The Abdus Salam
International Centre for Theoretical Physics

United Nations
Educational, Scientific and
Cultural Organization

IAEA
International Atomic Energy Agency

1967-5

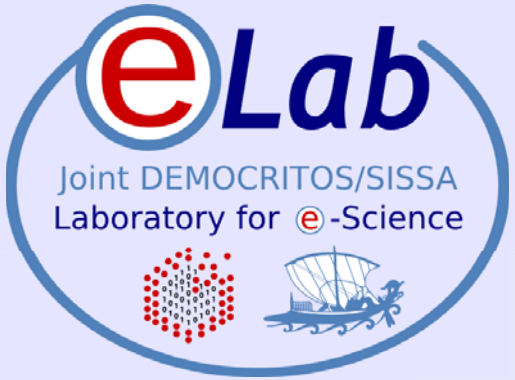
Advanced School in High Performance and GRID Computing

3 - 14 November 2008

Floating Points Numbers: How to work with them correctly and efficiently.

COZZINI Stefano
CNR-INFM Democritos
c/o SISSA
via Beirut 2-4
34014 Trieste
ITALY

Advanced School in
High Performance
and GRID Computing



Floating point numbers

Stefano Cozzini

CNR-INFM DEMOCRITOS, Trieste

Outline

- How to represent numbers on a computers.
- IEEE floating point formats
- Floating point arithmetic
- Few misconception about FP
- Pitfalls of FP arithmetic
- Role of the compilers in this game
- Some final tips to avoid (known) problems

Reality = real numbers

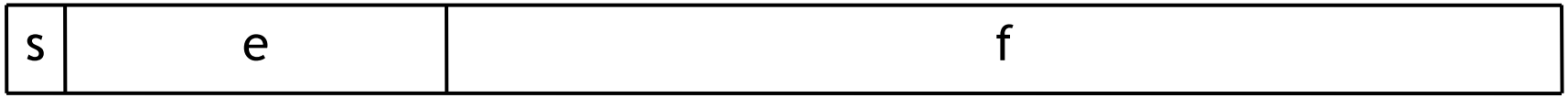
- Real number = unlimited accuracy
- How can we store a number on a computer ?
  - Binary coded decimal (BCD)
  - Rational Numbers
  - Fixed Point
  - Floating point

Floating-point representation

- floating numbers are stored using a kind of scientific notation.

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

- 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.

IEEE 754 standard for FP

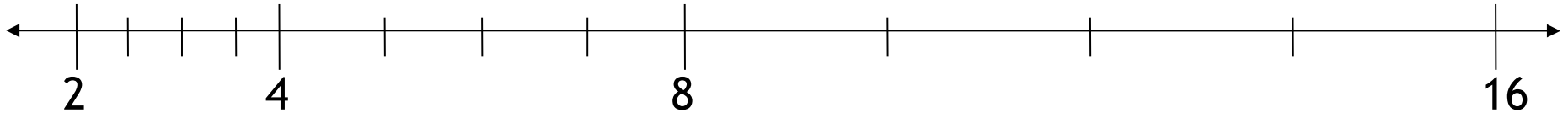
- Macheps = Machine epsilon = $2^{-\#\text{significand bits}}$ = relative error in each operation
- OV = overflow threshold = largest number
- UN = underflow threshold = smallest number

Format	# bits	#significand bits	macheps	#exponent bits	exponent range
Single	32	23+1	2^{-24} ($\sim 10^{-7}$)	8	2^{-126} - 2^{127} ($\sim 10^{+-38}$)
Double	64	52+1	2^{-53} ($\sim 10^{-16}$)	11	2^{-1022} - 2^{1023} ($\sim 10^{+-308}$)
Double Extended (*80 bits on all Intel machines*)	>=80	>=64	<=2^{-64}($\sim 10^{-19}$)	>=15	2^{-16382} - 2^{16383} ($\sim 10^{+-4932}$)

- 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?

