

# Floating Point Arithmetic is not Real



**Tim Mattson**

git clone <https://github.com/tgmattso/Princeton2025.git>

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

# 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.

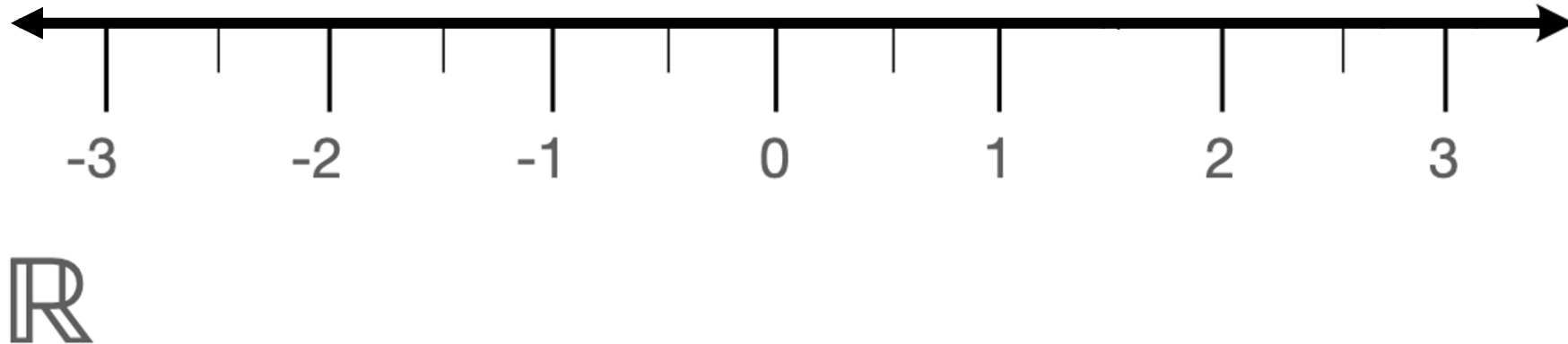
NASTRAN is the world's most widely used finite element code ... in heavy use since 1968

# 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
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  - Numerical Analysis
  - Alternatives to IEEE 754
- Wrap-up/Conclusion

git clone <https://github.com/tgmattso/CompSciForPhys.git>

# Numbers for Humans

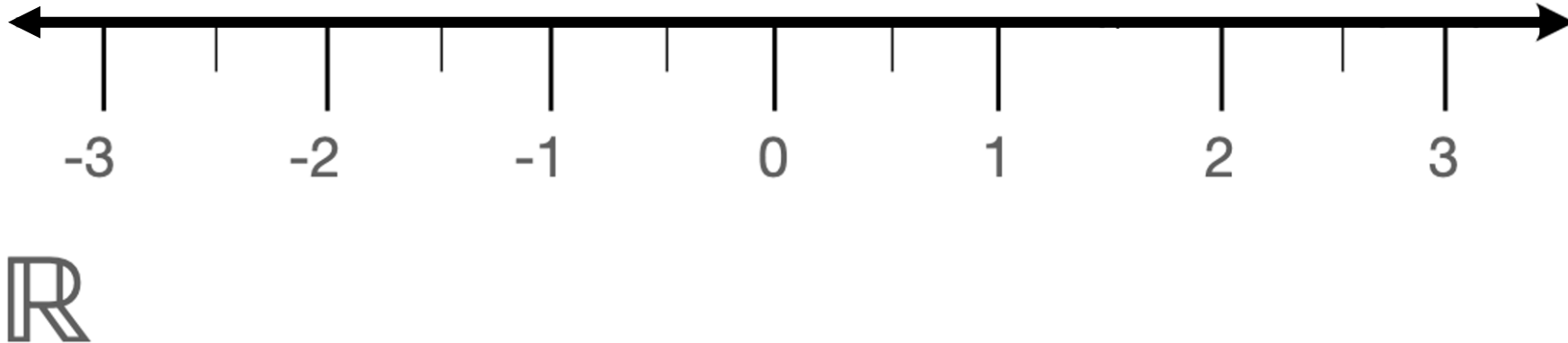


**Real Numbers:** viewed as points on a line ... pairs of real numbers can be arbitrarily close



# Numbers for Humans

## Arithmetic over Real Numbers



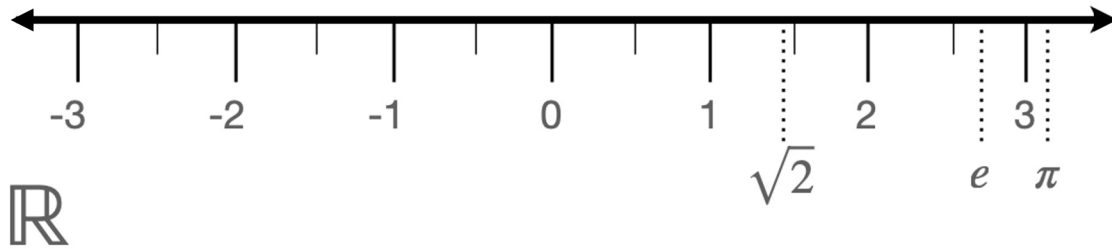
For the arithmetic operators, real numbers define a **closed set** ... for well defined operations and any input real numbers, the arithmetic operation returns a real number.

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

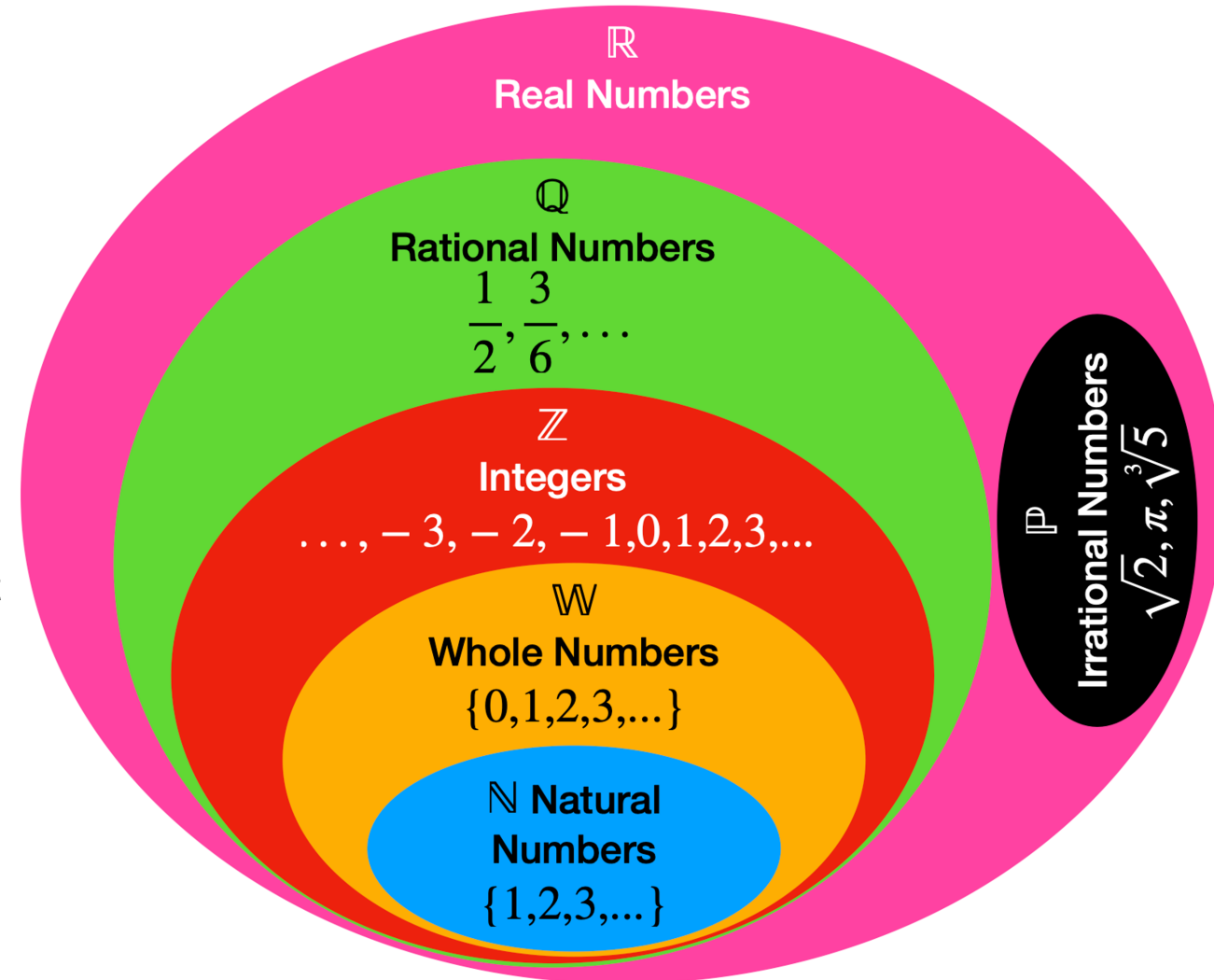
People use many different kinds of numbers



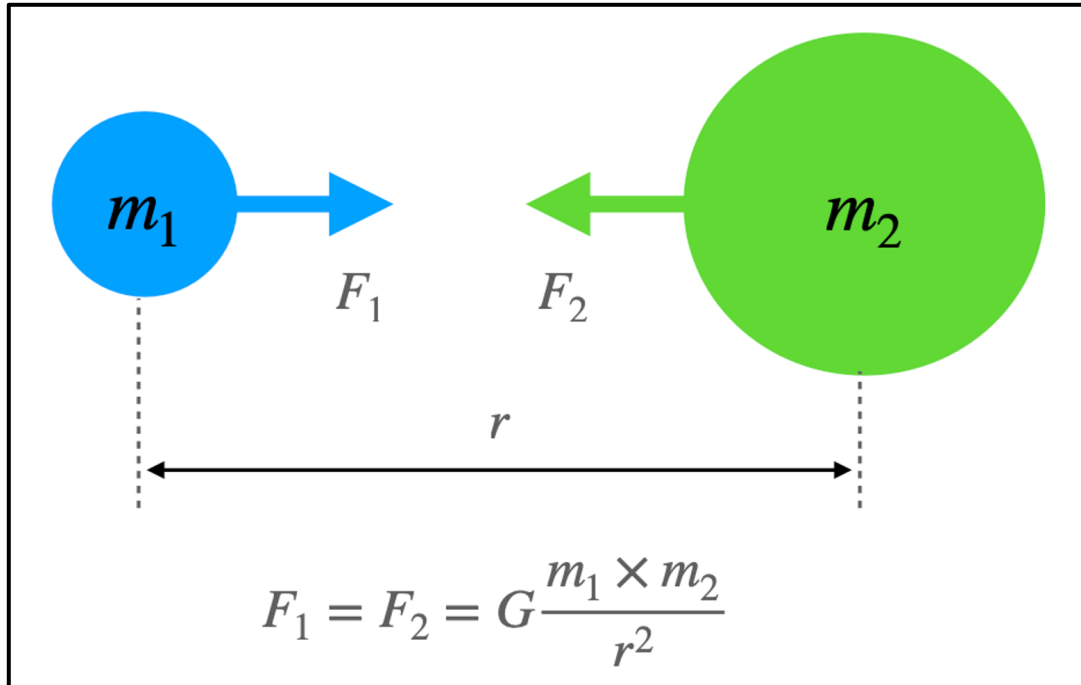
We are used to thinking of different kinds of numbers contained within the set of real numbers

- **Integers:** equally spaced numbers without a fractional part
- **Rational numbers:** Ratios of integers
- **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

The numbers on a computer are just another set of numbers embedded in the set of real numbers



# Numbers for Humans: Example



$$G \approx 0.000000000006674 \frac{\text{m}^3}{\text{kg} \cdot \text{s}^2}$$

Scientific Notation

$$0.000000000006674 \longrightarrow 6.674 \times 10^{-11}$$

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*

# Numbers for Humans

## How do we represent Real Numbers?

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

*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 ...

$$x = (-1)^{\text{sign}} \sum_{i=0}^{\infty} d_i \mathbf{b}^{-i} \mathbf{b}^{\text{exp}} \quad \dots \text{ where } \mathbf{b} \text{ is the } \text{radix}$$

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



# Numbers for Humans

## How do we represent Real Numbers?

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

*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 ...

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

Human's deal nicely with  $\infty$ .  
Computers do not.

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

Humans like a radix = 10.  
Which radix is best for a computer?

# 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.

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

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

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

# 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 tracks time by **accumulating N 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.
  - Do the computations in single precision (float) to make things “more interesting”
- What does your program generate for large N (500000)?

