Floating Point Arithmetic* is not Real

Tim Mattson

Acknowledgements: I borrowed heavily from lectures on floating point arithmetic by Ianna Osborne and Wahid Redjeb.

* specifically, numbers and the associated arithmetic defined by the IEEE 754 standard.

Should we trust computer arithmetic?

Sleipner Oil Rig Collapse (8/23/91). Loss: **$700 million.**

$1.6 Billion in 2024 dollars

See http://www.ima.umn.edu/~arnold/disasters/sleipner.html

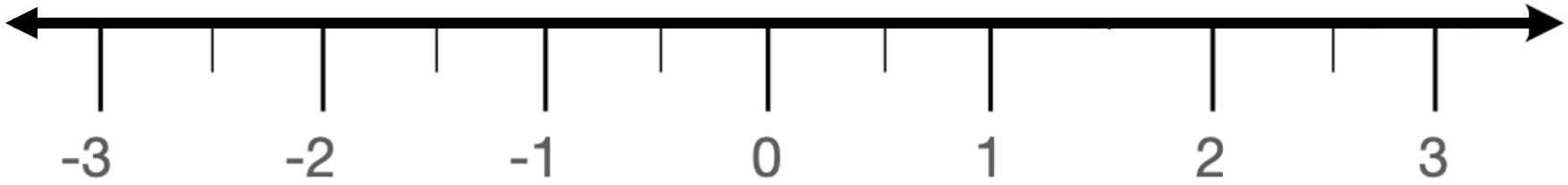
Linear elastic model using NASTRAN underestimated shear stresses by 47% resulted in concrete walls that were too thin.

NASTRAN is the world's most widely used finite element code … in heavy use since 1968

Outline

- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
- Wrap-up/Conclusion
- Additional Content
  - Numerical Analysis
  - Changing numbers of bits
  - Compiler options
  - A Few Exercises
    - Exercises
    - Solutions

Numbers for Humans

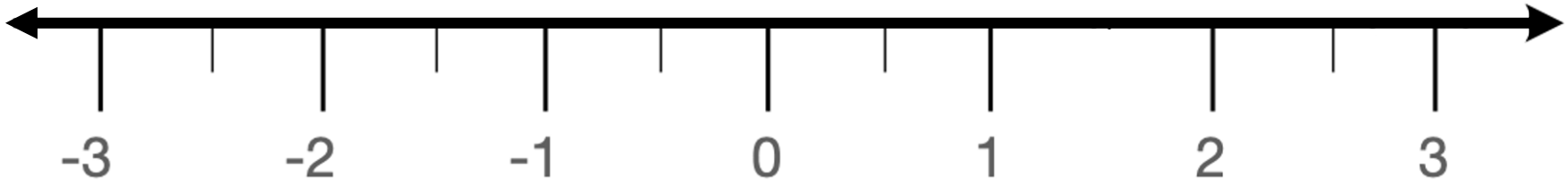


$\mathbb{R}$

Real Numbers: viewed as points on a line … pairs of real numbers can be arbitrarily close

Numbers for Humans

Arithmetic over Real Numbers



$\mathbb{R}$

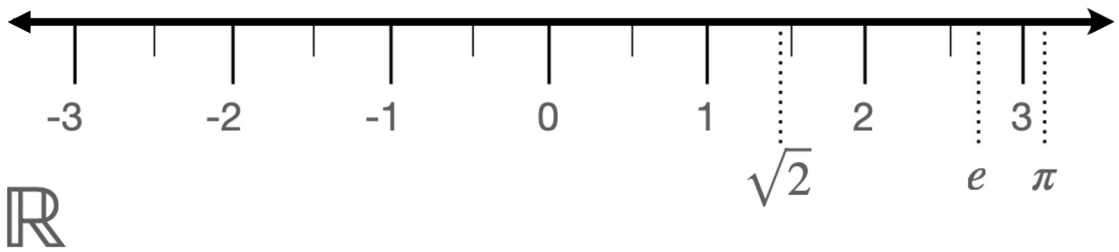
For the arithmetic operators, real numbers define a **closed set** … for well defined operations and any input real numbers, the arithmetic operation returns a real number.

A few key properties of Real Arithmetic:

- Commutative over addition and multiplication: $(a+b) = (b+a)$ $a*b = b*a$
- Associative: $(a+b)+c = a+(b+c)$ $(a*b)*c = a*(b*c)$
- Multiplication distributes over addition: $c * (a+b) = c*a + c*b$

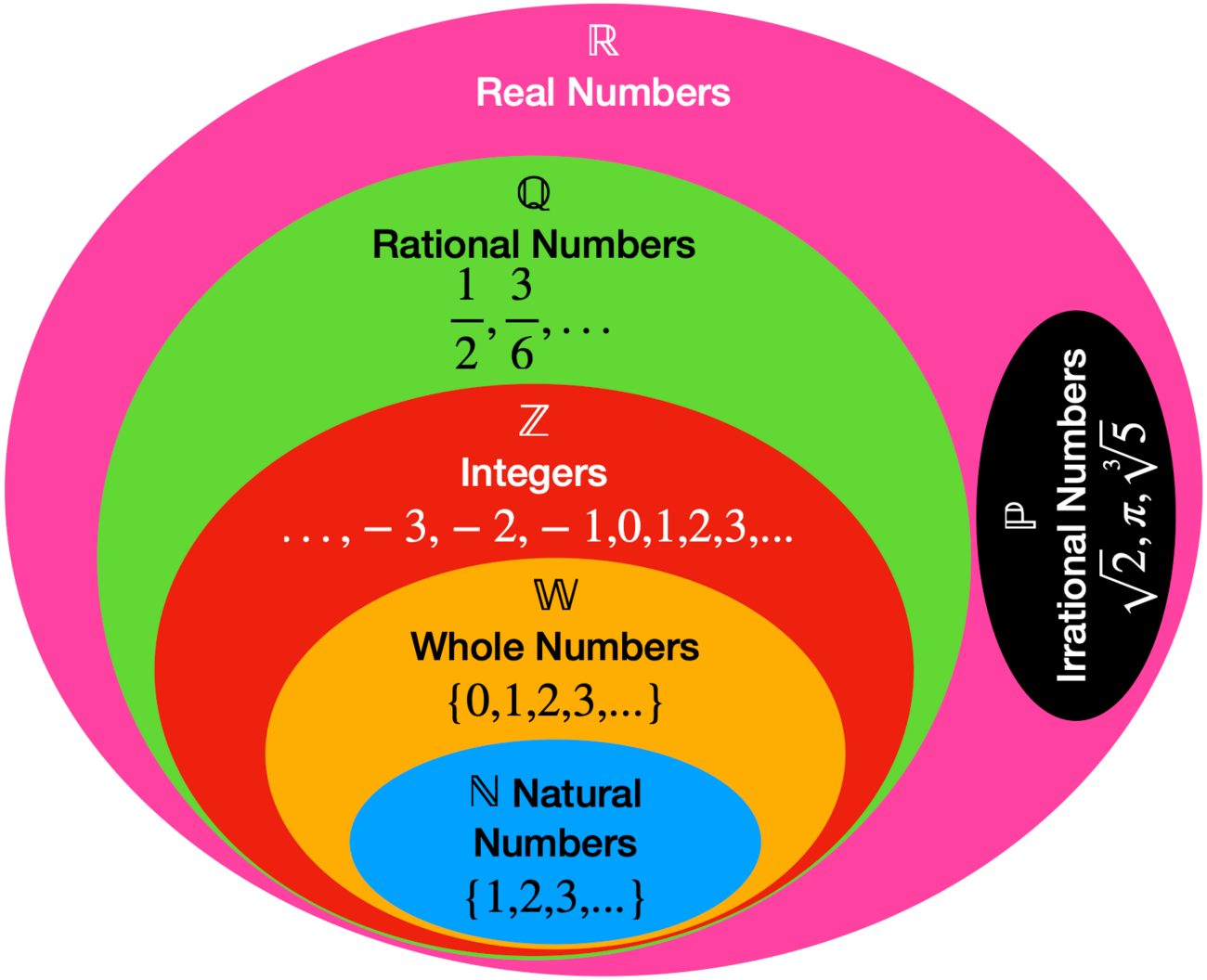
Numbers for Humans

People use many different kinds of numbers



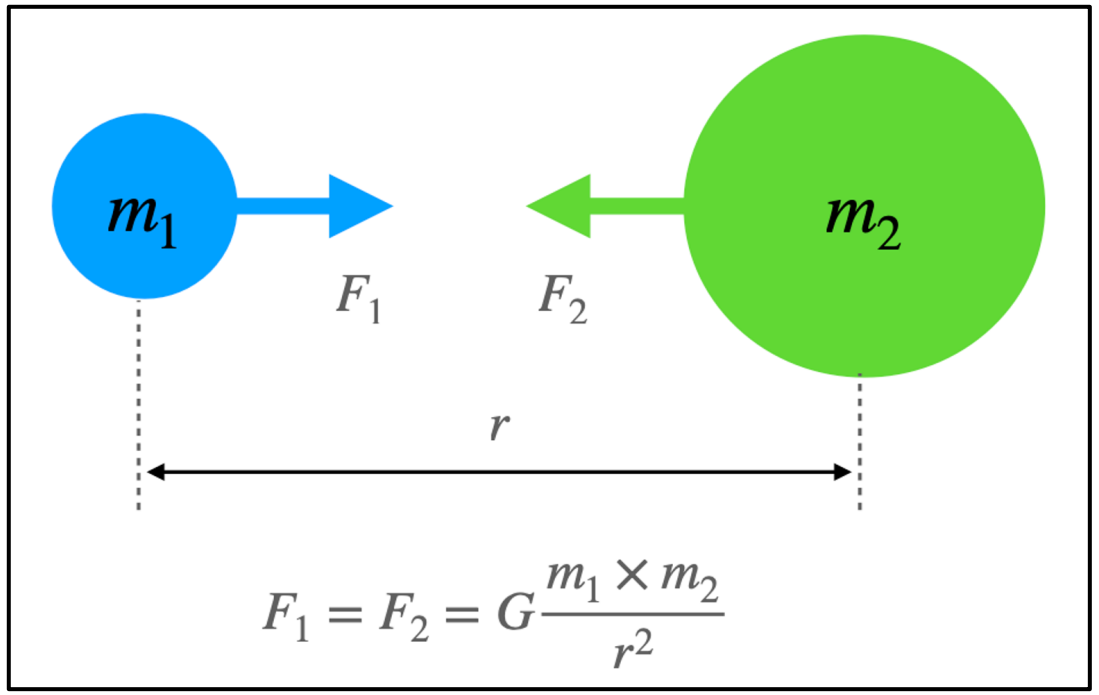
These different kinds of numbers are contained within the set of real numbers

- **Rational numbers**: Ratios of integers
- **Integers**: equally spaced numbers without a fractional part
- **Whole numbers**: Positive integers and zero
- **Natural numbers**: Positive integers and not zero
- **Irrational numbers**: numbers that cannot be represented as a ratio of integers



Based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

Numbers for Humans: Example



$$G \approx 0.00000000006674 \ \frac{m^3}{kg * s^2}$$

Scientific Notation

$$0.00000000006674 \longrightarrow 6.674 \times 10^{-11}$$

The exponent tells us how far to "float" the decimal point.

significand *exponent*

$$G \approx 6.674 \times 10^{-11} \ \text{m}^3\cdot\text{kg}^{-1}\cdot\text{s}^{-2}$$

radix

Based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

Numbers for Humans

How do we represent Real Numbers?

significand *exponent*

$$G \approx 6.674 \times 10^{-11} \ \mathrm{m^3 \cdot kg^{-1} \cdot s^{-2}}$$

radix

$$G \approx (6 \cdot 10^{0} + 6 \cdot 10^{-1} + 7 \cdot 10^{-2} + 4 \cdot 10^{-3}) \cdot 10^{-11}$$

We can generalize the above to any real number as …

$$x = (-1)^{sign} \sum_{i=0}^{\infty} d_i \boldsymbol{b}^{-i} \boldsymbol{b}^{exp}$$

… where $\boldsymbol{b}$ is the **radix**

$sign \in \{0,1\}, \quad b \geq 2, \quad d_i \in \{0, \ldots, (b-1)\}, \quad d_0 > 0 \ when \ x \neq 0, \quad \mathrm{b}, i, \exp \in [integer]$

Based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

Numbers for Humans

How do we represent Real Numbers?

significand *exponent*

$$G \approx 6.674 \times 10^{-11} \ \mathrm{m^3 \cdot kg^{-1} \cdot s^{-2}}$$

radix

What about numbers for computers?

$$G \approx (6 \cdot 10^{0} + 6 \cdot 10^{-1} + 7 \cdot 10^{-2} + 4 \cdot 10^{-3}) \cdot 10^{-11}$$

We can generalize the above to any real number as …

$$x = (-1)^{sign} \sum_{i=0}^{\infty} d_i b^{-i} b^{exp}$$

Human's deal nicely with ∞. Computers do not.

$sign \in \{0,1\}, \quad b \geq 2, \quad d_i \in \{0, \ldots, (b-1)\}, \quad d_0 > 0 \ when \ x \neq 0, \quad b, i, \exp \in [integer]$

Human's like a radix = 10. Which b is best for a computer?

Based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

Numbers for Computers

Computers work with a restricted subset of real numbers…

Finite precision … restricted to N digits.

N is tied to the length of a "word" in a computer's architecture. This is typically the width of the registers in a microprocessor's register file.

$$x = (-1)^{sign} \sum_{i=0}^{N} d_i b^{-i} b^{exp}$$

$$sign \in \{0,1\}, \quad b \geq 2, \quad d_i \in \{0, \ldots, (b-1)\}, \quad d_0 > 0 \; when \; x \neq 0, \quad b, i, \exp \in [integer]$$

Which radix (b) is best for a computer?

Binary has $d_i \in \{0,1\}$. Naturally maps onto representation as transistors used to implement computer logic.

