.wikipedia.org/wiki/Quad_precision
en.wikipedia.org/wiki/Half_precision

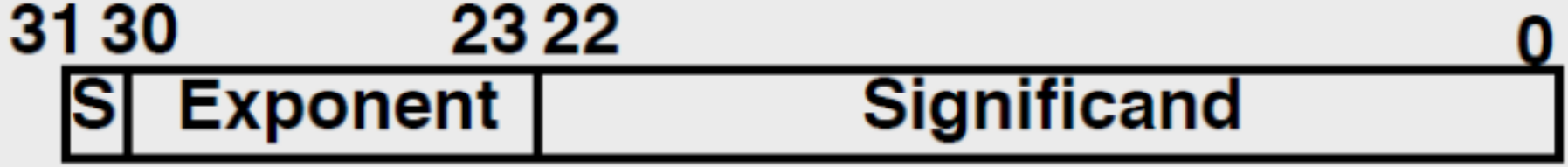
Format Size Comparison

Type	Sign	Exponent	Significand	Total bits	Exponent bias	Bits precision
Half (IEEE 754-2008)	1	5	10	16	15	11
Single	1	8	23	32	127	24
Double	1	11	52	64	1023	53
Quad	1	15	112	128	16383	113

IEEE 754 Floating Point Standard

- Called Biased Notation, where bias is number subtracted to get real number
  - IEEE 754 uses bias of 127 for single prec.
  - Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision

- Summary (single precision):



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - Double precision identical, except with exponent bias of 1023 (half, quad similar)

Exercise:

- **Given a binary real number using scientific notation (e.g., -1.000111 $\times$ 2^{100}), obtain its 32-bit representation pattern according to IEEE 754 standard.**

(solution: 11000001100011100000000000000000)

Exercise:

Given a binary real number using scientific notation (e.g., -1.000111×2^{100}), obtain its 32-bit representation pattern according to IEEE 754 standard.

(solution: 11000001100011100000000000000000)

Given the 32-bit IEEE 754 standard representation pattern of a real number (e.g., 01000001100011100000000000000000), obtain the BINARY real number in scientific notation.

(solution: 1.000111×2^{100})

IEEE 754 representation.

- Used by all modern computers.
- Scientific notation, but in binary.
- Single precision: `float` = 32 bits.
- Double precision: `double` = 64 bits.

Ex. Single precision representation of -0.453125.

bias phantom bit

$$-1 \times 2^{125 - 127} \times 1.8125 = -0.453125$$

In your favourite scientific programming language

- Fortran/ Fortran90
  - Real*4 real(kind=4) ::
  - Real*8 real(kind=8) ::
- C/C++
  - Float
  - Double
  - Long double (see discussion later)

IEEE 754

- During 80's Institute for electrical and electronics Engineers produced a standard for the FP format.
- Now the " *IEEE754 -1985 Standard for binary Floating Point Arithmetic* " is adopted by almost all the vendors..
- IEEE Standard specified all the details about FP system:
  - storage format
  - precise specification of the result of operations
  - special values
  - specified runtime behavior on illegal operations
- Life is easier now: portability of kernel computational code is ensured...

"Father" of the Floating point standard

IEEE Standard 754 for Binary Floating-Point Arithmetic.

Prof. Kahan

1989 ACM Turing Award Winner!

www.cs.berkeley.edu/~wkahan/
…/ieee754status/754story.html

Why do we need a standard ?

- Ensure portability
- Ensure provability
- Ensure that some important mathematical properties hold
  - People will assume that x + y == y + x
  - People will assume that x + 0 == x
  - People will assume that x==y ⇔ x−y==0
- These benefits should not come at a significant performance cost
- Obviously, we need to specify not only the formats but also the operations.

Range of single-precision numbers (32 bit)

BIASED exponent

$$(1 - 2s) * (1 + f) * 2^{e-127}.$$

- The smallest e is 00000001 (1).
- The smallest f is 00000000000000000000000 (0).
- The largest possible e is 11111110 (254).
- The largest possible f is 11111111111111111111111 ($1 - 2^{-23}$).

- The largest possible "normal" number is $(2 - 2^{-23}) * 2^{127} \approx 3.4 \times 10^{38}$.
- the smallest *positive* non-zero number is $1 * 2^{-126} = \approx 1.8 \times 10^{-38}$.
- In comparison, the range of possible 32-bit integers in two's complement are only -2^{31} and $(2^{31} - 1)$
- How can we represent so many more values in the IEEE 754 format, even though we use the same number of bits as regular integers?