# 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
%
```



# Why did summing 0.01 fail?



I saw this slogan on  
a T-shirt years ago

# 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
$\frac{1}{2^N}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$	$\frac{1}{1024}$	$\frac{1}{2048}$	$\frac{1}{4096}$	$\frac{1}{8192}$	$\frac{1}{16384}$	$\frac{1}{32768}$	$\frac{1}{65536}$	$\frac{1}{131072}$

# Real numbers on a computer are represented as finite precision numbers

- **Conclusion:** Not all decimal numbers 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.

# 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



Who cares?  
Does this really matter?

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



# Patriot Missile system

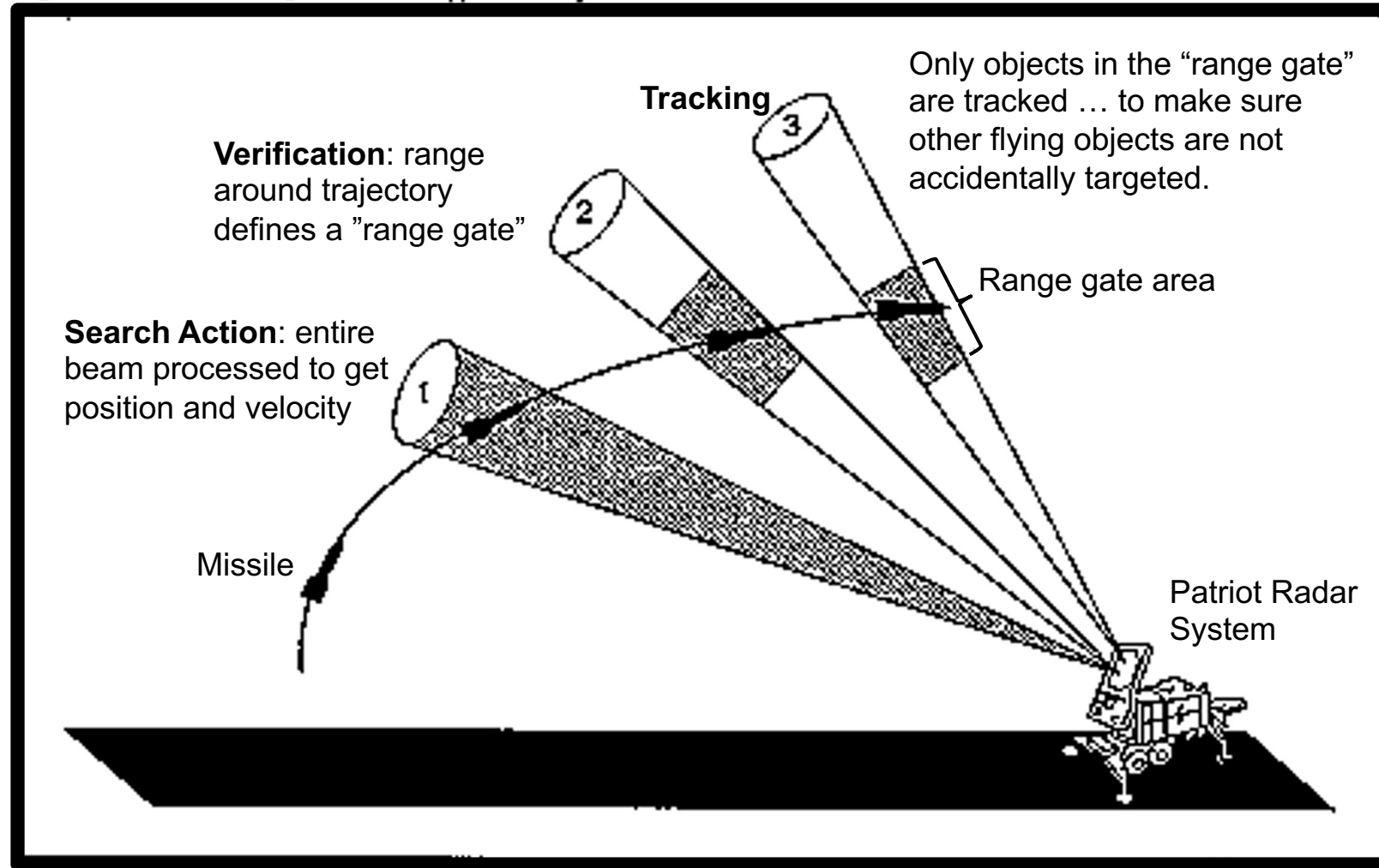
*Patriot missile incident (2/25/91)* . Failed to stop a scud missile from hitting a barracks,



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

# Patriot missile system: how it works

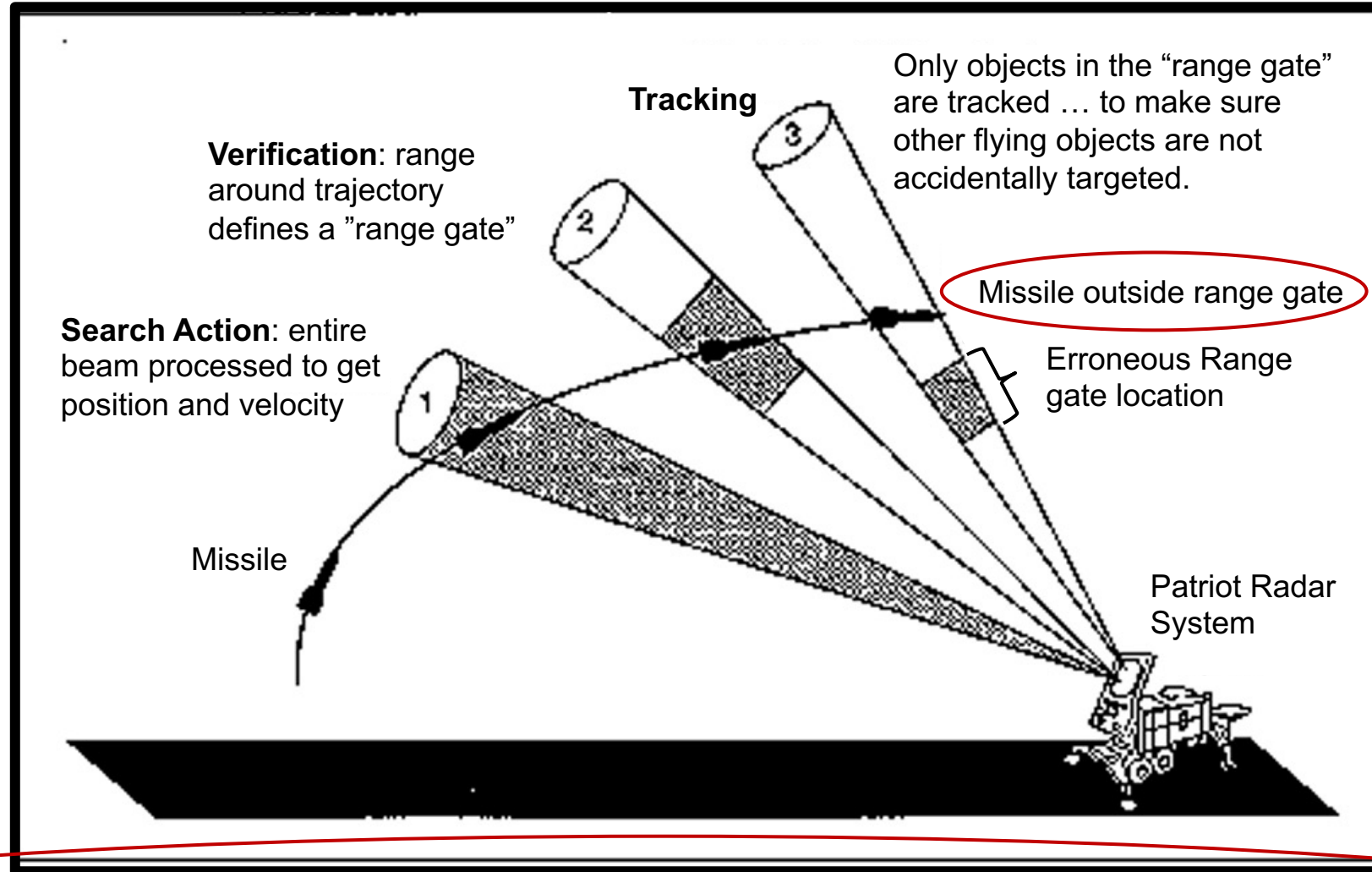
Incoming object detected as an enemy missile due to properties of the trajectory. Velocity and position from Radar fixes trajectory



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 tracks time by **accumulating N 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.**

- Write a program that calculates the sum of the first 10,000,000 numbers around  
10  