Decimal has $d_i \in \{0, \ldots, 9\}$. Requires four bits per digit … which wastes space (since four bits can encode {0,…,15}).

Exercise: Playing with “numbers for computers”

- You are a software engineer working on a device that tracks objects in time and space.
- The device increments time in “clock ticks” of 0.01 seconds.
- Write a program that inputs an integer for a maximum number of clock ticks. The program tracks time by **accumulating clock-ticks**. N is typically large … around 100 thousand. Output from the function is elapsed seconds expressed as a float.
  - Assume you are working with an embedded processor that does not support the type double.
  - This is part of an interrupt driven, real time system, hence track “time” not “number of ticks” since this time may be needed at any moment.
- What does your program generate for large N?

Accumulating clock ticks (0.01): Solution

```
#include <stdio.h>
#define time_step  0.01f

float CountTime(int Count)
{
    float sum     = 0.0f;

    for (int i=0; i<Count;i++)
        sum += time_step;

    return sum;
}
int main()
{
   int Count = 500000;
   float time_val;

   time_val = CountTime(Count);
   printf(" sum = %f or %f\n",time_val,time_step*Count);
}
```

```
% gcc -O0 hundreth.c
% ./a.out
 sum = 4982.411132. or 5000.000000
%
```

Converting a decimal number (0.01) to fixed point binary

0.01 is equal to $\frac{1}{100}$.

The fraction $\frac{1}{2^N}$ nearest but less than or equal to $\frac{1}{100}$ is $\frac{1}{128}$ (N=7)	$0.01_{10} \approx 0.0000001_2$
The remainder $\frac{1}{100} - \frac{1}{128} = \frac{7}{3200} \approx \frac{1}{457}$. The fraction $\frac{1}{2^N}$ nearest but less than or equal to this remainder is $\frac{1}{512}$ (N=9)	$0.01_{10} \approx 0.000000101_2$
The remainder $\frac{7}{3200} - \frac{1}{512} = \frac{3}{12800} \approx \frac{1}{4266}$. The fraction $\frac{1}{2^N}$ nearest but less than or equal to this remainder is $\frac{1}{8196}$ (N=13)	$0.01_{10} \approx 0.0000001010001_2$

- Continuing to 32 bits we get 0.00000010100011110101110000101000… but it's still not done.
- The denominator of the number 1/100 includes a relative prime (5) to the radix of binary numbers (2). Hence, there is no way to exactly represent 1/100 in binary!

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\frac{1}{2^N}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{254}$	$\frac{1}{512}$	$\frac{1}{1024}$	$\frac{1}{2048}$	$\frac{1}{4096}$	$\frac{1}{8196}$	$\frac{1}{16384}$	$\frac{1}{32768}$	$\frac{1}{65536}$	$\frac{1}{131072}$

Real numbers on a computer are represented as finite precision numbers

- **Conclusion**: Many decimal numbers do not have an exact representation as binary numbers.
  - You can have computations where the answer does NOT have an exact binary representation … in other words, **fixed precision arithmetic is NOT a closed set**.

```
float c, b = 1000.2f;
c = b - 1000.0;
printf (" %f", c);
```

Output: 0.200012

- The best we can hope for is that the computer does the computation "exactly" then rounds to the nearest binary number.

These numbers with finite precision, fractional parts, and exponents are called **floating point numbers**.

Real numbers on a computer are represented as finite precision numbers

- **Conclusion**: Many decimal numbers do not have an exact representation as binary numbers.
  - You can have computations where the answer does NOT have an exact binary representation … in other words, **fixed precision arithmetic is NOT a closed set**.

```
float c, b = 1000.2f;
c = b - 1000.0;
printf (" %f", c);
```

Output: 0.200012

- The best we can hope for is that the computer does the computation "exactly" then rounds to the nearest binary number.

These numbers with finite precision, fractional parts, and exponents are called **floating point numbers**.

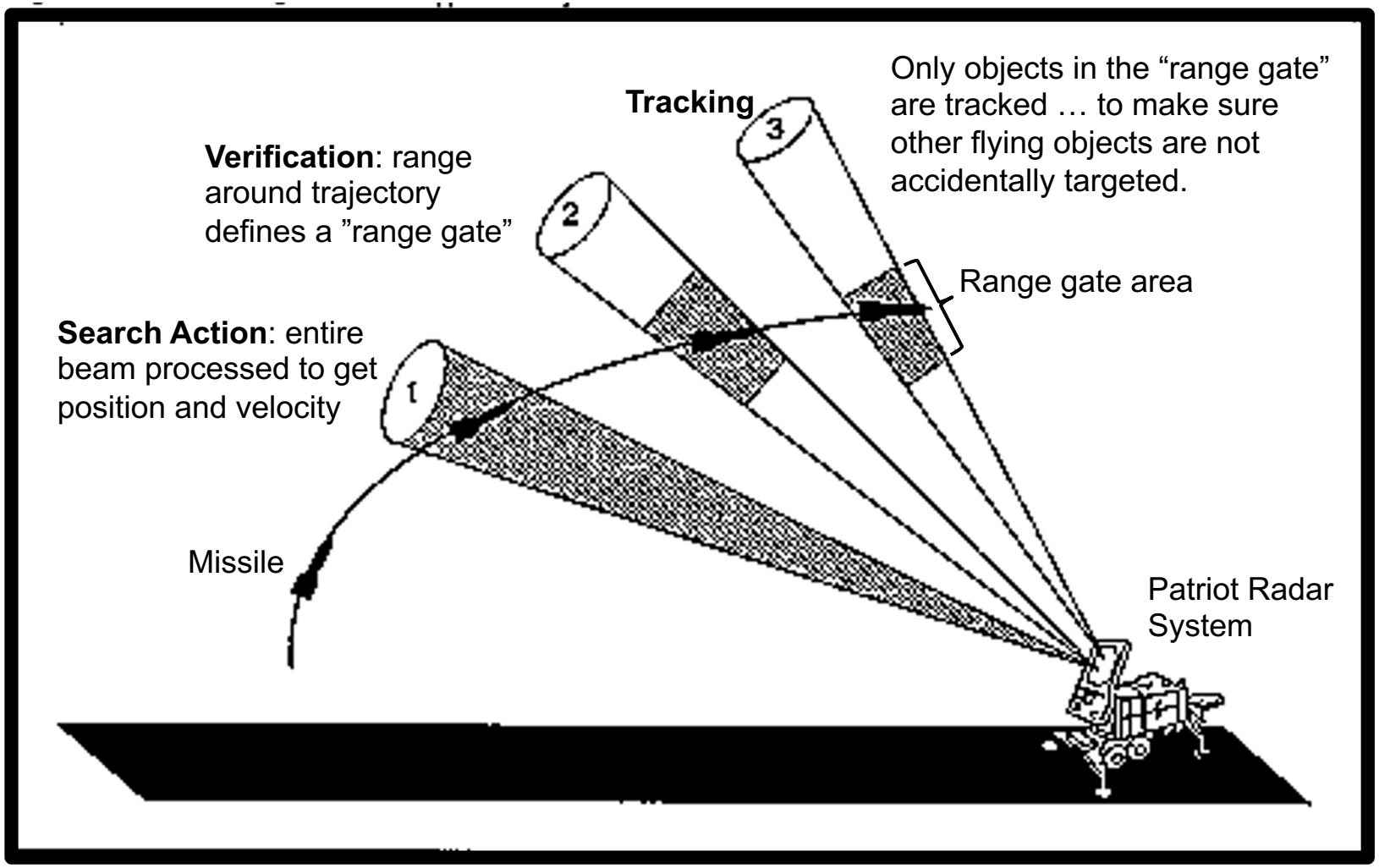
Patriot Missile system

A Patriot missile system failed to stop a scud missile from hitting an Army barracks *(2/25/91)* . 28 U.S. soldiers

See http://www.fas.org/spp/starwars/gao/im92026.htm

Patriot missile system: how it works

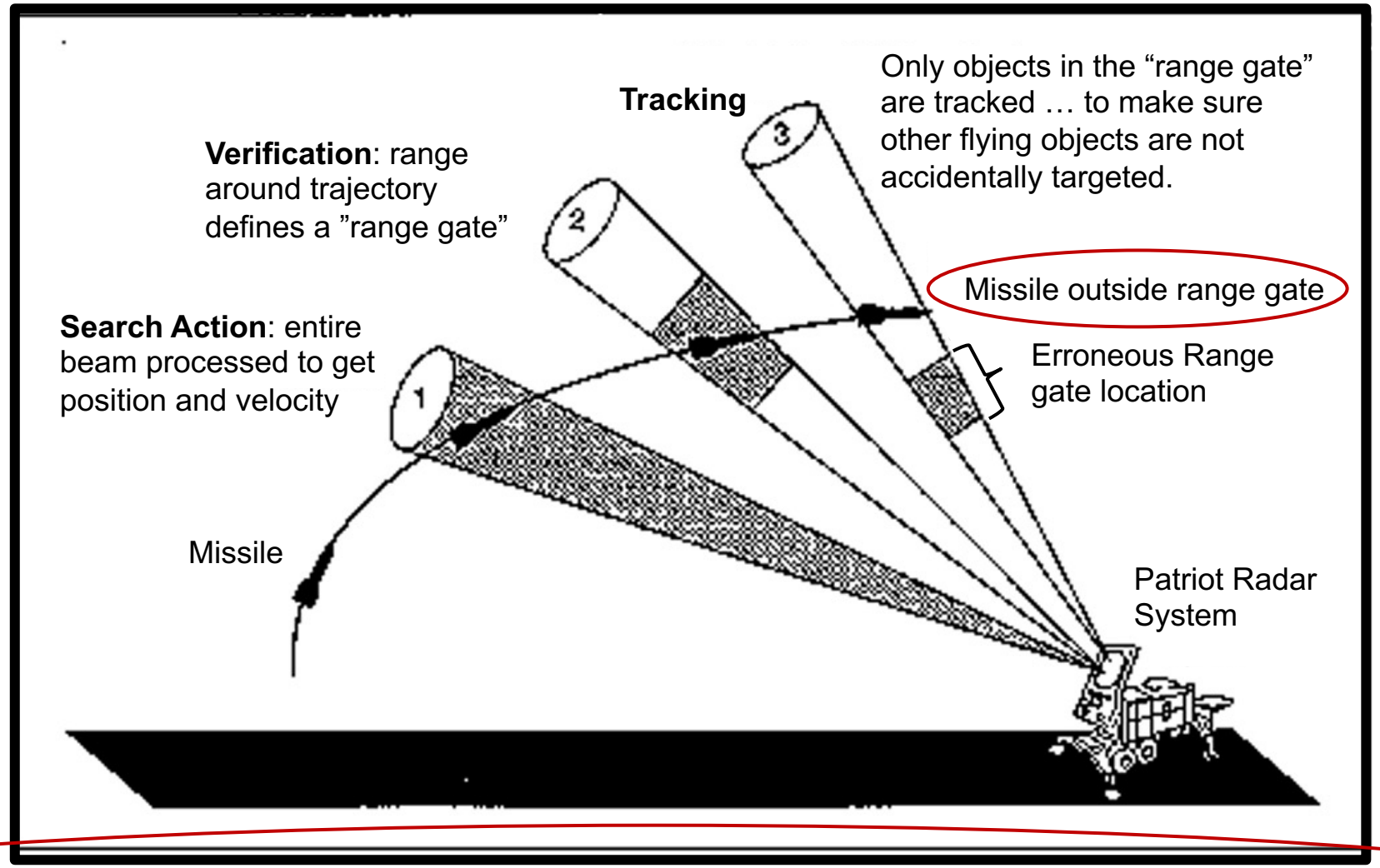
Incoming object detected as an enemy missile due to properties of the trajectory. Velocity and position from Radar fixes trajectory



24 bit clock counter defines time. Range calculations defined by real arithmetic, so convert to floating point numbers.

Patriot missile system: Disaster Strikes

Incoming object detected as an enemy missile due to properties of the trajectory. Velocity and position from Radar fixes trajectory



Accumulating clock-ticks (int) by the float representation of 0.01 led to an error of 0.3433 seconds after 100 hours of operation which, when you are trying to hit a missile moving at Mach 5, corresponds to an error of 687 meters

Exercise: Playing with “numbers for computers”

- You are a software engineer working on a device that tracks objects in time and space.
- The device increments time in “clock ticks” of 0.01 seconds. **Propose (and test) a value for the clock tick that makes the program work.**
- Write a program that inputs an integer for a maximum number of clock ticks. The program tracks time by accumulating clock-ticks. N is typically large … around 100 thousand. Output from the function is elapsed seconds expressed as a float.
  - Assume you are working with an embedded processor that does not support the type double.
  - This is part of an interrupt driven, real time system, hence track “time” not “number of ticks” since this time may be needed at any moment.
- What does your program generate for large N?

Exercise: Playing with “numbers for computers”

- You are a software engineer working on a device that tracks objects in time and space.
- The device increments time in “clock ticks” of 0.01 seconds. **Propose (and test) a value for the clock tick that makes the program work.**
- Wr[illegible]acks tim[illegible]the function is elapsed seconds expressed as a float.
  - Assume you are working with an embedded processor that does not support the type double.
  - This is part of an interrupt driven, real time system, hence track “time” not “number of ticks” since this time may be needed at any moment.

```
> ./a.out
 dt = 0.0078125000000000000.  // dt=1.0/128.0 … one over a power of two
 sum = 39062.500000000000000000, dt*Count=39062.500000000000000000
```

- What does your program generate for large N?

