Beyond the (standard) floating point computations

Bogusław Cyganek

cyganek@agh.edu.pl

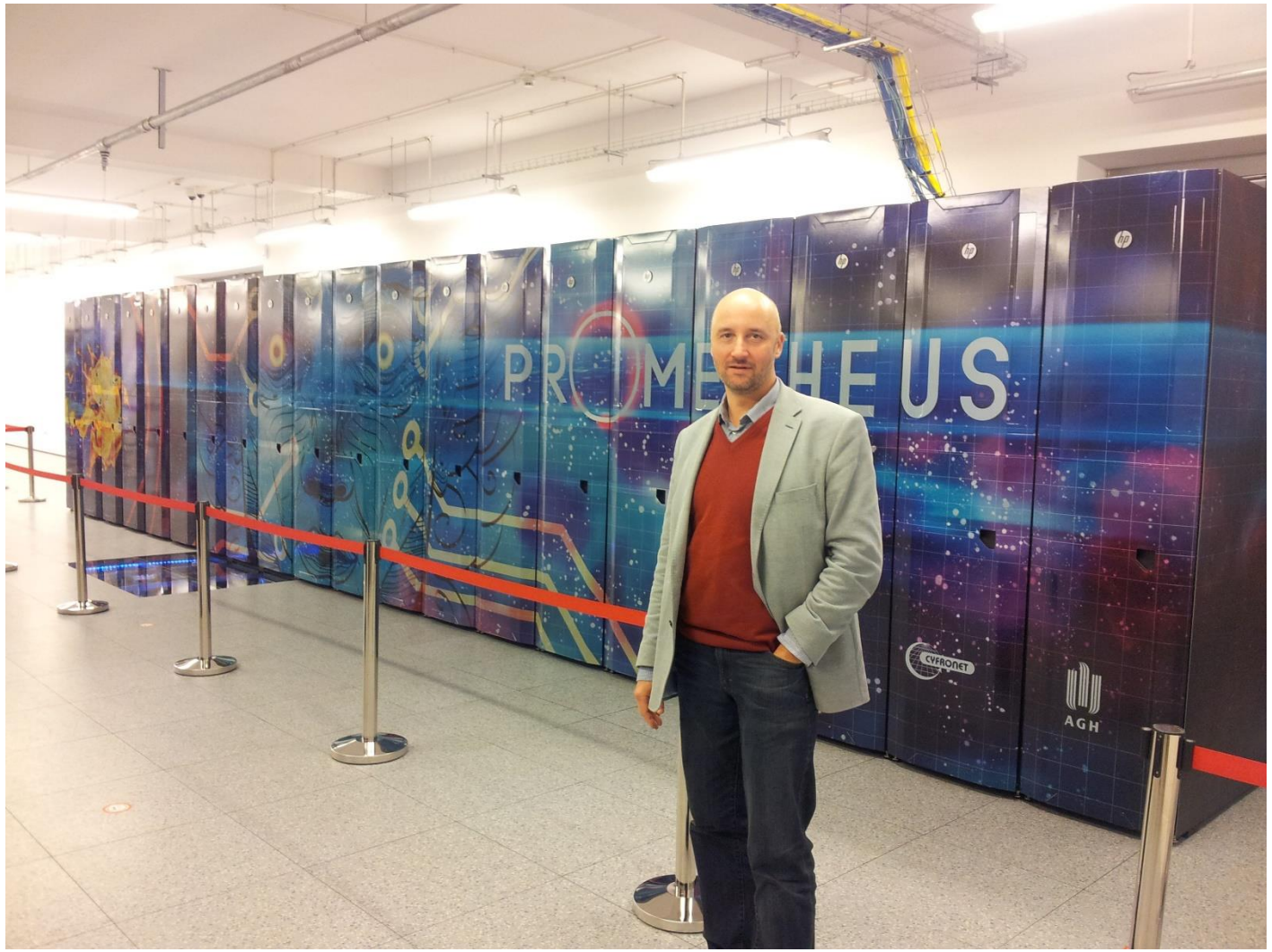
AGH University of Science and Technology, Krakow, Poland

C++ User Group Krakow, June 23rd 2022

Beyond the (standard) floating point computations – the plan

- Brief intro to the FP IEEE 754 standard, its pros, cons and pitfalls.
- Quest for the 16 bit FP format.
- How can we compress floating point data – shorter formats, lossless & lossy compressions, new ZFP and SZ methods and their use in C++.
- A new FP format – the POSIT arithmetic and how to use it.
- Intro to the relevant C++ libraries and examples with each topic.

Massive Data Processing Bottleneck



2021

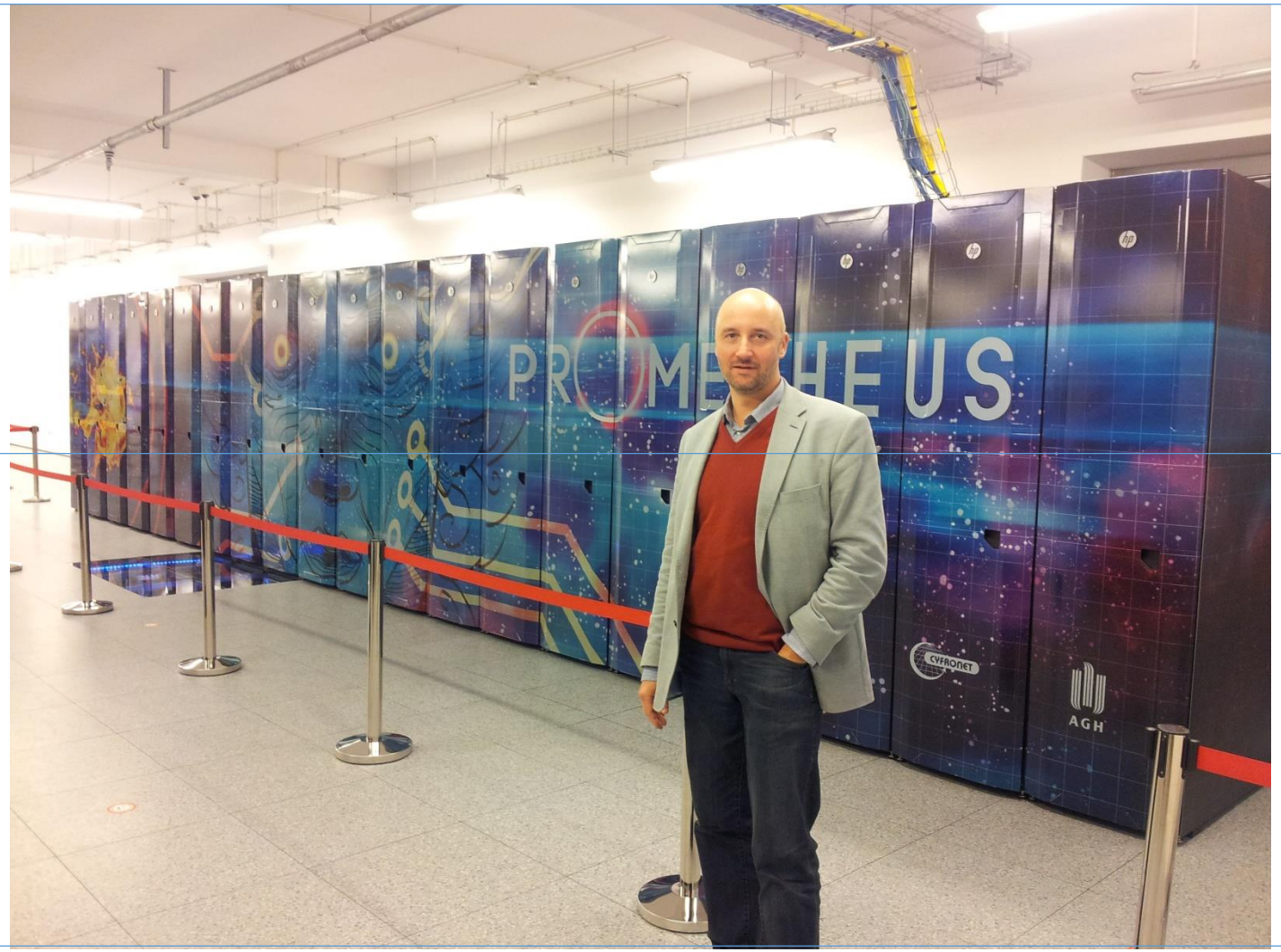
Myself with the PROMETHEUS computer at the Cyfronet Computer Center in Krakow, Poland.

Massive Data Processing Bottleneck

Exa
10^{18}

Tera
10^{12}

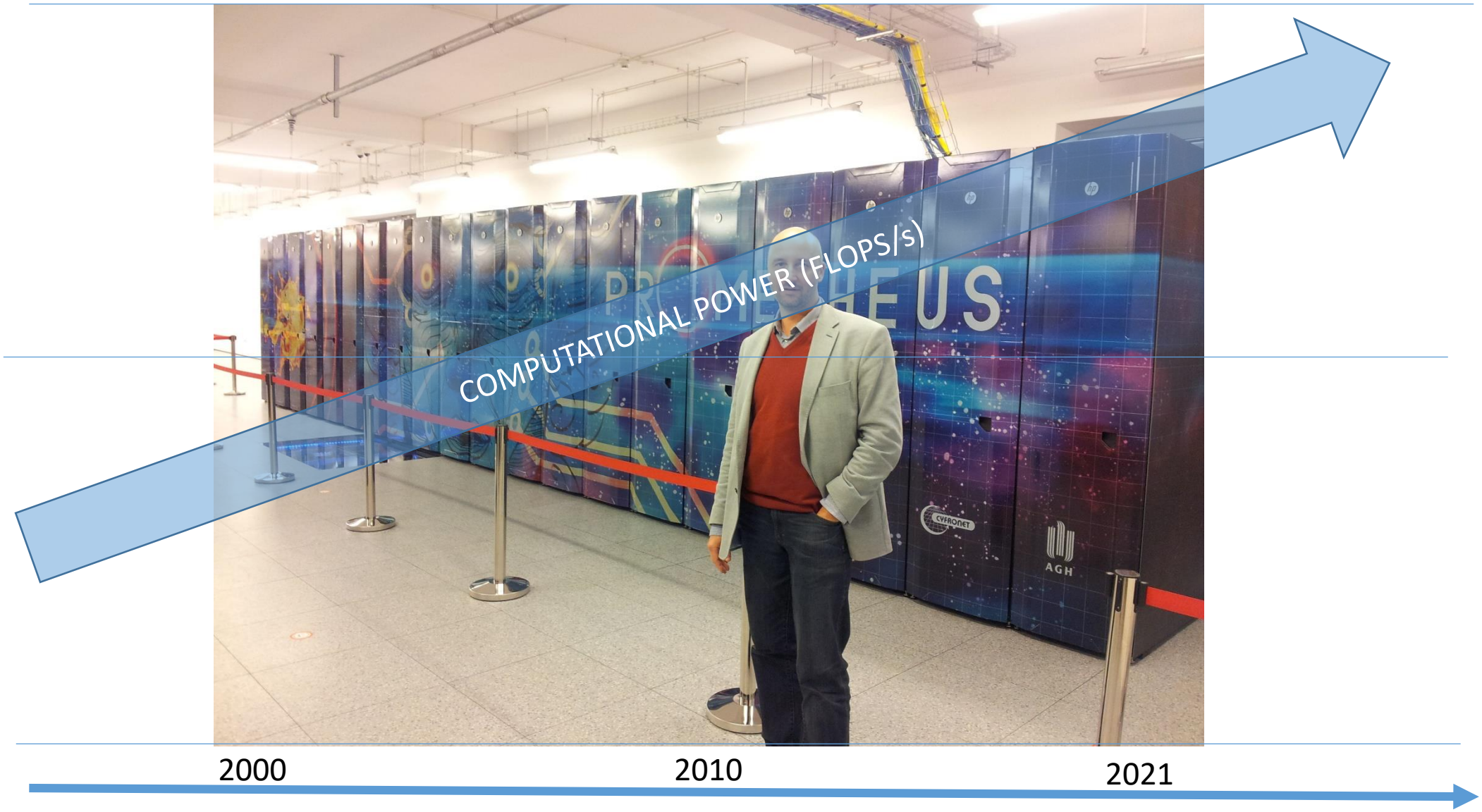
Mega
10^{6}



2000 2010 2021

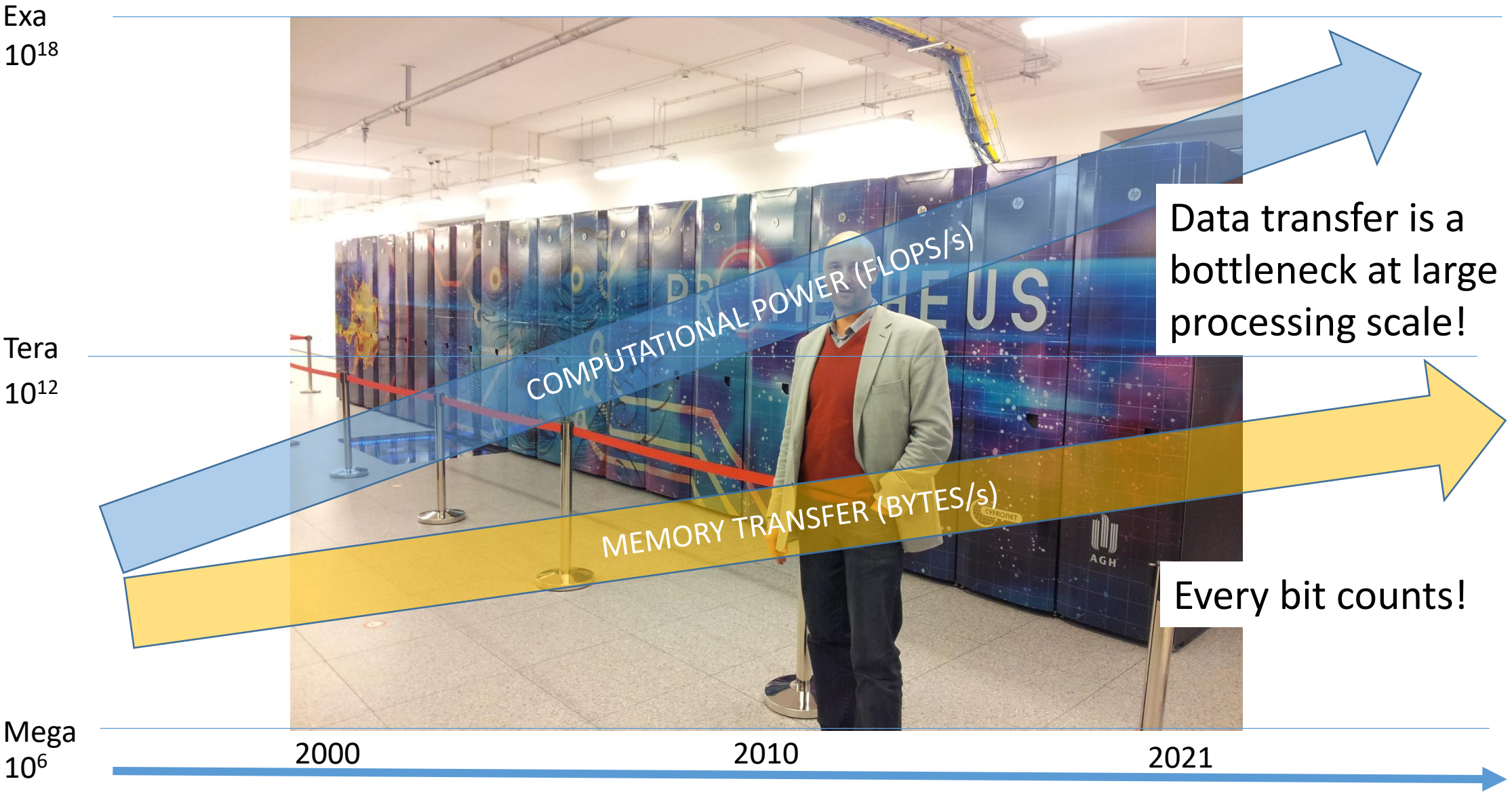
Myself with the PROMETHEUS computer at the Cyfronet Computer Center in Krakow, Poland.

Massive Data Processing Bottleneck



Myself with the PROMETHEUS computer at the Cyfronet Computer Center in Krakow, Poland.

Massive Data Processing Bottleneck



Myself with the PROMETHEUS computer at the Cyfronet Computer Center in Krakow, Poland.

Massive Data Processing Bottleneck

However having only float and double in C++ we frequently choose the ”fast track”, forgetting about the memory consumption. For example:

```
int slider_position = 256;
int next_slider_pos {}; // let's set this to 75% of the slider_position
// ...

next_slider_pos = 75 / 100 * slider_position; // oops, 0 ?
```

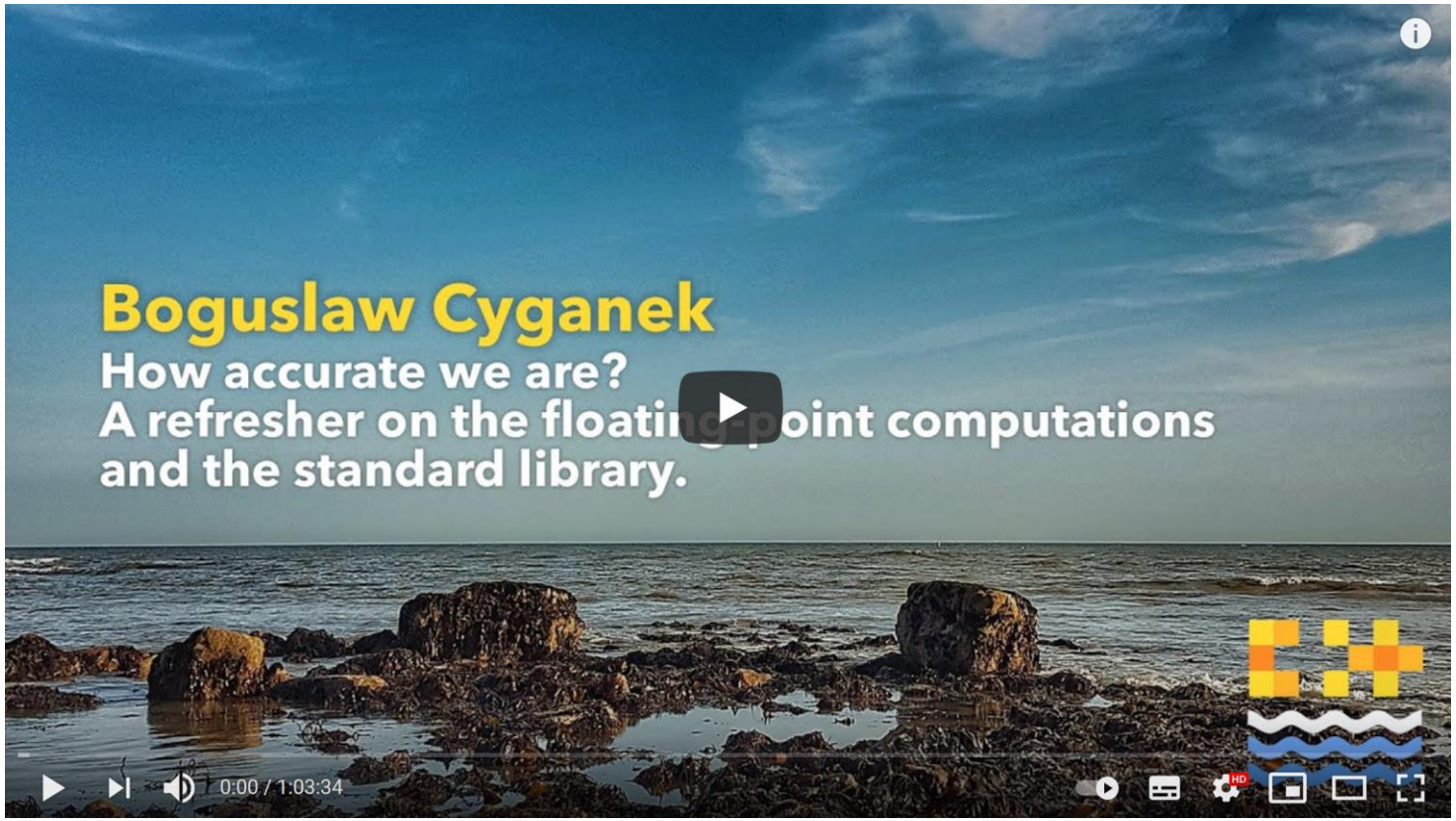
```
double slider_position = 256.;
double next_slider_pos {}; // let's set this to 75% of the slider_position
// ...

next_slider_pos = 75. / 100. * slider_position; // ok, but...
```

But what about massive data? E.g. image processing, convolutions require fractionals?