```
> ./a.out  
dt = 0.007812500000000000. // dt=1.0/128.0 ... one over a power of two  
sum = 39062.50000000000000000000, dt*Count=39062.500000000000000000
```

  - 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?

# 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

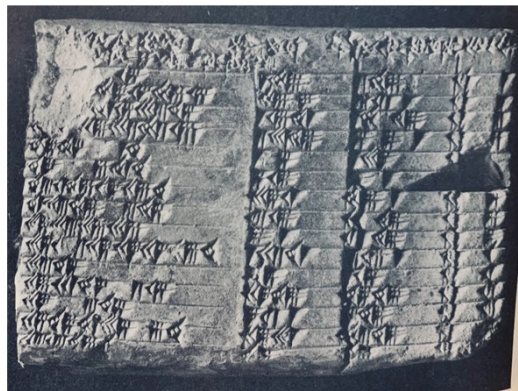
# Outline

- 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
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  - Numerical Analysis
  - Alternatives to IEEE 754
- Wrap-up/Conclusion

# Floating-point Arithmetic Timeline

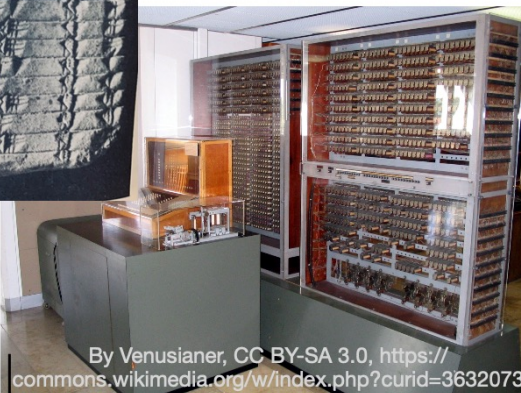
“...the next generation of application programmers and error analysts will face new challenges and have new requirements for standardization. Good luck to them!”

[https://grouper.ieee.org/groups/msc/ANSI\\_IEEE-Std-754-2019/background/ieee-computer.pdf](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<sup>[1]</sup>
- Arithmetic unit: Binary floating-point, 22-bit, add, subtract, multiply, divide, square root<sup>[1]</sup>



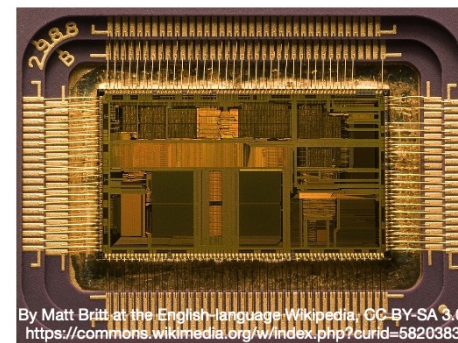
By Venusianer, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=3632073>

DEC PDP-11

1970's  
boom of  
minicomputers

IEEE p745

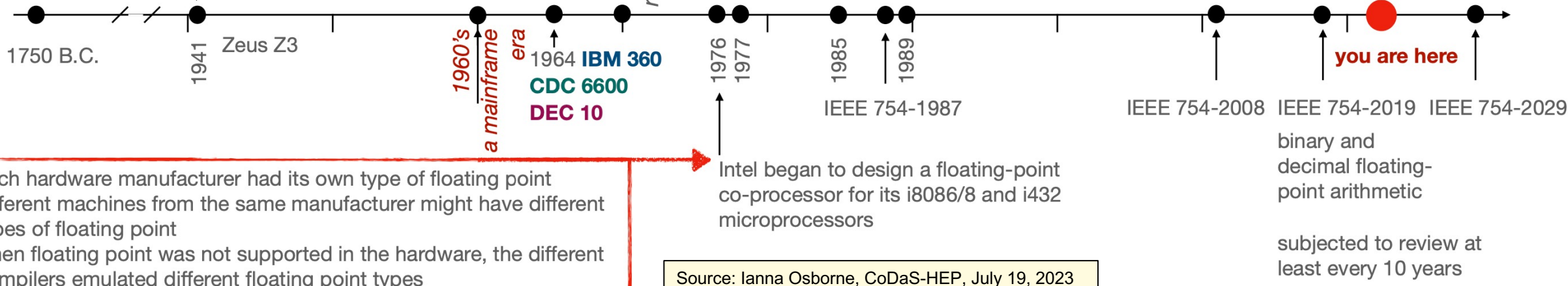
IEEE 754-1985



By Matt Britt at the English-language Wikipedia, CC-BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=5820383>

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.”



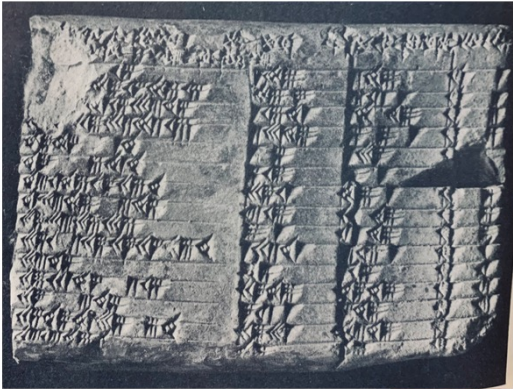




# Floating-point Arithmetic Timeline

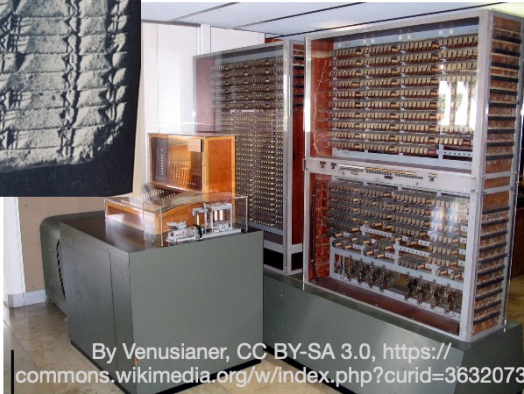
“...the next generation of application programmers and error analysts will face new challenges and have new requirements for standardization. Good luck to them!”

[https://grouper.ieee.org/groups/msc/ANSI\\_IEEE-Std-754-2019/background/ieee-computer.pdf](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<sup>[1]</sup>
- Arithmetic unit: Binary **floating-point**, 22-bit, add, subtract, multiply, divide, square root<sup>[1]</sup>



By Venusianer, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=3632073>

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

Each manufacturer had its own type of floating point. Since the same manufacturer's floating point was not supported in the hardware, the different compilers emulated different floating point types

The era of floating point chaos

DEC

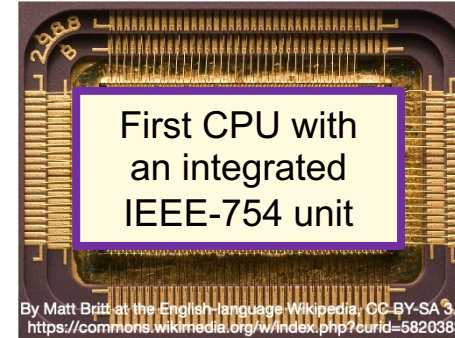
1970's boom of minicomputers

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

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

Intel 80486

the first tightly-pipelined 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**.



First CPU with an integrated IEEE-754 unit

By Matt Britt at the English-language Wikipedia, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=5820383>

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

“new kinds of might eventual kinds of stand fields like arti machine vision recognition, a Some of these accuracy by p faster rather th more precision constraints fro scientific computing.”

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

you are here

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

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 ...

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

$sign \in \{0,1\}$   
 $d_i \in \{0, \dots, (b-1)\}$   
 $e_{min} \leq e \leq e_{max}$

$b \geq 2$  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  $F(b, p, e_{min}, e_{max})$

# Floating Point Number systems: Normalized numbers

Consider representations of the decimal number 0.1

$$1.0 \times 10^{-1}, \quad 0.1 \times 10^0, \quad 0.01 \times 10^1$$

- 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

$$F^*(b, p, e_{\min}, e_{\max})$$

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

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

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

$$d_0 \neq 0$$

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

- $x = 0$  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  $1 - e_{\max}$ )

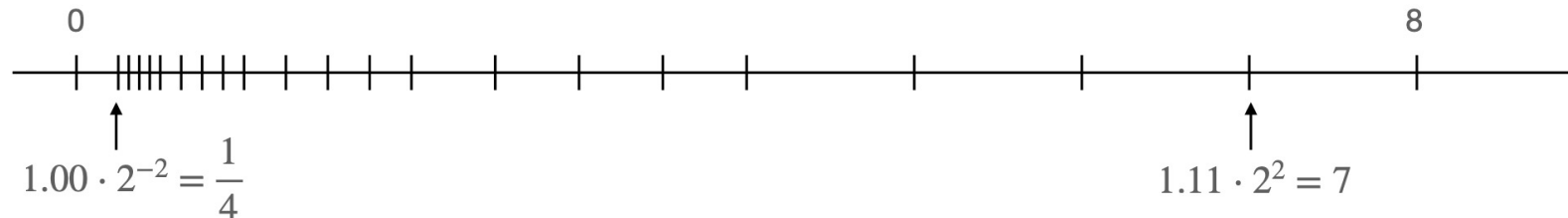


# Normalized Representation

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

$d_0.d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
$1.00_2$	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

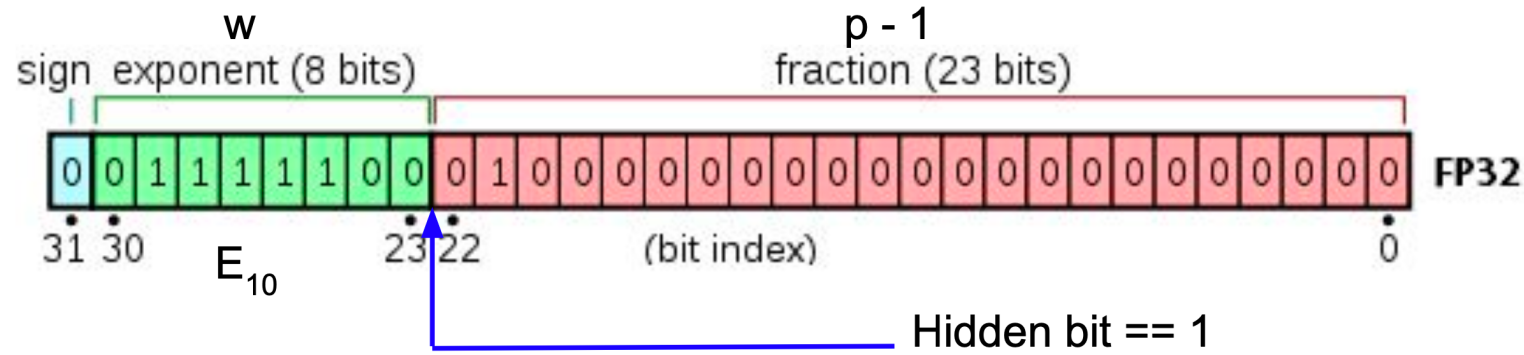




# Outline

- 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
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  - Numerical Analysis
  - Alternatives to IEEE 754
- Wrap-up/Conclusion

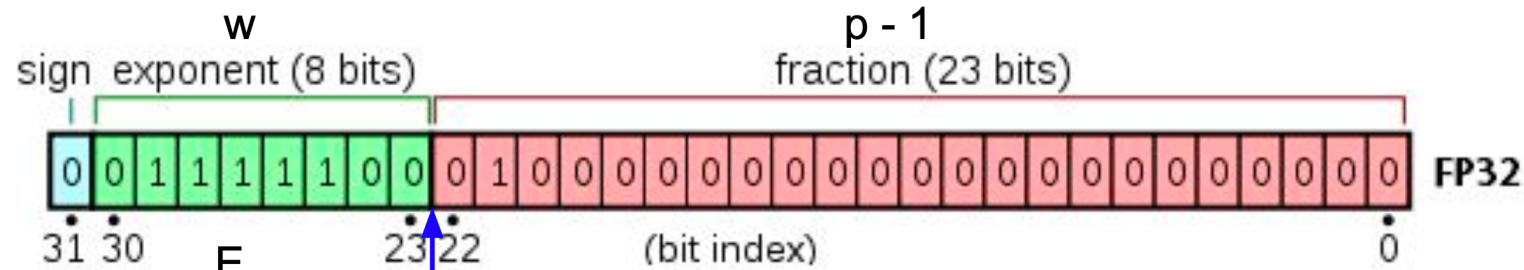
# IEEE 754 Floating Point Numbers



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
- $e_{\max} = -e_{\min} + 1$

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

# Exceptions

- Certain situations outside “normal” behavior are defined as **Exceptions**. Two cases:
  1. **Silent**: An exception occurs, a status flag is set, a result is returned and the computation proceeds. This is the typical case.
  2. **Signaled**: The exception occurs, a signal is raised, and an optional trap function is invoked. Trapping can be set through compiler switches but can seriously slow down code. This is very rarely done ... except by professionals writing low-level math libraries.
- 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

# Special values

- The IEEE 754 standard defines a number of special values

	The special value	exponent	fraction
Regular normalized floating point numbers.	$1. f \times 2^e$	$e_{min} \leq e \leq e_{max}$	Any pattern of 1's and 0's
Denormalized Numbers ... too small to represent as a normalized number.	$0. f \times 2^{e_{min}}$	All 0's ( $e_{min} - 1$ )	$f \neq 0$
	$\pm 0$	All 0's ( $e_{min} - 1$ )	$f = 0$
0 and $\infty$ have signs to work with limits	$\pm \infty$	All 1's ( $e_{max} + 1$ )	$f = 0$
Not a Number (undefined math such as 0/0).	NaN	All 1's ( $e_{max} + 1$ )	$f \neq 0$

- Typically, we do not test for these cases in code, but they do show up from time to time (especially NaN) so its good to be aware of them.

# More about NaNs (Not a Number)

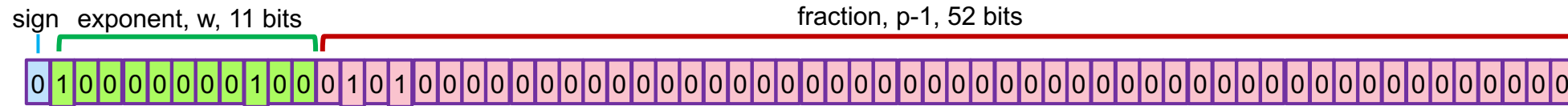
- Here are the cases where a NaN can be produced.

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

- There are 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.

# Writing IEEE 754 numbers in binary

- The number 42.0 written in binary

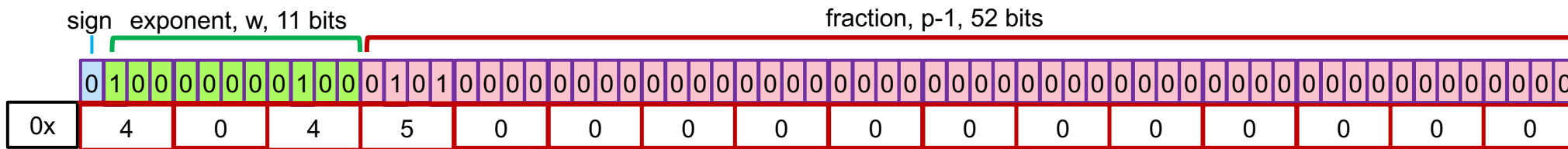


# Keeping track of all 64 locations and writing all those zeros is painful

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	0x7FEFFFFFFF
Smallest normal	0x0010000000000000
Largest subnormal	0x000FFFFFFFFF
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

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



# 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  $\varepsilon$ .
- Exercise ... Write a program that computes  $\varepsilon$ . Then compare your result to the theoretical value.

# Finding the machine epsilon

- The machine epsilon is the last number added to one that yields a sum that is greater than one.
- Note: our while loop ends when we can't see the difference between 1.0 and (1.0+eps), Hence, it goes one step too far and we have to adjust for that extra step.