FP Finiteness

- There *aren't* more IEEE numbers.
- With 32 bits, there are $2^{32}-1$, or about 4 billion, different bit patterns.
  - These can represent 4 billion integers or 4 billion reals.
  - But there are an infinite number of reals, and the IEEE format can only represent *some* of the ones from about -2^{128} to $+2^{128}$.
  - Represent same number of values between 2^n and 2^{n+1} as 2^{n+1} and 2^{n+2}

- Thus, floating-point arithmetic has "issues"
  - Small roundoff errors can accumulate with multiplications or exponentiations, resulting in big errors.
  - Rounding errors can invalidate many basic arithmetic principles such as the associative law, $(x + y) + z = x + (y + z)$.
- The IEEE 754 standard guarantees that all machines will produce the same results—but those results may not be mathematically correct!

- When a real number is exactly representable in a given floating-point system, it is called a machine number
- When this isn't the case, the number has to be approximated a *nearby* floating-point number
- The process of choosing this *nearby* floating-point number is called rounding
- Naturally, the error produced from this approximation is called rounding error or roundoff error

density of FP numbers..

Because the same number of bits are used to represent all normalized numbers, the smaller the exponent, the greater the density of representable numbers.

For example, there are approximately 8,388,607 single-precision numbers between 1.0 and 2.0, while there are only about 8191 between 1023.0 and 1024.0.

This means that for large numbers (both positive and negative) there are very few FP numbers to play with and many real numbers map to the same floating-point number.

Rounding

- When we perform math on real numbers, we have to worry about rounding to fit the result in the significant field.
- The FP hardware carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting:
  - double to a single precision value, or
  - floating point number to an integer

IEEE FP Rounding Modes

Examples in decimal (but, of course, IEEE754 in binary)

- **Round towards + ∞**
  - ALWAYS round "up": 2.001 → 3, -2.001 → -2
- **Round towards - ∞**
  - ALWAYS round "down": 1.999 → 1, -1.999 → -2
- **Truncate**
  - Just drop the last bits (round towards 0)
- **Unbiased (default mode). Midway? Round to even**
  - Normal rounding, almost: 2.4 → 2, 2.6 → 3, 2.5 → 2, 3.5 → 4
  - Round like you learned in grade school (nearest int)
  - Except if the value is right on the borderline, in which case we round to the nearest EVEN number
  - Insures fairness on calculation
  - This way, half the time we round up on tie, the other half time we round down. Tends to balance out inaccuracies

IEE 754 exception handling

.What happens when the "exact value" is not a real number, or too small or too large to represent accurately?

- Five exceptions:
  - Overflow - exact result > OV, too large to represent.
  - Underflow - exact result nonzero and < UN, too small to represent.
  - Divide-by-zero – nonzero/0.
  - Invalid - 0/0, sqrt(-1), …
  - Inexact - you made a rounding error (very common!).
- Possible responses
  - Stop with error message (unfriendly, not default).
  - Keep computing (default, but how?).

Rounding problem

- Many operations among floating points does not end in a floating point.
- IEEE 754 defines the way to handle this:
  - Take the exact value, and round it to the nearest floating point number (correct rounding).
  - Break ties by rounding to nearest floating point number whose bottom bit is zero (rounding to nearest even).
  - Other rounding options also available (up, down, towards 0).

Even some rather simple numbers can have FP rounding problem...

- 1/3 is NOT exactly representable
- 0.01 is NOT exactly representable
- Question to be answered during lab this afternoon
  - how many inverse are not exactly representable in the range [1, 100] ?

Floating Point Fallacy

- FP add associative: FALSE!
  - $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$
  - $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$
    $= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = \underline{0.0}$
  - $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$
    $= (0.0) + 1.0 = \underline{1.0}$
- Therefore, Floating Point add is not associative!
  - Why? FP result approximates real result!
  - This example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Precision and Accuracy

Don't confuse these two terms!

Precision is a count of the number bits in a computer word used to represent a value.

Accuracy is a measure of the difference between the actual value of a number and its computer representation.

High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.

Example: `float pi = 3.14;`

pi will be represented using all 24 bits of the significant (highly precise), but is only an approximation (not accurate).

Representation for ± ∞

- In FP, divide by 0 should produce ± ∞, not overflow.
- Why?
  - OK to do further computations with ∞ E.g., X/0 > Y may be a valid comparison
  - Ask math majors
- IEEE 754 represents ± ∞
  - Most positive exponent reserved for ∞
  - Significands all zeroes

Representation for 0

- Represent 0?
  - exponent all zeroes
  - significand all zeroes
  - What about sign? Both cases valid.

```
+0: 0 00000000 00000000000000000000000
-0: 1 00000000 00000000000000000000000
```

Special Numbers

- What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	???

