GWDG AG-C

float, double, long double, float128, bfloat16, and all that: what to use and how to use each

Dr. Freja Nordsiek

Table of contents

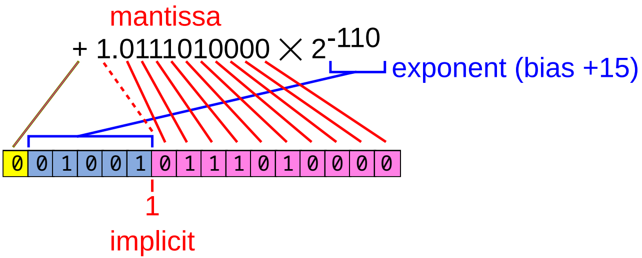
Other Number Types

- Fixed-width integers (hardware support up to 64-bit usually)
- Big-integers (software only)
- Fixed point arithmetic (fixed-width integers with a fixed divisor)
  - Most common example is financial with integers representing cents (0.01 EUR) instead of EUR
  - Very fast with simple accuracy properties
  - Poor range (can't represent very small or very large numbers)
  - Robust libraries exist (multiplication and division require care)
  - Good luck on sin, cos, $ln\Gamma(x)$, Bessel functions, etc.

Floating Point

- "Let the decimal float around"
- Store the mantissa and exponent separately
  - Example: 1.391×10^{-201}
    - mantissa is 1.391
    - exponent is -201
- Finite subset of extended-reals
- Wide magnitude range
- Approximately constant multiplicative resoltion over whole range
- Almost always base-2 (binary), not base-10

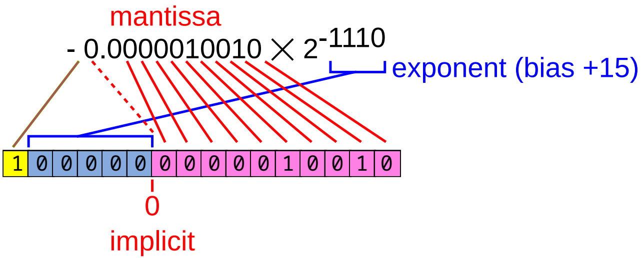
Representation



Special flag/representation values for

- zero (often mantissa and exponent zero)
- infinity (often maxed out exponent and zero mantissa)
- NaN (often maxed out exponent and non-zero mantissa)
- Subnormal numbers (often zero exponent and non-zero mantissa)

Representation – Subnormal



- Represent numbers smaller than can be done with a leading 1
- Loss of precision
- Not supported by all formats or all hardware
  - e.g. some hardware doesn't support them in `bfloat16`

Common Problems

- Mantissa issues
  - Catastrophic cancellation
  - Bigger mantissas reduce FLOPS and increase size
- Range issues
  - Overflow to $\pm\infty$
  - Underflow to 0
  - Loss of precision at small magnitudes (subnormals)
  - Bigger exponents reduce FLOPS and increase size
- $+0$ vs. -0
- NaN

Formats – Representation

Type	Width (k)	Impl. Mant. Bit	Mantissa (t)	Exponent (w)	Exp. Bias (e_{bias})	Bin. Dig.(p)	e_{max}	e_{min}
binary16 (IEEE)	16	yes	10	5	15	11	15	-14
binary32 (IEEE)	32	yes	23	8	127	24	127	-126
binary64 (IEEE)	64	yes	52	11	1023	53	1023	-1022
binary128 (IEEE)	128	yes	112	15	16383	113	16383	-16382
binary{k} (IEEE)	k	yes	$k - \mathrm{rnd}(4 \log_2 k)$	$k - t - 1$	$2^{2-1} - 1$	$t + 1$	e_{bias}	$1 - e_{max}$
bfloat16	16	yes	7	8	127	8	127	-126
x87 FPU 80-bit	80	no	64	15	16383	64	16383	-16382

- IEEE Floating-Point Working Group, “IEEE Standard for Floating-Point Arithmetic”, 2019
- Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture*, 2022
- Other formats exist like double-double

Formats – Capabilities

Type	Decimal Digits	Largest Finite	Smallest Positive Normal	Smallest Positive Subnormal
`binary16`	3.311	6.55×10^{4}	6.10×10^{-5}	5.96×10^{-8}
`binary32`	7.225	3.40×10^{38}	1.18×10^{-38}	1.40×10^{-45}
`binary64`	15.95	1.80×10^{308}	2.23×10^{-308}	4.94×10^{-324}
`binary128`	34.02	1.19×10^{4932}	3.36×10^{-4932}	6.48×10^{-4966}
`binary{k}`	$(t+1)\log_{10} 2$	$2^{e_{max}}(2 - 2^{1-p})$	$2^{e_{min}}$	$2^{1+e_{min}-p}$
`bfloat16`	2.408	3.39×10^{38}	1.18×10^{-38}	9.18×10^{-41}
x87 FPU 80-bit	19.27	1.19×10^{4932}	3.36×10^{-4932}	3.65×10^{-4951}

What Range Do I Need?

At Each Stage of The Calculation

- How big/small are the input numbers?
- How big/small are the output numbers?