```
#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++;
    }
    iters--;eps*2.0; //adjust for one step too far
    printf("epsilon is 2 to the -%ld or %g\n",
           sizeof(TYPE),iters,eps);
}
```

epsilon is 2 to the -23 or 1.19209e-07

---

## Theoretical Result

- 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 ( $F^*(2,24,-126,127)$ ). The number closest but larger than 1.0 is:

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

# 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
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  - Numerical Analysis
  - Alternatives to IEEE 754
- Wrap-up/Conclusion

# Addition with floating point numbers

- Lets keep things simple ... we will use  $F^*(10, 3, -2, 2)$
- Find the sum ...  $1.23 \times 10^1 + 3.11 \times 10^{-1}$ 
  - Align smaller number to the exponent of the larger number  
 $0.0311 \times 10^1$
  - Add the two aligned numbers .....

$$\begin{array}{r} 1 \ . \ 2 \ 3 \quad \quad \times 10^1 \\ + \ 0 \ . \ 0 \ 3 \ 1 \ 1 \quad \times 10^1 \\ \hline 1 \ . \ 2 \ 6 \ 1 \ 1 \quad \times 10^1 \end{array}$$

- 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).

# 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?

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

float or double

```
    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);
}
```

# 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 later 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 greatly reduces 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	16.695311265857270655	15.40368271
backward	16.695311265859963612	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
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
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  - Numerical Analysis
  - Alternatives to IEEE 754
- Wrap-up/Conclusion

# What can go wrong with subtraction? Cancellation

- Consider two numbers ...

3.141592653589793

16 digits of pi

3.141592653585682

12 digits of pi

Their difference (in real arithmetic) →

0.000000000004111

=  $4.111 \times 10^{-12}$

Storage of numbers and difference with float →

0.000000e+00

**Complete loss of  
accuracy**

Storage of numbers and difference with double →

4.110933815582030e-12

**Partial loss of  
accuracy**

- The machine epsilon for a double is  $2.22045e-16$ . The above error is large compared to epsilon.

Subtracting two number of similar magnitude cancels high order bits.

# Exercise: Implement a Series summation to find $e^x$

- A Taylor/Maclaurin series expansion for  $e^x$

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

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 $e^x$

Hints

- A Taylor/Maclaurin series expansion for  $e^x$

Compare computation of  $e^x$  directly and as  $e^{-x} = 1/e^x$ .

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

- 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:

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

- 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. Or think about what you did when computing the machine epsilon.

# Solution: Implement a Series summation to find e<sup>x</sup>

- A Taylor/Maclaurin series expansion for e<sup>x</sup>

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

- The computation of x<sup>n</sup> 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:

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

- 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<sup>x</sup> directly and as e<sup>x</sup> = 1/ e<sup>-x</sup>.

```
#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)	x	exp(x) math.h	MyExp(x)	1/MyExp( x )
5	148.413	148.413	-5	6.73795e-03	6.73714e-03	6.73795e-03
10	22026.5	22026.5	-10	4.53999e-05	-5.23423e-05	4.53999e-05
15	3.26902e+06	3.26902e+06	-15	3.05902e-07	-2.23869e-02	3.05902e-07
20	4.85165e+08	4.85165e+08	-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.

# 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
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
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  - Numerical Analysis
  - Alternatives to IEEE 754
- Wrap-up/Conclusion

# 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,-126,127)):

$$\begin{array}{r} (1.000000000000000000000000)_2 \cdot 2^0 \\ - (1.000000000000000000000001)_2 \cdot 2^{-25} \end{array}$$

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

$$\begin{array}{r} (1.000000000000000000000000)_2 \cdot 2^0 \\ - (0.0000000000000000000000000100000000000000000000001)_2 \cdot 2^0 \\ \hline = (0.1111111111111111111111111101111111111111111111110)_2 \cdot 2^0 \end{array}$$

- Then normalize the result

$$(1.111111111111111111111111011111111111111111111111)_2 \cdot 2^{-1}$$

- Then round to nearest to fit into the destination format

$$(1.111111111111111111111111)_2 \cdot 2^{-1}$$



# 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 ( $F^*(2,24,-126,127)$ ):

$$(1.000000000000000000000000)_2 \cdot 2^0$$

$$- (1.000000000000000000000001)_2 \cdot 2^{-25}$$

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

$$\begin{array}{r} (1.000000000000000000000000)_{2} \cdot 2^0 \\ - (0.0000000000000000000000001000000000000000000000001)_{2} \cdot 2^0 \\ \hline = (0.1111111111111111111111111011111111111111111111110)_{2} \cdot 2^0 \end{array}$$

- Then normalize the result

[illegible]

- Then round to nearest to fit into the destination format

$$(1.111111111111111111111111)_{\text{2}} \cdot 2^{-1}$$

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

# IEEE 754 arithmetic and rounding

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

[illegible]

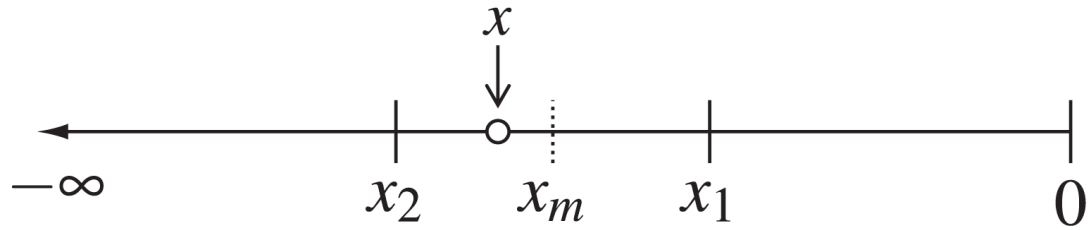
Next 2 bits in the answer:  
the **guard** and **rounding** bits

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

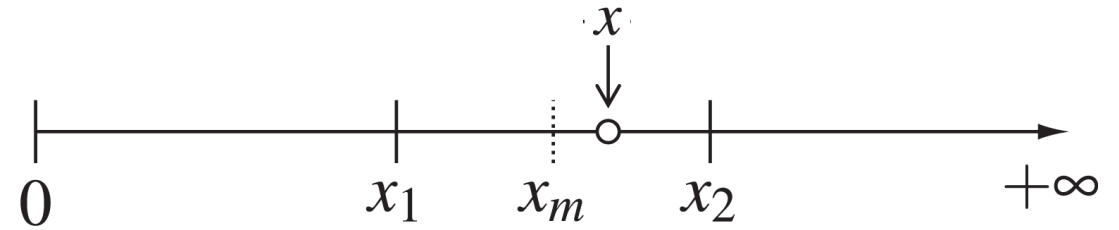
- The Guard, Rounding, and Sticky bits are sufficient to support all the IEEE 754 rounding modes to yield the same result you'd get from an exact computation followed by rounding into the target format.
- Correctly rounded results are required for the basic arithmetic operations (including FMA) but also **square root**, **remainder**, and **conversion between Integer and Floating point numbers** ... but not for conversion between decimal and binary floating point.

# IEEE 754 Rounding Modes

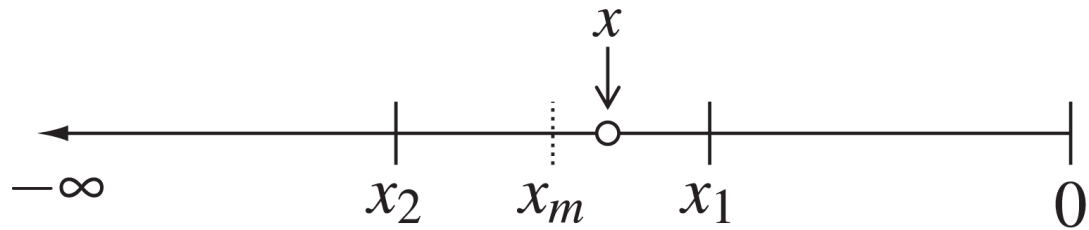
Consider a real number  $x$  that falls between its two nearest floating point numbers ( $x_1$  and  $x_2$ ). At the midpoint between  $x_1$  and  $x_2$  is the real number  $x_m$ . We have four cases to consider when thinking about rounding.



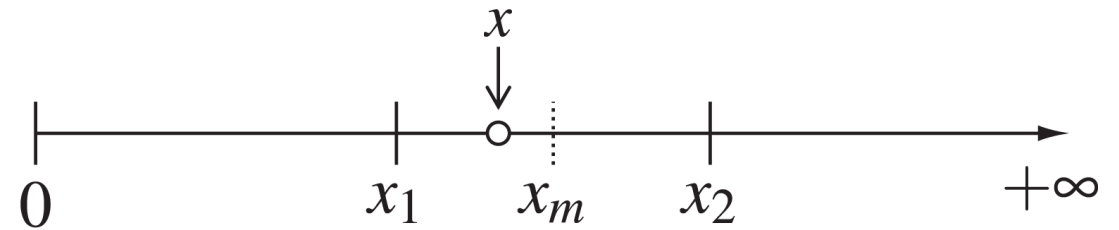
**(a)**  $x < x_m < 0$



**(b)**  $x > x_m > 0$



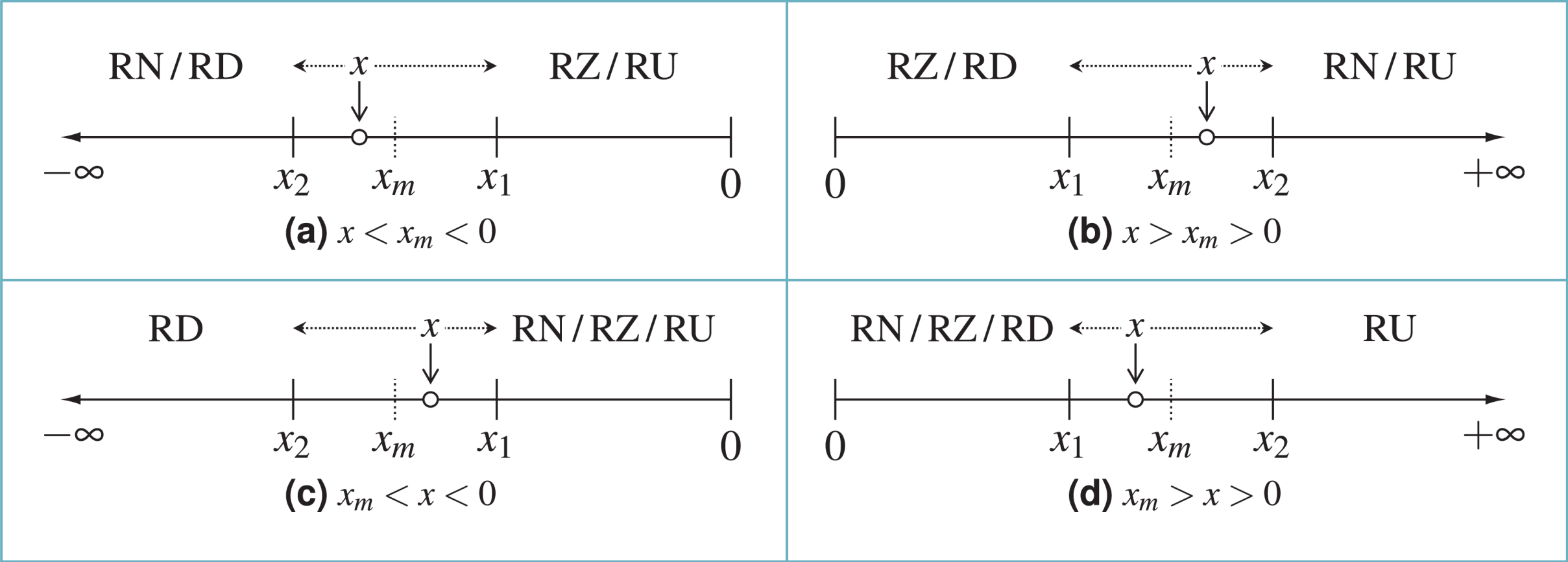
**(c)**  $x_m < x < 0$



**(d)**  $x_m > x > 0$

# 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

# You must be careful how you manage rounding...

*Vancouver stock exchange index* undervalued by 50%

(Nov. 25, 1983)



See <http://ta.twi.tudelft.nl/usersvuiik/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  $\text{REAL}^*4$  format, losing 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

The 4 rounding modes in IEEE 754

Two versions of round to nearest...

- Nearest, on a tie, round to even
- Nearest, on a tie, away from zero

Default  
rounding  
mode

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 do 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?