- Professor Kahan had clever ideas; "Waste not, want not"
  - We'll talk about Exp=0,255 & Sig!=0 later

Representation for Not a Number

- What do I get if I calculate `sqrt(-4.0)` or `0/0`?
  - If ∞ not an error, these shouldn't be either
  - Called Not a Number (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN, X) = NaN

Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable pos num:

    $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$

  - Second smallest representable pos num:

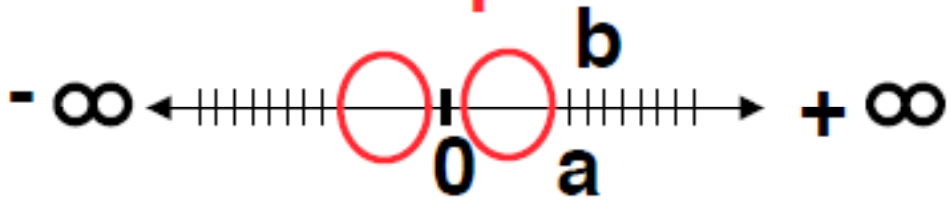
    $b = 1.000\ldots\ldots1_2 * 2^{-126}$
    $= (1 + 0.00\ldots1_2) * 2^{-126}$
    $= (1 + 2^{-23}) * 2^{-126}$
    $= 2^{-126} + 2^{-149}$

    Normalization and implicit 1 is to blame!

$a - 0 = 2^{-126}$

$b - a = 2^{-149}$

Gaps!



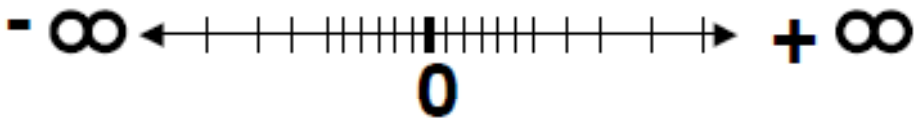
Representation for Denorms (2/2)

- Solution:
  - We still haven't used Exponent = 0, Significand nonzero
  - Denormalized number: no (implied) leading 1, implicit exponent = -126.
  - Smallest representable pos num:

    $a = 2^{-149}$

  - Second smallest representable pos num:

    $b = 2^{-148}$



Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

Checking Floating-Point Equality

- Sometimes okay to compare for equality
  - When calculations are known to be exact
  - To synthesize a comparison
  - Compare against 0.0 to avoid division by zero
- But floating-point results are usually inexact
  - Comparing floating-point numbers for equality may have undesirable results

```
do while (tot=1.0)
      tot=tot+0.1
end do
```

Using float/real*4 for computations

- Storing low-precision data as float is fine, but
- Generally not recommended to use float for computations
  - Float has less than half the precision of double
  - Using double intermediates greatly reduces the risk of roundoff problems polluting the answer
  - Round double value back to float to give a float result
- Extra internal precision is ablative armor against roundoff problems

be careful on float -> integer conversion

- FP numbers converted in integer number can lead to overflow/underflow ...

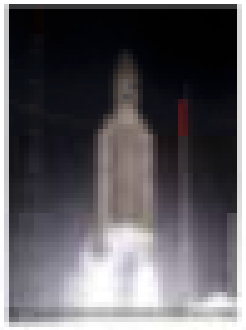
Data conversion

- On 4 June 1996, the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, exploded.
- The failure of the Ariane 501 was caused by the complete loss of guidance and attitude. This loss of information was due to specification and design errors in the software of the inertial reference system.
- The internal SRI* software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value.
- The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. http://www.ima.umn.edu/~arnold/disasters/ariane.html

Numerical Catastrophes

Ariane 5 rocket. [June 4, 1996]

- 10 year, $7 billion ESA project exploded after launch.
- 64-bit float converted to 16 bit signed int.
- Unanticipated overflow.



Copyright, Arianespace

Vancouver stock exchange. [November, 1983]

- Index undervalued by 44%.
- Recalculated index after each trade by adding change in price.
- 22 months of accumulated truncation error.

Patriot missile accident. [February 25, 1991]

- Failed to track scud; hit Army barracks, killed 28.
- Inaccuracy in measuring time in 1/20 of a second since using 24 bit binary floating point.