Brief intro to IEEE 754

Brief intro to IEEE 754

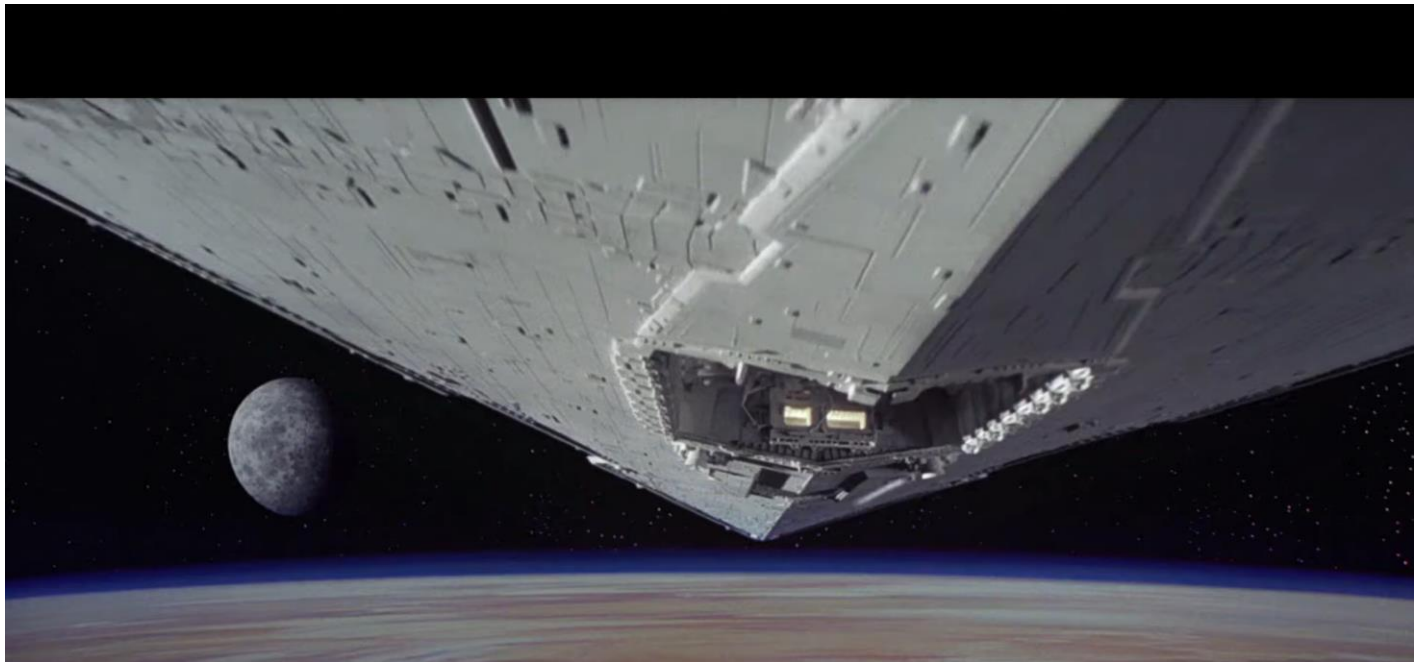
But if you wish more on IEEE 754 and FP computations then take a look at my talk 2020:



https://www.youtube.com/watch?v=7aZbYJ5UTC8

The IEEE 754 Standard – How it was Born?

Once upon a time in a galaxy… there was no FP standard …



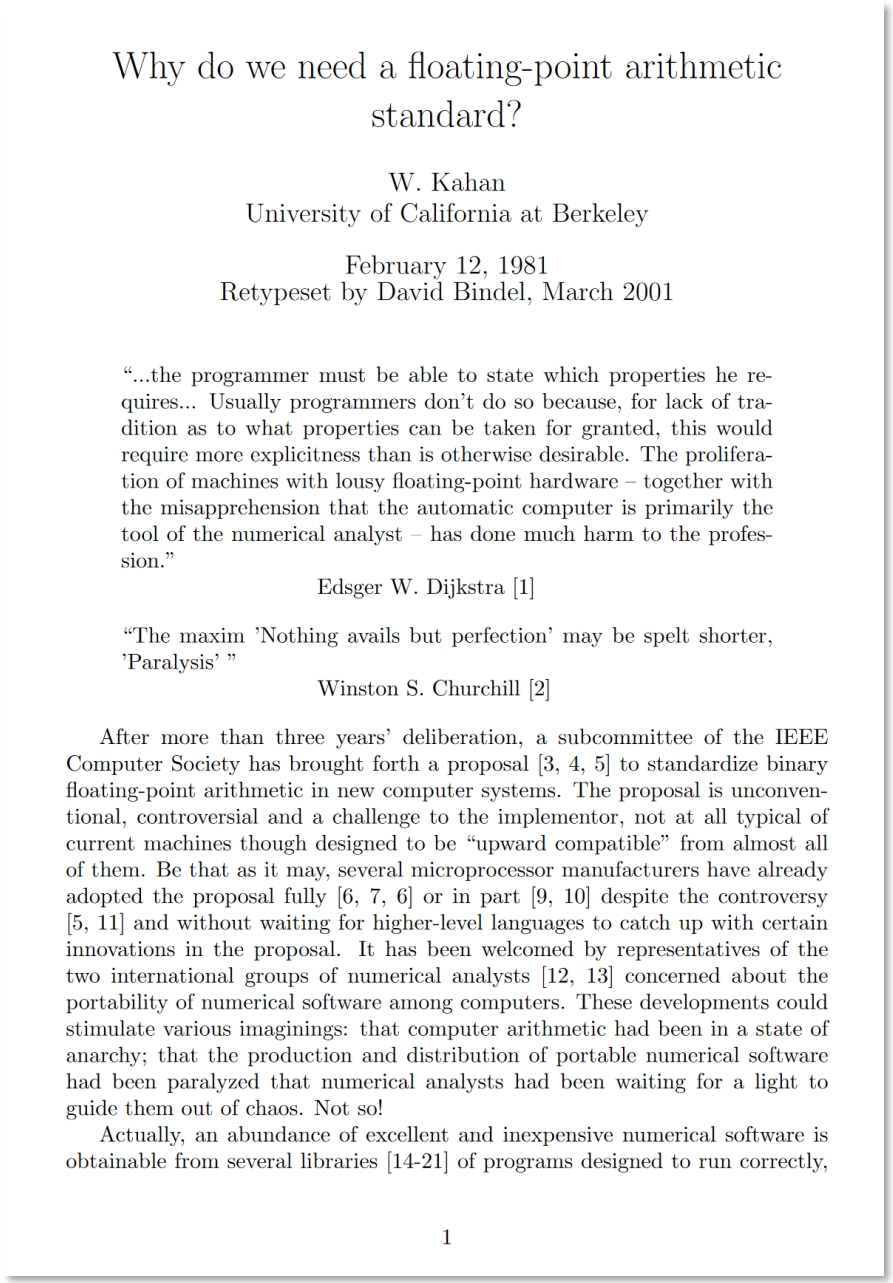
- Computers used many different representations for FP numbers.
- The lack of a FP standard was a hot problem by the early 1970s for writing and maintaining code.
- Each major company had its own standard or a variant – these differed in the word sizes, the representations, and the rounding behavior and general accuracy of operations.
- FP compatibility across multiple systems was in desperate need of standardization by the early 1980s, leading to the creation of the IEEE 754 standard (*https://en.wikipedia.org/wiki/Floating-point_arithmetic*).

The IEEE 754 Standard – How it was Born?

W. Kahan: *Why do we need a floating-point arithmetic standard?* Stanford University, 1981.

”Numerical software production is costly. We cannot afford it unless programming costs are distributed over a large market; this means **most programs must be portable over diverse machines**.
[…] we need an abstract model of their computational environment. Faithful models do exist, but they reveal that environment to be too diverse, forcing portable programmers to bloat even the simplest concrete tasks into abstract monsters.
We need something simple or, if not so simple, not so capriciously complex.”

”Alas, the proposed IEEE standard for binary floating point arithmetic will not guarantee correct results from all numerical programs. **But the standard weights the odds more in our favor**.”



Why do we need a floating-point arithmetic standard?

W. Kahan
University of California at Berkeley

February 12, 1981
Retypeset by David Bindel, March 2001

“...the programmer must be able to state which properties he requires... Usually programmers don’t do so because, for lack of tradition as to what properties can be taken for granted, this would require more explicitness than is otherwise desirable. The proliferation of machines with lousy floating-point hardware – together with the misapprehension that the automatic computer is primarily the tool of the numerical analyst – has done much harm to the profession.”

Edsger W. Dijkstra [1]

“The maxim ’Nothing avails but perfection’ may be spelt shorter, ’Paralysis’ ”

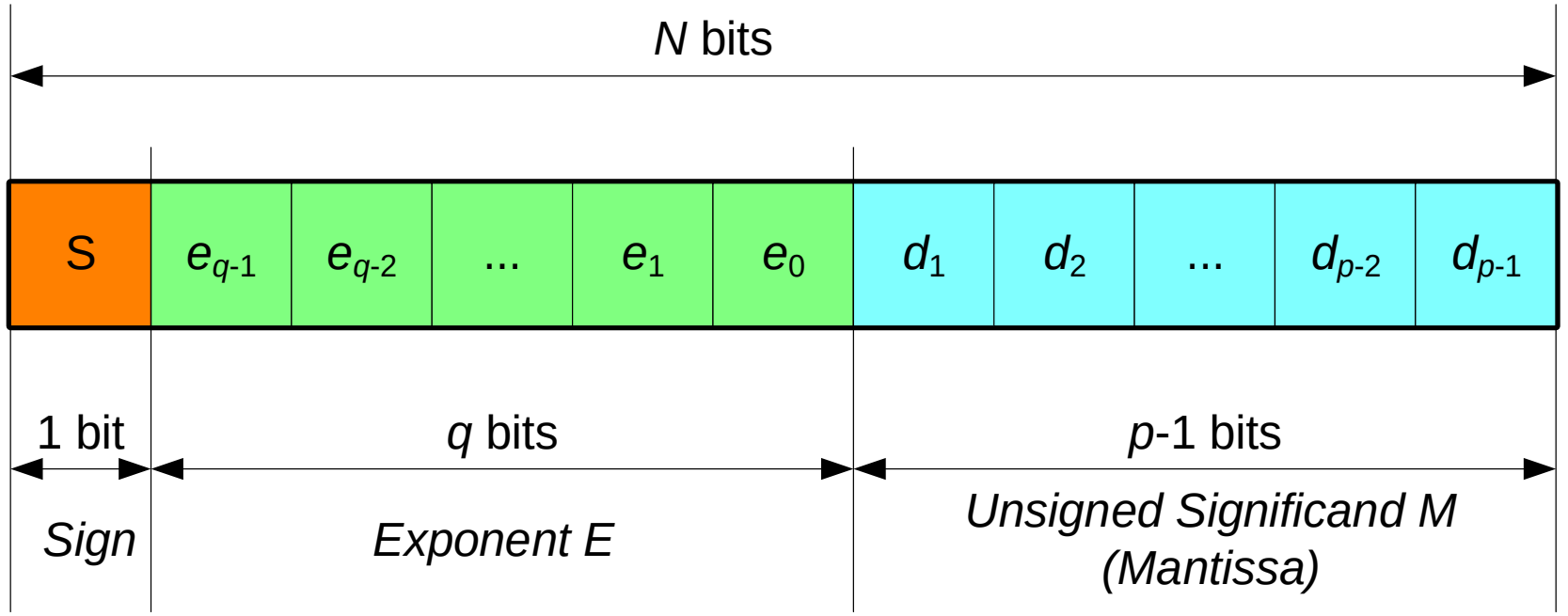
Winston S. Churchill [2]

After more than three years’ deliberation, a subcommittee of the IEEE Computer Society has brought forth a proposal [3, 4, 5] to standardize binary floating-point arithmetic in new computer systems. The proposal is unconventional, controversial and a challenge to the implementor, not at all typical of current machines though designed to be “upward compatible” from almost all of them. Be that as it may, several microprocessor manufacturers have already adopted the proposal fully [6, 7, 6] or in part [9, 10] despite the controversy [5, 11] and without waiting for higher-level languages to catch up with certain innovations in the proposal. It has been welcomed by representatives of the two international groups of numerical analysts [12, 13] concerned about the portability of numerical software among computers. These developments could stimulate various imaginings: that computer arithmetic had been in a state of anarchy; that the production and distribution of portable numerical software had been paralyzed that numerical analysts had been waiting for a light to guide them out of chaos. Not so!

Actually, an abundance of excellent and inexpensive numerical software is obtainable from several libraries [14-21] of programs designed to run correctly,

1

The IEEE 754 Standard for Floating-Point Arithmetic

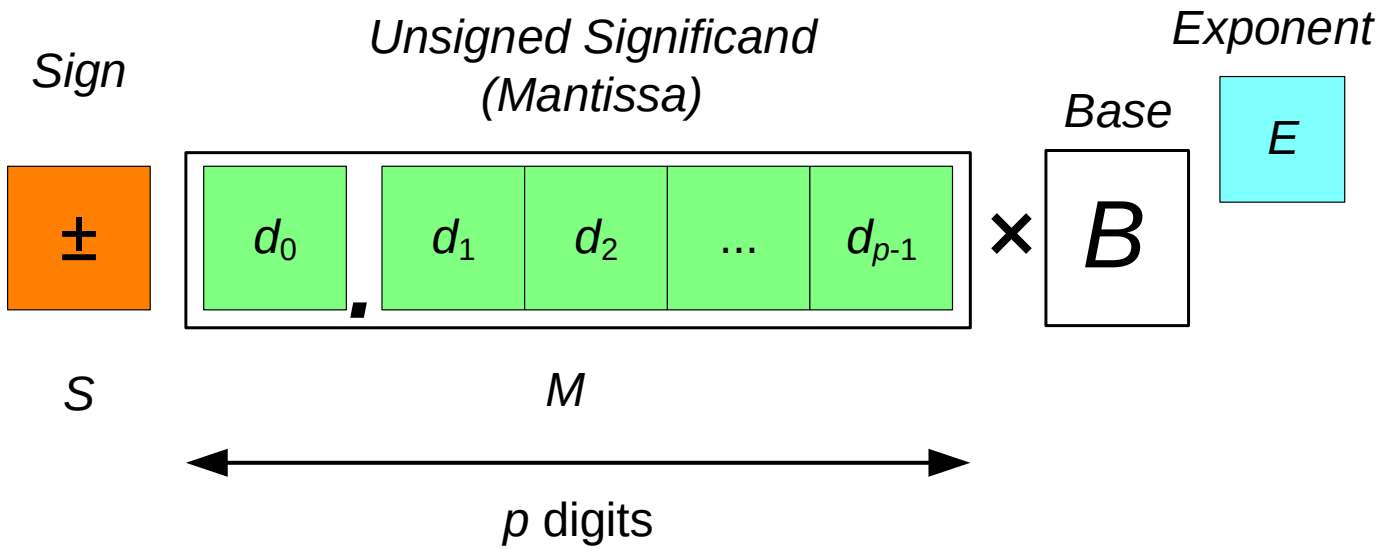


Bit representation of the floating-point values in the IEEE 754 standard.

The IEEE 754 Standard for Floating-Point Arithmetic

- IEEE 754 defines many FP formats from which the *single precision* and *double precision* are probably the most commonly encountered in C/C++ compilers.
  - Single precision format occupies 4 bytes (32 bits) and has p=24 and q=8 bits. In some C/C++ compilers represented with **`float`**.
  - Double precision format occupies 8 bytes (64 bits) and has p=53 and q=11 bits. In some C/C++ compilers represented with **`double`**.
  - Extended precision (double extended) format occupies 10 bytes (80 bits), p=64, q=15. In some C/C++ compilers represented with `long double`.
  - Quadruple precision occupies 16 bytes (128 bits) with p=113 bits (supported by some C/C++ compilers with `long double` or special types/flags).
  - Half precision 2 bytes – **Why?** Standard revised in 2008; more later …

Number representation in the floating-point format



A value D of a number is given as follows

$$D = (-1)^S \cdot M \cdot B^E = (-1)^S \cdot (d_o \bullet d_1 \ldots d_{p-1}) \cdot B^E$$

M is unsigned **significand** (**mantissa**, fraction), *B* is a base, and *E* denotes the exponent

$$E_{\min} \le E \le E_{\max}$$

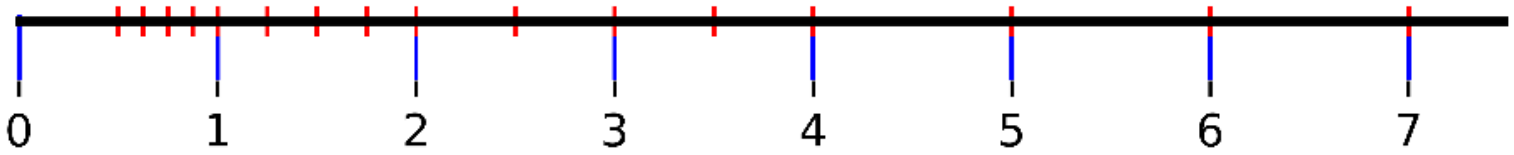
For *p* digits and the base *B*, a value of the significand is given as follows

$$M = d_0 + d_1 \cdot B^{-1} + \cdots + d_{p-1} \cdot B^{-(p-1)}$$

Floating-point number distribution

$$D = (-1)^S \cdot M \cdot B^E = (-1)^S \cdot (d_o \bullet d_1 \ldots d_{p-1}) \cdot B^E$$

p=3, B=2, E=[-1,2]



- Floating-point values are shown in **red**. It is easy to observe different groups of number "concentrations", corresponding to different exponents E.
- 4 groups are well visible which correspond to E=-1, 0, 1, and 2, respectively.
- Spacing within a group is the same.
- However, spacing between the groups increases by a factor of the base B.