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 do 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

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 <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
-  – Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  - Numerical Analysis
  - Alternatives to IEEE 754
- Wrap-up/Conclusion



# Properties of Floating point arithmetic

- IEEE 754 defines the concept of an ***correctly rounded*** results of operation.
  - An correctly rounded result is equivalent to a computation carried out to infinite precision rounded to fit in the designated storage format (e.g., float or double).
- The standard requires a set of operations that must be correctly rounded
  - Add, subtract, multiply, divide, remainder
  - Square root, fused multiply-add, minimum, maximum
  - Comparisons and total ordering
  - Conversions between formats
- Correctly rounding results are recommended (but not required) for:
  - Exponentials
  - Logarithms
  - Reciprocal square root
  - Trigonometric functions

# Floating point arithmetic is not associative or distributive

- **IEEE 754** guarantees that a single arithmetic operation produces a correctly rounded result ... but that guarantee does not apply to multiple operations in sequence.
- Floating point numbers are:
  - Commutative:  $A * B = B * A$
  - NOT Associative:  $A * (C * B) \neq (A * C) * B$
  - NOT Distributive:  $A*(B+C) \neq A*B + A*C$
- All these computations were done in a C program using type float

```
a = 11111113;  b= -11111113;  c = 7.51111111f;
(a + b) + c = 7.511111
a + (b + c) = 8.000000

Correct answer: 7.511111
```

```
a = 20000;  b= -6;  c = 6.0000003;
(a*b + a*c) = 0.007812
a*(b + c) = 0.009537

Correct answer: 0.006000
```

- Python promotes floating point number to double, but even with the extra precision, you can run into trouble

Python

```
a = 1.e20;  b= -1.e20;  c=1.0
(a+b) + c = 1.0
a+(b+c) = 0.0

Correct answer: 1.0
```

... But C using float gets this case right



```
a = 1.e20f;  b= -1.e20f;  c=1.0f
(a + b) + c = 1.000000
a + (b + c) = 1.000000
```

# Exercise: Summation with floating point arithmetic

- We have provided a programs that create a sequence of random numbers greater than zero. They are written in C (summation.c) and C++ (summationCpp.cc) so you can do this exercise in either language.
- In the functions in the file UtilityFunctions.c, 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).
- Add code to smmation.c or summationCpp.cc 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).
- Compile your program as:  
g++ summationCpp.cc UtilityFunctions.c  
gcc summationCpp.c UtilityFunctions.c
- Using what you know about floating point arithmetic, is there anything you can think of doing to improve the quality of your sum?

# Summation program

```
#include "UtilityFunctions.h" // FillSequence() comes from this module

#define N 100000 //length of sequence of numbers to work with
int main ()
{
    float seq[N]; //Sequence to sum
    float True_sum; //The best estimate of the actual sum
    float sum = 0.0f;

    FillSequence(N, seq, &True_sum); // Fill seq with N values > 0

    for(int i=0; i<N; i++)sum += seq[i];

    printf(" Sum = %f, Estimated sum = %f\n",sum,True_sum);
}
```

```
> gcc summation.c UtilityFunctions.c
> ./a.out
> Sum = 2502476.500000, Estimated sum = 2502458.750000
```

This result is kind  
of awful

# Summation program

Let's do the sum in parallel and see how the answer varies with the number of threads

```
#include <omp.h>
#include "UtilityFunctions.h" // FillSequence() comes from this module

#define N 100000 //length of sequence of numbers to work with
int main ()
{
    float seq[N]; //Sequence to sum
    float True_sum; //The best estimate of the actual sum
    float sum = 0.0f;

    FillSequence(N, seq, &True_sum); // Fill seq with N values > 0

    #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<N; i++)sum += seq[i];

    printf(" Sum = %f, Estimated sum = %f\n",sum,True_sum);
}
```

Values with 1 to 8 threads

1	2502476.5
2	2502457.0
4	2502459.25
8	2502459.0

True value = 2502458.75

# Sequence Generation

- I created a particularly awful sequence to sum

```
void FillSequence(int N, float *seq, float *True_sum)
{
    float shift_up    = 100.0f;
    float shift_down  =  0.000001f;
    double up_sum     = 0.0d,  down_sum = 0.0d;

    for(int i=0; i<N; i++){
        if(i%2==0){
            seq[i]    = (float) frandom() * shift_up;
            up_sum    += (double) seq[i];
        }
        else {
            seq[i]    = (float) frandom() * shift_down;
            down_sum += (double) seq[i];
        }
    }
    *True_sum = (float)(up_sum + down_sum);
}
```

Alternating big and small numbers to maximize opportunities for loss of precision when summing the numbers.

Notice how I estimate the “true” value by summing big numbers and little numbers separately before combining them.

# Summation program

Sort from small to large before summing the sequence

```
#include "UtilityFunctions.h" // FillSequence() comes from this module
```

```
#define N 100000 //length of sequence of numbers to work with
```

```
int main ()  
{
```

```
    float seq[N]; //Sequence to sum  
    float True_sum; //The best estimate of the actual sum  
    float sum = 0.0f;
```

```
    FillSequence(N, seq, &True_sum); // Fill seq with N values > 0
```

```
    qsort(seq, N, sizeof(int), compare); // Sort from smallest to largest  
    for(int i=0; i<N; i++)sum += seq[i];
```

```
    printf(" Sum = %f, Estimated sum = %f\n",sum,True_sum);
```

```
}
```

```
> gcc summation.c UtilityFunctions.c  
> ./a.out  
> Sum = 2502455.500000, Estimated sum = 2502458.750000
```

The sorted sequence decreases loss of precision since magnitudes of numbers are closer together in a sorted sequence


From 2502476.5 to 2502455.5  
That's a big improvement

# 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
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
Basic arithmetic operations over Real numbers are commutative, distributive and associative.	Basic operations over floating point numbers are commutative, but NOT associative or distributive.



# 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
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  -  – Changing the math
  - Numerical Analysis
  - Alternatives to IEEE 754
- Wrap-up/Conclusion

# Changing the math to make things better...

## Example: Solution to the quadratic equation.

- Consider a quadratic equation.

$$ax^2 + bx + c = 0$$

- Using real arithmetic the solution is.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Two sources of cancelation

# A numerically superior way to solve the quadratic equation. $ax^2 + bx + c = 0$

- In general, there are two solutions:

$$x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

- Recall that  $(\alpha - \beta)(\alpha + \beta) = \alpha^2 - \beta^2$  so we can rearrange the product  $x_-x_+$

$$x_-x_+ = \frac{-b - \sqrt{b^2 - 4ac}}{2a} * \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{b^2 - (b^2 - 4ac)}{4a^2} = \frac{4ac}{4a^2} = \frac{c}{a}$$

- Hence, compute  $x_-$  directly and use  $x_-x_+ = \frac{c}{a}$  to get the other solution

## Exercise: Refactoring functions to improve floating point behavior

- Evaluate the following function for large  $x$   
( $x = 10^k$  for  $k = 5, 6, 7, 8 \dots$ )
- Can you refactor the function to make it numerically more stable?

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

## Exercise: Refactoring functions to improve floating point behavior

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


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

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

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

# 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
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  -  – Numerical Analysis
  - Alternatives to IEEE 754
- Wrap-up/Conclusion

# 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 computer.

# 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:  $\Delta x$  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  $\Delta x \rightarrow$  small  $\Delta y$
  - We call this a **well conditioned** problem
- For a large condition number,  $C$ :
  - Small  $\Delta x \rightarrow$  large  $\Delta y$
  - 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:  $\Delta x$  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  $\Delta x \rightarrow$  small  $\Delta y$
  - We call this a **well conditioned** problem
- For a large condition number,  $C$ :
  - Small  $\Delta x \rightarrow$  large  $\Delta y$
  - 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 can we use the properties of floating-point arithmetic to improve summation?**

# Compensated summation

## (i.e., Kahan Summation)

- Summation is notorious for errors from loss of precision when two numbers of widely different magnitudes are added. Kahan Summation corrects for this error.

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

$cor = 0.0$

$sum = 0.0$

for  $i = 1$  to  $N$ :

$xcor = x[i] - cor$

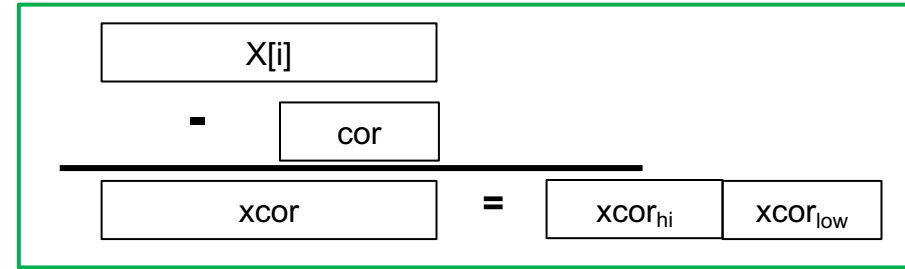
$tmpSum = sum + xcor$

$cor = (tmpSum - sum) - xcor$

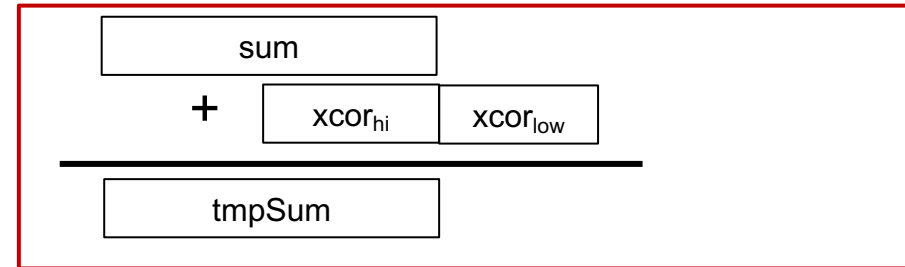
$sum = tmpSum$

}

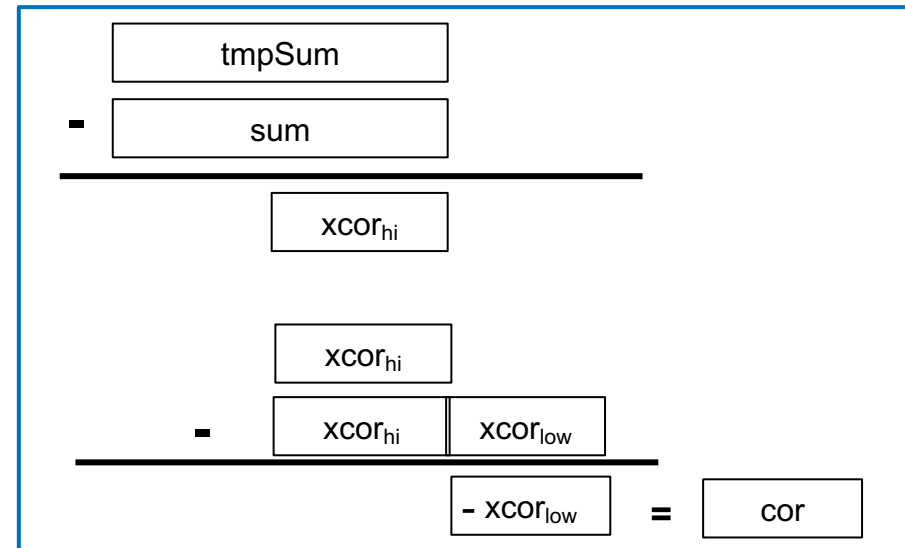
Output:  $sum$



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



Sum grows to be a “big” number. Add a small number ( $xcor$ ) and you lose low order bits ( $xcor_{low}$ )



Recover high order bits from  $xcor$

Recover the lost low order bits from  $xcor$  to apply as a correction when computing the next term in the sum

# Summation program

```
#include "UtilityFunctions.h" // FillSequence() comes from this module

#define N 100000 //length of sequence of numbers to work with
int main ()
{
    float seq[N]; //Sequence to sum
    float True_sum; //The best estimate of the actual sum
    float sum = 0.0f;

    FillSequence(N, seq, &True_sum); // Fill seq with N values > 0

    KahanSummation(N, seq, sum);


    printf(" Kahan Sum = %f, Estimated sum = %f\n", sum, True_sum);
}
```