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!

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..

Floating Point Arithmetic

- Representable numbers:
  - The way the numbers are stored
- Operations: (The way the number are handled)
  - arithmetic: +,-,x,/,...
  - comparison (<, =, >)
  - conversion between different formats - short to long FP numbers, FP to integer, etc.
  - exception handling - what to do for 0/0, 2*largest_number, inexact etc.
  - binary/decimal conversion - for I/O, when radix is not 10.
- Language/library support is required for all these operations.

error 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 point 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).

Floating Point Arithmetic: A Little History

- 1960s-1970s: Each computer handled FP arithmetic differently:
  - Format, accuracy, rounding mode and exception handling all differed.
  - It was very difficult to write portable and reliable technical software.
- 1982: IEEE-754 standard defined. First implementation: Intel 8087.
- 1985 the IEEE 754 standard was adopted.
  - Having a standard at least ensures that all compliant machines will produce the same outputs for the same program.
- 2000: IEEE FP arithmetic is now almost universally implemented in general purpose computer systems.

Fortran/C intrinsic data types for IEEE standards

- Fortran:
  - REAL*4 / REAL (32 bit)
  - REAL*8 /DOUBLE PRECISION (64 bit)
  - REAL*16/ QUADRUPLE PRECISION (128 bit: not always present and implemented)
- C/C++
  - Float (32 bit)
  - Double (64 bit)
  - Double extended   (80 bit on all commodity hardware (INTEL/AMD etc)

Misconception 1

- Floating-point arithmetic is fuzzily defined, programs involving floating-point should not be expected to be deterministic.
  - Since 1985 there is a IEEE standard for floating-point arithmetic.
  - Everybody agrees it is a good thing and will do his best to comply
  - ... but full compliance requires more cooperation between processor, OS, languages, and compilers than the world is able to provide.
  - Besides full compliance has a cost in terms of performance.
- Floating-point programs are deterministic, but should not be expected to be spontaneously portable...

Misconception 2

- I need 3 significant digits in the end, a double holds 15 decimal digits, therefore I shouldn’t worry about precision.

⊖ You can destroy 14 significant digits in one subtraction

⊖ it will happen to you if you do not expect it

⊕ It is relatively easy to avoid if you expect it

Misconception 3

- All floating-point operations involve a (somehow fuzzy) rounding error.
  - Many are exact, we know who they are and we may even force them into our programs
  - Since the IEEE-754 standard, rounding is well defined, and you can do maths about it

PITFALL: addition is not associative

$x = -1.5 \times 10^{38}$
$y = 1.5 \times 10^{38}$
$z = 1.0$

$$x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$$
$$= -1.5 \times 10^{38} + 1.5 \times 10^{38}$$
$$= 0$$

$$(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$$
$$= 0 + 1.0$$
$$= 1.0$$

EXERCISE ON SUM NUMBERS IN THE LAB #2

PITFALL: cancellation

- Cancellation: if you subtract numbers which were very close

  (example: 1.2345e0 - 1.2344e0 = 1.0000e-4)

  - you loose significant digits (and get meaningless zeroes)
  - although the operation is exact! (no rounding error)
- Problems may arise if such a subtraction is followed by multiplications or divisions
- You may get meaningless digits in your result

PITFALL: some numbers are not exactly representable..

- 1/3 is not exactly representable
- 0.01 is NOT exactly representable
- Question to be answered during lab 2
  - how many 1/N (inverse) are not exactly representable in the range [1, 100] ?

EXERCISE ON INVERSE S IN THE LAB #2

so do not expect exact results..