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set

Outline

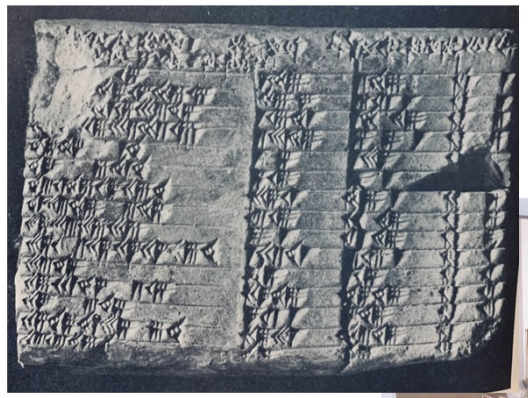
- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
- Wrap-up/Conclusion
- Additional Content
  - Numerical Analysis
  - Changing numbers of bits
  - Compiler options
  - A Few Exercises
    - Exercises
    - Solutions

PRINCETON UNIVERSITY

iris hep

Floating-point Arithmetic Timeline

"…the next generation of application programmers and error analysts will face new challenges and have new requirements for standardization. Good luck to them!"
https://grouper.ieee.org/groups/msc/ANSI_IEEE-Std-754-2019/background/ieee-computer.pdf



Babylonians worked with floating-point sexagesimal numbers

- Average calculation speed: addition – 0.8 seconds, multiplication – 3 seconds[1]
- Arithmetic unit: Binary floating-point, 22-bit, add, subtract, multiply, divide, square root[1]

Zeus Z3

DEC PDP-11

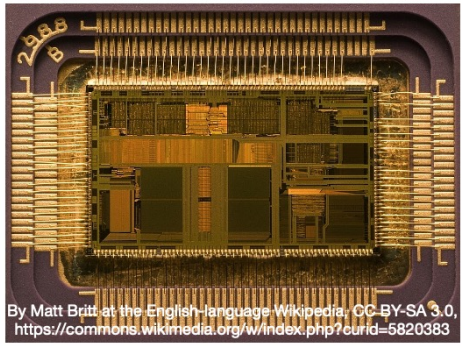
1970's boom of minicomputers

IEEE p745

IEEE 754-1985

Intel 80486

the first tightly-pipelined[c] x86 design as well as the first x86 chip to include more than one million transistors. It offered a large on-chip cache and an integrated floating-point unit.



The floating point unit fully conforms to the ANSI/ IEEE standard 754-1985 for floating point arithmetic.

"new kinds of computational demands might eventually encompass new kinds of standards, particularly for fields like artificial intelligence, machine vision and speech recognition, and machine learning. Some of these fields obtain greater accuracy by processing more data faster rather than by computing with more precision – rather different constraints from those for traditional scientific computing."

1750 B.C. | 1941 | 1960's a mainframe era | 1964 IBM 360, CDC 6600, DEC 10 | 1976 1977 | 1985 | 1989 | IEEE 754-1987 | IEEE 754-2008 | IEEE 754-2019 | you are here | IEEE 754-2029

- each hardware manufacturer had its own type of floating point
- different machines from the same manufacturer might have different types of floating point
- when floating point was not supported in the hardware, the different compilers emulated different floating point types

Intel began to design a floating-point co-processor for its i8086/8 and i432 microprocessors

binary and decimal floating-point arithmetic

subjected to review at least every 10 years

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

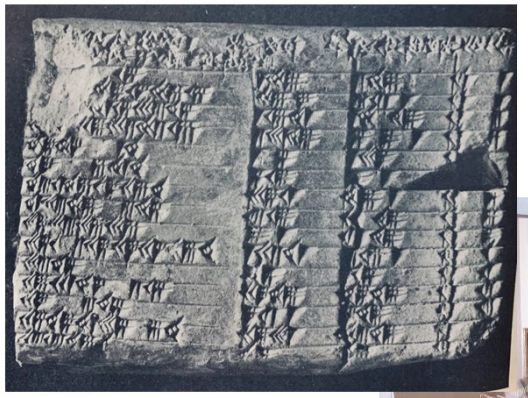
PRINCETON UNIVERSITY

Summarizing the key points …

iris hep

Floating-point Arithmetic Timeline

"…the next generation of application programmers and error analysts will face new challenges and have new requirements for standardization. Good luck to them!"
https://grouper.ieee.org/groups/msc/ANSI_IEEE-Std-754-2019/background/ieee-computer.pdf

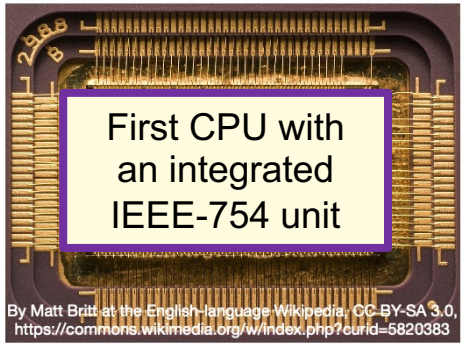


Babylonians worked with floating-point sexagesimal numbers

- Average calculation speed: addition – 0.8 seconds, multiplication – 3 seconds[1]
- Arithmetic unit: Binary floating-point, 22-bit, add, subtract, multiply, divide, square root[1]

Intel 80486

the first tightly-pipelined[c] x86 design as well as the first x86 chip to include more than one million transistors. It offered a large on-chip cache and an integrated floating-point unit.



The floating point unit fully conforms to the ANSI/ IEEE standard 754-1985 for floating point arithmetic.

"new kinds of … might eventua… kinds of stand… fields like arti… machine visio… recognition, a… Some of these… accuracy by p… faster rather th… more precisio… constraints fro… scientific computing."

IEEE-754 floating point is born … thanks to a team led by William Kahan

First CPU with an integrated IEEE-754 unit

IEEE-754 floating point continues to evolve … next version expected in 2029

The concept of floating point numbers is very old (1750 BC)

1941 Zeus Z3

1960's inframe era

DEC

1970's boom of minico…

CDC 6600

1976 1977

1985

1989

you are here

IEEE 754-1987

IEEE 754-2008 IEEE 754-2019 IEEE 754-2029

The era of floating point chaos

Intel produces the first chip to support IEEE-754 in 1980 (the 8087 coprocessor)

…nufacturer had its own typ…
…es from the same manufactu…
…point
…int was not supported in the hardware, the different compilers emulated different floating point types

…sign a floating-point … its i8086/8 and i432

binary and decimal floating-point arithmetic

subjected to review at least every 10 years

Floating Point Number systems

Computers work with finite precision, floating point numbers …

$$x = (-1)^{sign} \sum_{i=0}^{p} d_i b^{-i} b^e = \pm d_0.d_1 \ldots d_{p-1} \times b^e$$

$$sign \in \{0,1\}$$

$$d_i \in \{0, \ldots, (b-1)\}$$

$$e_{min} \leq e \leq e_{max}$$

$b \geq 2$	The radix … usually 2 or 10 (but occasionally 8 or 16)
$p \geq 1$	The precision … the number of digits in the significand
e_{max}	The largest exponent
e_{min}	The smallest exponent (generally 1 – e_{max})

These four numbers define a unique set of floating point numbers … written as F(b, p, e_{max}, e_{min})

Floating Point Number systems: Normalized numbers

Consider representations of the decimal number 0.1

$$1.0\times10^{-1}, \qquad 0.1\times10^{0}, \qquad 0.01\times10^{1}$$

- These are all the same number, just represented differently depending on the choice of exponent.
- That ambiguity is confusing, so we require that $d_0 \neq 0$ so numbers between b^{min} and b^{max} have a single unique representation.
- We call these normalized floating point numbers $F^*(b, p, e_{max}, e_{min})$

$$x = (-1)^{sign}\sum_{i=0}^{p} d_i b^{-i} b^{e} = \pm d_0 1.d_1 \ldots d_{p-1}\times b^{e}$$

$$sign \in \{0,1\}$$

$$d_i \in \{0, \ldots, (b-1)\}$$

$$d_0 \neq 0$$

$$e_{min} \leq e \leq e_{max}$$

- $x = 0 \ \ and \ x < b^{e_{min}}$ do not have normalized representations.

$b \geq 2$	The radix … usually 2 or 10 (but occasionally 8 or 16)
$p \geq 1$	The precision … the number of digits in the significand
e_{max}	The largest exponent
e_{min}	The smallest exponent (generally 1 – e_{max})

PRINCETON UNIVERSITY

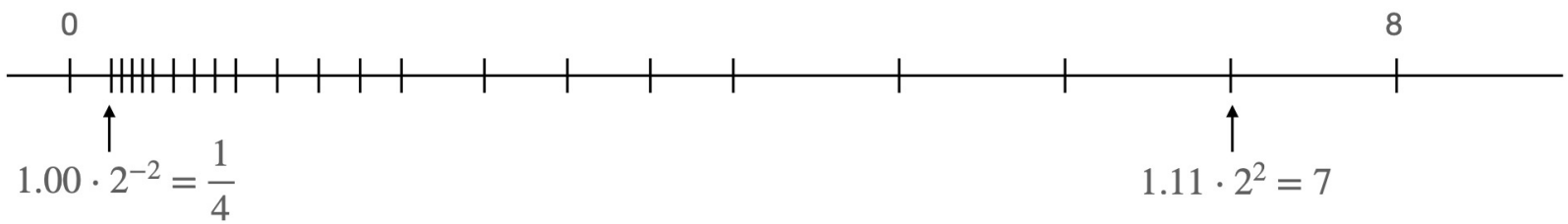
iris hep

Normalized Representation

$F^*(2,3,-2,2)$

$d_0 . d_1 d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Equivalent decimal values for all patterns of normalized binary digits and exponents



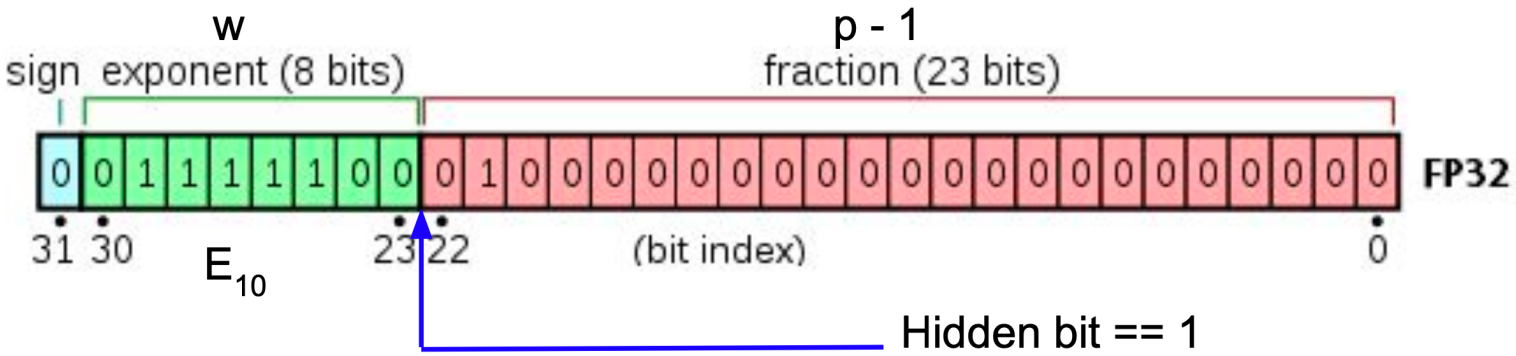
F*(Radix, Precision, emin, emax)

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

Outline

- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
- Wrap-up/Conclusion
- Additional Content
  - Numerical Analysis
  - Changing numbers of bits
  - Compiler options
  - A Few Exercises
    - Exercises
    - Solutions

IEEE 754 Floating Point Numbers

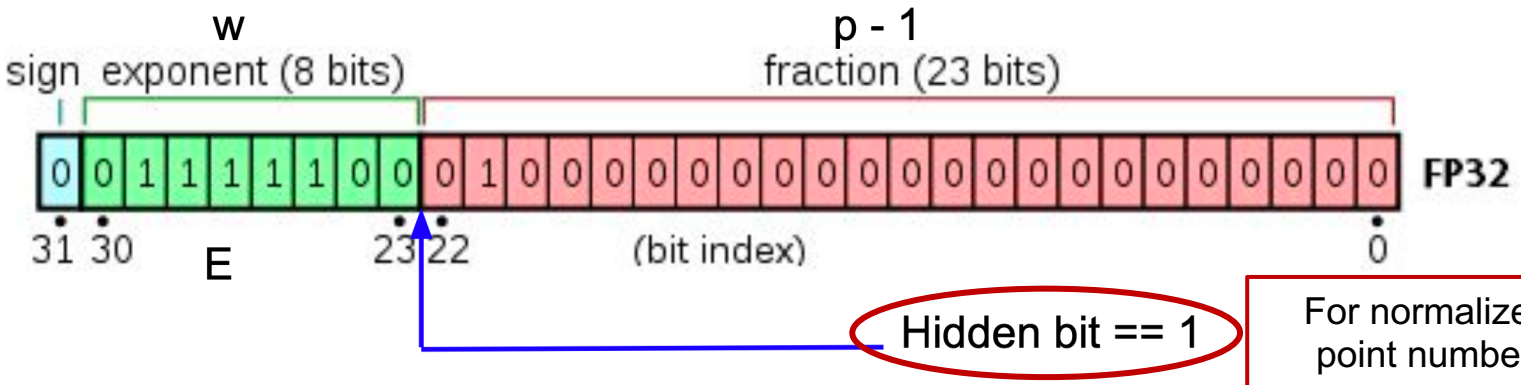


IEEE Name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

- Exponent: $E = e - e_{min} + 1$, *w* bits
- $e_{max} = -e_{min} + 1$

Source: Wahid Redjeb, ESC'22

IEEE 754 Floating Point Numbers



For normalized, binary floating point numbers, the lead bit is "always" one so there is no reason to store it. The bit is "hidden"

IEEE Name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

- Exponent: $E = e - e_{min} + 1$, *w* bits
- $e_{max} = -e_{min} + 1$

Source: Wahid Redjeb, ESC'22

Exceptions and associated special values

- Certain situations outside “normal” behavior are defined as **Exceptions**. Two cases:
  1. An exception occurs, a result is returned and the computation proceeds. This is the typical case.
  2. The exception is signaled. An optional trap function may be invoked. Trapping can be set through compiler switches but can seriously slow down code. Thbis very rarely done … except by professionals writing low-level math libraries.
- Associated with the exceptions, are a number of “special values”:

Regular normalized floating point numbers.