Catastrophic Cancellation

```
public static double f1(double x) {
   return (1.0 - Math.cos(x)) / (x* x);
}
```

Ex. Evaluate `f1(x)` for `x = 1.1e-8`.

- `Math.cos(x) = 0.99999999999999988897769753748434595763683319091796875`

  nearest floating point value agrees with exact answer to 16 decimal places.

- `(1.0 - Math.cos(x)) = 1.1102e-16`

  inaccurate estimate of exact answer ($6.05 \cdot 10^{-17}$)

- `(1.0 - Math.cos(x)) / (x*x) = 0.9175`

  80% larger than exact answer (about 0.5)

Catastrophic cancellation. Devastating loss of precision when small numbers are computed from large numbers, which themselves are subject to roundoff error.

FP Subtraction : the cancellation problem(1)

- Subtraction between two t-digit number having same sign and similar magnitude yields result with fewer that t digits, hence it is always representable.
- Reason is that the leading digits of two numbers cancel (I.e. their difference is zero)
- Example:
  - 1.92403 X102 -1.92275 X102 = 0.00128 X102

  ----> 1.28000X10-1
- This is correct, exact representable but has only 3 digits ..

Cancellation (2)

- Despite exactness of results, cancellation often implies serious loss of information.
- Operand are often uncertain, due to rounding or other previous errors, in which case relative uncertainty in difference may be large.
- Subtraction itself is not at fault: it signals loss of information that has already occurred.
- Problems may arise if such a subtraction is followed by multiplications or divisions

  You may get meaningless digits in your result

Casting floats to ints and vice versa

`(int)` *`floating_point_expression`*

Coerces and converts it to the nearest integer (C uses truncation)

```
i = (int) (3.14159 * f);
```

`(float)` *`integer_expression`*

converts integer to nearest floating point

```
f = f + (float) i;
```

int → float → int

```
if (i == (int)((float) i)) {
  printf("true");
}
```

- Will not always print "true"
- Most large values of integers don't have exact floating point representations!
- What about `double`?

float → int → float

```
if (f == (float)((int) f)) {
  printf("true");
}
```

- Will not always print "true"
- Small floating point numbers (<1) don't have integer representations
- For other numbers, rounding errors

What about compilers ?

- IEEE754 standard defines how FP are to be performed
- compilers which translates our high level language (fortran/C) to assembler is responsible to generate IEEE-compliant code
- there are generally specific flags to force the compiler to adhere to the standard
- By default some compiler do not adhere to it: you have to force them to adhere..

Search the compiler flags/options yourself (of your compiler) to enforce adherence to IEEE Floating Point Standard

X86 instruction sets

- Implemented in processors by Intel, AMD
- internal double-extended format on 80 bits: mantissa on 64 bits, exponent on 15 bits.
- (almost) perfect IEEE compliance on this double-extended format
- one status register which holds (among other things)
  - the current rounding mode
  - the precision to which operations round the mantissa: 24, 53 or 64 bits
  - but the exponent is always 15 bits
- For single and double, IEEE-754-compliant rounding and overflow handling (including exponent) performed when writing back to memory

What it means for us ?

Assume you want a portable program, i.e use double-precision.

- Fully IEEE-754 compliant possible, but slow:
  - set the status flags to "round mantissa to 53 bits"
  - then write the result of every single operation to memory (not every single but almost)
- Next best: compliant except for over/underflow handling:
  - set the status flags to "round mantissa to 53 bits"
  - but computations will use 15-bit exponents instead of 12 OK if if you may prove that your program doesn't generate huge nor tiny values
- Default behavior for C/gcc in Linux:
  - All the computations on registers are done in double-extended precision, even if the variables were declared as double.
  - Round to actual double only when writing to memory.
  - ⊕ More accurate in the common case (when portability not an issue)
  - ⊖ ... but it's the compiler who decides which variable is held in memory, and which is in register.

Intel behaviour on Floating Points

- From man ifort

-fp-model keyword

Controls the semantics of floating-point calculations

Default:

-fp-model fast=1 The compiler uses more aggressive optimizations on floating-point calculations.

Description:

This option controls the semantics of floating-point calculations.