0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+
0.1 != 1.0

- The literal "0.1" doesn’t equal 0.1
- Limited precision of floating-point implies roundoff

- More generally, exact results also fail from
  - Limited range (overflow, underflow)
  - Special values

horrible story #1 0.10

- During the Gulf War in 1991, a U.S. Patriot missile failed to intercept an Iraqi Scud missile, and tens of peoples were killed.
- A later study determined that the problem was caused by the inaccuracy of the binary representation of 0.10.
  - The Patriot incremented a counter once every 0.10 seconds.
  - It multiplied the counter value by 0.10 to compute the actual time.
- However, the (24-bit) binary representation of 0.10 actually corresponds to 0.099999904632568359375, which is off by 0.000000095367431640625.
- This doesn’t seem like much, but after 100 hours the time ends up being off by 0.34 seconds—enough time for a Scud to travel 500 meters!

http://www.mc.edu/campus/users/travis/syllabi/381/patriot.htm

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
- compilers should respect the semantic of the language.

About the semantic..

- the semantic of most recent languages is to respect your parentheses:
- if you write (a + b) + c the compiler should not replace it with a + (b + c), unless it can prove that both computations always yield the same result. Even if it would be faster!
- if you write r := b - ((a + b)- a) ; the compiler shouldn’t replace it with r:=0 ;

Again on compilers

- 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..

Compiler IEEE flags

- Gnu suite:
- -ffloat-store

.. a few programs rely on the precise definition of IEEE floating point. Use -ffloat-store for such programs, after modifying them to store all pertinent intermediate computations into variables.

- PGI:

-Kieee -Knoieee (default) Perform floating-point operations in strict conformance with the IEEE 754 standard. Some optimizations are disabled with -Kieee, and a more accurate math library is used. The default -Knoieee uses faster but very slightly less accurate methods.

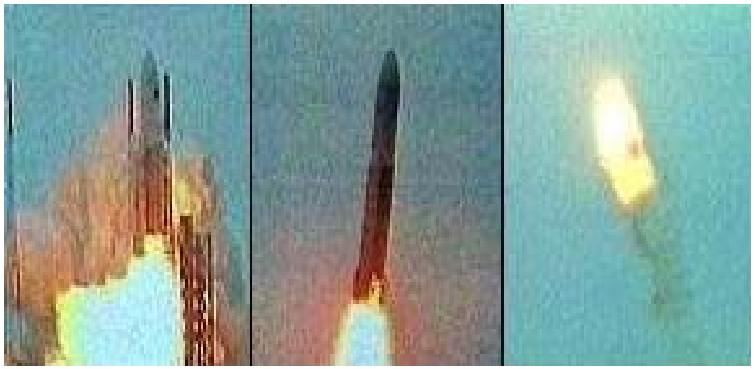
- intel:

-mp Maintains floating-point precision (while disabling some optimizations). The -mp option restricts optimization to maintain declared precision and to ensure that floating-point arithmetic conforms more closely to the ANSI* and IEEE standards. This is the same as specifying -fltconsistency or -mieee-fp.

final tip #1: be careful on float -> integer conversion

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

horrible story #2 : 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

Final tip #2 : single vs double precision

- 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

the last slide

- There are many situations in floating-point calculations that can generate results that are surprising to the programmer. There are four general rules that should be followed:

- 1. In a calculation involving both single and double precision, the result will not usually be any more accurate than single precision. If double precision is required, be certain all terms in the calculation, including constants, are specified in double precision.

- 2. Never assume that a simple numeric value is accurately represented in the computer. Most floating-point values can't be precisely represented as a finite binary value.

- 3. Never assume that the result is accurate to the last decimal place. There are always small differences between the "true" answer and what can be calculated with the finite precision of any floating point processing unit.

- 4. Never compare two floating-point values to see if they are equal or not-equal. This is a corollary to rule 3. There are almost always going to be small differences between numbers that "should" be equal. Instead, always check to see if the numbers are nearly equal. In other words, check to see if the difference between them is very small or insignificant.

References on Floating Point Arithmetic

- *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, published in the March, 1991 issue of Computing Surveys. Copyright 1991
- Available as link on our wiki
- 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

Final citation

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

A. Householder

- Any questions ?