Numbers too small to normalize.

0’s and ∞‘s have signs, to work with limits in math.

Not a Number (undefined math such as 0/0).

The special value	exponent	fraction
$1.f \times 2^e$	$e_{min} \leq e \leq e_{max}$	Any pattern of 1’s and 0’s
$0.f \times 2^{e_{min}}$	All 0’s $(e_{min} - 1)$	$f \neq 0$
± 0	All 0’s $(e_{min} - 1)$	$f = 0$
$\pm \infty$	All 1’s $(e_{max} + 1)$	$f = 0$
NaN	All 1’s $(e_{max} + 1)$	$f \neq 0$

- The Exceptions defined by IEEE 754 include the following
  - **Underflow**: The result is too small to be represented as a normalized float. Produces a signed zero or a denormalized float.
  - **Overflow**: The result is too large to be represented by a normalized float. Produces a signed infinity.
  - **Divide-by-zero**: A float is divided by zero. The appropriate infinity is returned.
  - **Invalid**: The operation or its result is ill-defined (such as 0.0/0.0). A NaN is returned.
  - **Inexact**: The result of the floating point operation is not exact and must be rounded. The rounded result is returned

More about NaNs

- Here are the cases where a NaN can be produced.

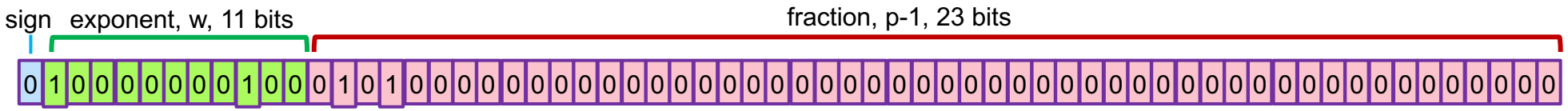
Operation	NaN produced by …
$+$	$\infty + (-\infty)$
$\times$	$0 \times \infty$
$/$	$0/0, \qquad \infty/\infty$
REM	$x\ REM\ 0, \qquad \infty REM\ y$
$\sqrt{}$	$\sqrt{x} \quad when \quad x < 0$

- There are actually two kinds of NaNs:
  - A quiet NaN … A NaN condition is identified but no further information is provided. The fraction bits are all zero other than the first one.
  - A signaling NaN … Additional implementation dependent information is encoded into the fraction bits.

Source: What every computer computer scientist should know about floating point arithmetic, David Goldberg, Computing Surveys, 1991 https://dl.acm.org/doi/pdf/10.1145/103162.103163

Writing IEEE 754 numbers in binary

- The number 42.0 written in binary

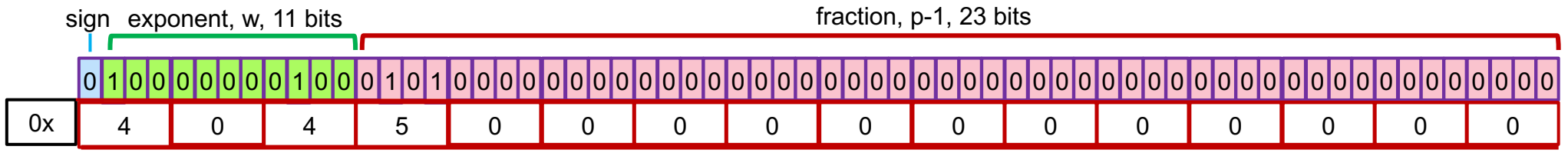


Keeping track of all 64 locations and
writing all those zeros is painful

IEEE name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary 64	double	64	11	53	-1022	1023

Writing IEEE 754 numbers in binary/hexadecimal

- The number 42.0 written in binary with the equivalent hexadecimal (base 16) form beneath.



- It is dramatically easier to write things down in hexadecimal than binary.
- The following are notable examples of key “numbers” in hexadecimal.

-42	0xC045000000000000
Largest normal	0x7FEFFFFFFFFFFF
Smallest normal	0x0010000000000000
Largest subnormal	0x000FFFFFFFFFFF
Smallest subnormal	0X0000000000000001

+ zero	0x0000000000000000
-zero	0x8000000000000000
+infinity	0x7FF0000000000000
-infinity	0x8FF0000000000000
NaN	0X7FF-anything but all-zero

IEEE name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary 64	double	64	11	53	-1022	1023

Decimal, hexadecimal, and binary

0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

NaN: not a number

A normal is a number that can be written in a normalized floating point format

A subnormal is too small to be written as a normalized number … the exponent would need to be less than e_{min}.

Outline

- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
- Wrap-up/Conclusion
- Additional Content
  - Numerical Analysis
  - Changing numbers of bits
  - Compiler options
  - A Few Exercises
    - Exercises
    - Solutions

Addition with floating point numbers

- Lets keep things simple and work with F*(10, 3, -2, 2)

- Find the sum … $1.23 \times 10^1 + 3.11 \times 10^{-1}$
  - Align smaller number to the exponent of the larger number

    0.0311×10^1
  - Add the two aligned numbers …..

$$\begin{array}{cccccccl} 1 & . & 2 & 3 & & & & \times 10^1 \\ 0 & . & 0 & 3 & 1 & 1 & & \times 10^1 \\ 1 & . & 2 & 6 & 1 & 1 & & \times 10^1 \end{array}$$

  - Round to nearest (the default rounding in IEEE 754).

$$1 . 2 6 \times 10^1$$

- **Adding numbers with greatly different magnitudes causes loss of precision** (you lose the low order bits from the exact result).

F*(Radix, Precision, emin, emax)

Floating point arithmetic is not associative

- IEEE 754 guarantees that a single arithmetic operation produces a correctly rounded exact result … but that guarantee does not apply to multiple operations in sequence.

- Floating point numbers are:
  - Commutative: A * B = B * A
  - NOT Associative: A * (C * B) ≠ (A * C) * B
  - NOT Distributive: A*(B+C) ≠ A*B + A*C

- And non-associativity can be inconsistent.

  (0.7+0.1) + 0.3 = 1.0999999999999

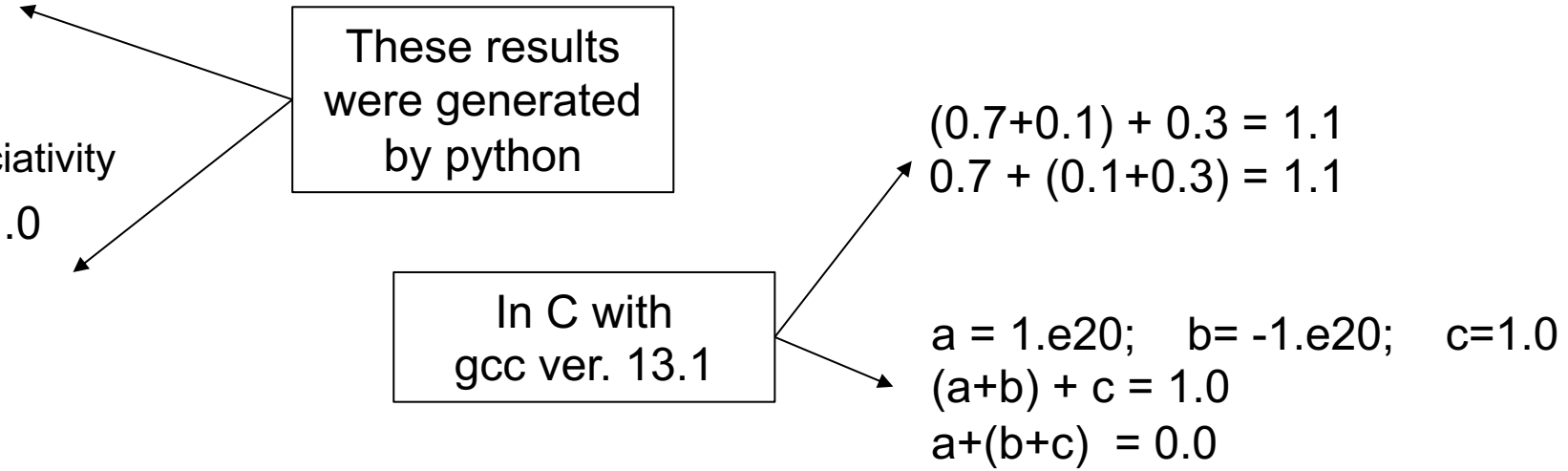
  0.7 + (0.1+0.3) = 1.1

  - A more difficult case of non-associativity

    a = 1.e20; b= -1.e20; c=1.0

    (a+b) + c = 1.0

    a+(b+c) = 0.0



Exercise: summing numbers

- Compute the finite sum: $$sum = \sum_{i=1}^{N} \frac{1.0}{i}$$

- This is a simple loop. Run it forward (i=1,N) and backwards (i=N,1) for large N (`10000000`). Try both double and float

- Are the results different? Why?

Exercise: summing numbers

- Compute the finite sum: $$sum = \sum_{i=1}^{N} \frac{1.0}{i}$$

- This is a simple loop. Run it forward (i=1,N) and backwards (i=N,1) for large N (10000000). Try both double and float

- Are the results different? Why?
  - In the forward direction, the terms in the sum get smaller as you progress. This leads to loss of precision as the smaller terms lates in the summation are added to the much larger accumulated partials sum.
  - In the backwards direction, the terms in the sum start small and grow … so reduced loss of precision adding small numbers to much larger numbers.
  - Using double precision eliminated this problem.

```
#include<stdio.h>
int main(){

    float sum=0.0;
    long N = 10000000;


    for(int i= 1;i<N;i++){
       sum += 1.0/(float)i;
    }
    printf(" sum forward  = %14.8f\n",sum);


    sum = 0.0;
    for(int i= N-1;i>=1;i--){
       sum += 1.0/(float)i;
    }
    printf(" sum backward = %14.8f\n",sum);
}
```

float or double

	double	float
forward	16.69531127	15.40368271
backward	16.69531127	16.68603134

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set
Basic arithmetic operations over Real numbers are commutative, distributive and associative.	Basic operations over floating point numbers are commutative, but NOT associative or distributive.
With arbitrary precision, there is no loss of accuracy when adding real numbers	Adding numbers of different sizes can cause loss of low order bits.

Outline

- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
- Wrap-up/Conclusion
- Additional Content
  - Numerical Analysis
  - Changing numbers of bits
  - Compiler options
  - A Few Exercises
    - Exercises
    - Solutions

PRINCETON UNIVERSITY

iris hep

Cancellation

- Cancellation occurs when we subtract two almost equal numbers
- The consequence is the error could be much larger than the machine epsilon
- For example, consider two numbers

$$x = 3.141592653589793 \text{ (16-digit approximation to } \pi)$$

$$y = 3.141592653585682 \text{ (12-digit approximation to } \pi)$$

Their difference is

$$z = x - y = 0.000000000004111 = 4.111\text{x}10^{-12}$$

In a C program, if we store x, y in single precision and display z in single precision, the difference is

0.000000e+00 Complete loss of accuracy

If we store x, y in double precision and display z in double precision, the difference is

4.110933815582030e-12 Partial loss of accuracy

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

Exercise: Implement a Series summation to find e^x

- A Taylor/Maclaurin series expansion for e^x

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ ... = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

1. Compare to the exp(x) function in math.h for a range of x values **greater than zero**.
   - How do your results compare to the exp(x) library function?

2. Compute e^x for x<0. Consider small negative to large negative values.
   - Do you continue to match the exp(x) library function?

Exercise: Implement a Series summation to find e^x

- A Taylor/Maclaurin series expansion for e^x

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ ... = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

- The computation of x^n and n! are expensive but worse … they lead to large numbers that could overflow the storage format.
- A better approach is to use the relation:

$$\frac{x^n}{n!} = \frac{x}{n} \bullet \frac{x^{n-1}}{(n-1)!}$$

- Terminating the sum … obviously you don't want to go to infinity. How do you terminate the sum? A good approach is to end the sum when new terms do not significantly change the sum.

- For x< 0, compare computation of e^x directly and as $e^x = 1/ e^x$.

```
#define TYPE float
TYPE MyExp (TYPE x) {
    long counter = 0;
    TYPE delta = (TYPE)1.0;
    TYPE e_tothe_x = (TYPE)1.0;
    while((1.0 + delta) != 1.0) {
        counter++;
        delta *= x/counter;
        e_tothe_x +=delta;
    }
    return e_tothe_x;
}
```

When x>0 in series, no cancelation and MyExp matches exp from the standard math library (math.h)

x	exp(x) math.h	MyExp(x)
5	148.413	148.413
10	22026.5	22026.5
15	3.26902e+06	3.26902e+06
20	4.85165e+08	4.85165e+08

x	exp(x) math.h	MyExp(x)	1/MyExp(|x|)
-5	6.73795e-03	6.73714e-03	6.73795e-03
-10	4.53999e-05	-5.23423e-05	4.53999e-05
-15	3.05902e-07	-2.23869e-02	3.05902e-07
-20	2.06115e-09	-1.79703	2.06115e-09

When x<0 in series, MyExp does not match exp from math.h due to cancelation. Results become nonsensical for x= -10 and beyond.

Solution: Numerical Analysis

Refactoring functions to improve floating point behavior

- Evaluate the following function for large x ($x = 10^k$ for k = 5,6,7,8 …)

$$f(x) = \frac{1}{\sqrt{x^2 + 1} - x}$$

- Can you refactor the function to make it numerically more stable?