The keywords can be considered in groups:

- Group A: precise, fast, strict
- Group B: source
- Group C: except (or the negative form)

Example:

let us compute how many inverse are not correctly represented by FP

```
program inverse
!a simple program to check how many inverse are not accurate.
   real*4 :: X,Y,Z
   integer:: I

   i=0
   do j=1,100,1
        X=real(j)
        Y=1.0/X
        Z=Y*X
        IF (Z.ne.1.0) then
          write(*,'(A,F28.26)') "this is not correct=", Z
          i=i+1
        end if
   end do
     write(*,*)"found", I

   end program inverse
```

Task

- **Compile with different compilers:**
  - **using default (no flag)**
  - **using standard optimization**
  - **forcing IEEE754 Standard**
- **Change precision of data XYZ and repeat..**

Inverse Exercise Result

- **In the case of Single precision (real*4)**
  - **There are 8 inverses not correctly computed**
  - **pgf90 and gfortran compiler spot out the wrong values for all the options**
  - **ifort compiler only when IEEE standard is forced**
- **In the case of Double precision (real*8)**
  - **There are 2 inverses not correctly computed**
  - **pgf90 and gfortran compiler spot out the wrong values for all the options**
  - **ifort compiler only when IEEE standard is forced**
  - LESSON LEARNED: be careful with intel precision

A few final warnings..

- Only about 7 decimal digits are representable in single-precision IEEE format, and about 16 in double-precision IEEE format.
  - Use double/real*8 almost everywhere
- Remember: Every time numbers are transferred from external decimal to internal binary or vice-versa, precision can be lost.
- Always use safe comparisons.
- be careful on Conversions between data types
- Don't expect identical results from two different floating-point implementations.

A final citation..

Excerpt from The Art of Computer Programming by Donald E. Knuth:

"Floating-point computation is by nature inexact, and it is not difficult to misuse it so that the computed answers consist almost entirely of 'noise'.

One of the principal problems of numerical analysis is to determine how accurate the results of certain numerical methods will be; a 'credibility gap' problem is involved here: we don't know how much of the computer's answers to believe.

Novice computer users solve this problem by implicitly trusting in the computer as an infallible authority; they tend to believe all digits of a printed answer are significant.

Disillusioned computer users have just the opposite approach, they are constantly afraid their answers are almost meaningless."

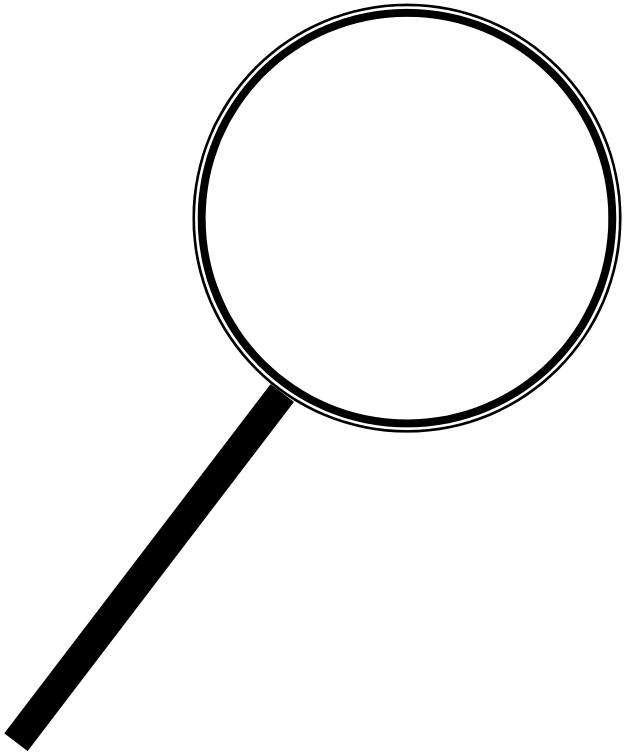
Yet another citation..

“It makes me nervous to fly on airplanes since I know they are designed using floating-point arithmetic.”

A. Householder

Further References on Floating Point Arithmetic

- Chapter 2 of the book: Numerical mathematics and Computing, Sixth Ed., by Chenney and Kinciad.
- Prof. Kahan's "Lecture Notes on IEEE 754"
  - » www.cs.berkeley.edu/~wkahan/ieeestatus/ieee754.ps
- Prof. Kahan's "The Baleful Effects of Computer Benchmarks on Applied Math, Physics and Chemistry
  - » www.cs.berkeley/~wkahan/ieee754status/baleful.ps
- *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, published in the March, 1991 issue of Computing Surveys. Copyright 1991
- Lecture of Dr. Stefano Cozzini at smr2174 ICTP activity.
- inst.eecs.berkeley.edu/~cs61c
  - – www.cs.berkeley.edu/~ddgarcia
- *The pitfall of verifying floating point computations* by D. Monniaux
  - » hal.archivesouvertes.fr/docs/00/28/14/29/PDF/floating-point-article.pdf



Questions

THANK YOU