Floating-point – basic facts

- In the FP domain, (**many) real values cannot be <u>exactly</u> represented**.
- Due to the **roundoff errors**, in the FP domain some algebraic conditions do not always hold. For example it might happen that *the commutative low does not hold*, that is

$$(a + b) + c \neq a + (b + c) \quad \text{for} \quad a, b, c \in \text{FP}.$$

- Floating point **representation of numbers is not unique**. The preferable representation of significand is with no leading zeros to retain the maximum number of significant bits, that is $d_0>0$. This is so called a **normalized form**.
- Normalization makes some problems with convenient representation of zero (preferably with all bits set to 0). Hence, *a special encoding is necessary (➔ denormal)*.
- Usually the exponent *E*, which can be negative, is represented in *a biased format* $E=E_{true}+bias$, in which its value is shifted, so *E is always positive* (this is so called the excess method). Such representation simplifies comparison since with this representation we can compare a number [*S*,*E*,*M*] as signed-magnitude numbers.

- The most-important-bit of the significand is not stored since in the normalized FP representation it is always 1. This is so called *a hidden bit* trick (Goldberg, 1991). This also explains why using the binary base *B*=2 is beneficial (the smallest wobble). This feature explains also how the above bit partitions are organized considering also the *S* sign bit.
- Standard requires only that double **is at least as precise** as float and long double as double, respectively.

The IEEE 754 Standard for Floating-Point Arithmetic

- The standard defines some special values which are especially encoded:
  - Positive infinity (+∞).
  - Negative infinity (−∞).
  - Negative zero (0^-).
  - Positive zero (0^+).
  - Not-a-number (NaN) –used to represent for example results of forbidden mathematical operations avoiding an exception, such as a square root of a negative value, etc. **Exponent values of all ones** are used to represent special values (but what with the rest? Waste of bits?)
- The significand and exponent are especially encoded to represent the above special values. For other than above special encodings, a value 127 in a single, and 1023 in a double precision, are added to the exponent, respectively.

The IEEE 754 Standard for Floating-Point Arithmetic

- The ‘standard’ FP formats there is a problem with precise representation of 0 and values close to 0. To remedy this, there is the special encoding (all 0’s in the exponent bit and d_0=0 of the significand) to represent this group of values, which are called *denormals* (*subnormals*). However, in some systems using denormals comes with a significant run-time cost.

The IEEE 754 Standard for Floating-Point Arithmetic

- The IEEE 754 standard defines all necessary mathematical operations on FP values, such as addition, multiplication, but also roundings. Rounding is inevitable if the result cannot fit in the FP representation, e.g. due to insufficient number of bits for precision. There are five rounding modes, as follows:
  - **Default round to nearest** (ties round to the nearest even digit, i.e. to a value that makes the significand end in an even digit). It can be shown that rounding to nearest even leads to lowest errors.
  - Optional round to nearest (ties round away from 0).
  - Round up (toward $+\infty$).
  - Round down (toward $-\infty$).
  - Round toward 0 (cuts off fractional). This is also **the default rounding for the integer types**.

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000

1

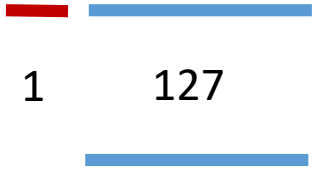
The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000

1 127

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000

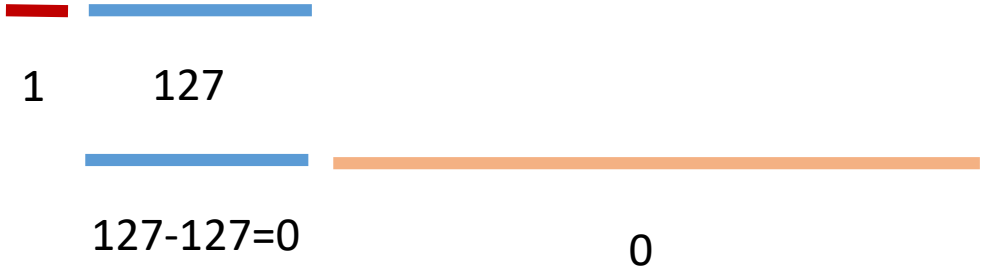


1 127

127-127=0

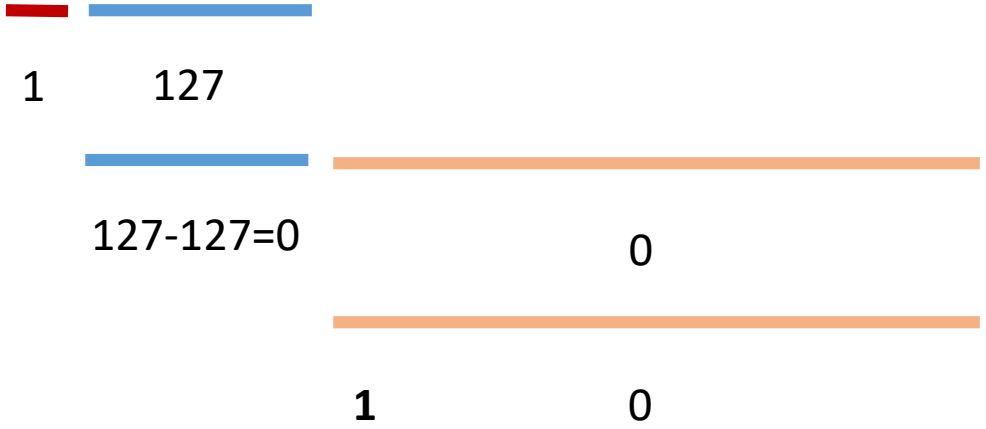
The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000



The IEEE 754 Standard for Floating-Point Arithmetic

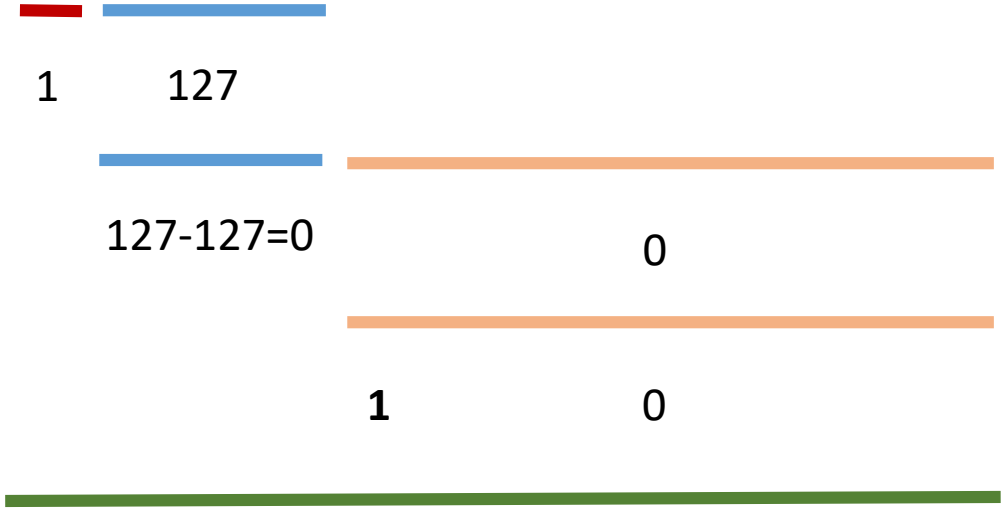
IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000



Hidden bit

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of -1.000000 is :
1 | 01111111 | 00000000000000000000000



Hidden bit

$$D = (-1)^S \cdot M \cdot B^E = -1 \cdot (1.0\ldots0) \cdot 2^0 = -1$$

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

0

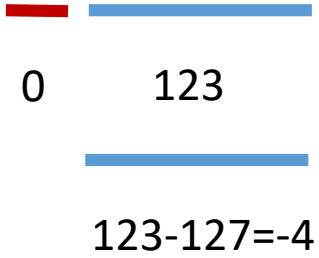
The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

0 123

The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

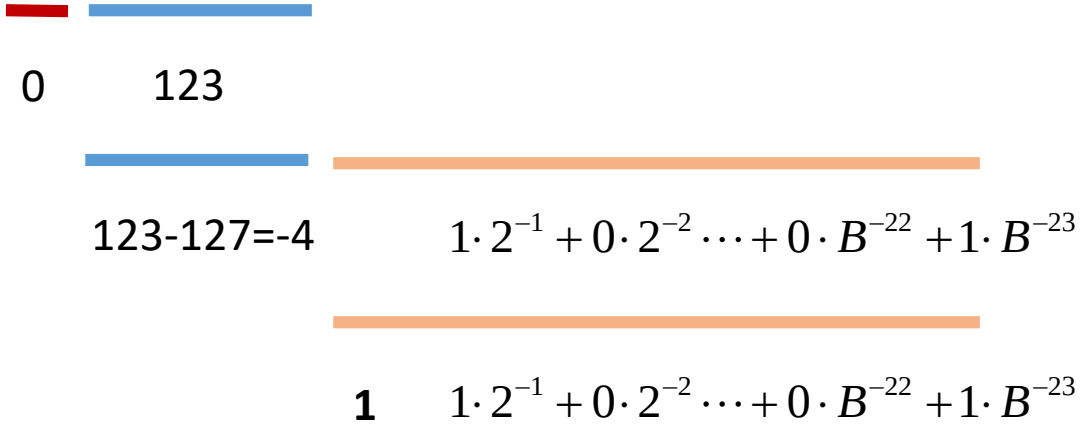


The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101

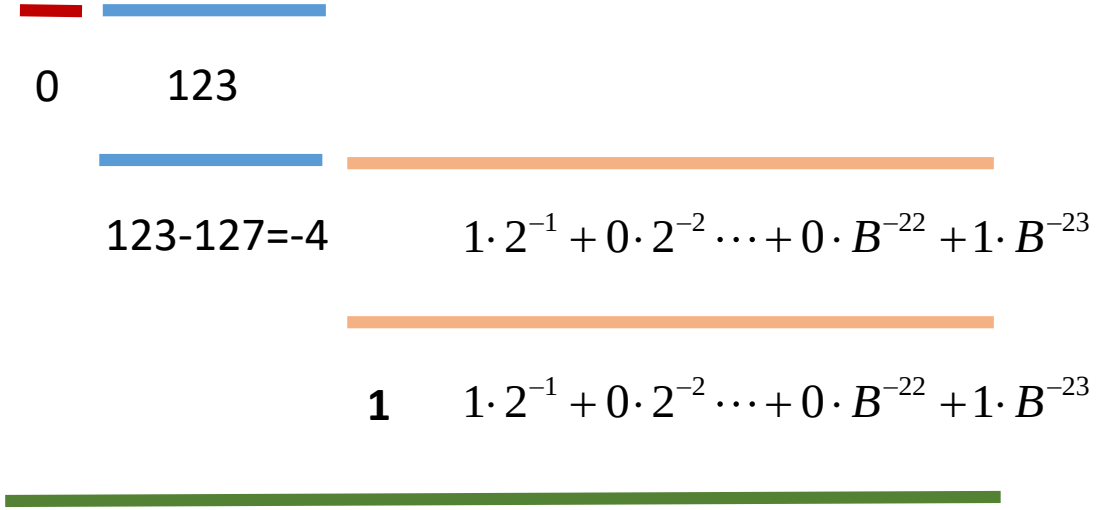
The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101



The IEEE 754 Standard for Floating-Point Arithmetic

IEEE 754 representation of 0.100000 is :
0 | 01111011 | 10011001100110011001101



$$D=(-1)^S\cdot M\cdot B^E=1\cdot\left(1+1\cdot 2^{-1}+0\cdot 2^{-2}\cdots+0\cdot B^{-22}+1\cdot B^{-23}\right)\cdot 2^{-4}\approx 0.1$$

The IEEE 754 Standard for Floating-Point Arithmetic

What is the minimal value in the IEEE 754 single precision representation?

The IEEE 754 Standard for Floating-Point Arithmetic

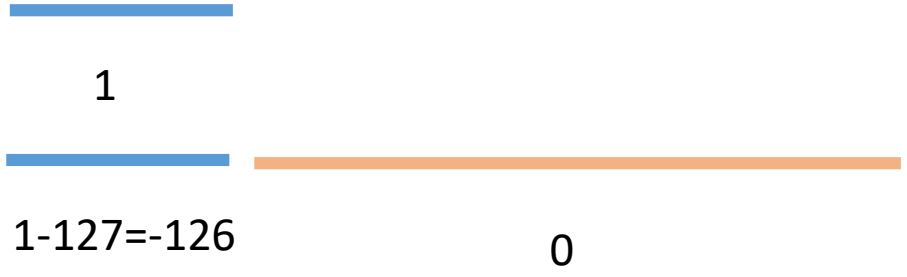
What is the minimal value in the IEEE 754 single precision representation?

0 | 00000001 | 00000000000000000000000 (all 0s in the exponent are reserved)

The IEEE 754 Standard for Floating-Point Arithmetic

What is the minimal value in the IEEE 754 single precision representation?

0 | 00000001 | 00000000000000000000000 (all 0s in the exponent are reserved)

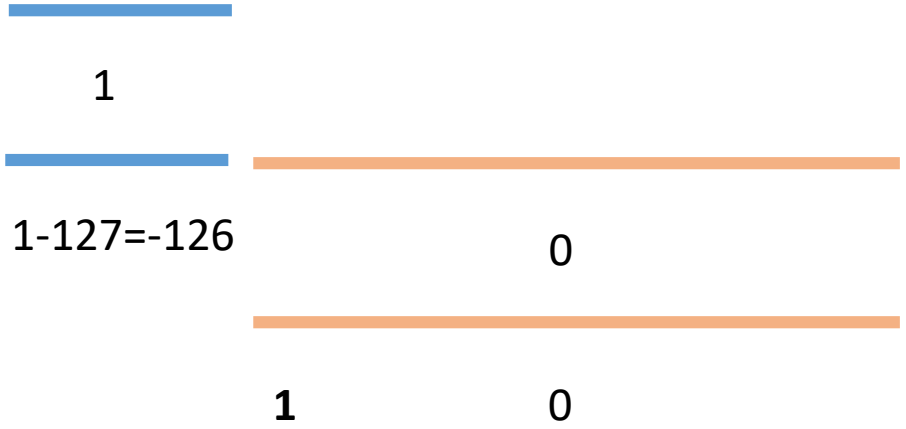


The IEEE 754 Standard for Floating-Point Arithmetic

What is the minimal value in the IEEE 754 single precision representation?

0 | 00000001 | 00000000000000000000000

(all 0s in the exponent are reserved)



Hidden bit

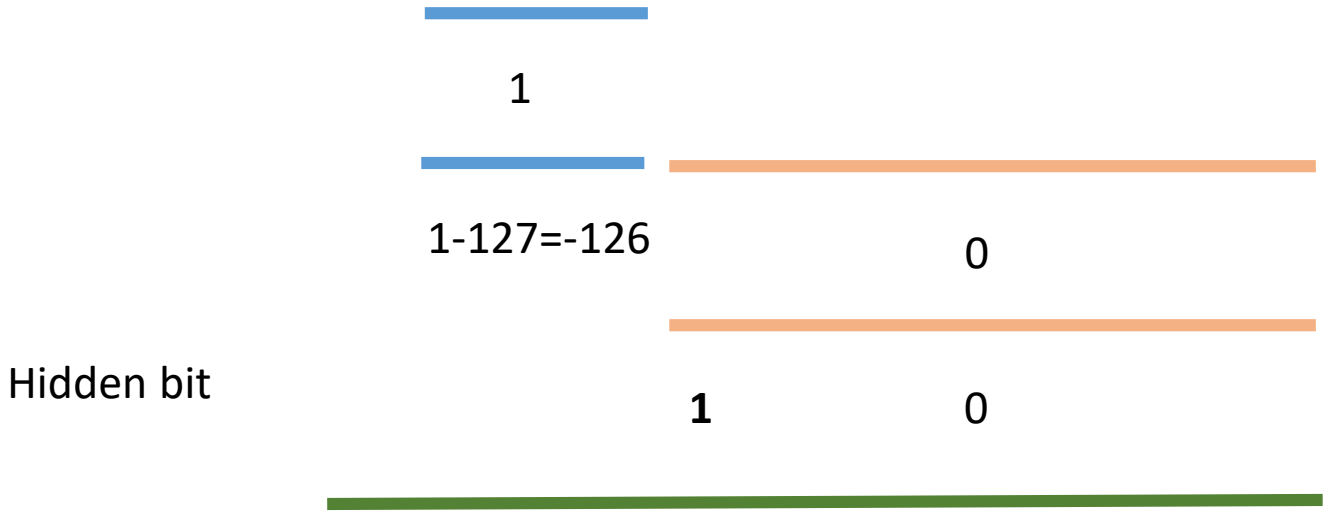
1 0

The IEEE 754 Standard for Floating-Point Arithmetic

What is the minimal value in the IEEE 754 single precision representation?

0 | 00000001 | 00000000000000000000000

(all 0s in the exponent are reserved)



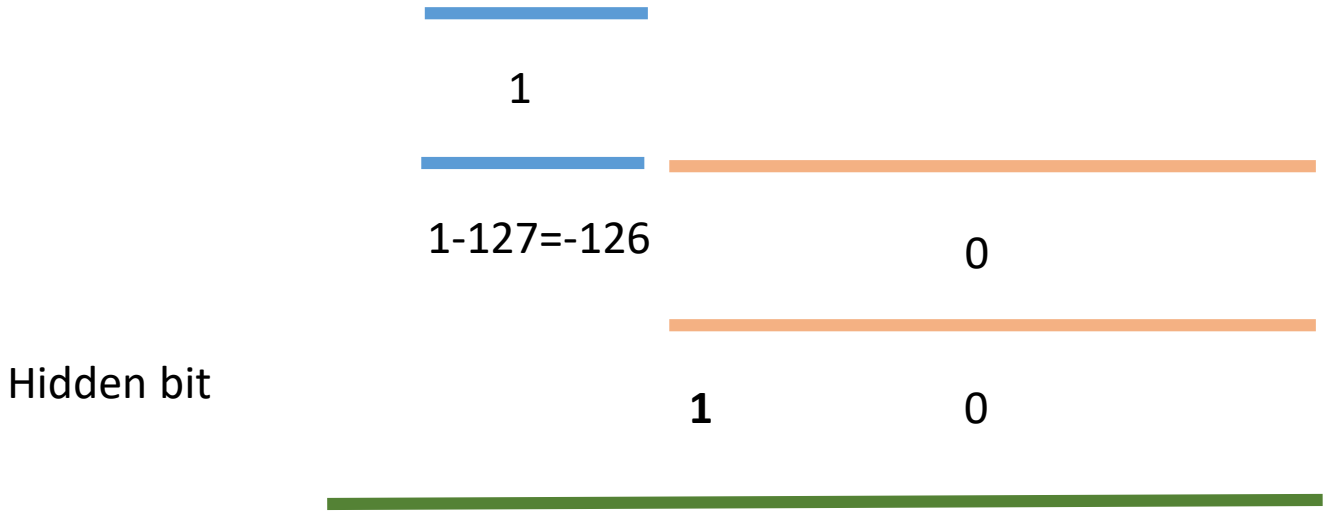
$$D = (-1)^S \cdot M \cdot B^E = (1.0\ldots0) \cdot 2^{-126} \approx 1.17549435\text{e-}38$$

The IEEE 754 Standard for Floating-Point Arithmetic

What is the minimal value in the IEEE 754 single precision representation?

0 | 00000001 | 00000000000000000000000

(all 0s in the exponent are reserved)



$$D = (-1)^S \cdot M \cdot B^E = (1.0\ldots0) \cdot 2^{-126} \approx 1.17549435\text{e-}38$$

```
constexpr float kMin { std::numeric_limits< float >::min() };
```

The IEEE 754 Standard for Floating-Point Arithmetic

What is the **maximal** value in the IEEE 754 single precision representation?