Example (1)

$$3 \times 10^{-25} \quad \bullet \quad 1 \times 10^{-23} = 3 \times 10^{-48}$$

`binary64` enough for inputs, but not output.

Example (2)

$$3 \times 10^{5} \quad \bullet \quad 2 \times 10^{-8} = 6 \times 10^{-3}$$

`binary16` enough for output, but not inputs.

What Precision/Accuracy Do I Need?

Define your tolerances:

$$A_{apparent} = \left(1 \pm \delta_{relative}\right) A_{real} \pm \delta_{absolute}$$

where

- $\delta_{relative}$ is the max relative tolerance
- $\delta_{absolute}$ is the max absolute tolerance

Then

1. Determine the tolerances/errors in your inputs
2. Determine the desired tolerances for your outputs
3. Go through the calculation steps determining how much accuracy you need at each stage to achieve this

Addition And Subtraction

If you have only 2-digits of precision, then

$$1.0 \times 10^4 \quad + \quad 9.9 \times 10^2 \quad \rightarrow \quad 1.0 \times 10^4 \tag{1}$$

which is NO change despite adding something due to lack of digits

If your $\delta_{relative} = 0.1$:

- This is OK if it happens once
- But imagine you are summing one element of the bigger number and a million elements of the smaller – NOT SO GOOD

Multiplication And Division

Multiplying or dividing two numbers in the range can easily

- Overflow to infinity
- Underflow to zero
- Go into subnormal range and lose precision

If your format has a range of $10^{-4} - 10^{4}$

- $10^{3} \bullet 10^{3}$ overflows to $+\infty$
- $10^{-3} \bullet -10^{-3}$ underflows to -0
- $10^{-2} \bullet 10^{-2}$ is at the bottom of the range and thus loses precision

Other Operations

- May require doing research on the operation
- May require doing research on the implementation
  - e.g. not every libc give correctly rounded sin, cos, etc. for each precision or over the whole range
- Some hardware can do Fused-Multipy-Add ($ax + b$) in hardware correctly rounded
- Sometimes possible to determine by testing
  - Brute force for small parameter spaces
  - If you can figure out the worst case inputs that would cause the most problems

Return to The Table And Determine Which Are Sufficient

Type	Decimal Digits	Largest Finite	Smallest Positive Normal	Smallest Positive Subnormal
`binary16`	3.311	6.55×10^{4}	6.10×10^{-5}	5.96×10^{-8}
`binary32`	7.225	3.40×10^{38}	1.18×10^{-38}	1.40×10^{-45}
`binary64`	15.95	1.80×10^{308}	2.23×10^{-308}	4.94×10^{-324}
`binary128`	34.02	1.19×10^{4932}	3.36×10^{-4932}	6.48×10^{-4966}
`binary{k}`	$(t+1)\log_{10} 2$	$2^{e_{max}}(2 - 2^{1-p})$	$2^{e_{min}}$	$2^{1+e_{min}-p}$
`bfloat16`	2.408	3.39×10^{38}	1.18×10^{-38}	9.18×10^{-41}
x87 FPU 80-bit	19.27	1.19×10^{4932}	3.36×10^{-4932}	3.65×10^{-4951}

Pick Fastest One/s on Your Hardware

Smaller is usually faster

- SIMD/Vector instructions operate on wider vectors
- Faster convergence for expensive operations (e.g. sin, cos, $\ln\Gamma(x)$, etc.)
- Each memory read/write operation gets/puts more elements
- Cache can hold more elements
- Arrays more likely to fit entirely in TLB

Exceptions are:

- Pure software implementation when there is no hardware support (e.g. `binary128` on x86-64)
- Hardware vendor might not be prioritizing performance on a format (e.g. x87 FPU 80-bit on x86-64)
- Conversion penalties if you have to convert formats

CPU Availability

Type	Intel x86-64	AMD x86-64	ARM 64 (aarch64)
`binary16`	full since Sapphire Rapids (convert only since Ivy Bridge)	convert only since Jaguar	many/most
`binary32`	all	all	all
`binary64`	all	all	all
`binary128`	software only	software only	software only
`binary{k}`	software if you write it	software if you write it	software if you write it
`bfloat16`	convert only since Sapphire Rapids	convert only since Zen 5	some
x87 FPU 80-bit	all	all	software if you write it

`binary128` hardware support only found in

- IBM Power 9 and newer
- IBM Z series and s/390 since G5 in 1998 (mainframes)

RISC-V has it defined in the Q extension, but no hardware implements that yet.