Solution: Numerical Analysis

Refactoring functions to improve floating point behavior

- Evaluate the following function for large x (x = 10^k for k = 5,6,7,8 …)

$$f(x) = \frac{1}{\sqrt{x^2+1}-x}$$

- For large x, $\sqrt{x^2+1} \approx x$ so we expect problems with cancelation in the denominator.
- We can refactor the expression to remove the cancelation.

$$f(x) = \frac{1}{\sqrt{x^2+1}-x} = \frac{\sqrt{x^2+1}+x}{(\sqrt{x^2+1}-x)(\sqrt{x^x+1}+x)} = \frac{\sqrt{x^2}+x}{x^2+1-x^2} = \sqrt{x^2+1}+x$$

x	$\frac{1}{\sqrt{x^2+1}-x}$	$\sqrt{x^2+1}+x$
10^5	200000.223331	200000.000005
10^6	1999984.771129	2000000.000001
10^7	19884107.851852	20000000.000000
10^8	inf	200000000.000000

Solutions degrades until divide by zero

Solution: Numerical Analysis

Refactoring functions to improve floating point behavior

- Evaluate the following function for large x (x = 10^k for k = 5,6,7,8 …)

$$f(x) = \frac{1}{\sqrt{x^2+1} - x}$$

- For large x, $\sqrt{x^2+1} \approx x$ so we expect problems with cancelation in the denominator.
- We can refactor the expression to remove the cancelation.

This sort of mathematical refactoring to produce numerically stable algorithms is prohibitively difficult to apply to "real" problems. Very few people have formal training in numerical analysis.

x	$\frac{1}{\sqrt{x^2+1} - x}$	$\sqrt{x^2+1} + x$
10^5		
10^6		
10^7		
10^8		

Computer Science has changed over my lifetime. Numerical Analysis seems to have turned into a sliver under the fingernails of computer scientists

Prof. W. Kahan, Desperately needed Remedies … Oct. 14, 2011

Solutions degrades until divide by zero

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set
Basic arithmetic operations over Real numbers are commutative, distributive and associative.	Basic operations over floating point numbers are commutative, but NOT associative or distributive.
With arbitrary precision, there is no loss of accuracy when adding real numbers	Adding numbers of different sizes can cause loss of low order bits.
With arbitrary precision, there is no loss of accuracy when subtracting real numbers	Subtracting two numbers of similar size cancels higher order bits

Outline

- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
- Wrap-up/Conclusion
- Additional Content
  - Numerical Analysis
  - Changing numbers of bits
  - Compiler options
  - A Few Exercises
    - Exercises
    - Solutions

IEEE 754 arithmetic and rounding

- The IEEE 754 standard requires that the result of basic arithmetic ops (+, -, *, /, FMA) be equal to the result from “infinitely precise arithmetic” rounded to the storage format (e.g., float or double).

- Consider the following problem … subtract two IEEE 754 32 bit numbers (F*(2,24,127,-126)):

$$(1.00000000000000000000000)_2 \cdot 2^0$$
$$- (1.00000000000000000000001)_2 \cdot 2^{-25}$$

- We normalize them to the same exponent and carry out the operation exactly

$$(1.00000000000000000000000 \quad)_2 \cdot 2^0$$
$$- \ (0.00000000000000000000000\,01000000000000000000001)_2 \cdot 2^0$$
$$= \ (0.11111111111111111111110\,11111111111111111110)_2 \cdot 2^0$$

- Then normalize the result

$$(1.11111111111111111111101\,11111111111111111111)_2 \cdot 2^{-1}$$

- Then round to nearest to fit into the destination format

$$(1.11111111111111111111)_2 \cdot 2^{-1}$$

FMA: Fused Multiple Add … d = a*b+c

F*(Radix, Precision, emin, emax)

Content based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

IEEE 754 arithmetic and rounding

- The IEEE 754 standard requires that the result of basic arithmetic ops (+, -, *, /, FMA) be equal to the result from “infinitely precise arithmetic” rounded to the storage format (e.g., float or double).

- Consider the following problem … subtract two IEEE 754 32 bit numbers (F*(2,24,127,-126)):

$$(1.00000000000000000000000)_2 \cdot 2^0$$
$$- (1.00000000000000000000001)_2 \cdot 2^{-25}$$

- We normalize them to the same exponent and carry out the operation exactly

$$(1.00000000000000000000000)_2 \cdot 2^0$$
$$- (0.00000000000000000000000\,01000000000000000000001)_2 \cdot 2^0$$
$$= (0.11111111111111111111110\,11111111111111111110)_2 \cdot 2^0$$

The exact result doubled the number of bits in the fraction. Do we really need all those bits?

- Then normalize the result

$$(1.11111111111111111111101\,11111111111111111111)_2 \cdot 2^{-1}$$

- Then round to nearest to fit into the destination format

$$(1.11111111111111111111)_2 \cdot 2^{-1}$$

FMA: Fused Multiple Add … d = a*b+c

F*(Radix, Precision, emin, emax)

IEEE 754 arithmetic and rounding

Next 2 bits in the answer:
the guard and rounding bits

- Turns out you only need three extra bits … the Guard bit, the Rounding bit, and the Sticky Bit (GRS)

The Sticky bit: the logical or of all the other bits
(i.e., if any bit is “1” then the sticky bit is “1”

$$
\begin{array}{rll}
 & (1.00000000000000000000000000)_2 & \cdot 2^0 \\
- & (0.00000000000000000000000011)_2 & \cdot 2^0 \\
\hline
 & (0.11111111111111111111111101)_2 & \cdot 2^0 \\
\text{Normalize} & (1.11111111111111111111110 1)_2 & \cdot 2^{-1} \\
\text{Round to nearest} & (1.11111111111111111111111)_2 & \cdot 2^{-1}
\end{array}
$$

- The Guard, Rounding, and Sticky bits are sufficient to support all the IEEE 754 rounding modes to yield the same result you’d get from an exact computation followed by rounding into the target format.

- Exactly rounded results are required for the basic arithmetic operations (including FMA) but also **square root**, **remainder**, and **conversion between Integer and Floating point numbers** … but not for conversion between decimal and binary floating point.

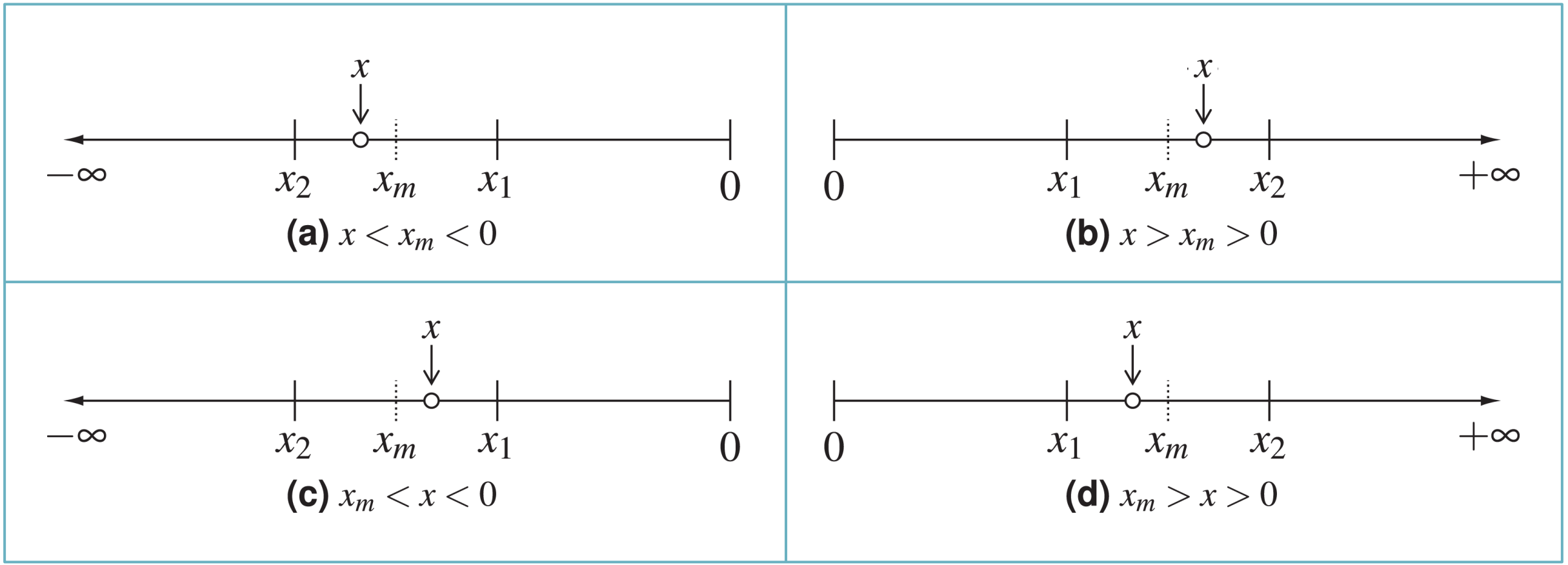
FMA: Fused Multiple Add … d = a*b+c

F*(Radix, Precision, emin, emax)

Content based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

IEEE 754 Rounding Modes

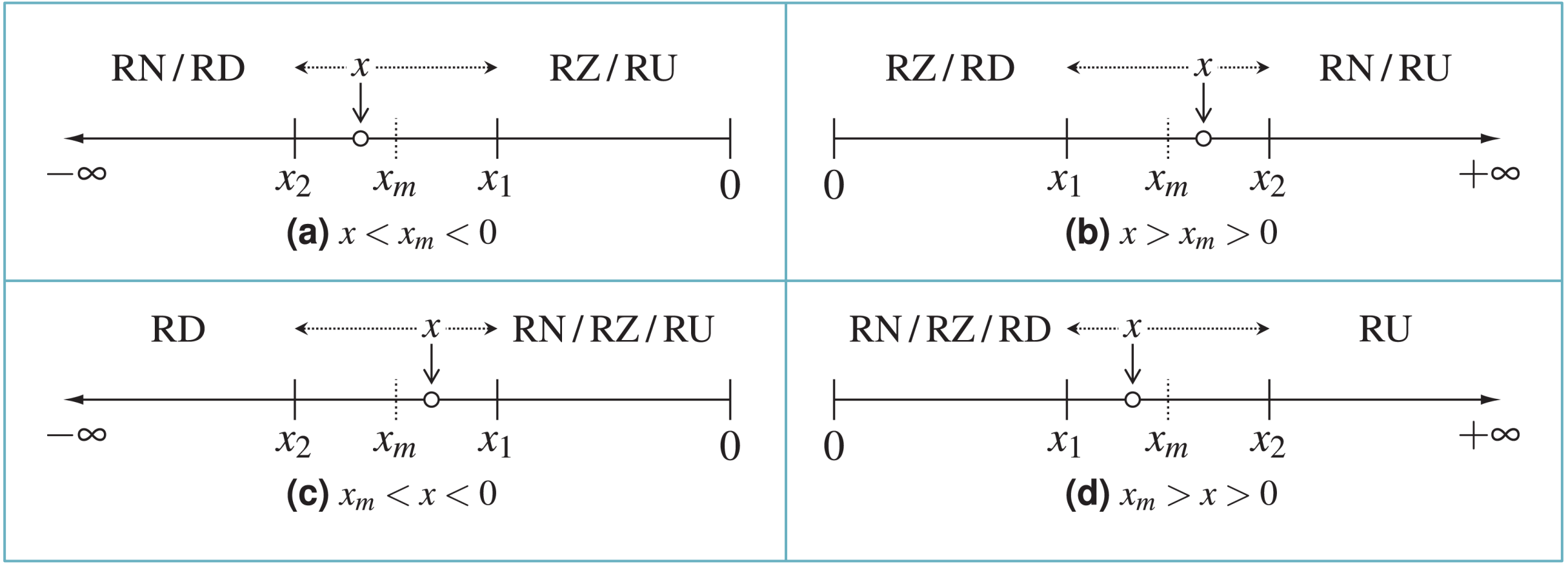
Consider a real number x that falls between its two nearest floating point numbers (x_1 and x_2). At the midpoint between x_1 and x_2 is the real number x_m. We have four cases to consider when thinking about rounding.



Source: Numerical behavior of NVIDIA tensor cores, M. Fasi, N. J. Higham, M. Mikaitis, and S. Pransesh, *PeerJ Comp. Sci.* 7:e330, Feb 10, 2021, DOI: 10.7717/peerj-cs.330/fig-1

IEEE 754 Rounding Modes

Consider a real number x that falls between its two nearest floating point numbers (x_1 and x_2). At the midpoint between x_1 and x_2 is the real number x_m. The horizontal dotted line shows the floating point numbers selected for the different rounding modes (RN, RD, RZ, RU) for position of x vs x_m and which side of zero x is on.



RN: Round to Nearest. RD: Round Downward RZ: Round towards zero RU: Round upward

Source: Numerical behavior of NVIDIA tensor cores, M. Fasi, N. J. Higham, M. Mikaitis, and S. Pransesh, *PeerJ Comp. Sci.* 7:e330, Feb 10, 2021, DOI: 10.7717/peerj-cs.330/fig-1

You must be careful how you manage rounding…

Vancouver stock exchange index undervalued by 50%

(Nov. 25, 1983)

See http://ta.twi.tudelft.nl/usersvuik/wi211/disasters.html

Index managed on an IBM/370. 3000 trades a day and for each trade, the index was truncated to the machine's REAL*4 format, loosing 0.5 ULP per transaction. After 22 months, the index had lost half its value.

ULP … Unit in the last place

Working with IEEE 754 rounding modes

Two versions of round to nearest…
- Nearest, on a tie, round to even
- Nearest, on a tie, away from zero

Default rounding mode

The 4 rounding modes in IEEE 754

rounding mode	C name
to nearest	FE_TONEAREST
toward zero	FE_TOWARDZERO
to +infinity	FE_UPWARD
to −infinity	FE_DOWNWARD

Three directed roundings

C

```
#include <fenv.h>
//#pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = fegetround( );

// establish the desired rounding mode
fesetround(FE_TOWARDZERO);

// do whatever you need to do …
// … and restore the original mode afterwards
fesetround(originalRounding);
```

Clang and GCC compilers to not recognize the STDC pragma (even though they are technically required to).

Fortunately, rounding mode control seems to work without it.