Our original sum value  
was 2502476.50 which  
was an awful result

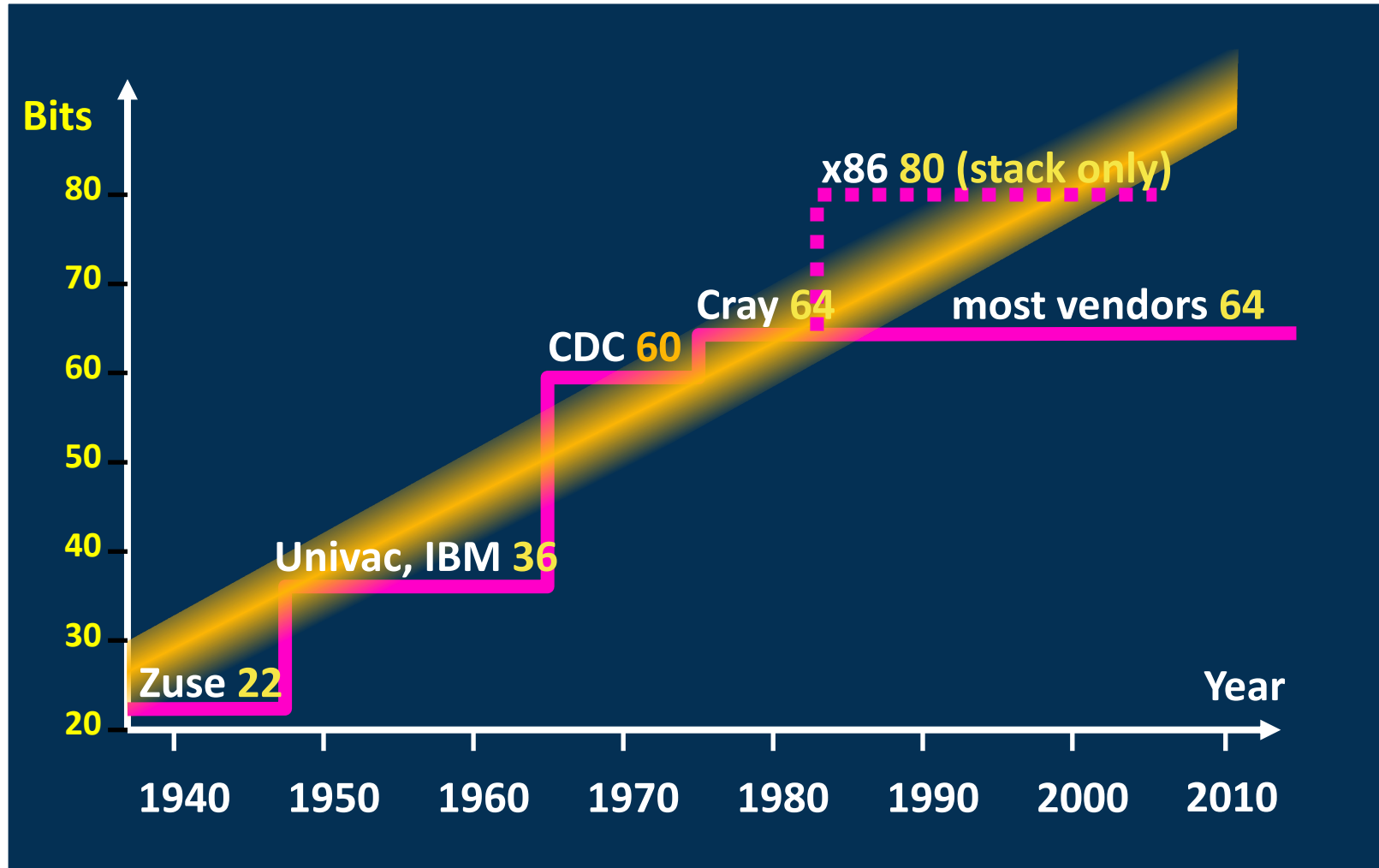
The Kahan Sum and the  
estimated correct sum  
match  
2502458.75

```
> gcc summation.c UtilityFunctions.c
> ./a.out
> Kahan sum = 2502458.750000, Estimated sum = 2502458.750000
```

# 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
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  - Numerical Analysis
  -  – Alternatives to IEEE 754
- Wrap-up/Conclusion

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



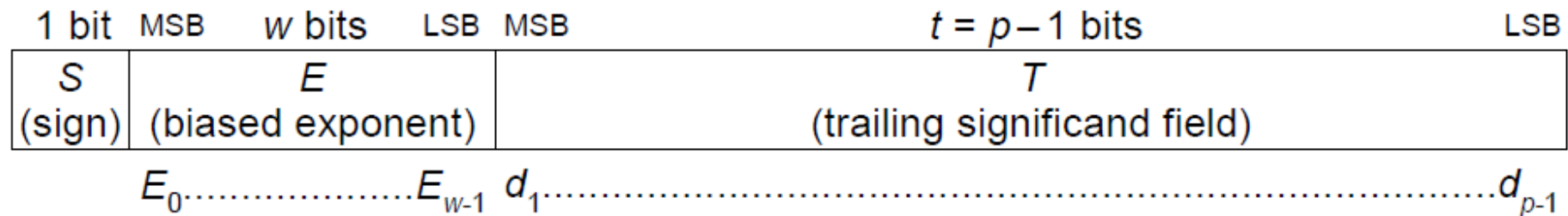
Is 64 bits enough? Is it too much? We're *guessing*.

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

# Quad Precision

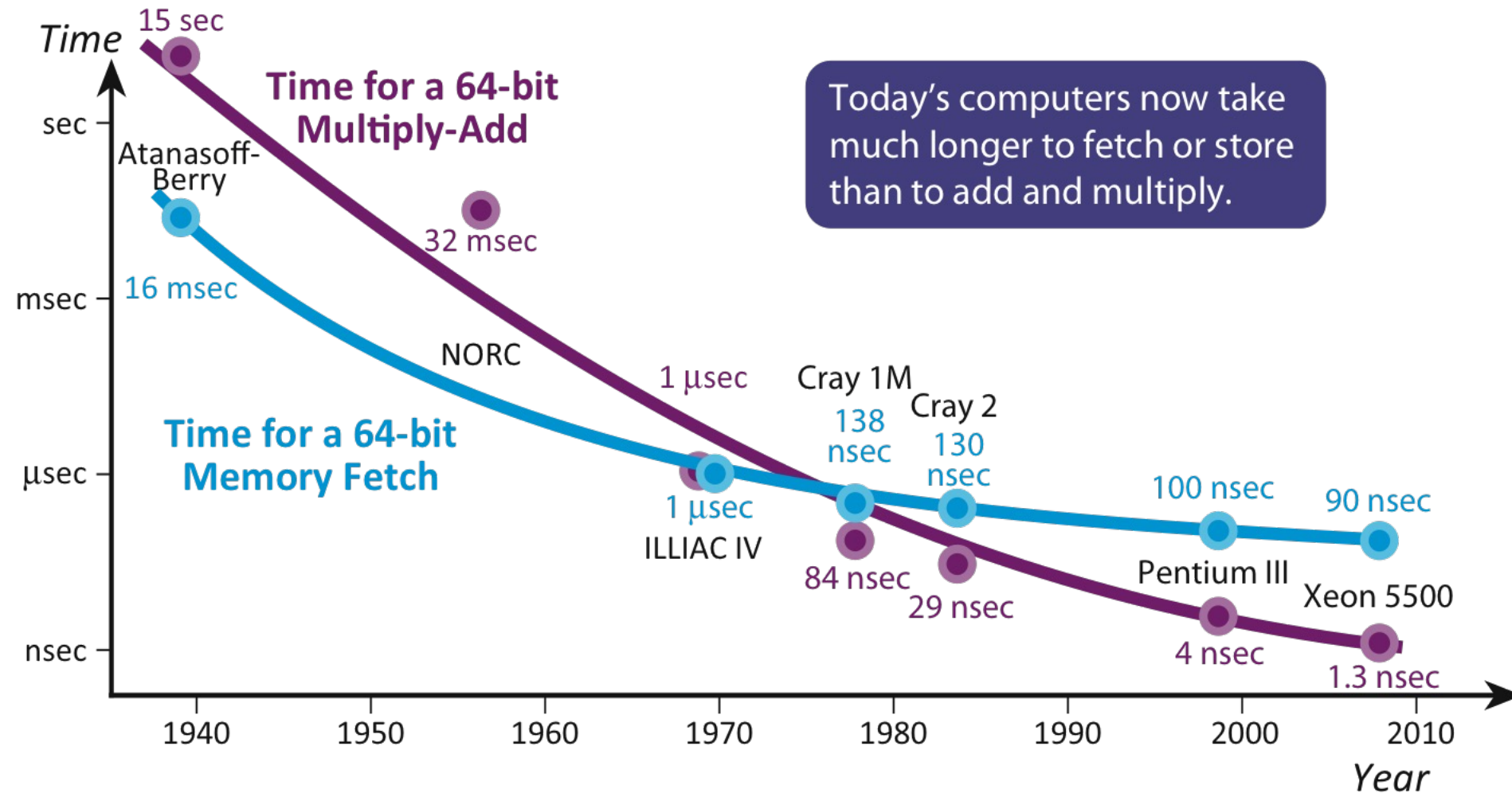
- IEEE 754™ defines a range of formats including quad (128)

	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



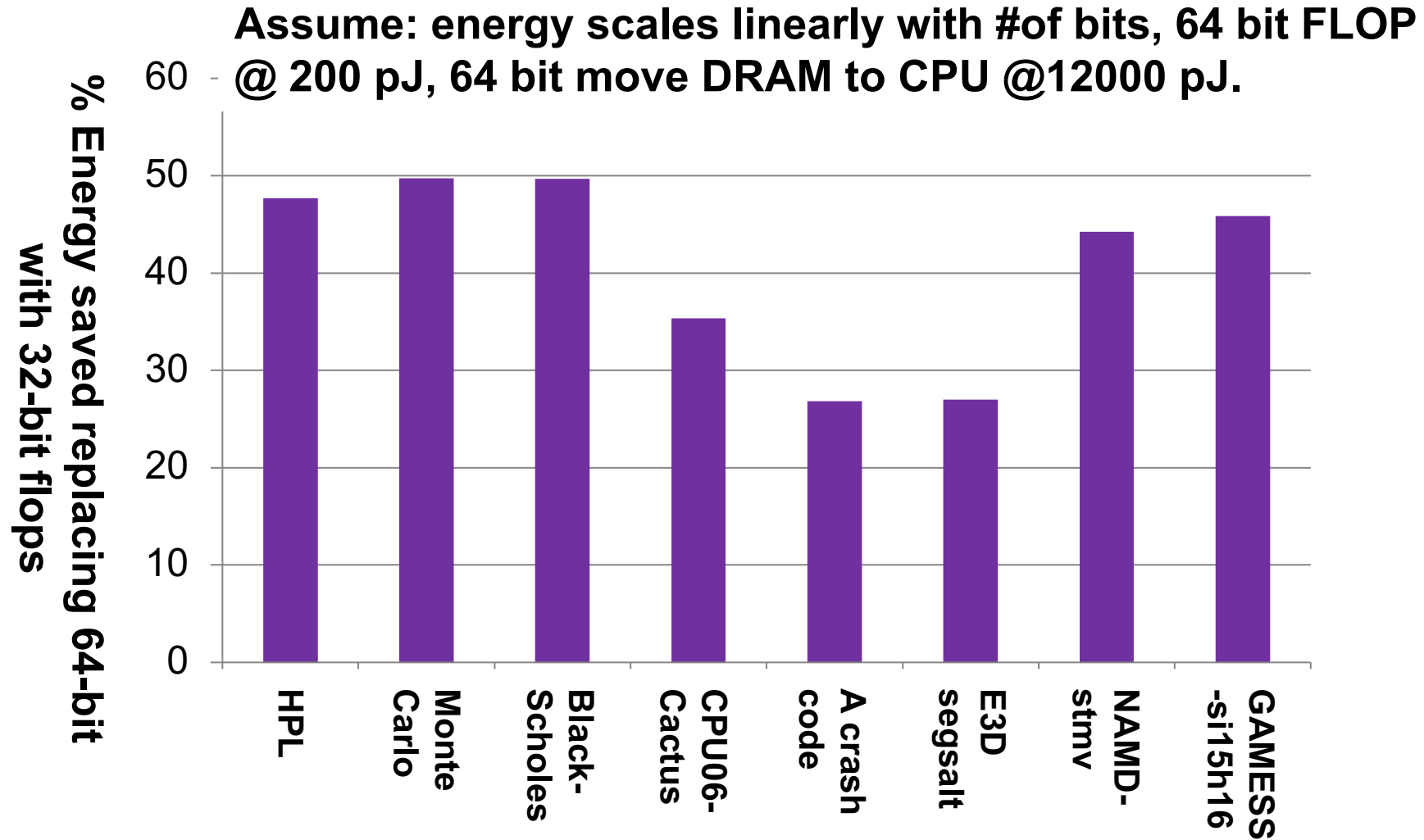
# Energy implications of floating point numbers: 32 bit vs. 64 bit numbers