The IEEE 754 Standard for Floating-Point Arithmetic

What is the **maximal** value in the IEEE 754 single precision representation?

0 | 11111110 | 11111111111111111111111 (all 1s in the exponent are reserved)

The IEEE 754 Standard for Floating-Point Arithmetic

What is the **maximal** value in the IEEE 754 single precision representation?

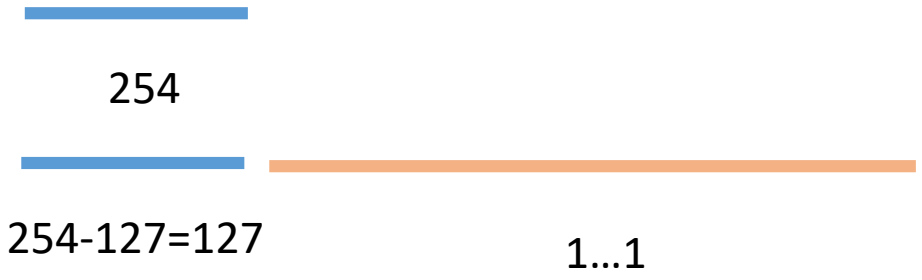
0 | 11111110 | 11111111111111111111111 (all 1s in the exponent are reserved)

254

The IEEE 754 Standard for Floating-Point Arithmetic

What is the **maximal** value in the IEEE 754 single precision representation?

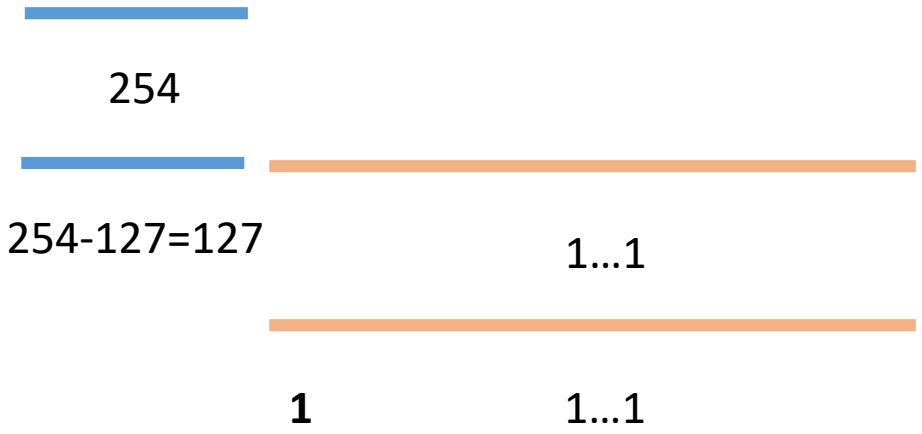
0 | 11111110 | 11111111111111111111111 (all 1s in the exponent are reserved)



The IEEE 754 Standard for Floating-Point Arithmetic

What is the **maximal** value in the IEEE 754 single precision representation?

0 | 11111110 | 11111111111111111111111 (all 1s in the exponent are reserved)

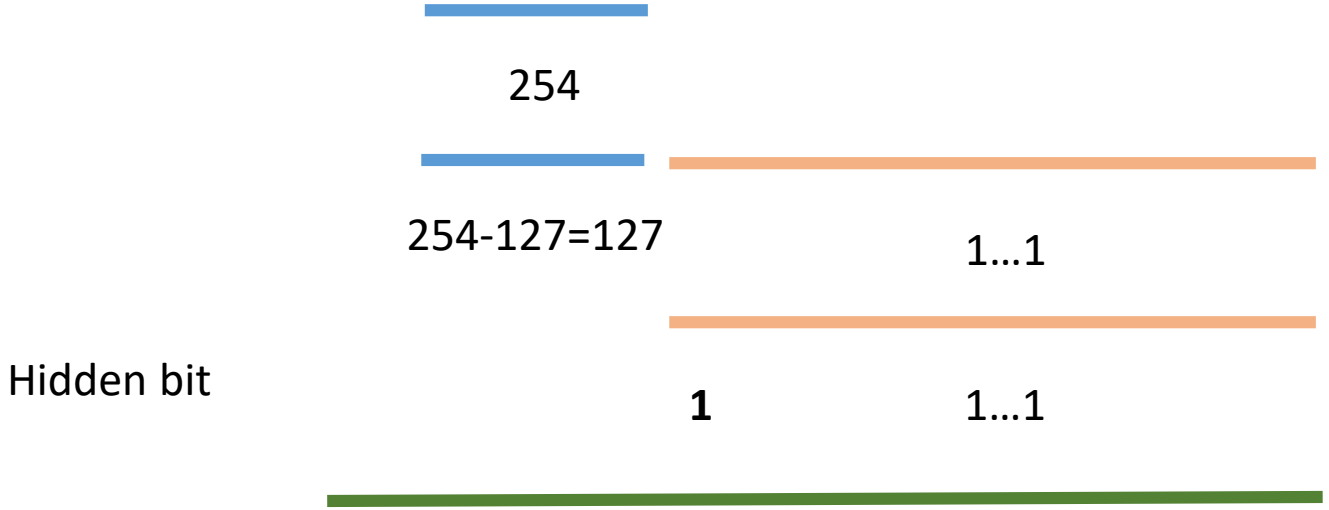


The IEEE 754 Standard for Floating-Point Arithmetic

What is the **maximal** value in the IEEE 754 single precision representation?

0 | 11111110 | 11111111111111111111111

(all 1s in the exponent are reserved)



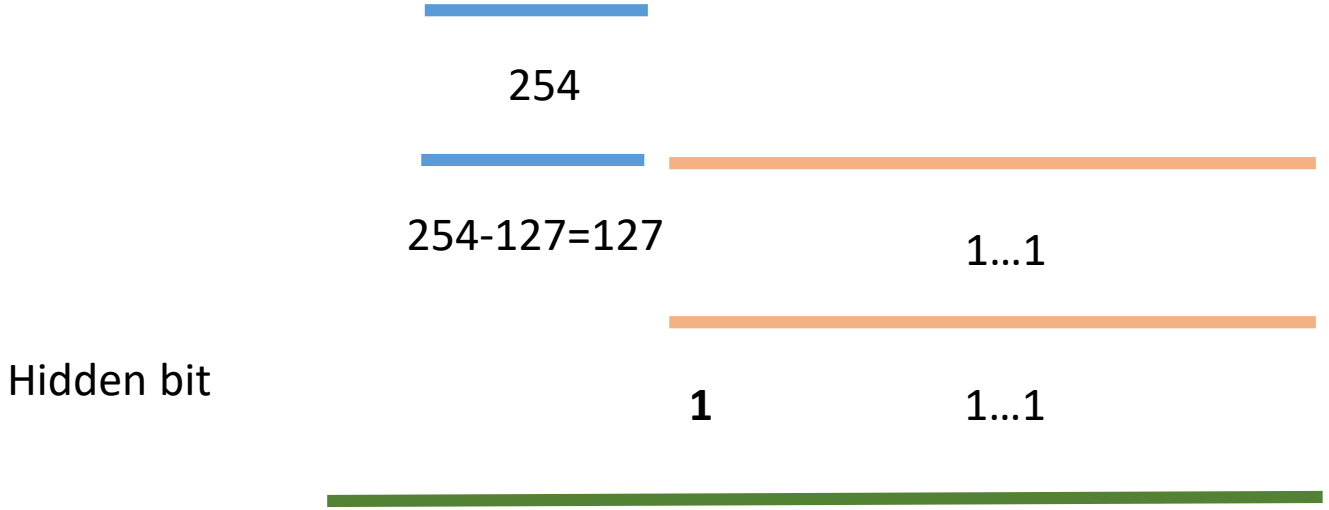
$$D = (-1)^S \cdot M \cdot B^E = 1 \cdot \left(1 + 1 \cdot 2^{-1} + \cdots + 1 \cdot B^{-23}\right) \cdot 2^{127} = \left(2^{24} - 1\right) \cdot 2^{104} \approx 3.40282347\text{e+}38$$

The IEEE 754 Standard for Floating-Point Arithmetic

What is the **maximal** value in the IEEE 754 single precision representation?

0 | 11111110 | 11111111111111111111111

(all 1s in the exponent are reserved)



$$D = (-1)^S \cdot M \cdot B^E = 1 \cdot \left(1 + 1 \cdot 2^{-1} + \cdots + 1 \cdot B^{-23}\right) \cdot 2^{127} = \left(2^{24} - 1\right) \cdot 2^{104} \approx 3.40282347\text{e+}38$$

```
constexpr float kMax { std::numeric_limits< float >::max() };
```

Floating-point – basic facts

- **In the FP domain we always work with approximations of real values**.
- Computations with the FP numbers can result in overflow, underflow, or are burdened by **roundoff errors**.
- **In FP the commutative low does not hold:**

  $(a + b) + c \neq a + (b + c)$ for $a, b, c \in$ FP.

  In FP **ORDER OF ADDITIONS DOES MATTER!**

- Machine epsilon conveys a value represented by the lowest bit of the significand. This is a difference between 1.0 and the next closest higher value representable in the FP format. A product with D provides a spacing assessment ➔ **thresholds** in iterations.
- **Adding values with different exponents leads to large errors.**
- **Subtracting close values can lead to severe cancellation errors**.

Floating-point – computation hints

- When subtracting/comparing FP values, if:

$$|x_{n+1} - x_n| \le \varepsilon \cdot \max\{|x_n|, |x_{n+1}|\}$$

then x_{n+1} and x_n are "FP-equal" (`std::numeric_limits<T>::epsilon()`).

- When dividing FP values – put an upper bound a_{max} on the divident

$$c = a / b \quad \Leftrightarrow \quad a \le a_{\max} \quad and \quad b \ge a_{\max} / k_{\max}$$

or a lower on the divisor

$$c = a / b \quad \Leftrightarrow \quad a \le b_{\min} \cdot k_{\max} \quad and \quad b \ge b_{\min}$$

(`std::numeric_limits<T>::min()`, `std::numeric_limits<T>::max()`)

For more details on FP computations see my talk 2020 & my book.

The Quest for a 16-bit Floating-Point Format

The Quest for a 16-bit Floating-Point Format

In 2008 a revision of IEEE 754 was published, defining a floating point format that occupies exactly 16 bits, alco called a **binary16**:

- 1 bit for the sign
- 5 bits for the exponent
- 10 bits for the significand (+1 hidden bit)

There are many „custom”” versions of a 16-bit float:

- Nvidia and Microsoft defined the half datatype in the Cg language (2002).
- Then implemented in silicon in the GeForce FX (2002).
- Used in few computer graphics formats, such as MATLAB, OpenEXR, JPEG XR, GIMP, OpenGL, Cg, Direct3D, and D3DX.
- Many more: *https://en.wikipedia.org/wiki/Half-precision_floating-point_format*

The Quest for a 16-bit Floating-Point Format – C++ Short-Float Proposal

Adding a fundamental type to the C++ standard for the Short Float by Boris Fomitchev et al.

But, will it be accepted?

Adding Fundamental Type for Short Float

Committee: ISO/IEC JTC1 SC22 WG14
Document Number: **N2016**

Committee: ISO/IEC JTC1 SC22 WG21 EWG Evolution
Document Number: **P0192R1**

Date: 2016-02-14

Authors: Boris Fomitchev, Sergei Nikolaev, Olivier Giroux, Lawrence Crowl

Reply to:
Boris Fomitchev boris@stlport.com
Sergei Nikolaev me@cvmlib.com
Olivier Giroux ogiroux@nvidia.com
Lawrence Crowl Lawrence@Crowl.org

Contents

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2016.pdf

The Quest for a Short-Float – a 16-bit Floating-Point Extensions

GCC supports half-precision (16-bit) floating point on ARM and AArch64 targets, via the `__fp16` type defined in the ARM C Language Extensions.

6.13 Half-Precision Floating Point

On ARM and AArch64 targets, GCC supports half-precision (16-bit) floating point via the `__fp16` type defined in the ARM C Language Extensions. On ARM systems, you must enable this type explicitly with the `-mfp16-format` command-line option in order to use it.

ARM targets support two incompatible representations for half-precision floating-point values. You must choose one of the representations and use it consistently in your program.

Specifying `-mfp16-format=ieee` selects the IEEE 754-2008 format. This format can represent normalized values in the range of *2^{-14}* to 65504. There are 11 bits of significand precision, approximately 3 decimal digits.

Specifying `-mfp16-format=alternative` selects the ARM alternative format. This representation is similar to the IEEE format, but does not support infinities or NaNs. Instead, the range of exponents is extended, so that this format can represent normalized values in the range of *2^{-14}* to 131008.

The GCC port for AArch64 only supports the IEEE 754-2008 format, and does not require use of the `-mfp16-format` command-line option.

The `__fp16` type may only be used as an argument to intrinsics defined in `<arm_fp16.h>`, or as a storage format. For purposes of arithmetic and other operations, `__fp16` values in C or C++ expressions are automatically promoted to `float`.

The ARM target provides hardware support for conversions between `__fp16` and `float` values as an extension to VFP and NEON (Advanced SIMD), and from ARMv8-A provides hardware support for conversions between `__fp16` and `double` values. GCC generates code using these hardware instructions if you compile with options to select an FPU that provides them; for example, `-mfpu=neon-fp16 -mfloat-abi=softfp`, in addition to the `-mfp16-format` option to select a half-precision format.

Language-level support for the `__fp16` data type is independent of whether GCC generates code using hardware floating-point instructions. In cases where hardware support is not specified, GCC implements conversions between `__fp16` and other types as library calls.

It is recommended that portable code use the `_Float16` type defined by ISO/IEC TS 18661-3:2015. See Floating Types.

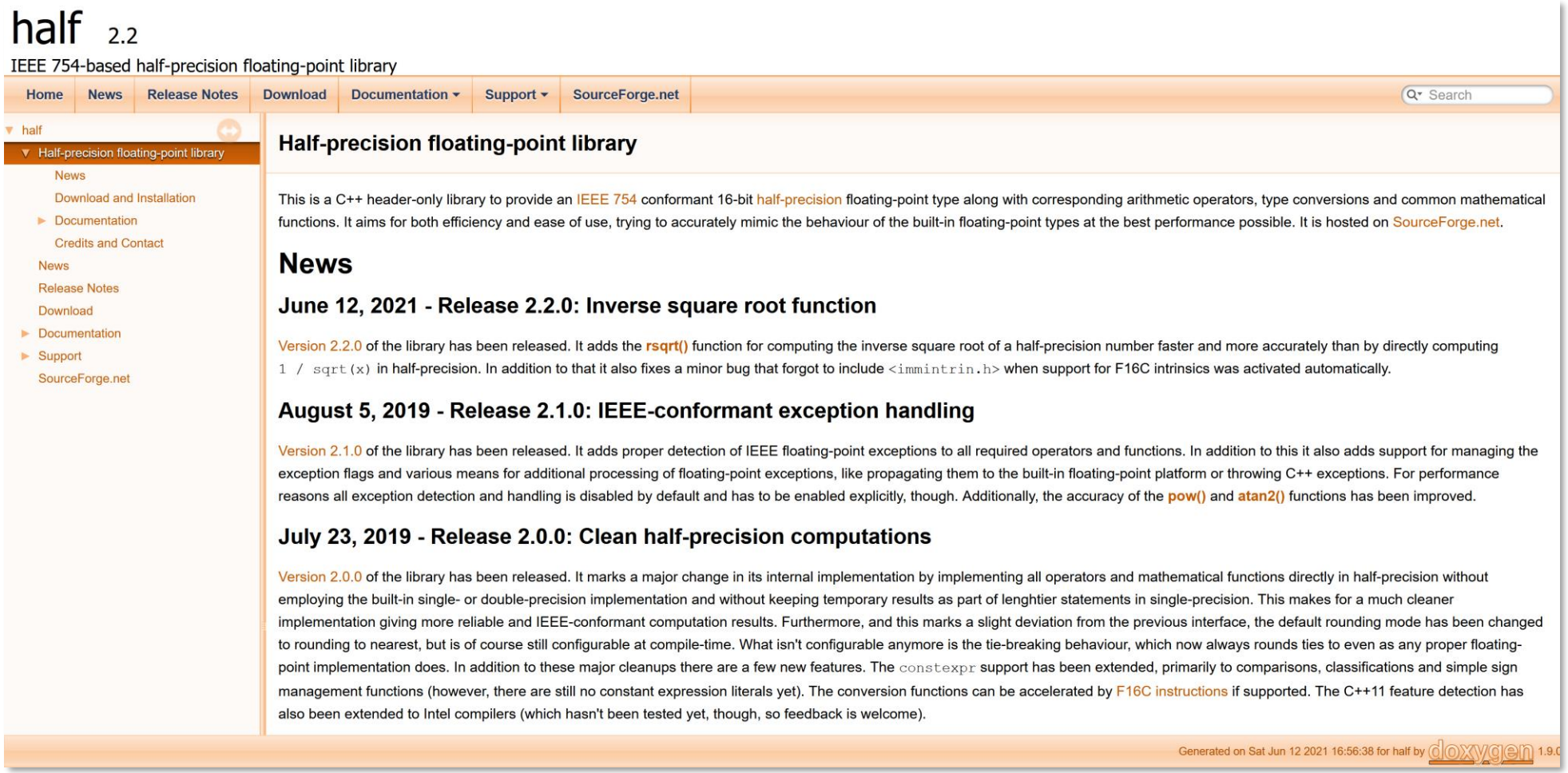
Next: Decimal Float, Previous: Floating Types, Up: C Extensions [Contents][Index]

https://gcc.gnu.org/onlinedocs/gcc/Half-Precision.html

The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Developed by Christian Rau and released under the MIT License.

http://half.sourceforge.net/



The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Examples:

```
#include "half.hpp"
```

```
using half_float::half;

half r( 12.13 );
const half    kPi( 3.14159 );

std::cout << "area = " << kPi * r * r << std::endl;
```

```
area = 462.25
```

The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Examples:

```
#include "half.hpp"
#include <numbers>          // Yes, finally we have constants in C++20

using half_float::half;

half r( 12.13 );
const auto kPi_h = std::numbers::pi_v< half >;

std::cout << "area = " << kPi_h * r * r << std::endl;
```

```
area = 462.25
```

The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Examples:

```
#include "half.hpp"
#include <numbers>          // Yes, finally we have constants in C++20

using half_float::half;

half r( 12.13 );
const auto kPi_h = std::numbers::pi_v< half >;

std::cout << "area = " << kPi_h * r * r << std::endl;
```

```
area = 462.25
```

The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Examples:

```
#include "half.hpp"
```

```
using half_float::half;

half r( 12.13 );
const auto kPi_h = half_float::acos( half(-1.f) );

std::cout << "area = " << kPi_h * r * r << std::endl;
```

```
area = 462.25
```

The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Examples:

```
#include "half.hpp"
#include <numbers>                    // Yes, finally we have constants in C++20

auto kPi_f = std::numbers::pi_v< float >;
auto kPi_d = std::numbers::pi_v< double >;

auto area_f { kPi_f * 12.13f *12.13f };
auto area_d { kPi_d * 12.13 *12.13 };

std::cout << "area_f = " << area_f << std::endl;
std::cout << "area_d = " << area_d << std::endl;

std::cout << "area_diff = " << std::fabs( area_d - area_h ) << std::endl;
```