If not, try the compiler flag –frounding-math

C++

```
#include <cfenv>
// #pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = std::fegetround( );

// establish the desired rounding mode
std::fesetround(FE_TOWARDZERO);

// do whatever you need to do …
// … and restore the original mode afterwards
std::fesetround(originalRounding);
```

Exercise: IEEE 754 rounding modes

- Explore how different rounding modes change the answers of programs you have on your system.
- What does it tell you if answers change as rounding modes change?

The 4 rounding modes in IEEE 754

```
rounding mode     C name
------------------------------
to nearest        FE_TONEAREST
toward zero       FE_TOWARDZERO
to +infinity      FE_UPWARD
to -infinity      FE_DOWNWARD
```

C

```
#include <fenv.h>
//#pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = fegetround( );

// establish the desired rounding mode
fesetround(FE_TOWARDZERO);

// do whatever you need to do ...
// ... and restore the original mode afterwards
fesetround(originalRounding);
```

Clang and GCC compilers to not recognize the STDC pragma (even though they are technically required to).

Fortunately, rounding mode control seems to work without it.

If not, try the compiler flag –frounding-math

C++

```
#include <cfenv>
// #pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = std::fegetround( );

// establish the desired rounding mode
std::fesetround(FE_TOWARDZERO);

// do whatever you need to do ...
// ... and restore the original mode afterwards
std::fesetround(originalRounding);
```

Outline

- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
- Wrap-up/Conclusion
- Additional Content
  - Numerical Analysis
  - Changing numbers of bits
  - Compiler options
  - A Few Exercises
    - Exercises
    - Solutions

Should we trust computer arithmetic?

Sleipner Oil Rig Collapse (8/23/91). Loss: **$700 million.**

$1.6 Billion in 2024 dollars

See http://www.ima.umn.edu/~arnold/disasters/sleipner.html

Linear elastic model using NASTRAN underestimated shear stresses by 47% resulted in concrete walls that were too thin.

NASTRAN is the world's most widely used finite element code … in heavy use since 1968

We can’t trust FLOPS … let’s give up and return to slide rules

(an elegant weapon for a more civilized age)

Image source: Presbrey advertising agency for International Business Machines, 1951

Sleipner Oil Rig Collapse: The slide-rule wins!!!

It was recognized that finding and correcting the flaws in the computer analysis and design routines was going to be a major task. Further, with the income from the lost production of the gas field being valued at perhaps $1 million a day, it was evident that a replacement structure needed to be designed and built in the shortest possible time.

A decision was made to proceed with the design using the pre-computer, slide-rule era techniques that had been used for the first Condeep platforms designed 20 years previously. By the time the new computer results were available, all of the structure had been designed by hand and most of the structure had been built. On April 29, 1993 the new concrete gravity base structure was successfully mated with the deck and Sleipner was ready to be towed to sea (See photo on title page).

The failure of the Sleipner base structure, which involved a total economic loss of about $700 million, was probably the most expensive shear failure ever. The accident, the subsequent investigations, and the successful redesign offer several lessons for structural engineers. No matter how complex the structure or how sophisticated the computer software it is always possible to obtain most of the important design parameters by relatively simple hand calculations. Such calculations should always be done, both to check the computer results and to improve the engineers' understanding of the critical design issues. In this respect it is important to note that the design errors in Sleipner were not detected by the extensive and very formal quality assurance procedures that were employed.

Conclusion

- Floating point arithmetic usually works and you can “almost always” be comfortable using it.
- Floating point arithmetic is mathematically rigorous. You can prove theorems and develop rigorous error bounds. This is the field of numerical analysis.
- Unfortunately, almost nobody learns numerical analysis these days.
- As scientists using computers in your research, maintain a healthy skepticism of your results … don’t be shy about testing the fidelity of your computations*.
  1. Repeat the computation with arithmetic of increasing precision, increasing it until a desired number of digits in the results agree.
  2. Repeat the computation in arithmetic of the same precision but rounded differently, say *Down* then *Up* and perhaps *Towards Zero,* then compare results (this wont work with libraries that require a particular rounding mode).
  3. Repeat computation a few times in arithmetic of the same precision but with slightly different input data, and see how widely results vary.

*Source: W. Kahan: How futile are mindless Assessments of Roundoff in floating-point computation?

References

- What every computer computer scientist should know about floating point arithmetic, David Goldberg, Computing Surveys, 1991.
  - https://dl.acm.org/doi/pdf/10.1145/103162.103163

- W. Kahan: How futile are mindless Assessments of Roundoff in floating-point computation?
  - https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf

- History of IEEE-754: an interview with William Kahan
  - https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html

Additional content

- Numerical Analysis
- Changing numbers of bits
- Compiler options
- A Few Exercises
  - Exercises
  - Solutions

Numerical Analysis

- The details of how we do arithmetic on computers and the branch of mathematics that studies the consequences of computer arithmetic (numerical analysis) is fundamentally boring.
  - Even professionals who work on computer arithmetic (other than W. Kahan*) admit (maybe only in private) that it's boring.

> Computer Science has changed over my lifetime. Numerical Analysis seems to have turned into a sliver under the fingernails of computer scientists
>
> Prof. W. Kahan, Desperately needed Remedies … Oct. 14, 2011

- It's fine to take floating point arithmetic for granted … until something breaks.

- The scary part of this … is that you don't know something is wrong with your program until disaster strikes!!!!

> You don't need to be paranoid, but be skeptical of *ANYTHING* you compute on a comptuer.

*Professor William Kahan of UC Berkeley is the father of modern floating point arithmetic (IEEE-754)

Error Analysis: error in input → error in output

- View a program as taking input, x, and evaluating a function, f(x), to compute output, y. The numerical analyst is interested in the following evaluation:

$$f(x + \Delta x) = y + \Delta y$$

- Note: Δx includes all sources of error including roundoff errors, loss of precision, cancelation or even errors in collected data.

- Numerical analysts summarize the stability of a problem in terms of a ratio … the ratio of the error in the generated result to the error in the input. This is normalized to the range of values in y and x leading to what is called the condition number, C:

$$C = \frac{\left|\frac{\Delta y}{y}\right|}{\left|\frac{\Delta x}{x}\right|} = \frac{|x|}{|y|} \cdot \frac{|\Delta y|}{|\Delta x|} = \frac{|x \cdot f'(x)|}{|f(x)|}$$

- For a small condition number, C:
  - Small Δx → small Δy
  - We call this a **well conditioned** problem
- For a large condition number, C:
  - Small Δx → large Δy
  - We call this a **ill conditioned** problem

This is NOT a lecture on numerical analysis ….

- Numerical analysis is a complex topic well beyond the scope of this lecture.
- The goal here is to make you aware of it and the general concept of well-conditioned vs ill-condition problems … not how to derive and work with condition numbers.

Error Analysis: error in input → error in output

- View a program as taking input, x, and evaluating a function, f(x), to compute output, y. The numerical analyst is interested in the following evaluation:

$$f(x + \Delta x) = y + \Delta y$$

- Note: Δx includes all sources of error including roundoff errors, loss of precision, cancelation or even errors in collected data.
- Numerical analysts summarize the stability of a problem in terms of a ratio … the ratio of the error in the generated result to the error in the input. This is normalized to the range of values in y and x leading to what is called the condition number, C:

$$C = \frac{\left|\frac{\Delta y}{y}\right|}{\left|\frac{\Delta x}{x}\right|} = \frac{|x|}{|y|} \cdot \frac{|\Delta y|}{|\Delta x|} = \frac{|x \cdot f'(x)|}{|f(x)|}$$

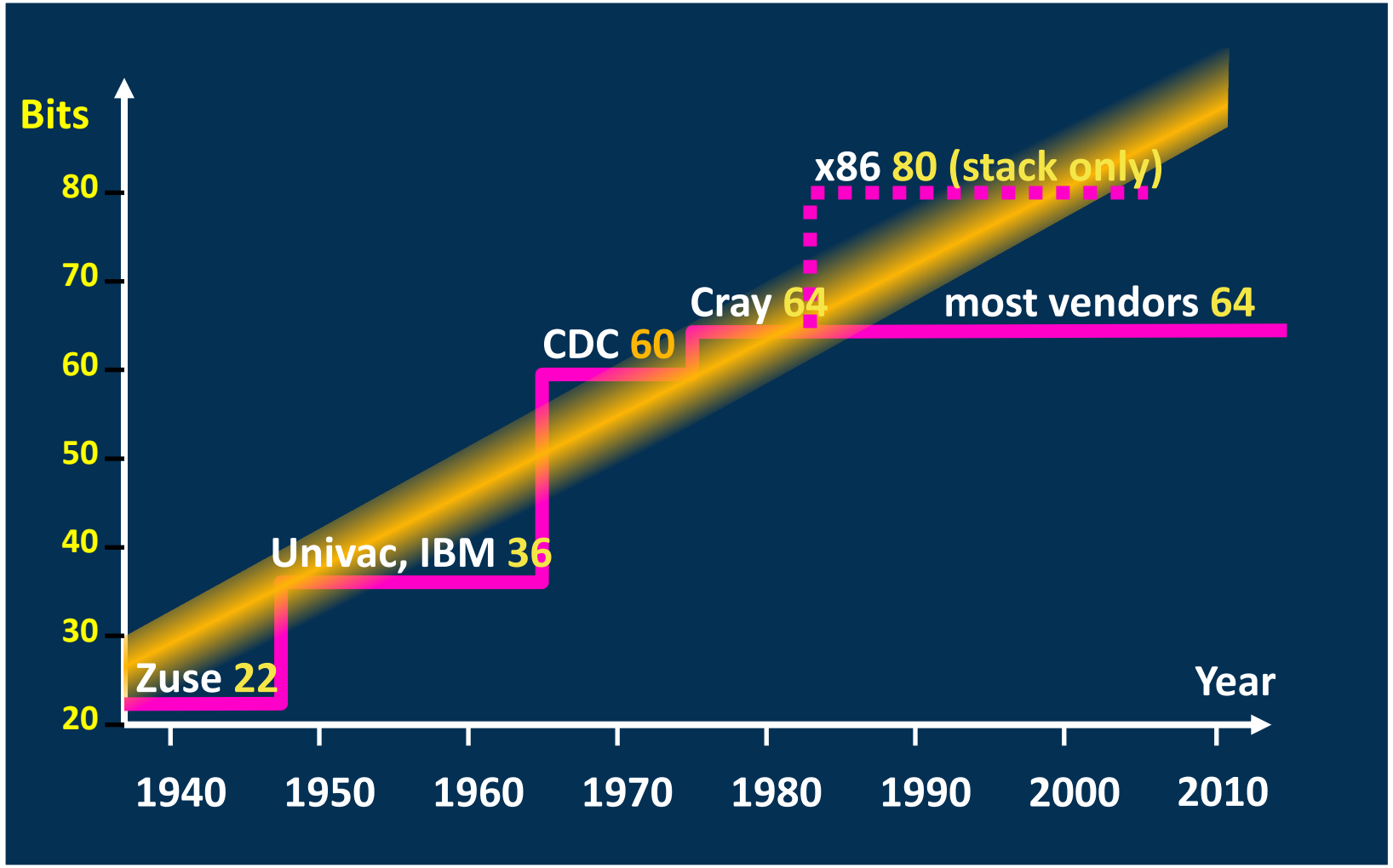
- For a small condition number, C:
  - Small Δx → small Δy
  - We call this a **well conditioned** problem
- For a large condition number, C:
  - Small Δx → large Δy
  - We call this a **ill conditioned** problem

… So rather than a long diversion into the details of numerical analysis, lets focus on a single problem numerical analysts work on … **how does the properties of floating point arithmetic influenced a computation?**

Additional content

- Numerical Analysis
- Changing numbers of bits
- Compiler options
- A Few Exercises
  - Exercises
  - Solutions

Floating point arithmetic …just use : use lots of bits and hope for the best …



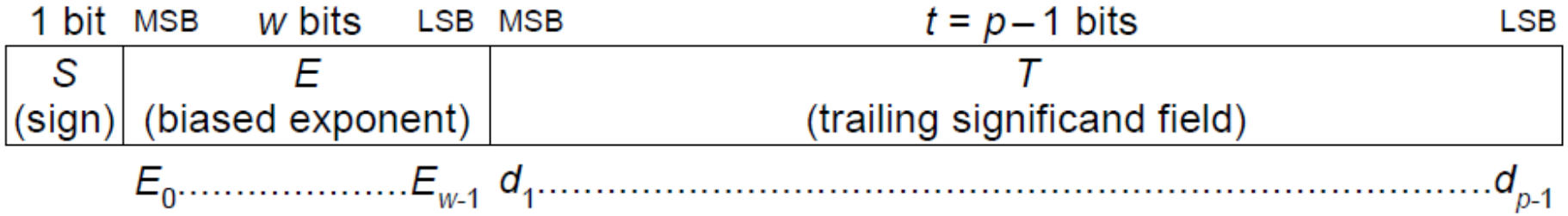
Is 64 bits enough? Is it too much? We're *guessing*.