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

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



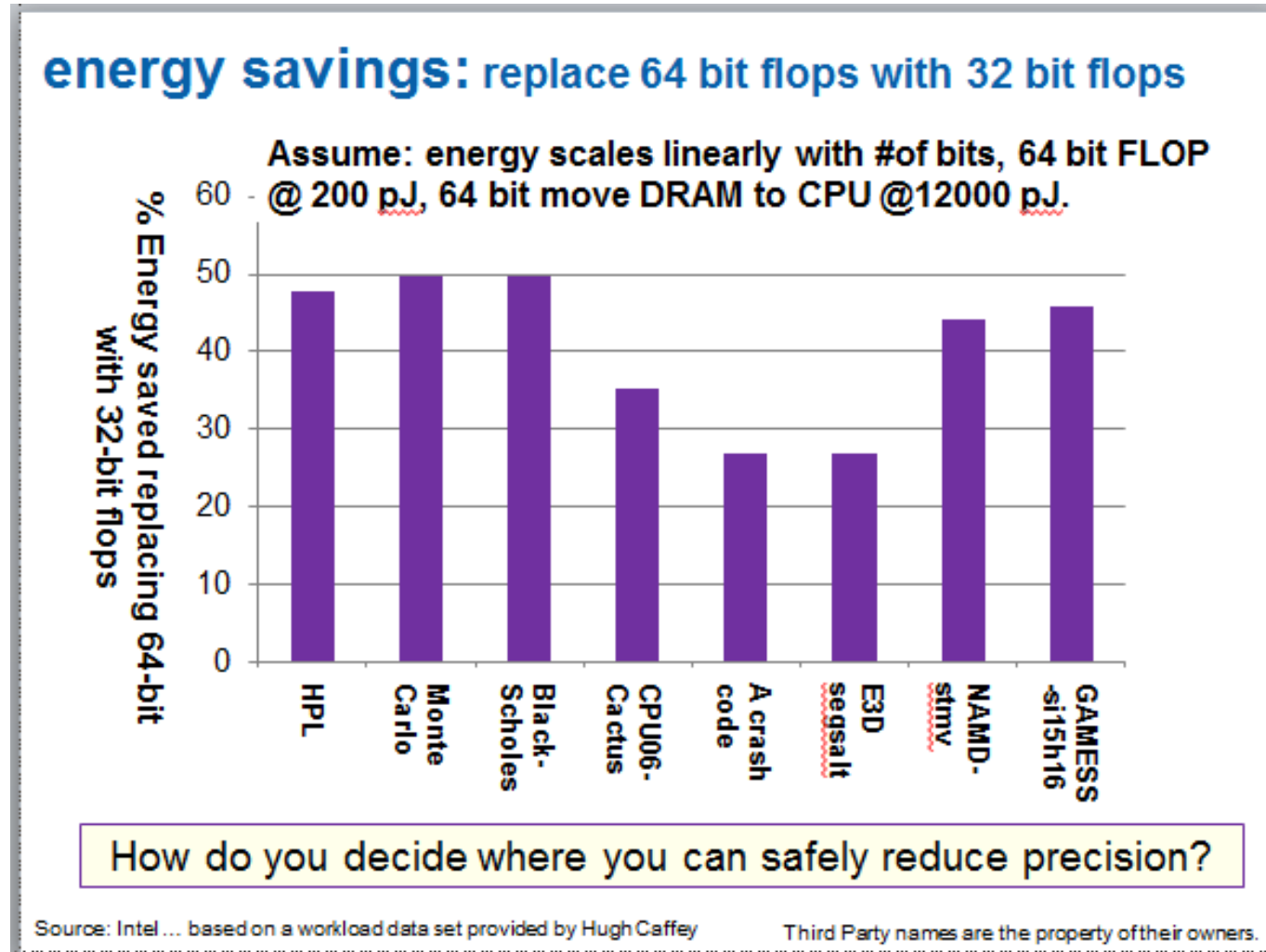
# energy savings: replace 64 bit flops with 32 bit flops



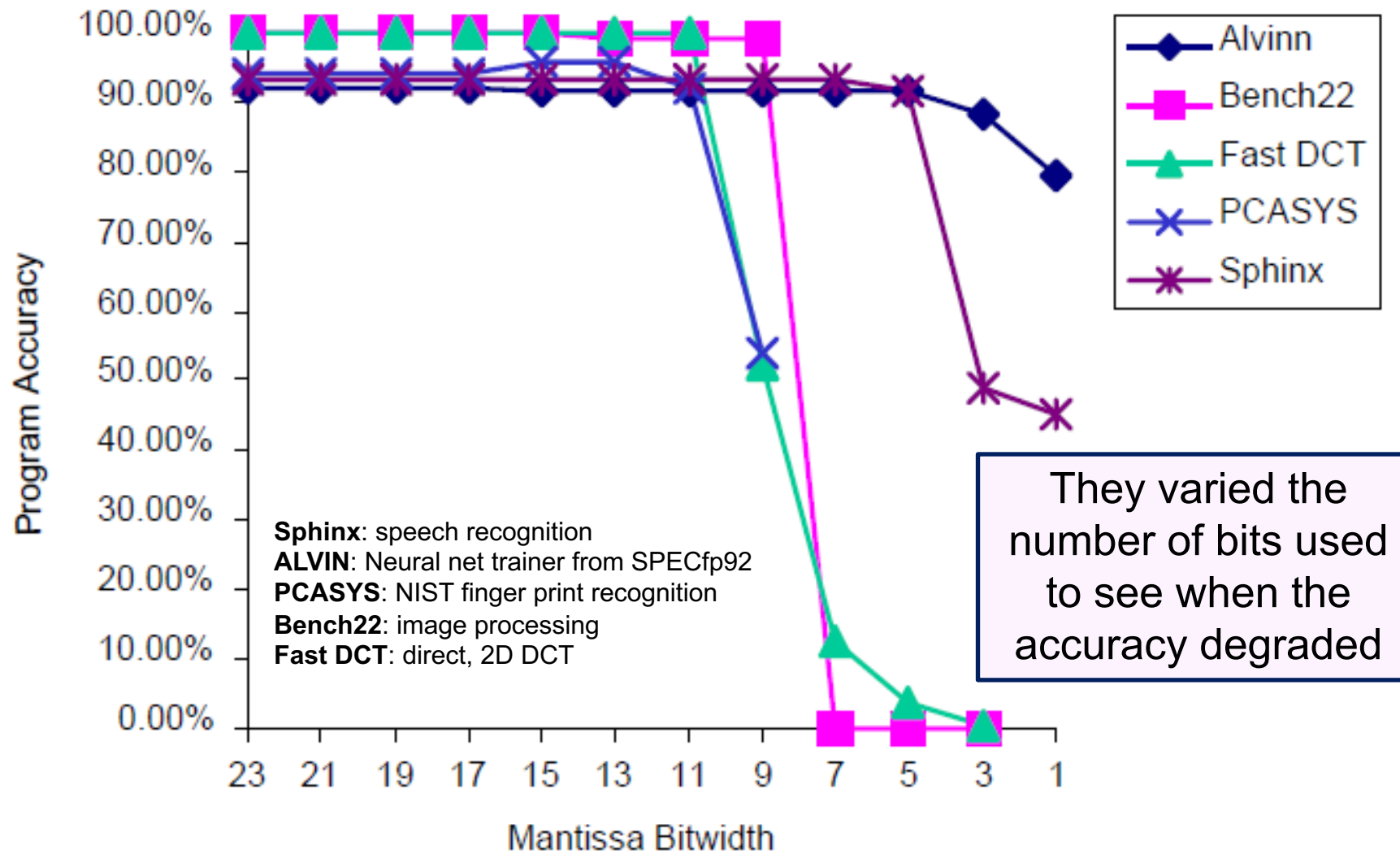
How do you decide where you can safely reduce precision?

# 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,

**... or give up on floating point numbers and use a safe arithmetic system instead.**

**Interval Arithmetic**

# Interval Numbers

- Interval number: the range of possible values within a closed set

$$\mathbf{x} \equiv [\underline{x}, \bar{x}] := \{x \in R \mid \underline{x} \leq x \leq \bar{x}\}$$

- Representing real numbers:

- A single floating point number

$$1/3 \approx 0.333333$$

- An interval that bounds the real number

$$1/3 \in [0.33333, 0.33334]$$

- Representing physical quantities:

- An single value (e.g. an average)

$$radius_{earth} \approx 6371 \text{ km}$$

- The range of possible values

$$radius_{earth} \in [6353, 6384] \text{ km}$$

# Interval Arithmetic

Let  $x = [a, b]$  and  $y = [c, d]$  be two interval numbers

1. Addition  $x + y = [a, b] + [c, d] = [a + c, b + d]$

2. Subtraction  $x - y = [a, b] - [c, d] = [a - d, b - c]$

3. Multiplication  $xy = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$

4. Reciprocal  $1 / y = [1/d, 1/c]$

5. Division 
$$x/y = \begin{cases} x \cdot 1/y & c, d \neq 0 & y \notin 0 \\ [-\infty, \infty] & c, d \neq 0 & y \in 0 \end{cases}$$

# Properties of Interval Arithmetic

Let  $x$ ,  $y$  and  $z$  be interval numbers

## 1. Commutative Law

$$x + y = y + x$$

$$xy = yx$$

## 2. Associative Law

$$x + (y + z) = (x + y) + z$$

$$x(yz) = (xy)z$$

## 3. *Distributive Law does not always hold, but*

$$x(y + z) \subseteq xy + xz$$

# Functions and Interval arithmetic

- Interval extension of a function

$$[f]([x]) \supseteq \{f(y) | y \in [x]\}$$

- Naively can just replace variables with intervals. But be careful ... you want an interval extension that produces bounds that are as narrow as possible. For example ...

$$f(x) = x - x$$

$$\text{let } x = [1,2]$$

$$f[x] = [1 - 2, 2 - 1] = [-1,1]$$

- An interval extension with tighter bounds can be produced by modifying the function so the variable  $x$  appears only once.

$$f(x) = x - x = x(1 - 1) = 0$$



# 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
  - Rounding
  - Addition
  - Subtraction
  - Algebraic Properties of Floating Point Arithmetic
- Responding to “issues” in floating point arithmetic
  - Changing the math
  - Numerical Analysis
  - Alternatives to IEEE 754

## • Wrap-up/Conclusion

# 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
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
Basic arithmetic operations over Real numbers are commutative, distributive and associative.	Basic operations over floating point numbers are commutative, but NOT associative or distributive.

# The Problem

- How often do we have “working” software that is “silently” producing inaccurate results?
  - We don’t know ... nobody is keeping count.
- But we do know this is an issue for 2 reasons:  
(see Kahan’s desperately needed Remedies...)
  - Numerically Naïve (and unchallenged) formulas in text books (e.g. solving quadratic equations).
  - Errors found after years of use (Rank estimate in use since 1965 and in LINPACK, LAPACK, and MATLAB (Zlatko Drmac and Zvonimir Bujanovic 2008, 2010). Errors in LAPACK’s `_LARFP` found in 2010.)

... and then every now and then, a disaster reminds us  
that floating point arithmetic is not Real

# Here is a famous example ...

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



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

Inaccurate linear elastic model used with NASTRAN underestimated shear stresses by 47% resulted in concrete walls that were too thin.



# We can't trust FLOPS ... let's give up and return to slide rules



**150 Extra Engineers**

An IBM Electronic Calculator speeds through thousands of intricate computations so quickly that on many complex problems it's like having 150 EXTRA Engineers.

No longer must valuable engineering personnel . . . now in critical shortage . . . spend priceless creative time at routine repetitive figuring.

Thousands of IBM Electronic Business Machines . . . vital to our nation's defense . . . are at work for science, industry, and the armed forces, in laboratories, factories, and offices, helping to meet urgent demands for greater production.

**IBM** INTERNATIONAL BUSINESS MACHINES

Public Domain, <https://commons.wikimedia.org/w/index.php?curid=17480483>

(an elegant weapon for a more civilized age)



# *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.

# How should we respond?

- Programmers should conduct mathematically rigorous analysis of their floating point intensive applications to validate their correctness.
- But this won't happen ... training of modern programmers all but ignores numerical analysis. The following tricks\* help and are better than nothing ...
  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 won't 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.

These are useful techniques, but they don't go far enough. How can the discerning skeptic confidently use FLOPs?

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

# Conclusion

- Floating point arithmetic usually works and you can “almost always” be comfortable using it.
- We covered the most famous issues with floating point arithmetic, but we largely skipped numerical analysis. Floating point arithmetic is mathematically rigorous. You can prove theorems and develop formal error bounds.
- Unfortunately, almost nobody learns numerical analysis these days ... so be careful.
  - Modify rounding modes as an easy way to see if round-off errors are a problem.
  - Recognize that unless you impose an order of association, every order is equally valid. If your answers change as the number of threads changes, that is valuable information suggesting an ill-conditioned problem.
  - Anyone who suggests the need for bitwise identical results from a parallel code should be harshly criticized/punished.

My favorite picture of my wife



Kayaker: Pat Welle at Cascade head. Photo by T. Mattson.