We need to control precision & dynamics

```
area_f = 462.244
area_d = 462.244
area_diff = 0.00579589
```

The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Examples:

```
#include <limits>

template < typename FP >
void FP_limits()
{
        const auto FP_name { typeid( FP ).name() };

        std::cout << "FP_name: " << FP_name << std::endl;
        std::cout << "-------- " << std::endl;
        std::cout << "epsilon = "            << std::numeric_limits< FP >::epsilon()       << std::endl;

        std::cout << "radix = "              << std::numeric_limits< FP >::radix           << std::endl;
        std::cout << "mantissa_digits = "    << std::numeric_limits< FP >::digits          << std::endl;

        std::cout << "min = "                << std::numeric_limits< FP >::min()           << std::endl;
        std::cout << "max = "                << std::numeric_limits< FP >::max()           << std::endl;

        std::cout << "has_denorm = "         << std::numeric_limits< FP >::has_denorm      << std::endl;
        std::cout << "denorm_min = "         << std::numeric_limits< FP >::denorm_min()    << std::endl;
        std::cout << "lowest = "             << std::numeric_limits< FP >::lowest()        << std::endl;
        std::cout << "has_infinity = "       << std::numeric_limits< FP >::has_infinity    << std::endl;
        std::cout << "round_style = " <<
                ( std::numeric_limits< FP >::round_style == std::round_to_nearest
                ? "round_to_nearest" : "round_toward_zero" ) << std::endl;
        std::cout << "round_error = "                << std::numeric_limits< FP >::round_error() << std::endl;
}
```

The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Examples:

`FP_limits< half >();`	`FP_limits< float >();`	`FP_limits< double >();`
FP_name: class half_float::half -------- epsilon = 0.000976562 radix = 2 mantissa_digits = 11 min = 6.10352e-05 max = 65504 has_denorm = 1 denorm_min = 5.96046e-08 lowest = -65504 has_infinity = 1 round_style = round_to_nearest round_error = 0.5	FP_name: float -------- epsilon = 1.19209e-07 radix = 2 mantissa_digits = 24 min = 1.17549e-38 max = 3.40282e+38 has_denorm = 1 denorm_min = 1.4013e-45 lowest = -3.40282e+38 has_infinity = 1 round_style = round_to_nearest round_error = 0.5	FP_name: double -------- epsilon = 2.22045e-16 radix = 2 mantissa_digits = 53 min = 2.22507e-308 max = 1.79769e+308 has_denorm = 1 denorm_min = 4.94066e-324 lowest = -1.79769e+308 has_infinity = 1 round_style = round_to_nearest round_error = 0.5

The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Pros:

- IEEE-754 conformance (!)
- A header only library
- `std::numeric_limits` aware
- `std::hash` aware
- Well documented
- MIT license

Cons:

- Slow (no hardware support)
- No `constexpr`
- No `std::numbers`

The Quest for a Short-Float – IEEE 754-based Half-precision FP Library

Alternatives:

A lot of research into deep learning

- NVidia Mixed-Precision Programming Package

https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/

- FlexFloat (G. Tagliavini et al. *FlexFloat: A software library for transprecision computing*, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 39, no. 1, pp. 145–156, 2020)
- Brain Floating-Point BFLOAT16 by Google in tensor-processing unit
- Intel Flexpoint (U. Köster et al., *Flexpoint: An adaptive numerical format for efficient training of deep neural networks*, in Proc. Advances Neural Information Processing Systems, 2017)
- `std::ratio` (*https://en.cppreference.com/w/cpp/numeric/ratio*)
- Fixed-point arithmetic (e.g. `FxFor` class *https://github.com/BogCyg/BookCpp*)

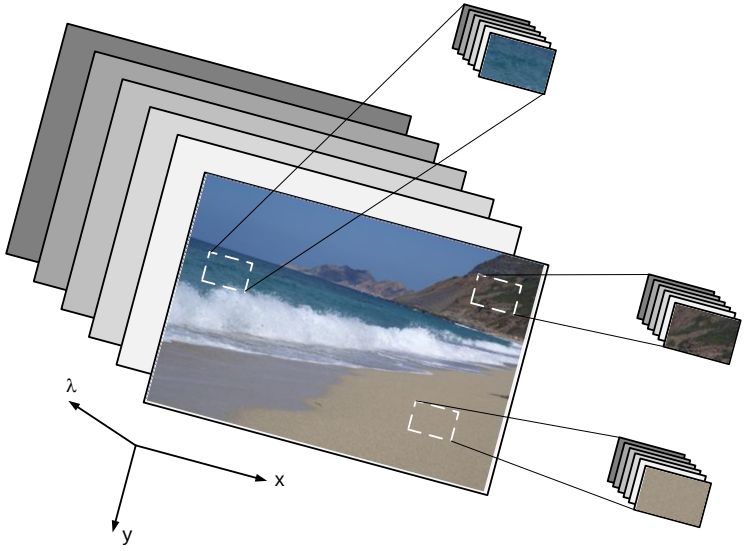
Floating-point Compression

Floating-point Compression

For large blocks of FP numbers the FP LOSSY compression schemes can be used

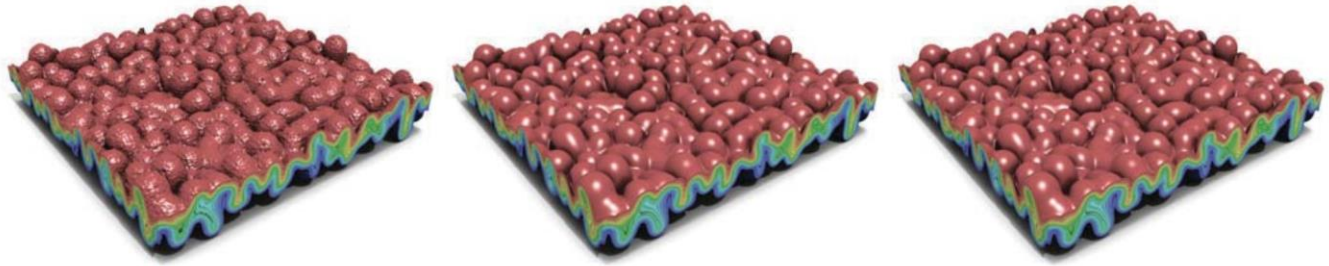
– ZFP proposed by Peter Lindstrom.

The main observation of the multi-dimensional data (tensors) is that elements of local ”sub-cubes” retain many similarities (autocorrelation), also in the exponent of their FP representation.



Fixed-Rate Compressed Floating-Point Arrays

Peter Lindstrom, *Senior Member, IEEE*



(a) 1 bit/double (b) 4 bits/double (c) 64 bits/double (no compression)

Fig. 1: Interval volume renderings of compressed double-precision floating-point data on a 384 × 384 × 256 grid. At 4 bits/double (16x compression) the image is visually indistinguishable from full 64-bit precision.

Abstract—Current compression schemes for floating-point data commonly take fixed-precision values and compress them to a variable-length bit stream, complicating memory management and random access. We present a fixed-rate, near-lossless compression scheme that maps small blocks of 4^d values in d dimensions to a fixed, user-specified number of bits per block, thereby allowing read and write random access to compressed floating-point data at block granularity. Our approach is inspired by fixed-rate texture compression methods widely adopted in graphics hardware, but has been tailored to the high dynamic range and precision demands of scientific applications. Our compressor is based on a new, lifted, orthogonal block transform and embedded coding, allowing each per-block bit stream to be truncated at any point if desired, thus facilitating bit rate selection using a single compression scheme. To avoid compression or decompression upon every data access, we employ a software write-back cache of uncompressed blocks. Our compressor has been designed with computational simplicity and speed in mind to allow for the possibility of a hardware implementation, and uses only a small number of fixed-point arithmetic operations per compressed value. We demonstrate the viability and benefits of lossy compression in several applications, including visualization, quantitative data analysis, and numerical simulation.

Index Terms—Data compression, floating-point arrays, orthogonal block transform, embedded coding

Peter Lindstrom, *Fixed-Rate Compressed Floating-Point Arrays*, IEEE Transactions on Visualization and Computer Graphics, Vol. 20, No. 12, 2014

Compressed Floating-Point Arrays

In ZFP a near-lossless compression scheme is proposed.

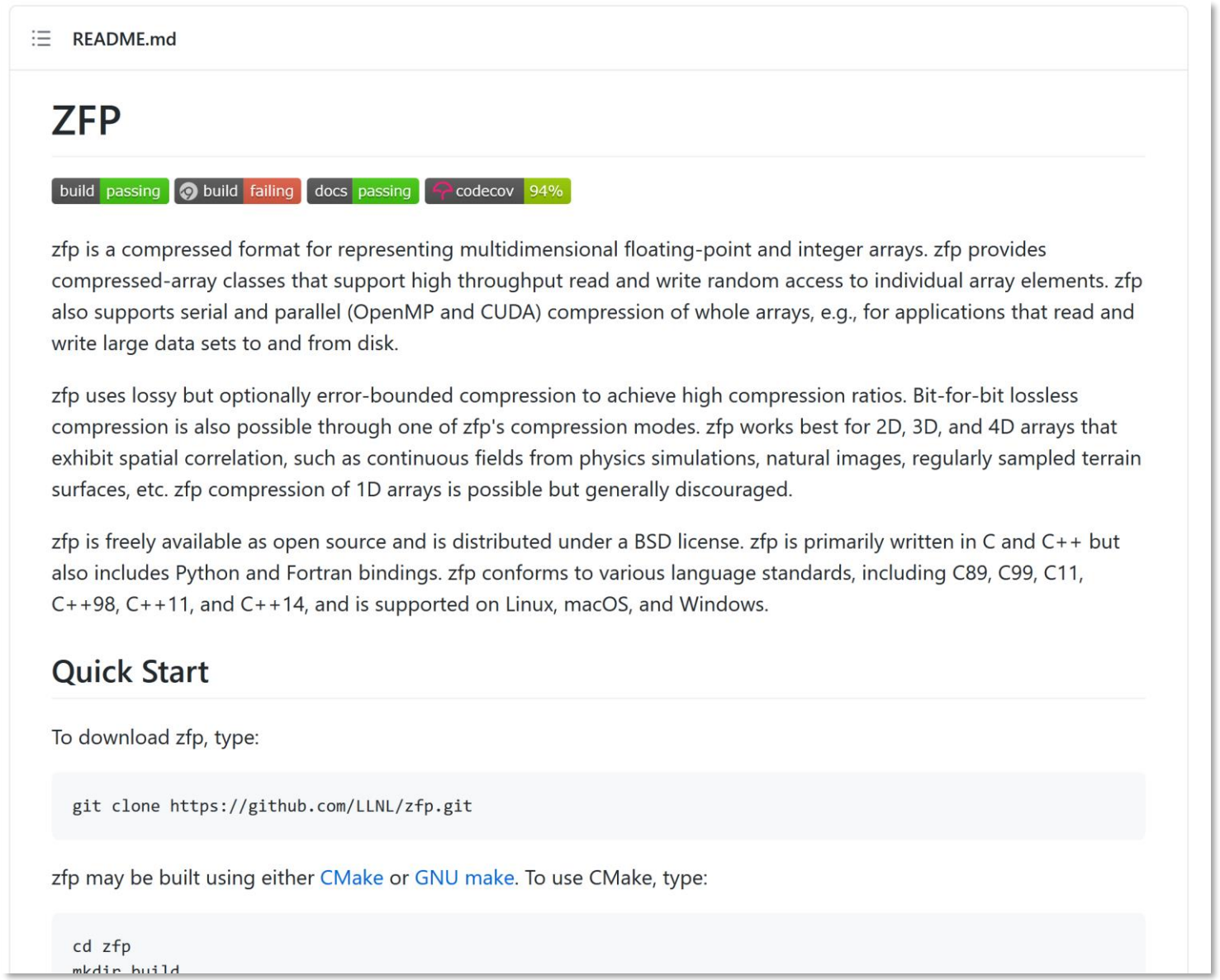
- It maps small blocks of 4^d values in d dimensions to a fixed number of bits per block (defined by the user). In 3D these are blocks (sub-arrays) of 4 x 4 x 4, etc.
- Each such a block is stored using the same, user-defined, number of bits.
- Each block is compressed in 5 steps:
  1. Align block values to a common exponent;
  2. Convert floating-point number ➔ fixed-point representation;
  3. Run an orthogonal block transform to decorrelate the values;
  4. Order the transform coeffs by their magnitude;
  5. Encode the coeffs, one ”bit plane” at a time.
- ZFP allows for random read & write access to compressed FP data at block granularity.

Peter Lindstrom, *Fixed-Rate Compressed Floating-Point Arrays*, IEEE Transactions on Visualization and Computer Graphics, Vol. 20, No. 12, 2014

Compressed Floating-Point Arrays – the ZFP Library

Characteristics of the ZFP library:

- **C++ wrapper** to the compressed array primitive. To the user appears as an 'normal' array that can be substituted into the existing code.
- Cache to optimize the frequency of compression & decompression (when updating the blocks).
- ZFP is strongly limited to regular gridded data.
- Applications in computer graphics but also in scientific multi-dimensional data analysis.



README.md

ZFP

build passing | build failing | docs passing | codecov 94%

zfp is a compressed format for representing multidimensional floating-point and integer arrays. zfp provides compressed-array classes that support high throughput read and write random access to individual array elements. zfp also supports serial and parallel (OpenMP and CUDA) compression of whole arrays, e.g., for applications that read and write large data sets to and from disk.

zfp uses lossy but optionally error-bounded compression to achieve high compression ratios. Bit-for-bit lossless compression is also possible through one of zfp's compression modes. zfp works best for 2D, 3D, and 4D arrays that exhibit spatial correlation, such as continuous fields from physics simulations, natural images, regularly sampled terrain surfaces, etc. zfp compression of 1D arrays is possible but generally discouraged.

zfp is freely available as open source and is distributed under a BSD license. zfp is primarily written in C and C++ but also includes Python and Fortran bindings. zfp conforms to various language standards, including C89, C99, C11, C++98, C++11, and C++14, and is supported on Linux, macOS, and Windows.

Quick Start

To download zfp, type:

```
git clone https://github.com/LLNL/zfp.git
```

zfp may be built using either CMake or GNU make. To use CMake, type:

```
cd zfp
mkdir build
```

https://github.com/LLNL/zfp (BSD license)

Compressed Floating-Point Arrays – the ZFP Library

Characteristics of the ZFP library (ctd):

- There are 5 compression modes:
  1. Expert;
  2. Fixed-rate (a fixed number of bits per 4^d block);
  3. Fixed-precision (the number of bits may vary, but the number of bit planes – i.e. the precision – is fixed);
  4. Fixed-accuracy (all bit planes are encoded, preferrable in many cases);
  5. Reversible (lossless).
- Parallel operation (OpenMP, CUDA).
- Different APIs:
- Low/high-level C.
- Bit-stream.
- **Compressed C++ arrays.**

Compressed Floating-Point Arrays – the ZFP Library

Operation of the compressed C++ arrays:

- ZFP arrays can easily substitute C++ arrays and STL vector.
- However, there are limitations on direct usage of addresses/references directly to the array elements.
- Each multi-dimensional ZFP array can be procesed as a linear buffer 1D but there are multi-dimension ZFP iterators.
- Some FP symbols, like NaN, are not supported (although subnormals are ok).
- There are 6 classes for 1D, 2D, and 3D arrays – single & double precision.

```
std::generate( cpp_array.begin(), cpp_array.end(), [&] () { return uni_distr( mtRandomEngine ); } );
```

Examples:

```
#include "zfparray2.h"

#include <cassert>
#include <numeric>
#include <random>
```

```
const int cols { 113 }, rows { 117 };
std::vector< double > cpp_array( cols * rows );

std::mt19937 mtRandomEngine( ( std::random_device() )() );
std::uniform_real_distribution uni_distr( -123.45, 123.45 );


std::generate( cpp_array.begin(), cpp_array.end(),
               [&] () { return uni_distr( mtRandomEngine ); };
```

Compressed Floating-Point Arrays – the ZFP Library

```
const int cols { 113 }, rows { 117 };
std::vector< double > cpp_array( cols * rows );

std::mt19937 mtRandomEngine( ( std::random_device() )() );
std::uniform_real_distribution uni_distr( -123.45, 123.45 );


std::generate( cpp_array.begin(), cpp_array.end(),
               [&] () { return uni_distr( mtRandomEngine ); };
```

```
double rate { 4. * 4. };

// The rate - how many bits per value to store in the compressed representation.
zfp::array2d my_zfp_array ( cols, rows, rate );

assert( cpp_array.size() == my_zfp_array.size() );

std::copy( cpp_array.begin(), cpp_array.end(), my_zfp_array.begin() );
```

Compressed Floating-Point Arrays – the ZFP Library

```
double rate { 4. * 4. };

// The rate - how many bits per value to store in the compressed representation.
zfp::array2d my_zfp_array ( cols, rows, rate );

assert( cpp_array.size() == my_zfp_array.size() );

std::copy( cpp_array.begin(), cpp_array.end(), my_zfp_array.begin() );
```

```
auto diff = std::transform_reduce(        // Compute Mean Square Error
                        cpp_array.begin(), cpp_array.end(), my_zfp_array.begin(),
                        double(),

                        [] ( const auto a, const auto b ) { return a + b; },
                        [] ( const auto a, const auto b ) { return ( a - b ) * ( a - b ); } );

const auto org_size { cpp_array.size() * sizeof( double ) };

const auto compr_size { my_zfp_array.compressed_size() };  // number of bytes of compressed data
```

Compressed Floating-Point Arrays – the ZFP Library

```
auto diff = std::transform_reduce(        // Compute Mean Square Error
                        cpp_array.begin(), cpp_array.end(), my_zfp_array.begin(),
                        double(),

                        [] ( const auto a, const auto b ) { return a + b; },
                        [] ( const auto a, const auto b ) { return ( a - b ) * ( a - b ); } );

const auto org_size { cpp_array.size() * sizeof( double ) };

const auto compr_size { my_zfp_array.compressed_size() };  // number of bytes of compressed data
```

```
std::cout << "           size [bytes] = " << org_size << std::endl;
std::cout << "compressed size [bytes] = " << compr_size << std::endl;

std::cout << "    compressed ratio [%] = "
          << ( 1.0 - (double) compr_size / (double) org_size ) * 100. << std::endl;

std::cout << "                    MSE = " << std::sqrt( diff ) << std::endl;
```

```
           size [bytes]  = 105768
compressed size [bytes]  =  27840
   compressed ratio [%]  = 73.6782
                    MSE  = 0.634047
```

Compressed Floating-Point Arrays

Alternatives and the future:

- ZFP hardware implementation (FPGA) ➔ *SystemC* language.