Source: John Gustafson from long long ago when he was at Intel

Quad Precision

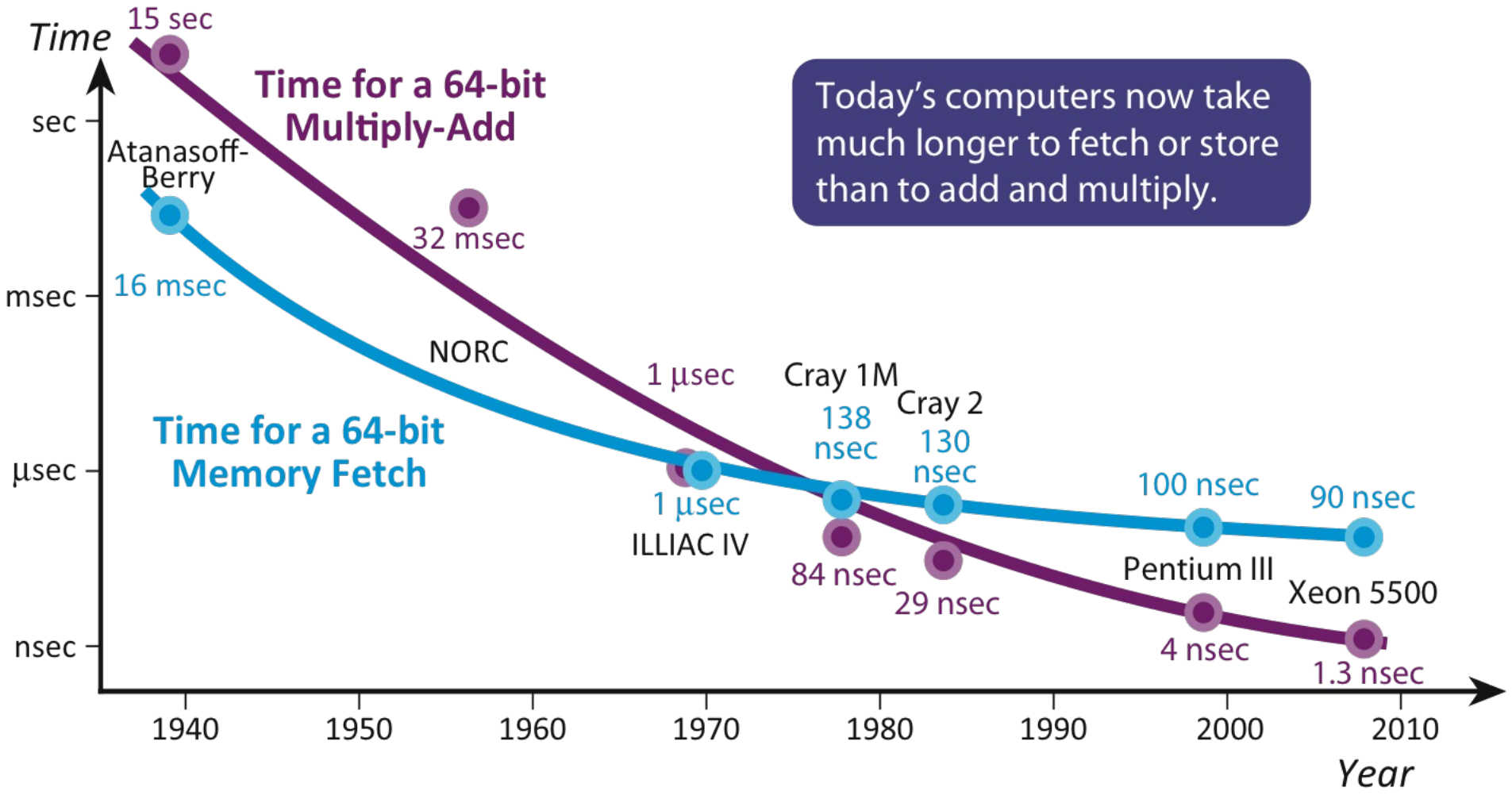
- IEEE 754™ defines a range of formats including quad (128)

	binary32	binary64	binary128
P, digits	24	53	113
emax	+127	+1023	+16383



- There are pathological cases where you lose all the precision in an answer, but the more common case is that you lose only half the digits.
- Hence, for 32 or 64 bit input data, quad precision (113 significant bits) is probably adequate to make most computations safe (Kahan 2011).

Wider floating point formats turn compute bound problems into memory bound problems



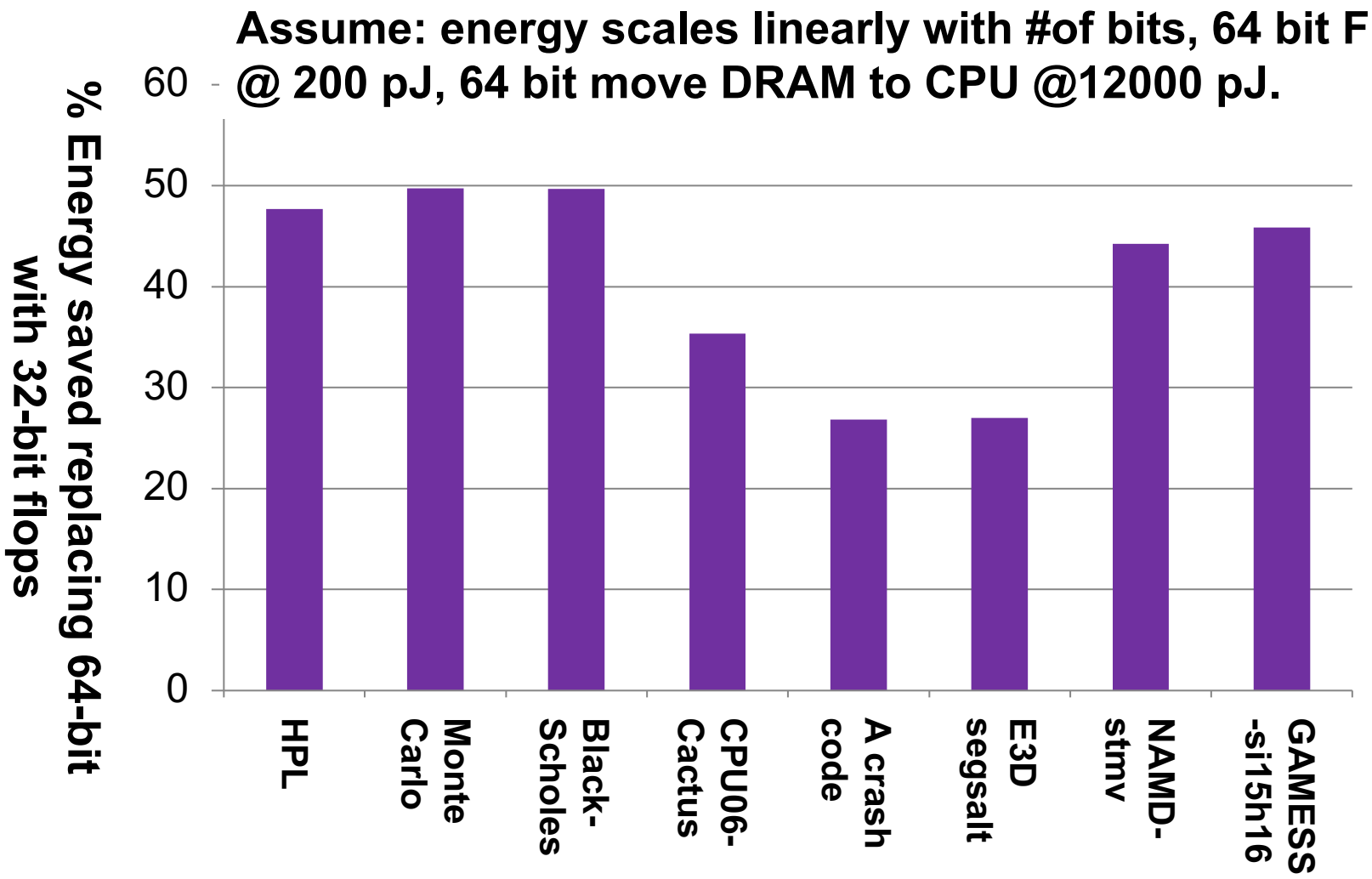
Energy implications of floating point numbers: 32 bit vs. 64 bit numbers

Operation	Approximate energy consumed today
64-bit multiply-add	64 pJ
Read/store register data	6 pJ
Read 64 bits from DRAM	**4200 pJ**
Read 32 bits from DRAM	***2100 pJ***

Simply using single precision in DRAM instead of double saves as much energy as *30* on-chip floating-point operations.

Source: S. Borkar, Intel. Data is for 32 nm technology ca. 2010

energy savings: replace 64 bit flops with 32 bit flops

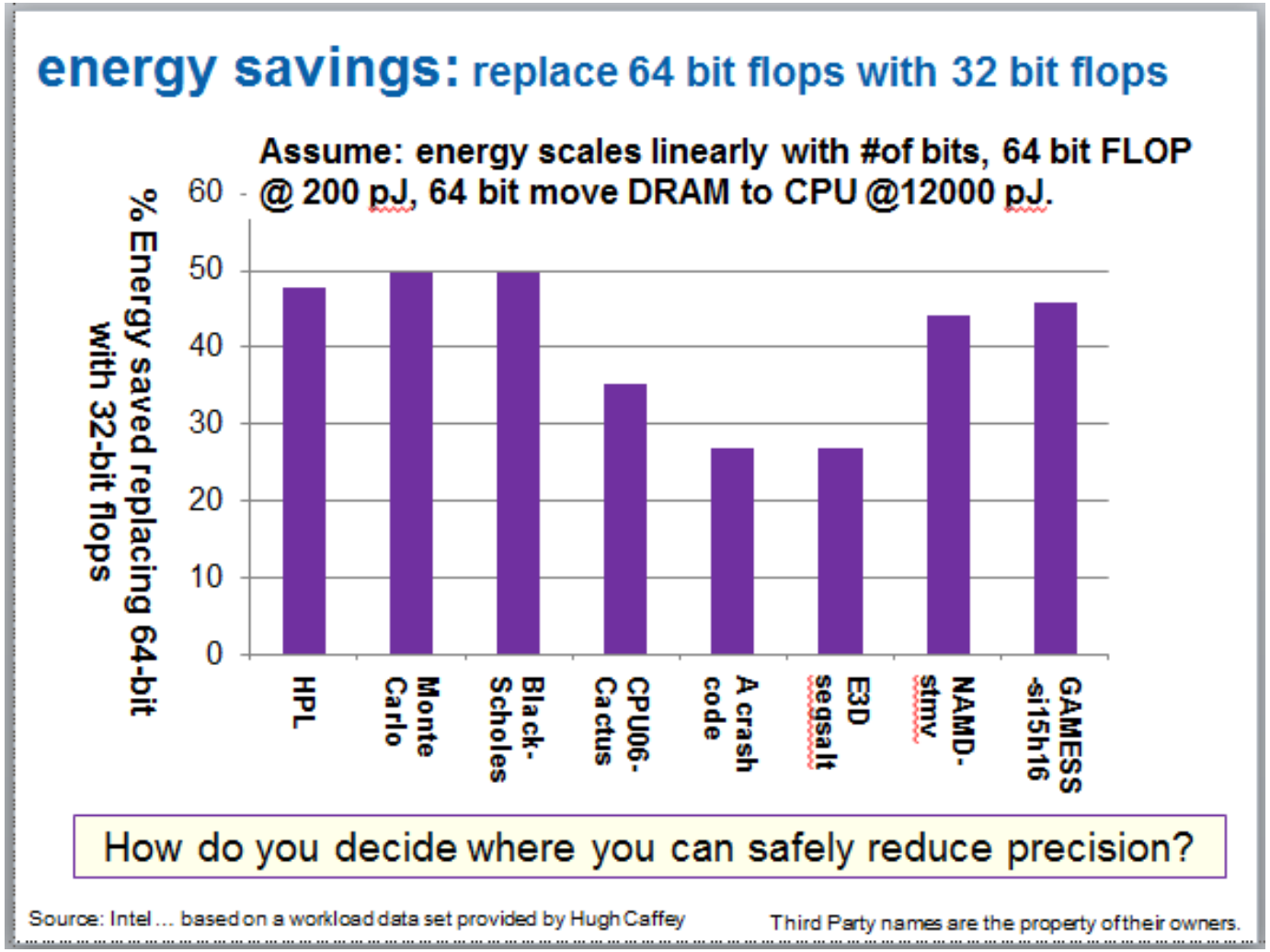


How do you decide where you can safely reduce precision?

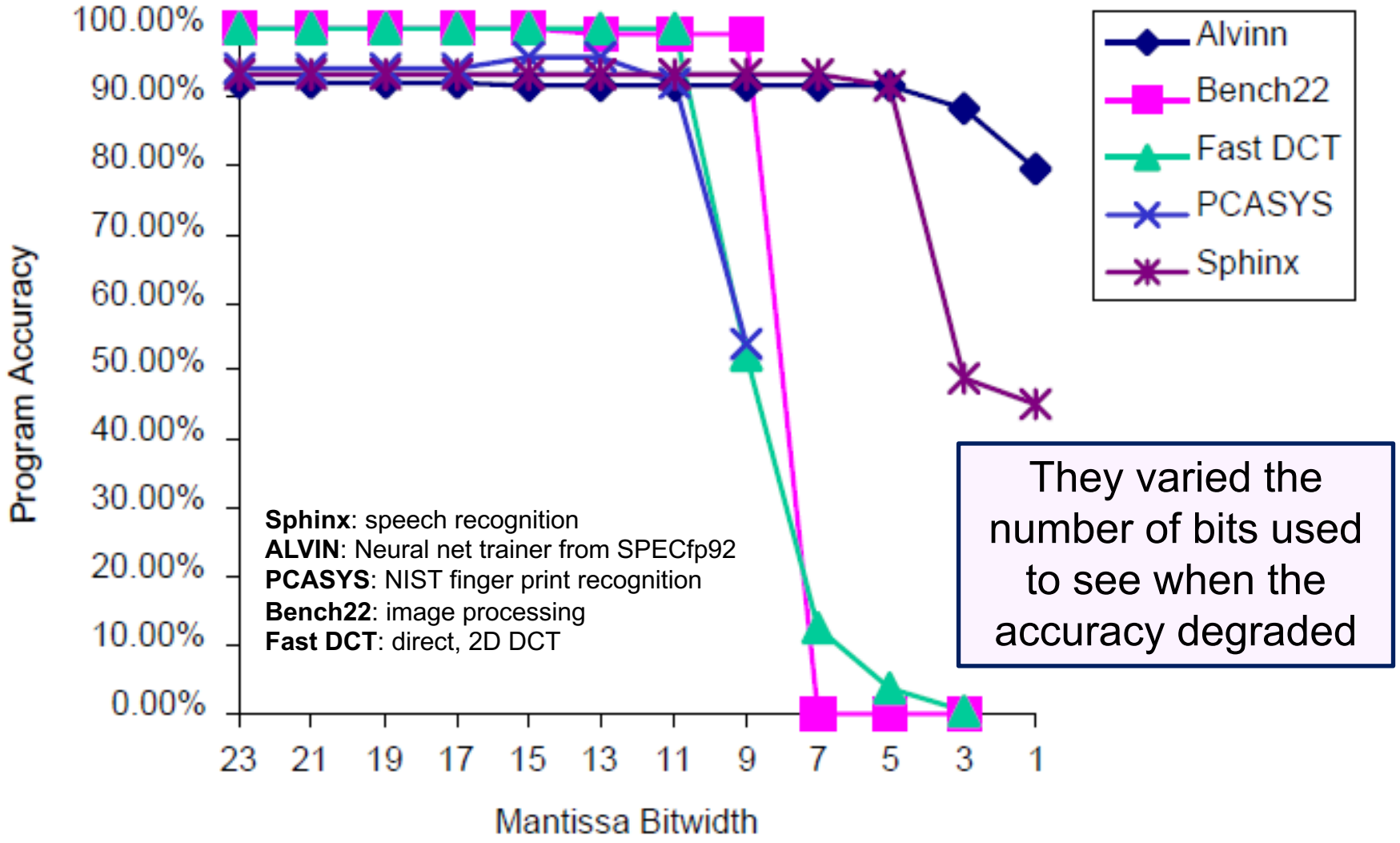
Source: Intel … based on a workload data set provided by Hugh Caffey (2010)

Maybe we don't want Quad after all?