# References

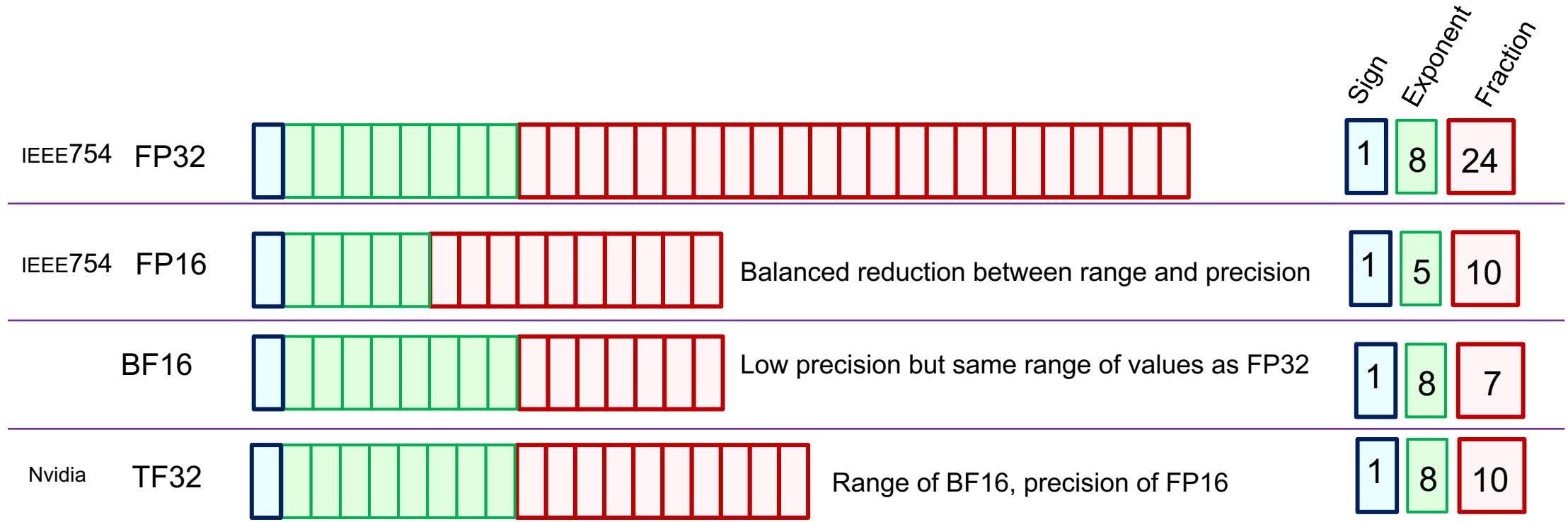
- What every computer scientist should know about floating point arithmetic, David Goldberg, Computing Surveys, 1991.
  - <https://dl.acm.org/doi/pdf/10.1145/103162.103163>
- W. Kahan: How futile are mindless Assessments of Roundoff in floating-point computation?
  - <https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf>
- History of IEEE-754: an interview with William Kahan
  - <https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html>

git clone <https://github.com/tgmattso/CompSciForPhys.git>

**But wait ... before we leave this topic, What about a future dominated by floating point numbers optimized for AI applications.**

**Can we use them for HPC?**

# Reduced Precision Floating Point types



For AI, range is more important than precision, so BF16 and TF32 are the best reduced precision AI options

Nvidia H100 support all these types plus FP8, FP64, and int8

# An Nvidia H100 GPU

An Nvidia H100 GPU with 144 Streaming multiprocessors (SM) and 576 4<sup>th</sup> generation tensor cores



Performance\* for Nvidia H100 SXM

- FP16: 989 TFLOPS, FP32: 67 TFLOPS, FP64: 34 TFLOPS

\*SGEMM-cube: Emulating FP32 GEMM on Ascend NPUs using FP16 cube Units with precision recovery, Weicheng Xue, et. Al. <https://www.arxiv.org/pdf/2507.23387>

<https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>

SM



# Nvidia Tensor Cores and FMA operations

- Nvidia H100 GPU tensor cores perform FMA operations where FMA stands for fused multiply add
- The add operation accumulates results across FMA operations and can use a higher precision, FP32 data type.
- Matrix blocks at the hardware level are 4x4.

$$\begin{array}{c} \mathbf{D} = \\ \text{FP16 or FP32} \end{array} \begin{pmatrix} \begin{array}{|c|c|c|c|} \hline A_{0,0} & A_{0,1} & A_{0,\dots} & A_{0,15} \\ \hline A_{1,0} & A_{1,1} & A_{1,\dots} & A_{1,15} \\ \hline A_{\dots,0} & A_{\dots,1} & A_{\dots,\dots} & A_{\dots,15} \\ \hline A_{15,0} & A_{15,1} & A_{15,\dots} & A_{15,15} \\ \hline \end{array} & \begin{pmatrix} \begin{array}{|c|c|c|c|} \hline B_{0,0} & B_{0,1} & B_{0,\dots} & B_{0,15} \\ \hline B_{1,0} & B_{1,1} & B_{1,\dots} & B_{1,15} \\ \hline B_{\dots,0} & B_{\dots,1} & B_{\dots,\dots} & B_{\dots,15} \\ \hline B_{15,0} & B_{15,1} & B_{15,\dots} & B_{15,15} \\ \hline \end{array} & \begin{array}{|c|c|c|c|} \hline C_{0,0} & C_{0,1} & C_{0,\dots} & C_{0,15} \\ \hline C_{1,0} & C_{1,1} & C_{1,\dots} & C_{1,15} \\ \hline C_{\dots,0} & C_{\dots,1} & C_{\dots,\dots} & C_{\dots,15} \\ \hline C_{15,0} & C_{15,1} & C_{15,\dots} & C_{15,15} \\ \hline \end{array} \\ \hline \end{pmatrix} \begin{array}{c} \text{FP16} \quad \text{FP16} \quad \text{FP16 or FP32} \end{array}$$

*A warp performs  $D=A*B+C$  where  $A$ ,  $B$ ,  $C$  and  $D$  are  $16 \times 16$  matrices*

<https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/>

# FP32 Emulation with BF16

- We can represent an IEEE754 floating point number by multiple lower precision values.
- First, some definitions
  - $(B_{16})$  casts an FP32 number into BF16
  - $(F_{32})$  casts a BF16 number into FP32
- Let's split an FP32 number,  $\alpha$ , into three BF16 numbers\*:  $b^{(0)}$ ,  $b^{(1)}$ , and  $b^{(2)}$

$$\begin{aligned}b^{(0)} &= (B_{16}) \alpha \\b^{(1)} &= (B_{16}) ((F_{32}) (\alpha - (F_{32})b^{(0)})) \\b^{(2)} &= (B_{16}) ((F_{32}) (\alpha - (F_{32})b^{(0)} - (F_{32})b^{(1)}))\end{aligned}$$

- The result is that  $\alpha = b^{(0)} + b^{(1)} + b^{(2)}$
- We can do this element wise to a matrix of FP32 values,  $A$ , to represent the matrix as a sum  $A^{(0)} + A^{(1)} + A^{(2)}$

You can also split the numbers by a shifting procedure ... i.e. capture higher order bits, round, grab next set of bits, shift exponent. Result: FP32 from 2 FP16 numbers with with loss of ~2 bits.

---

\*This discussion is based on an example from the paper “Leveraging the Bfloat 16 AI datatype for higher-precision computations by Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke, <https://arxiv.org/abs/1904.06376>, 2019,



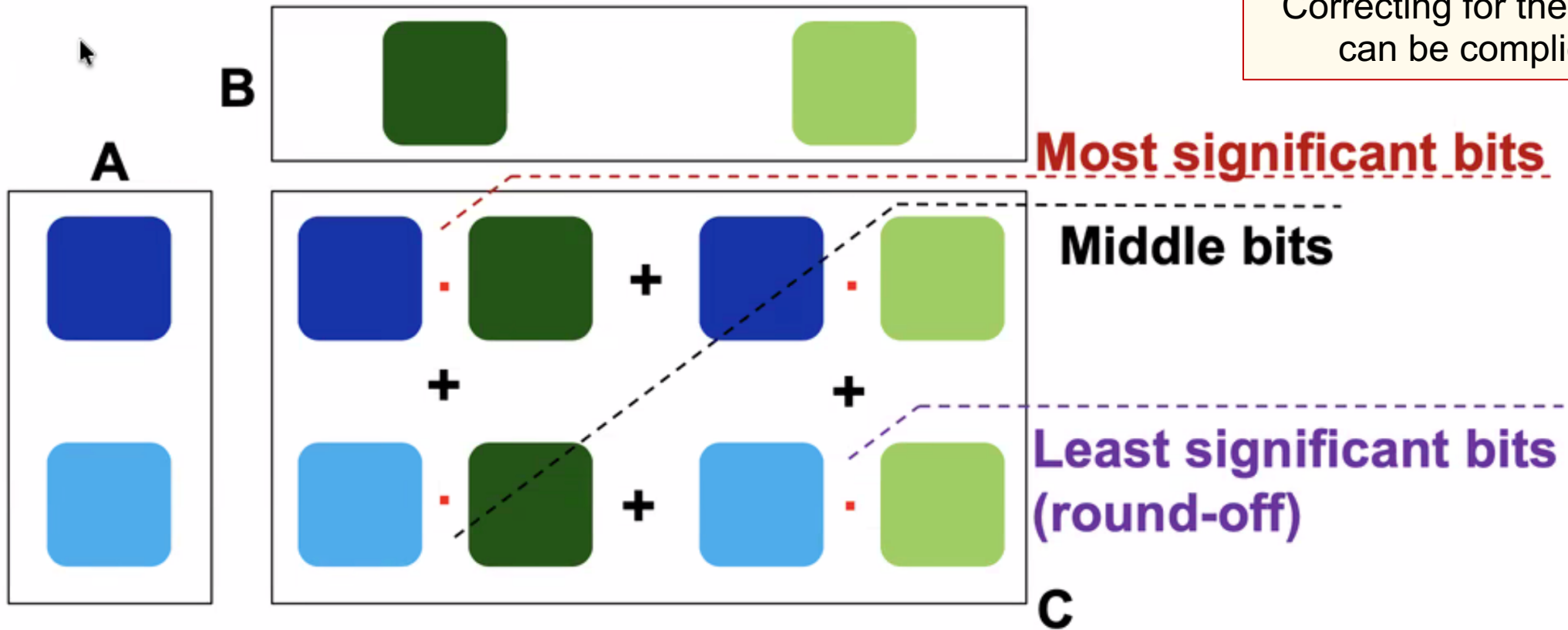
# Multiplication of FP32 matrices using FP16 matrices



## Visualization of Emulation with 2 Splits

$$AB = \left( \text{blue square} + \text{light blue square} \right) \left( \text{dark green square} + \text{light green square} \right) = C$$

Round-off errors accumulate at each matrix operation. Correcting for these errors can be complicated





# Why we cannot afford to ignore tensor cores

NVIDIA GPU	FP64 Tflop/s	AMD GPU	FP64 Tflop/s
K40	1.7	MI60	7.4
P100	2.9	MI100	23.1
V100	7.5	MI210	45.3
A100	9.7	MI250	90.5
H100	60	MI300X	163.4
B100	40	MI350*	72

\*Announced but not released as of 9/2025

**The mainstays of scientific computing, FP64 and FP32 have been deemphasized by the GPU hardware vendors at the cost of lower-precision (8, 6, and 4 bits) and integer (fixed-point) compute units suitable for AI workloads.**

This slide is based on a talk from HPEC'25. I added the title and the red circles/boxes for emphasis

Performance and Numerical Aspects of Decompositional Factorizations with FP64 Floating-Point Emulation in Int8, Piotr Luszczek, et. al., HPEC 2025



# Matrix Multiplication results ... promising but frustrating

- Clear performance results for H100 GPUs and FP16 are difficult to find for SGEMM.
  - Future work ... I will produce these numbers myself ... Hopefully I time for ESC'26
- The most careful work for SGEMM uses the Ozaki scheme, but results are unclear for modern GPUs based on what I can find in the literature.
- Nvidia claims that using TF32 on an H100 they can achieve SGEMM performance of one Petaflop using their mixed precision library (but they do not explain how they do this ... it must use FP16 as well since FP16 on H100 has a peak performance of 1.6 PFLOPS while TF32 has a peak performance of only 0.98 PFLOPS)

---

Nvidia H100 performance numbers: <https://www.nvidia.com/en-us/data-center/h100/>

DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions, Daichi Mukunoki, K. Ozaki, T. Ogita, and T. Imamura, <https://pmc.ncbi.nlm.nih.gov/articles/PMC7295351/>, 2020

# Other approaches for using mixed precision

- Use a solver in low precision to produce a preconditioner for a full precision iterative method.
- Iterative solvers with error evaluation per iteration at full precision.
- Forget computation in reduced precision. Data movement dominates runtimes for many problems. Use reduced precision for compressed storage using reduced precision types.
- ... and more. This is an active area of research and how it will impact HPC is promising but there are many details to nail down. I am particularly interested in how these AI accelerators can be used for Sparse Linear Algebra.