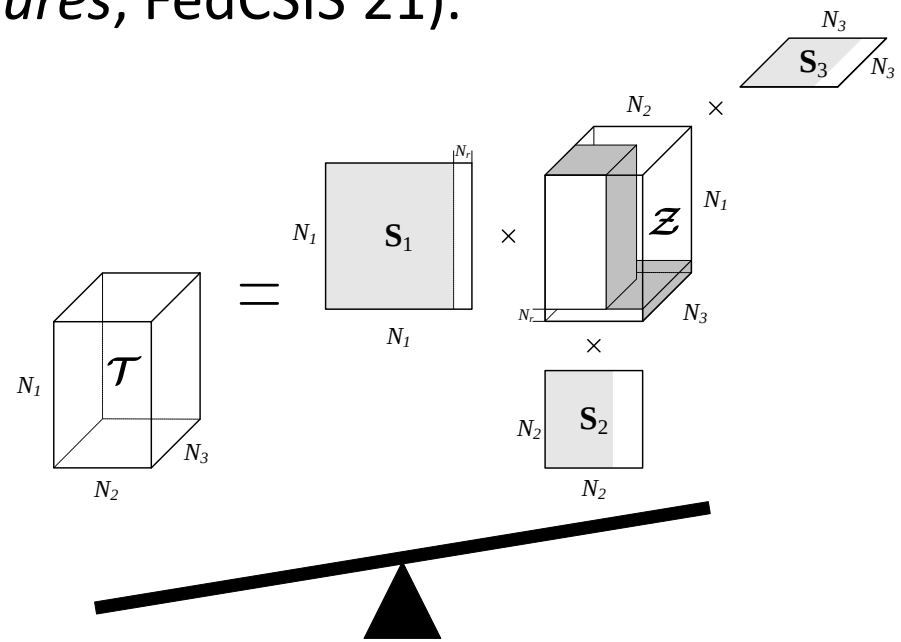
  (*https://github.com/LLNL/zhw*)

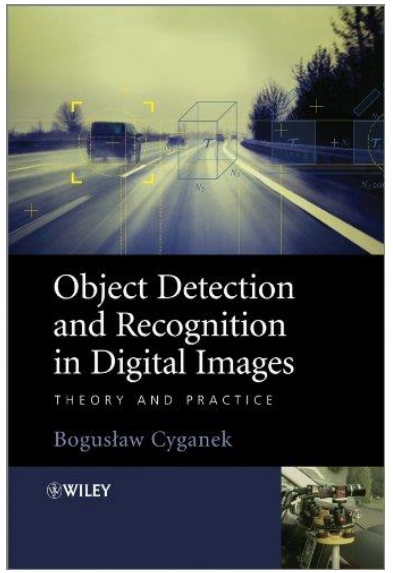
- SZ error-bounded lossy compression method with slightly improved linear-scaling quantization which can be beneficial in some applications (compression of nonzero weights in deep convolutional networks – DeeepSZ. (*https://github.com/szcompressor/SZ*)

Compressed Floating-Point (Arrays) Tensors & Deep Learning

Alternatives and the future:

- Tensor based lossy compression ➔ a large and separate subject. An example of deep learning + tensor decomposition + ZFP in our paper showing Our experiments show data compressions of 94%-97% that result in only 0.6-0.7% accuracy drop of CNNs (J. Grabek & B. Cyganek: *An impact of tensor-based data compression methods on the training and prediction accuracy of the popular deep convolutional neural network architectures*, FedCSIS'21).





New format – POSIT Arithmetic

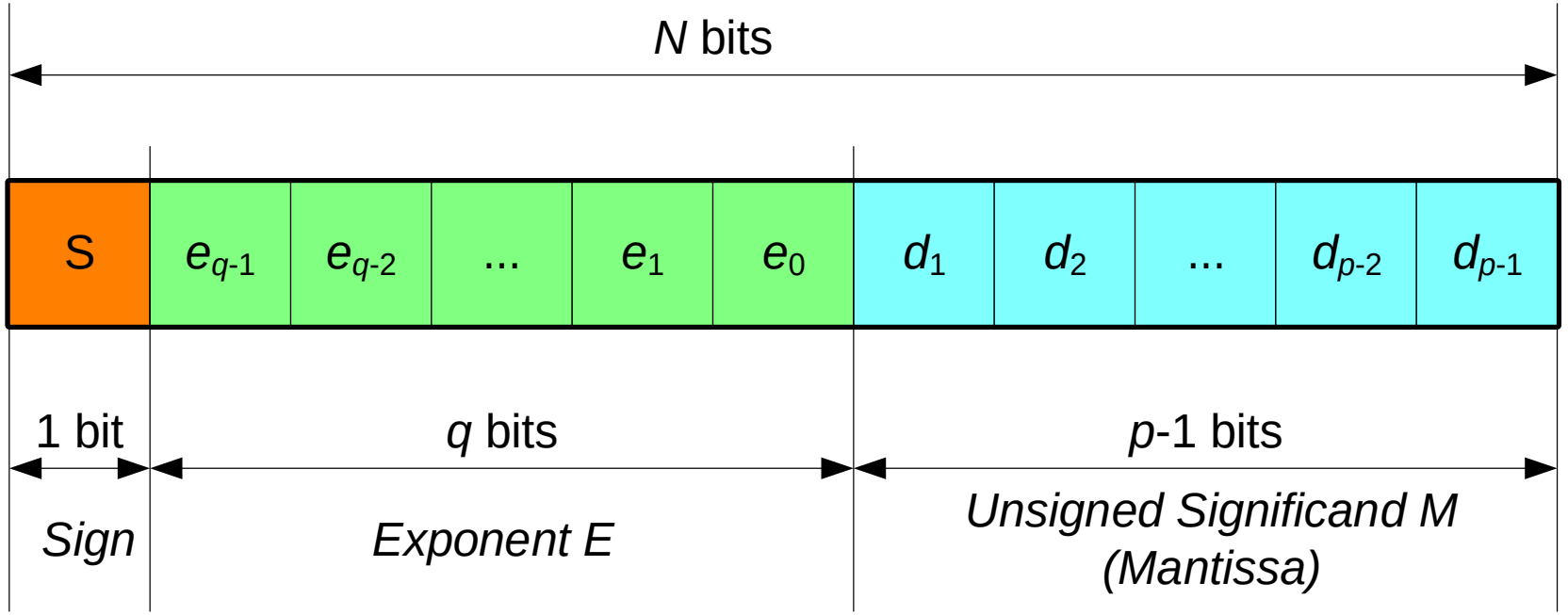
New format – POSIT Arithmetic

Developed by John L. Gustafson as **a direct replacement** for the IEEE 754 FP standard. Posits provide compelling advantages over floats.

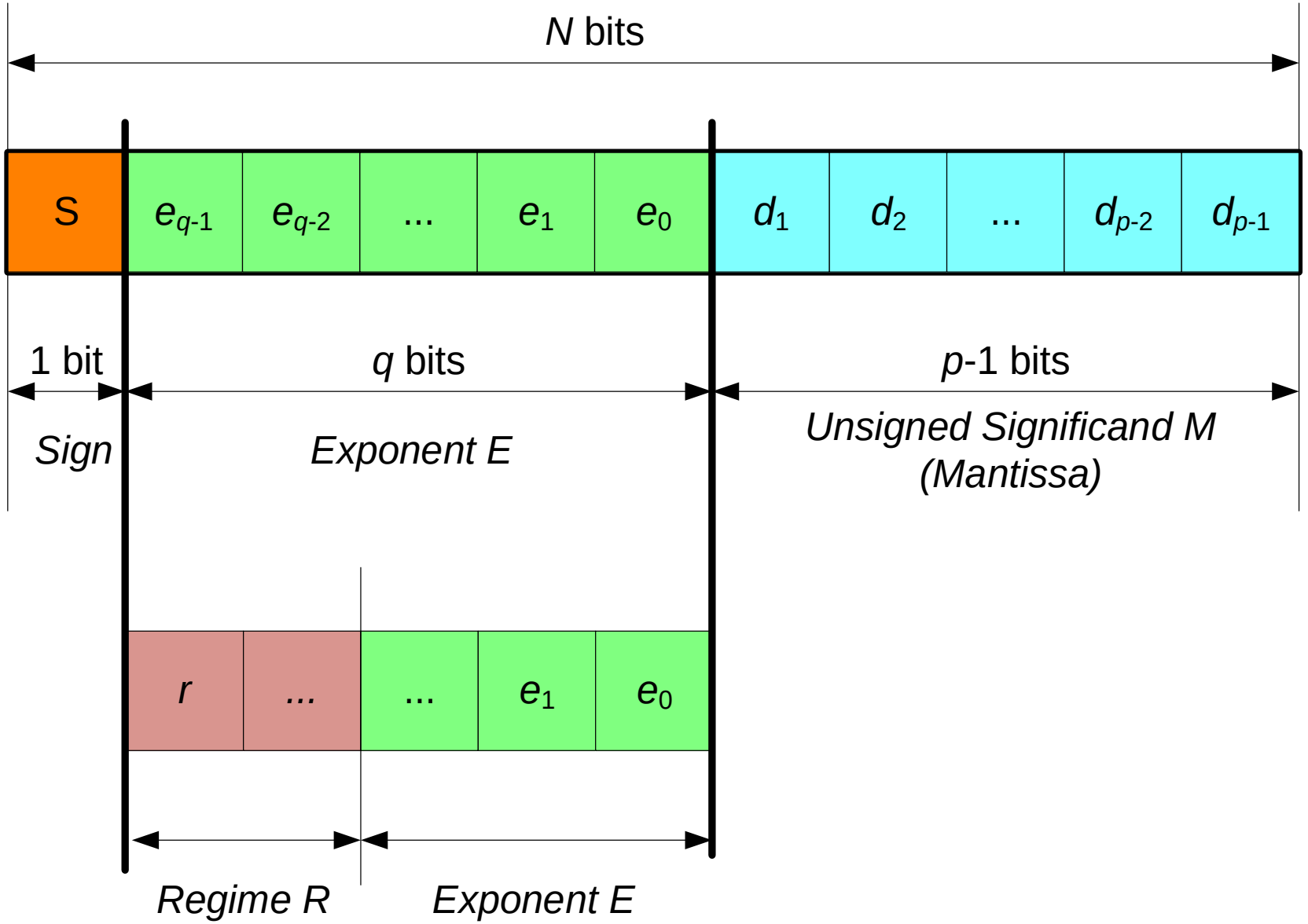
- Larger dynamic range due to the **variable-length encoding of exponent** (Golomb-Rice).
- More faction bits if small exponent is sufficient.
- Never overflow to infinity or underflow to zero.
- NaN indicates an action instead of a bit pattern.
- More concise bit assignment (only one 0, NaN is not a bit representation – its interruption).
- Posit environment mandates the fused operations (MultAdd, Sum, Dot, etc.)

J. L. Gustafson and I. T. Yonemoto. *Beating floating point at its own game: Posit arithmetic*. Supercomputing Frontiers and Innovations , 4(2):71–86, 2017.