- If Performance/Watt is the goal, using Quad everywhere to avoid careful numerical analysis is probably a bad idea.



How many bits do we really need?



J.Y.F. Tong, D. Nagle, and R. Rutenbar, "Reducing Power by Optimizing the Necessary Precision Range of Floating Point Arithmetic," in *IEEE Transactions on VLSI systems*, Vol. 8, No.3, pp 273-286, June 2000. [2] M. Stevenson, J. Babb,

Workload	Description	Accuracy Measurement
Sphinx III	CMU's speech recognition program based on fully continuous hidden Markov models. The input set is taken from the DARPA evaluation test set which consists of spoken sentences from the Wall Street Journal.	Accuracy is estimated by dividing the number of words recognized correctly over the total number of words in the input set.
ALVINN	Taken from SPECfp92. A neural network trainer using backpropagation. Designed to take input sensory data from a video camera and a laser range finder and guide a vehicle on the road.	The input set consists of 50 road scenes and the accuracy is measured as the number of correct travel direction decisions made by the network.
PCASYS	A pattern-level finger print classification program developed at NIST. The program classifies images of fingerprints into six pattern-level classes using a probabilistic neural network.	The input set consists of 50 different finger print images and the classification result is measured as percentage error in putting the image in the wrong class. The accuracy of the recognition is simply (1 - percentage error).
Bench22	An image processing benchmark which warps a random image, and then compares the warped image with the original one.	Percentage deviation from the original outputs are used as a measure of accuracy.
Fast DCT	A direct implementation of both 2-dimensional forward Discrete Cosine Transform (DCT) and inverse DCT of blocks of 8x8 pixels.	100 random blocks of 8x8 pixels are transformed by forward DCT and then recovered by inverse DCT. Accuracy measured as percentage of correctly recovered pixels.

Notes to support the "how many bits do we need" slide

Additional content

- Numerical Analysis
- Changing numbers of bits
- Compiler options
- A Few Exercises
  - Exercises
  - Solutions

PRINCETON UNIVERSITY

iris hep

Optimization levels and related options

- GCC has a rich optimization pipeline that is controlled by approximately a hundred command line options
- The default is to not optimize. You can specify this optimization level on the command line as -O0. It is often used when developing and debugging a project. This means it is usually accompanied with the command line switch -g so that debug information is emitted. As no optimizations take place, no information is lost because of it. No variables are optimized away, the compiler only inlines functions with special attributes that require it, and so on. As a consequence, the debugger can almost always find everything it searches for in the running program and report on its state very well. On the other hand, the resulting code is big and slow
- The most common optimization level for release builds is -O2 which attempts to optimize the code aggressively but avoids large compile times and excessive code growth.
- Optimization level -O3 instructs GCC to simply optimize as much as possible, even if the resulting code might be considerably bigger and the compilation can take longer.
- Note that neither -O2 nor -O3 imply anything about the precision and semantics of floating-point operations. Even at the optimization level -O3 GCC implements math functions so that they strictly follow the respective IEEE and/or ISO rules. This often means that the compiled programs run markedly slower than necessary if such strict adherence is not required. The command line switch -ffast-math is a common way to relax rules governing floating-point operations.
- The most aggressive optimization level is `-Ofast` which does imply `-ffast-math` along with a few options that disregard strict standard compliance. In GCC 11 this level also means the optimizers may introduce data races when moving memory stores which may not be safe for multithreaded applications. Additionally, the Fortran compiler can take advantage of associativity of math operations even across parentheses and convert big memory allocations on the heap to allocations on stack. The last mentioned transformation may cause the code to violate maximum stack size allowed by **`ulimit`** which is then reported to the user as a segmentation fault.

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Optimization level recommendation

- Usually we(*) recommend using -O2, because at this level the compiler makes balanced size and speed trade-offs when building a general-purpose operating system
- However, we suggest using -O3 if you know that your project is compute-intensive and is either small or an important part of your actual workload
- Moreover, if the compiled code contains performance-critical floating-point operations, we strongly advise that you investigate whether -ffast-math or any of the fine-grained options it implies can be safely used

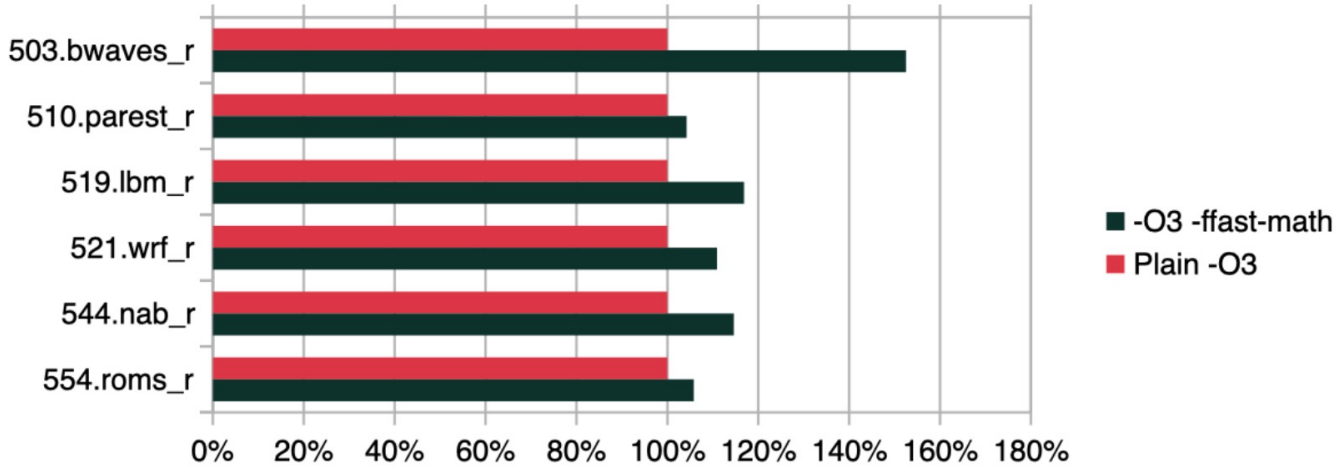


FIGURE 18: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF SELECTED FLOATING-POINT BENCHMARKS BUILT WITH GCC 11.2 AND -O3 -MARCH=NATIVE, WITHOUT AND WITH -FFAST-MATH

- (*) https://documentation.suse.com/sbp/server-linux/single-html/SBP-GCC-11/

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

Additional content

- Numerical Analysis
- Changing numbers of bits
- Compiler options
- A Few Exercises
  - Exercises
  - Solutions

Exercise: The machine epsilon

- When you compute the relative error between a result using real arithmetic, $\tilde{x}$, and the analogous result using floating point arithmetic, x, we define the error in two different ways:
  - Absolute error: $|\tilde{x} - x|$
  - Relative error: $\frac{|\tilde{x}-x|}{|\tilde{x}|}$

- The relative error is normalized, so the smallest relative error is the distance between 1.0 and the next largest floating point number. This goes by a number of names but it is traditionally called the machine epsilon or ε.
  - Exercise … Part 1: Write a program that computes ε.
  - Exercise … Part 2: For IEEE 754 float, derive the value of ε.

Exercise: Summation with floating point arithmetic

- We have provided a C program called summation.c which is supported by a number of utilities in the file UtilityFunctions.c. DO NOT look at UtilityFunctions.c (at least, not until you have finished this exercise.

- In the program, we generate a sequence of floating point numbers (all greater than zero).
  - **Don't look at how we create that sequence** … treat the sequence generator as a black box (in other words, just work on the sequence, don't use knowledge of how it was generated).

- Write code to sum the sequence of numbers. You can compare your result to the estimate of the correct result provided by the sequence generator.
  - Only use float types (it's cheating to use double … at least to start with).

- Using what you know about floating point arithmetic, is there anything you can think of doing to improve the quality of your sum?

git clone https://github.com/tgmattso/ParProgForPhys.git then go to the directory Flop_Exercises

Exercise: Refactoring functions to improve floating point behavior

- Evaluate the following function for large x (x = 10^k for k = 5,6,7,8 …)

$$f(x) = \frac{1}{\sqrt{x^2 + 1} - x}$$

- Can you refactor the function to make it numerically more stable?

Exercise: Compensated Summation.

- Summation is notorious for errors due to loss of precision when two numbers of widely different magnitudes are added.
- Define a correction that would account for this loss of precision? Use it with one of our summation problems to see if your Compensated summation approach works.

```
Input: a sequence of N values, x[i] i=1,N

   correction = 0.0
   sum = 0.0

   for i = 1 to N:

      xcor = x[i] ⊙ correction


      tmpSum =  sum + xcor

      correction = ????????

      sum = tmpSum
   }
Output: sum
```

apply a correction to x[i] to account for bits lost in the ***previous*** loop iteration

sum is big, but xcor is small. → Low order digits of xcor are lost

Note: It is cheating to look this up online. With what we've covered and the hints I've provided, you should be able to figure this out on your own.

Additional content

- Numerical Analysis
- Changing numbers of bits
- Compiler options
- A Few Exercises
  - Exercises
  - Solutions

Finding the machine epsilon

solution

- The rounding machine epsilon is the last number added to one that yields a sum that is greater than one.
- When rounding to nearest is used (the default in IEEE 754) it is twice the size of the machine epsilon we derived above. Why?

```
#include <stdio.h>
#define TYPE    float
int main(). {
   TYPE one = (TYPE)1.0;
   TYPE eps = (TYPE)1.0;
   long iters = 0;
   while( (one+(TYPE)eps)>one){
      eps = eps/(TYPE)2.0;
      iters++;
   }
   printf("epsilon is 2 to the -%ld or %g\n",
                        sizeof(TYPE),iters,eps);
}
```

- The machine epsilon is the gap between 1.0 and the closest number larger than one.
- For an IEEE 754 32 bit floating point number (F*(2,24,127,-126)). The number closest but larger than 1.0 is:

$$\varepsilon = (1.00\ldots1)2^0 - (1.00\ldots0)2^0 = (0.00\ldots1)2^0 = 2^{-(p-1)} = 2^{-23}$$

F*(Radix, Precision, emin, emax)

Exercise: Summation with floating point arithmetic

solution

- We have provided a C program called sum.c
- In the program, we generate a sequence of floating point numbers (all greater than zero).
  - **Don't look at how we create that sequence** … treat the sequence generator as a black box (in other words, just work on the sequence, don't use knowledge of how it was generated).
- Write code to sum the sequence of numbers. You can compare your result to the estimate of the correct result provided by the sequence generator.
  - Only use float types (it's cheating to use double … at least to start with).
- Using what you know about floating point arithmetic, is there anything you can think of doing to improve the quality of your sum?

Exercise: Refactoring functions to improve floating point behavior

solution

- Evaluate the following function for large x (x = 10^k for k = 5,6,7,8 …)

$$f(x) = \frac{1}{\sqrt{x^2+1} - x}$$

- For large x, $\sqrt{x^2+1} \approx x$ so we expect problems with cancelation in the denominator.
- We can refactor the expression to remove the cancelation.

$$f(x) = \frac{1}{\sqrt{x^2+1} - x} = \frac{\sqrt{x^2+1} + x}{(\sqrt{x^2+1} - x)(\sqrt{x^x+1} + x)} = \frac{\sqrt{x^2} + x}{x^2 + 1 - x^2} = \sqrt{x^2+1} + x$$

x	$\frac{1}{\sqrt{x^2+1} - x}$	$\sqrt{x^2+1} + x$
10^5	200000.223331	200000.000005
10^6	1999984.771129	2000000.000001
10^7	19884107.851852	20000000.000000
10^8	inf	200000000.000000

Solutions degrades until divide by zero

Exercise: The Kahan* Summation Algorithm

solution

- Using the properties of floating point arithmetic, algorithms that reduce round-off errors can be designed.
- A famous one is the Kahan Summation Algorithm. Here it is in pseudo-code

```
Input: a sequence of N values, x[i] i=1,N

   correction = 0.0
   sum = 0.0

   for i = 1 to N:

      xcor = x[i] – correction


      tmpSum =  sum + xcor

      correction = (tmpSum – sum) – xcor

      sum = tmpSum
   }
Output: sum
```

apply a correction to x[i] to account for bits lost in the previous loop iteration

sum is big, but xcor is small. → Low order digits of xcor are lost

(tmpSum-sum) removes high order/common bits through cancellation on subtraction.

What's left are the bits provided by xcor

Subtract xcor so you're left with the lost bits.

*Professor William Kahan of UC Berkeley is the father of modern floating point arithmetic (IEEE-754)