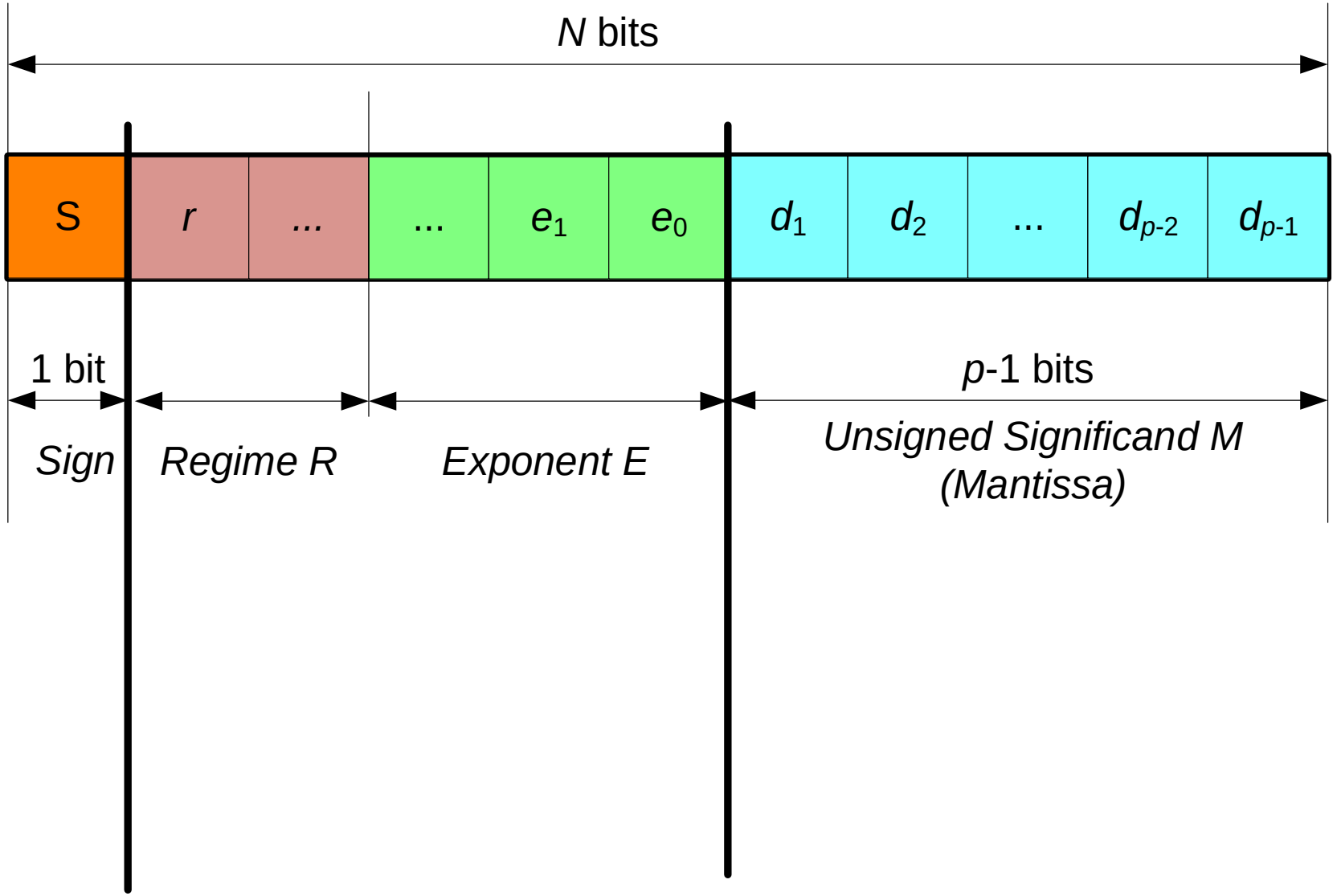
New format – POSIT Arithmetic



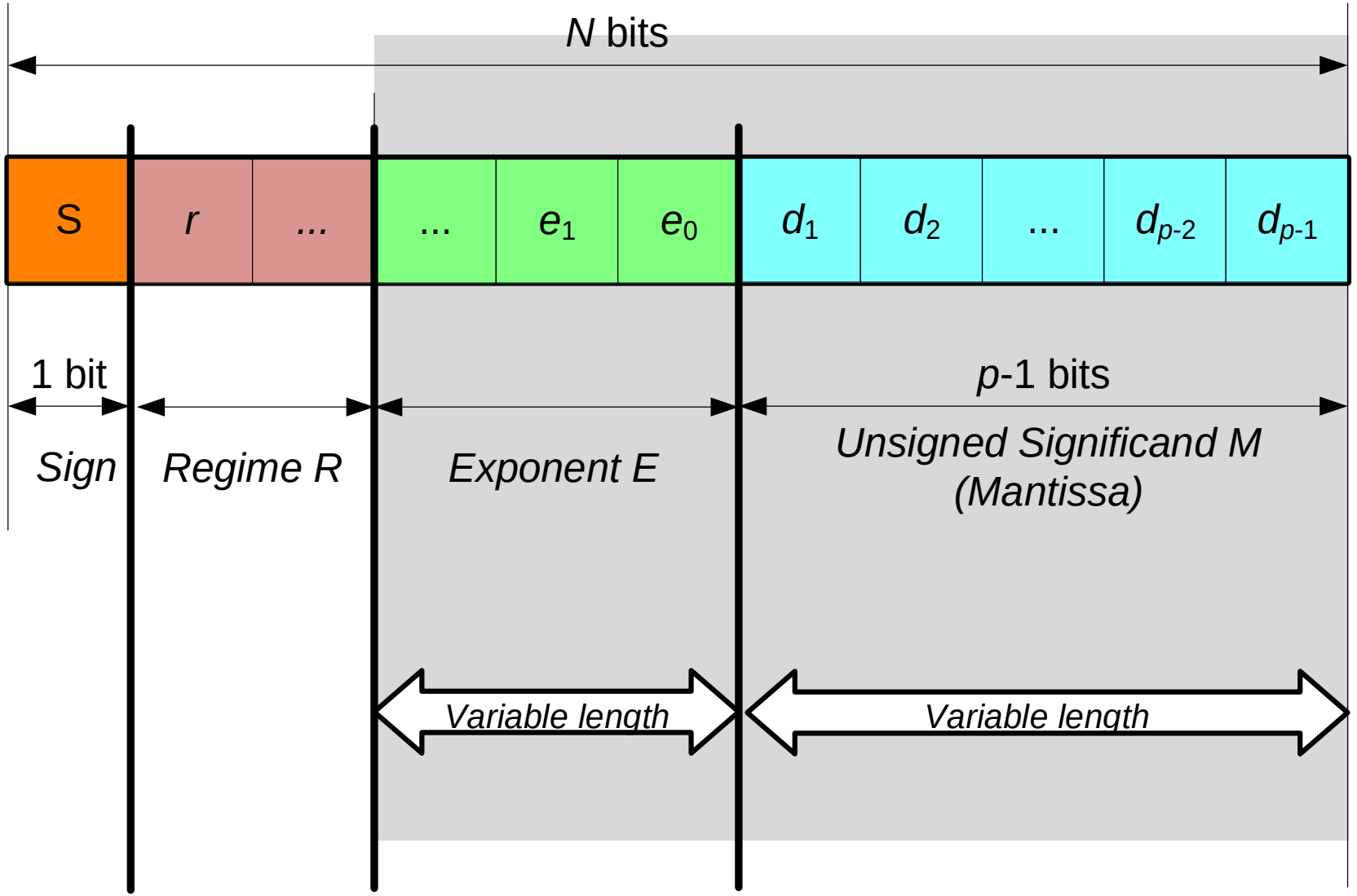
New format – POSIT Arithmetic



New format – POSIT Arithmetic

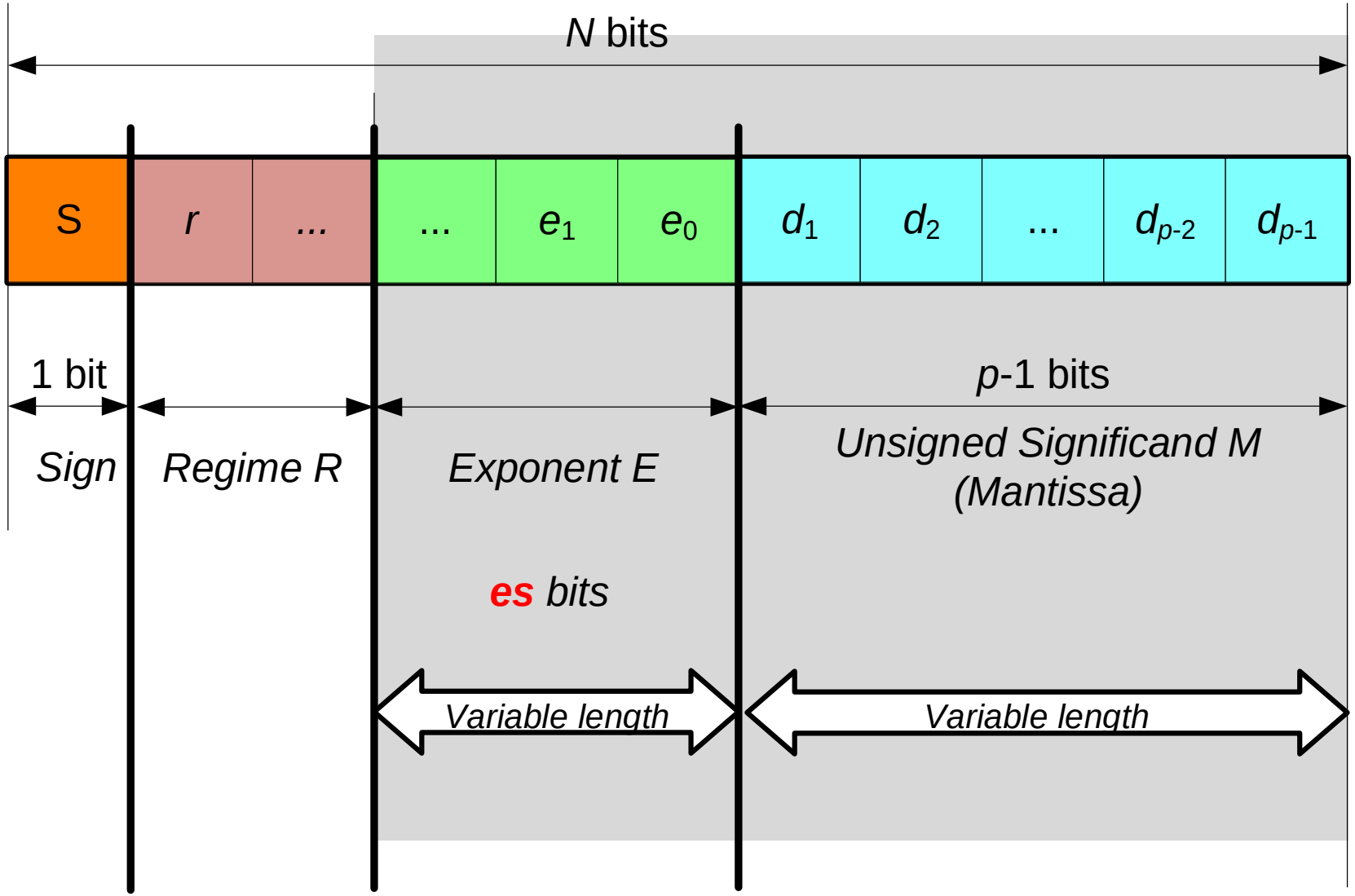


New format – POSIT Arithmetic



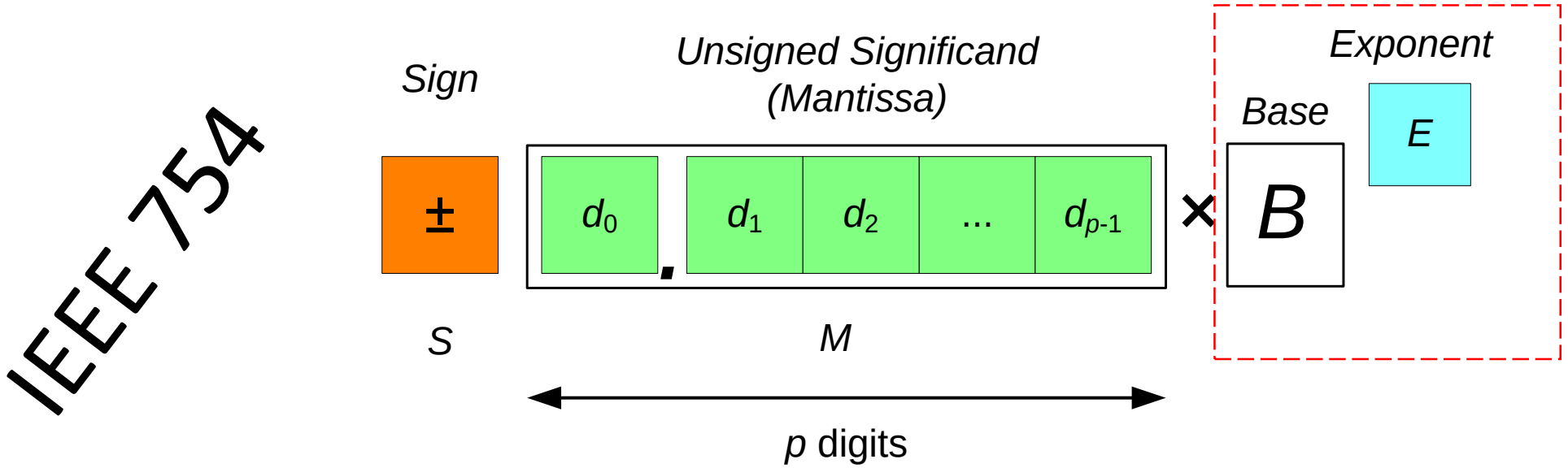
The variable-length encoding of exponent (Golomb-Rice)

New format – POSIT Arithmetic



The variable-length encoding of exponent (Golomb-Rice)

New format – POSIT Arithmetic



A value D of a number is given as follows

$$D = (-1)^S \cdot M \cdot B^E = (-1)^S \cdot (d_o \bullet d_1 \ldots d_{p-1}) \cdot B^E$$

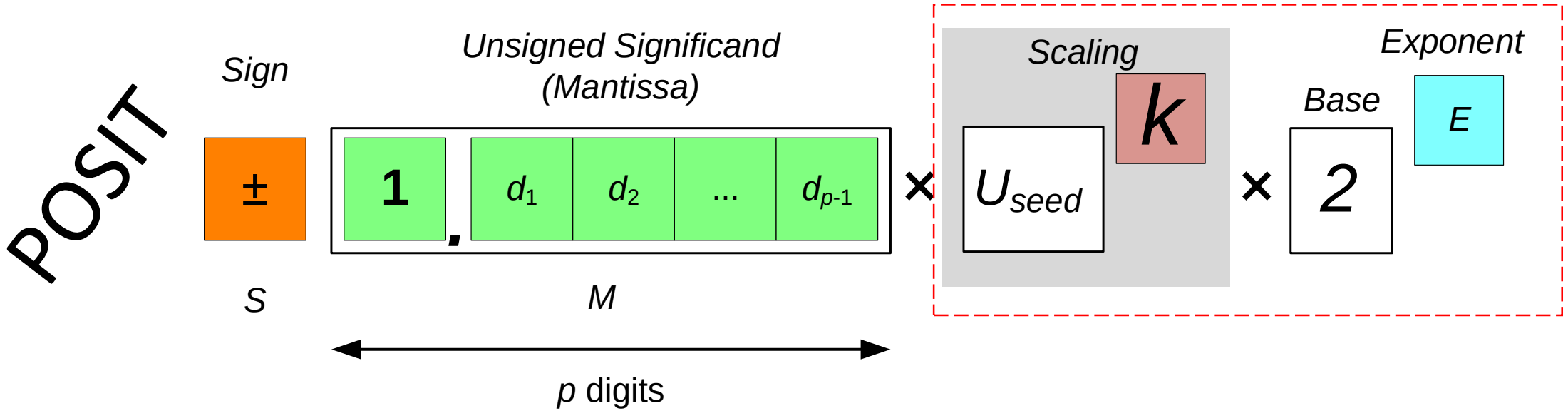
M is unsigned **significand** (**mantissa**, fraction), *B* is a base, and *E* denotes the exponent

$$E_{\min} \le E \le E_{\max}$$

For *p* digits and the base *B*, a value of the significand is given as follows

$$M = d_0 + d_1 \cdot B^{-1} + \cdots + d_{p-1} \cdot B^{-(p-1)}$$

New format – POSIT Arithmetic



A POSIT value *P* of a number is given as follows

$$P = (-1)^S \cdot M \cdot \left(U_{seed}{}^k \cdot 2^E\right) = (-1)^S \cdot \left(1 \bullet d_1 \ldots d_{p-1}\right) \cdot \left(U_{seed}{}^k \cdot 2^E\right)$$

M is unsigned **significand** (**mantissa**, fraction), *B* is a base, and *E* denotes the exponent

For *p* digits and the base *B=2*, a value of the significand is given as follows

$$M = 1 + d_1 \cdot 2^{-1} + \cdots + d_{p-1} \cdot 2^{-(p-1)} = 1 + \frac{d_1 \cdot 2^{(p-2)} + \cdots + d_{p-1}}{2^{(p-1)}}$$

New format – POSIT Arithmetic

A COMPLETE POSIT value *X* of a number is given as follows

$$P = \begin{cases} 0 & 00-0 \\ \pm\infty & 10-0 \\ (-1)^S \cdot M \cdot \left(U_{seed}{}^k \cdot 2^E\right) = (-1)^S \cdot \left(1 \bullet d_1 \ldots d_{p-1}\right) \cdot \left(U_{seed}{}^k \cdot 2^E\right) & other \end{cases}$$

The RULES (in many aspects different than for IEEE 754):

- The sign bit: 0 for positive numbers, 1 for negative numbers
- **BUT** - If negative, take the 2's complement before decoding the regime, exponent, and significand (fraction).

New format – POSIT Arithmetic

The REGIME bits – the numerical value *k* is determined by the run length of the bits

$$P = (-1)^S \cdot M \cdot \left(U_{seed}{}^k \cdot 2^E \right)$$

Binary string	0000	0001	001x	01xx	10xx	110x	1110	1111
Value *k*	-4	-3	-2	-1	0	1	2	3

A stream of 0s or 1s is terminated either when **the next bit is opposite**, or the end of the string is reached.

For the number *m* of identical bits in the run:

- if the bits are 0, then $k = -m$
- if they are 1, then $k = m - 1$

The regime indicates a scale factor of $U_{seed}{}^k$, where $U_{seed} = 2^{2^{es}}$

es	0	1	2	3	4
$U_{seed} = 2^{2^{es}}$	2	2^2=4	2^4=16	2^8=256	2^{16}=65536

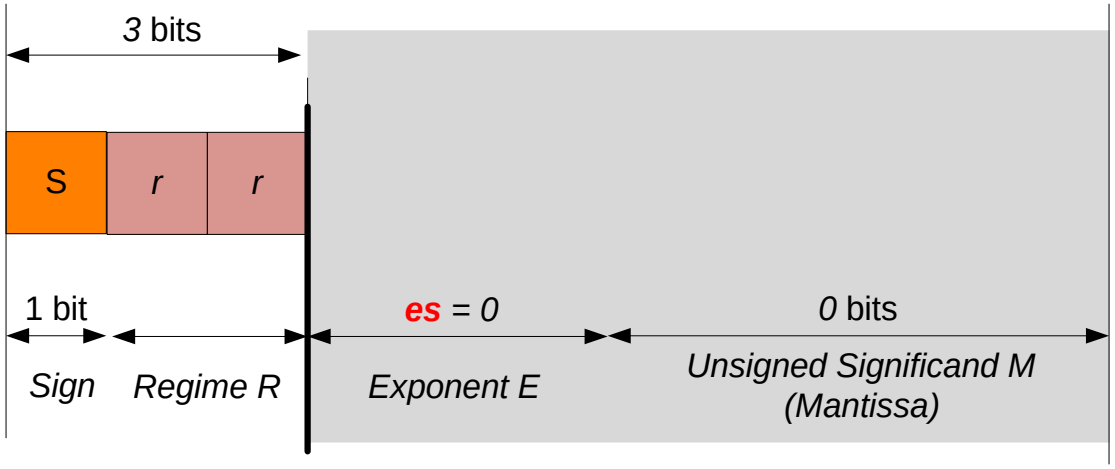
New format – POSIT Arithmetic

POSITS features:

- The exponent **E is regarded as an unsigned integer**. There is no bias as there is for floats.
- There can be up to *es* exponent bits, depending on how many bits remain to the right of the regime.
- A tapered accuracy – numbers near 1 in magnitude have more accuracy than extremely large or extremely small numbers (less common in calculations).
- If there are any bits remaining after the regime and the exponent bits, they represent the fraction (mantissa), *f*, just like the fraction 1.*f* in a float, with a hidden bit that is always 1.
- There are **no subnormal** numbers with a hidden bit of 0, there are for floats.

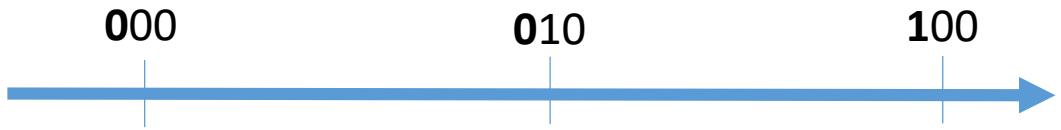
New format – POSIT Arithmetic

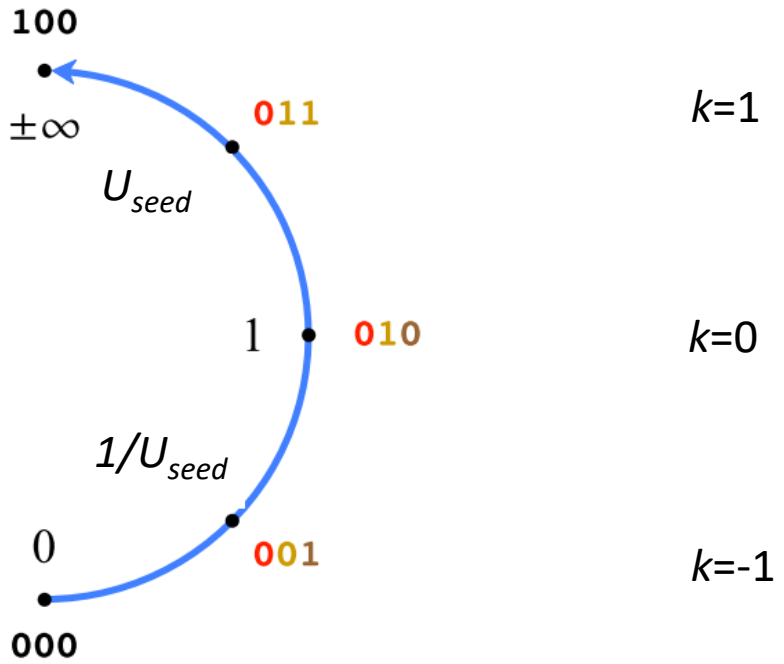
This will be the first example how to interpret POSITS format and how this numer system builds-up.



es=0, so: the scale factor of $U_{seed} = 2^{2^{es}} = 2^{2^0} = 2$

Positive values for a 3-bit posit (2^3=8)





Instead of a linear scale a circular representation is more intuitive:

New format – POSIT Arithmetic

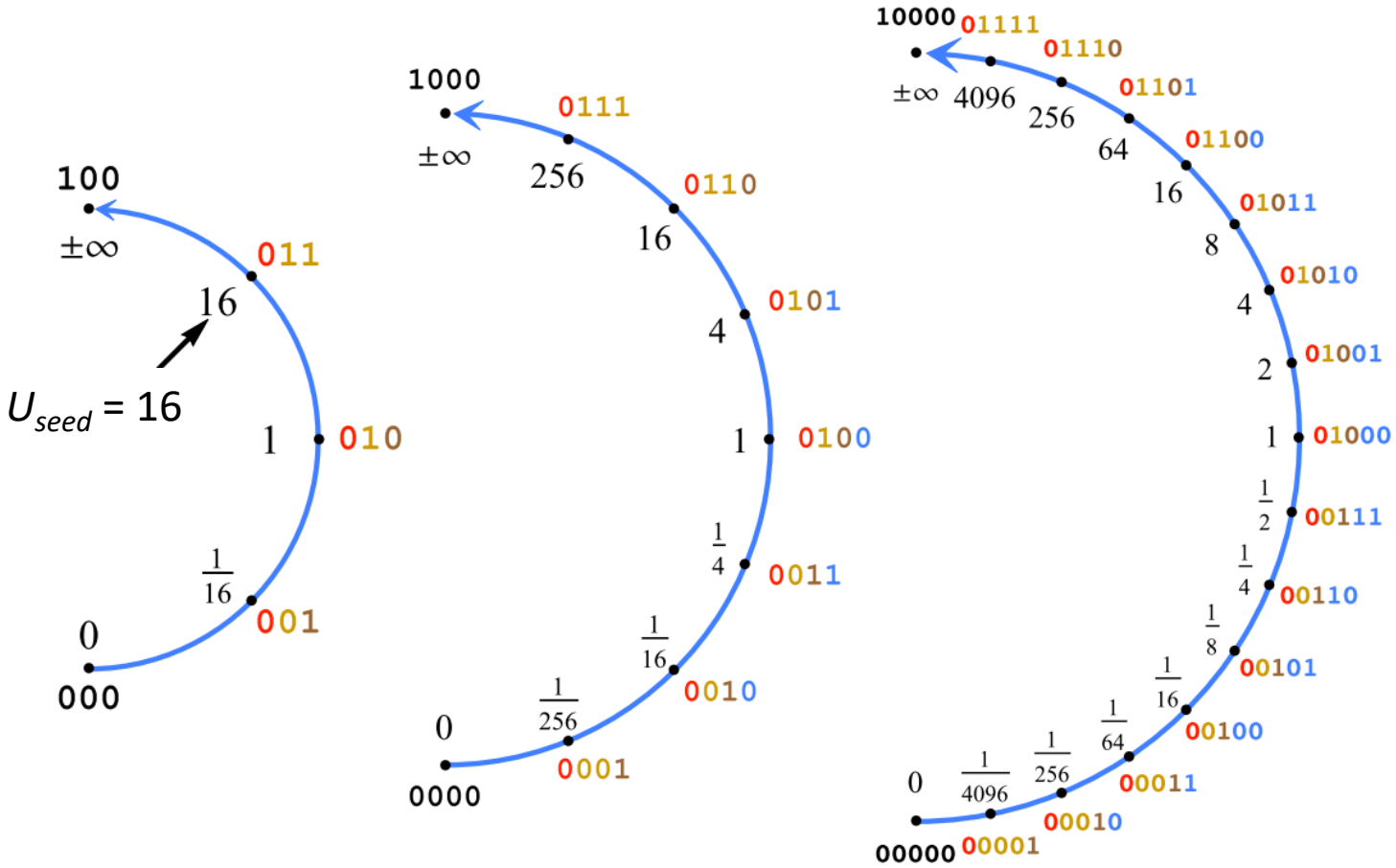
Extending values POSITS:

- Posit precision increases by appending bits, and values remain where they are on the circle when a 0 bit is appended.
- Appending a 1 bit **creates a new value between two posits** on the circle.
- There are special interpolation rules (with *maxpos* and *minpos* being the largest and the smallest positive):
  1. Between *maxpos* and inf the new value *maxpos* * U_{seed}, and between 0 and *minpos* the new value is *minpos* / U_{seed} (new ”amber” regime bit added).
  2. Between the existing values for $x=2^m$ and $y=2^n$, for m, n different by more than 1, the *geometric mean* $\text{sqrt}(x*y)=2^{(m+n)/2}$ (new ”blue” exponent bit added).
  3. For a midway between closest neighboring values the *arithmetic mean* $(x+y)/2$ (new ”black” fraction bit).

J. L. Gustafson and I. T. Yonemoto. *Beating floating point at its own game: Posit arithmetic*. Supercomputing Frontiers and Innovations , 4(2):71–86, 2017.

New format – POSIT Arithmetic

Posit construction from 3 bits to 5 bits with two exponent bits, *es* = 2, U_{seed} = 16:



3 bits 4 bits 5 bits

New format – POSIT Arithmetic

An example, $es = 3$, $p = (1+8)$, $U_{seed} = 2^{2(es)}=256$:

0 0001 010 10011001

sgn regime exponent fraction

$$+\ 256^{-3} * 2^{2} * (1 + 153/256) \approx 3.8091e\text{-}07$$

New format – IEEE 754 vs. POSIT

PARAMETER	IEEE 754	POSITS
Word Length	16, 32, 64, 128	Fixed overall length but with dynamic exponent and fraction

New format – IEEE 754 vs. POSIT

PARAMETER	IEEE 754	POSITS
Word Length	16, 32, 64, 128	Fixed overall length but with dynamic exponent and fraction
Sign	Sign/Magnitude (-0 vs. +0)	Two's complement (one 0)

New format – IEEE 754 vs. POSIT

PARAMETER	IEEE 754	POSITS
Word Length	16, 32, 64, 128	Fixed overall length but with dynamic exponent and fraction
Sign	Sign/Magnitude (-0 vs. +0)	Two's complement (one 0)
Exponent	Fixed length 5, 8, 11, 15, biased	**Variable** length, no bias

New format – IEEE 754 vs. POSIT

PARAMETER	IEEE 754	POSITS
Word Length	16, 32, 64, 128	Fixed overall length but with dynamic exponent and fraction
Sign	Sign/Magnitude (-0 vs. +0)	Two's complement (one 0)
Exponent	Fixed length 5, 8, 11, 15, biased	**Variable** length, no bias
Mantissa	Fixed length 1.d…d	**Variable** length 1.d…d

New format – IEEE 754 vs. POSIT

PARAMETER	IEEE 754	POSITS
Word Length	16, 32, 64, 128	Fixed overall length but with dynamic exponent and fraction
Sign	Sign/Magnitude (-0 vs. +0)	Two's complement (one 0)
Exponent	Fixed length 5, 8, 11, 15, biased	**Variable** length, no bias
Mantissa	Fixed length 1.d…d	**Variable** length 1.d…d
Infinities	$-\infty$ $+\infty$	$\pm\infty$

New format – IEEE 754 vs. POSIT

PARAMETER	IEEE 754	POSITS
Word Length	16, 32, 64, 128	Fixed overall length but with dynamic exponent and fraction
Sign	Sign/Magnitude (-0 vs. +0)	Two's complement (one 0)
Exponent	Fixed length 5, 8, 11, 15, biased	**Variable** length, no bias
Mantissa	Fixed length 1.d…d	**Variable** length 1.d…d
Infinities	$-\infty$ $+\infty$	$\pm\infty$
NaNs	Many (lost of bit combinations)	One

New format – IEEE 754 vs. POSIT

PARAMETER	IEEE 754	POSITS
Word Length	16, 32, 64, 128	Fixed overall length but with dynamic exponent and fraction
Sign	Sign/Magnitude (-0 vs. +0)	Two's complement (one 0)
Exponent	Fixed length 5, 8, 11, 15, biased	**Variable** length, no bias
Mantissa	Fixed length 1.d…d	**Variable** length 1.d…d
Infinities	$-\infty$ $+\infty$	$\pm\infty$
NaNs	Many (lost of bit combinations)	One
Underflow	Gradual (subnormals)	Gradual (natural)

New format – IEEE 754 vs. POSIT

PARAMETER	IEEE 754	POSITS
Word Length	16, 32, 64, 128	Fixed overall length but with dynamic exponent and fraction
Sign	Sign/Magnitude (-0 vs. +0)	Two's complement (one 0)
Exponent	Fixed length 5, 8, 11, 15, biased	**Variable** length, no bias
Mantissa	Fixed length 1.d…d	**Variable** length 1.d…d
Infinities	$-\infty$ $+\infty$	$\pm\infty$
NaNs	Many (lost of bit combinations)	One
Underflow	Gradual (subnormals)	Gradual (natural)
Overflow	Yes	No, only 1 / 0 = ∞

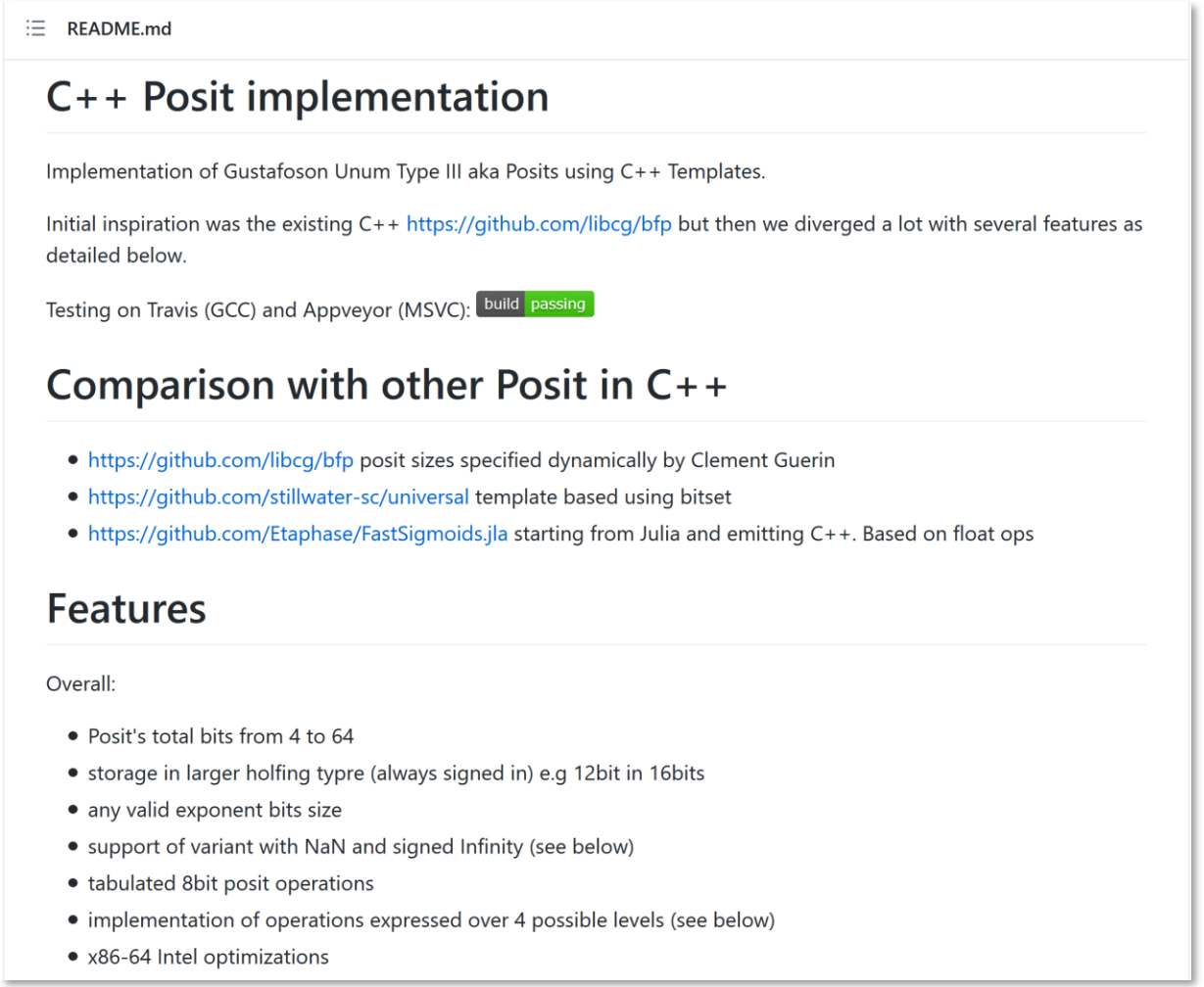
New format – IEEE 754 vs. POSIT

PARAMETER	IEEE 754	POSITS
Word Length	16, 32, 64, 128	Fixed overall length but with dynamic exponent and fraction
Sign	Sign/Magnitude (-0 vs. +0)	Two's complement (one 0)
Exponent	Fixed length 5, 8, 11, 15, biased	**Variable** length, no bias
Mantissa	Fixed length 1.d…d	**Variable** length 1.d…d
Infinities	$-\infty$ $+\infty$	$\pm\infty$
NaNs	Many (lost of bit combinations)	One
Underflow	Gradual (subnormals)	Gradual (natural)
Overflow	Yes	No, only 1 / 0 = ∞
Base	2, 10	2, 4, 16, 256, …

New format – POSIT Arithmetic

Software implementations of Posits:

- SoftPosit – a library developed by the Next Generation Arithmetic Committee.
- Beyond Floating Point – an older Posit library (*https://github.com/libcg/bfp*).
- StillWater – a modern C++ template based library (*github.com/stillwater-sc/universal*).
- **cppPosit** – also a modern C++ rich of features, defining 4 operational levels of different efficiency/features ratios. Used in many applications, mostly for operations of the **convolutional deep neural networks** (for many architectures 12-16 bit Posits can be a perfect replacement for the Float32).



README.md

C++ Posit implementation

Implementation of Gustafoson Unum Type III aka Posits using C++ Templates.

Initial inspiration was the existing C++ https://github.com/libcg/bfp but then we diverged a lot with several features as detailed below.

Testing on Travis (GCC) and Appveyor (MSVC): build passing

Comparison with other Posit in C++

- https://github.com/libcg/bfp posit sizes specified dynamically by Clement Guerin
- https://github.com/stillwater-sc/universal template based using bitset
- https://github.com/Etaphase/FastSigmoids.jla starting from Julia and emitting C++. Based on float ops

Features

Overall:

- Posit's total bits from 4 to 64
- storage in larger holfing typre (always signed in) e.g 12bit in 16bits
- any valid exponent bits size
- support of variant with NaN and signed Infinity (see below)
- tabulated 8bit posit operations
- implementation of operations expressed over 4 possible levels (see below)
- x86-64 Intel optimizations

https://github.com/eruffaldi/cppPosit

New format – POSIT Arithmetic

Examples:

```
#include "posit12.hpp"
```

```
// es == 2
posit12 a( 20.0 );
posit12 b( -10.0 );

std::cout << "a = " << (float) a << std::endl;
std::cout << "1/a = " << (float) ( posit12( 1. ) / a ) << std::endl;
```

```
a = 20
1/a = 0.0498047
```

New format – POSIT Arithmetic

Examples:

```
#include "posit12.hpp"
```

```
// es == 2
posit12 a( 20.0 );
posit12 b( -10.0 );

std::cout << „b = " << (float) b << std::endl;
std::cout << "1/b = " << (float) ( posit12( 1. ) / b ) << std::endl;
```

```
b = -10
1/b = -0.0996094
```

New format – POSIT Arithmetic

Examples:

```
#include "posit12.hpp"
```

```
// es == 2
posit12 a( 20.0 );
posit12 b( -10.0 );

std::cout << "a*b = " << (float) ( a * b ) << std::endl;
std::cout << "b/a = " << (float) ( b / a ) << std::endl;
```

```
a*b = -200
b/a = -0.5
```

New format – POSIT Arithmetic

Examples:

```
#include "posit12.hpp"
```

```
posit12 r( 12.13 );
const posit12 kPi( 3.14159 );

std::cout << "area_p = " << (float) ( kPi * r * r ) << std::endl;
std::cout << "area_f = " <<  (float)kPi * (float)r * (float)r << std::endl;
```

```
area_p = 456
area_f = 461.721
```

1.2% loss of accuracy @ 62.5% data compression

New format – POSIT Arithmetic

Examples:

```
#include "posit12.hpp"
```

We must always control dynamics & precision!

```
std::cout << "std::numeric_limits< posit12 >::min = "
      << (float) std::numeric_limits< posit12 >::min() << std::endl;
std::cout << "std::numeric_limits< posit12 >::max = "
      << (float) std::numeric_limits< posit12 >::max() << std::endl;
std::cout << "std::numeric_limits< posit12 >::epsilon = " <<
      (float) std::numeric_limits< posit12 >::epsilon() << std::endl;
```

```
std::numeric_limits< posit12 >::min = 9.09495e-13
std::numeric_limits< posit12 >::max = 1.09951e+12
std::numeric_limits< posit12 >::epsilon = 0.0078125
```

Literature & Conclusions

Literature & Conclusions

- *https://en.wikipedia.org/wiki/Floating-point_arithmetic*
- *https://en.wikipedia.org/wiki/Half-precision_floating-point_format*
- *https://en.wikipedia.org/wiki/Bfloat16_floating-point_format*
- *http://www.mrob.com/pub/math/floatformats.html*
- *http://www.mrob.com/pub/math/altnum.html*
- Web IEEE 754 calculator: *http://weitz.de/ieee/*

Literature & Conclusions

- D. E. Knuth, *The Art of Computer Programming*, Volume 2, Seminumerical Algorithms. Addison-Wesley, Boston, MA, USA, 1997.
- W. Kahan, *Further Remarks on Reducing Truncation Errors*. Communications of the ACM, Vol. 8, No. 1, 1965, pp. 40.
- **J. L. Gustafson and I. T. Yonemoto. Beating floating point at its own game: Posit arithmetic. Supercomputing Frontiers and Innovations , 4(2):71–86, 2017.**
- P. Elias. Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory , 21(2):194–203, 1975.
- D. Goldberg, *What every computer scientist should know about floating-point arithmetic*. CSUR, 1991, pp. 5–48.
- Y.-K. Zhu and W. B. Hayes, *Algorithm 908: Online Exact Summation of Floating-Point Streams*. ACM Transactions on Mathematical Software, Vol. 37, No. 3, 2010, pp. 1–13.

Literature & Conclusions

- 754-2008 - *IEEE Standard for Floating-Point Arithmetic*, DOI: 10.1109/IEEESTD.2008.5976968, ISBN: 978-0-7381-6981-1, 2008.
- GMP a library for arbitrary precision arithmetic (https://gmplib.org/).
- ttmath – A library for math operations on big integer/floating point numbers (www.ttmath.org).
- **Peter Lindstrom. *Fixed-Rate Compressed Floating-Point Arrays*. IEEE Transactions on Visualization and Computer Graphics, 20(12):2674-2683, December 2014.**
- Cococcioni, Marco & Rossi, Federico & Ruffaldi, Emanuele & Dinechin, Benoît. Novel Arithmetics in Deep Neural Networks Signal Processing for Autonomous Driving: Challenges and Opportunities. IEEE Signal Processing Magazine. 38, Jan. 2021.

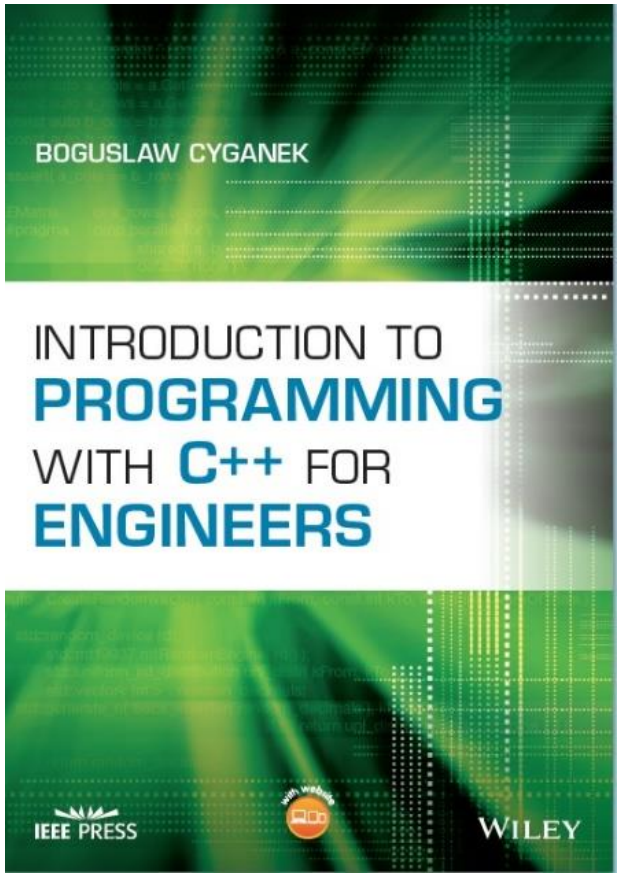
Literature & Conclusions

- Cyganek B., Wiatr K.: *How orthogonal are we? A note on fast and accurate inner product computation in the floating-point arithmetic*. Int. Conference Societal Automation Technological & Architectural Frameworks, 2019.
- Grabek J. & Cyganek B. : *An impact of tensor-based data compression methods on the training and prediction accuracy of the popular deep convolutional neural network architectures*, FedCSIS'21.
- **Cyganek B.: *Introduction to Programming with C++ for Engineers*, Wiley, 2020.**

Literature & Conclusions

- We all are indebted to the IEEE 754 standard! The entire modern numerical world depends on it! But we can do more.
- **We must always control dynamics & precision of the numbers**!
- New technologies launch new demands on data representation:
  - Machine learning & Artificial Intelligence (*Deep learning!*);
  - Big data – **data transfer constitutes a bottleneck** of the contemporary computations;
- New directions:
  - Data **compressions** (mostly lossy, but everything is an approximation…);
  - New FP formants – 16 bit FP, but most of all **Posits**!
  - We need hardware integration/implementations for new FP representations (FPGA, GPU, microprocessors, RISC-V …, direct IEEE 754 replacement?).
  - We are still ahead of having new formats integrated into the systems (What with old software?)

Thank you!

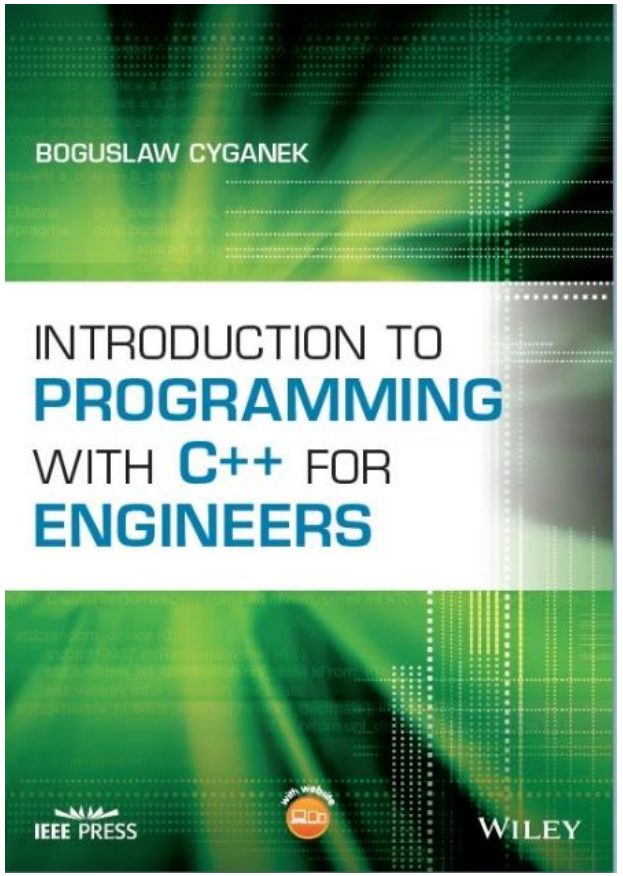


The new book for beginners and advanced programmers:

Cyganek B.: ***Introduction to Programming with C++ for Engineers*****, Wiley, 2020**

https://github.com/BogCyg/BookCpp

Thank you!



The new book for beginners and advanced programmers:

Cyganek B.: ***Introduction to Programming with C++ for Engineers*****, Wiley, 2020**

https://github.com/BogCyg/BookCpp

Literature & Conclusions