GPU Availability

Type	NVIDIA	AMD	Intel
binary16	compute capability $\geq$ 5.3 (Pascal)	since GCN 5 (Vega)	since Gen 8 (Broadwell)
binary32	all	all	all
binary64	all	all	since Gen 7 (Ivy Bridge)
binary128	none	none	none
binary{k}	none	none	none
bfloat16	compute capability $\geq$ 8.0 (Ampere)	since CDNA 1 and RDNA 3	since Gen 12.5 (Ponte Vecchio)
x87 FPU 80-bit	none	none	none

Many 8-bit floating point formats are beginning to come into use on GPUs

Tricks

- Use different formats at different stages
  - Gains must be greater than the conversion penalties
- Transform problem into a different form/space
  - log-space is often better for mult/div. over a wide range
- Doing pencil-paper math to convert equations into forms more suitable for floating point
- Don't use naive numerical algorithms
  - e.g. use LU decomposition rather than matrix inversion
- Try the calculations with different precisions to see if you didn't make mistakes in your analysis

Alternative names

Type	Alternative Names
`binary16`	`float16`, `fp16`, `f16`
`binary32`	`float32`, `fp32`, `f32`
`binary64`	`float64`, `fp64`, `f64`
`binary128`	`float128`, `fp128`, `f128`, quad
`binary{k}`	
`bfloat16`	`bf16`, `bfp16`
x87 FPU 80-bit	FPU float, x87 float

half/short, single/float, double, long/long double

- Many programming use these terms but don't rigourously define them so as to support a wide range of hardware.
- Most of them require for both the precision and range that:

$$\text{half}/\text{short} \leq \text{single}/\text{float} \leq \text{double} \leq \text{long}/\text{longdouble}$$

- In many, it is perfectly valid for the OS/environment and/or compiler to choose that
  - `long double` to be the same as `double`
  - `long double` and `double` to be the same as `float` (e.g. Arduino Uno)
  - No half/short at all
  - short to be the same as `float`
- Many make no or few guarantees as to the format
  - e.g. `long double` could be `binary64`, x87 FPU 80-bit, `binary128`, or double-double

C

C has few restricitons on `float`, `double`, `long double`

Usually,

C Type	x86-64	ARM 64	IBM Power	RISC-V
`float`	`binary32`	`binary32`	`binary32`	`binary32`
`double`	`binary64`	`binary64`	`binary64`	`binary64`
`long double`	x87 FPU 80-bit	`binary128` (pure software)	`binary128` since Power 9, double-double before	`binary128` (pure software so far)

C Annex X

Optional support for specifying the format exactly in Annex X

Format	Type in C	Literal suffix	Good <float.h> preprocessor macro
binary16	_Float16	f16	FLT16_MIN
binary32	_Float32	f32	FLT32_MIN
binary64	_Float64	f64	FLT64_MIN
binary128	_Float128	f128	FLT128_MIN
binary{k}	_Float{k}	f{k}	FLT{k}_MIN
bfloat16	not yet	not yet	not yet
x87 FPU 80-bit	often _Float64x	often f64x	often FLT64X_MIN

You must define __STDC_WANT_IEC_60559_TYPES_EXT__ before including <float.h> to get the preprocessor definitions:

```c
#define __STDC_WANT_IEC_60559_TYPES_EXT__

#include <float.h>
```

C - Example

```c
#define __STDC_WANT_IEC_60559_TYPES_EXT__

#include <float.h>
#include <stdio.h>
void main()
{
#ifdef FLT128_MIN
    if (1.0f128 + 2.0f128 >= 2.5f128)
        printf("binary128 math works!\n");
    else
        printf("binary128 math present but wrong!\n");
#else
    printf("binary128 not present.\n");
#endif
}
```

```
binary128 math works!
```

C++

C++ has few restricitons on `float`, `double`, `long double`

Usually,

C++ Type	x86-64	ARM 64	IBM Power	RISC-V
`float`	`binary32`	`binary32`	`binary32`	`binary32`
`double`	`binary64`	`binary64`	`binary64`	`binary64`
`long double`	x87 FPU 80-bit	`binary128` (pure software)	`binary128` since Power 9, double-double before	`binary128` (pure software so far)

C++23

Can now pick format explicitly

`#include <stdfloat>`

Format	Type in C++23	Literal Suffix	Preprocessor macro
binary16	std::float16_t	f16 or F16	__STDCPP_FLOAT16_T__
binary32	std::float32_t	f32 or F32	__STDCPP_FLOAT32_T__
binary64	std::float64_t	f64 or F64	__STDCPP_FLOAT64_T__
binary128	std::float128_t	f128 or F128	__STDCPP_FLOAT128_T__
binary{k}	not yet	not yet	not yet
bfloat16	std::bfloat16_t	bf16 or BF16	__STDCPP_BFLOAT16_T__
x87 FPU 80-bit	none	none	none

C++ - Example

```cpp
#include <stdio.h>
#include <stdfloat>
int main()
{
#ifdef __STDCPP_FLOAT128_T__
    if (1.0f128 + 2.0f128 >= 2.5f128)
        printf("binary128 math works!\n");
    else
        printf("binary128 math present but wrong!\n");
#else
    printf("binary128 not present.\n");
#endif
    return 0;
}
```

```
binary128 math works!
```

Fortran - Requirement Selecting

`SELECTED_REAL_KIND(P, R)`

Since Fortran 90

- selects some floating point format with precision of at least P base-10 digits whose exponent can reach R (e.g. 1.4×10^{R}).
- Result is negative if the compiler can't provide any

```
PROGRAM test
    INTEGER, PARAMETER :: myreal = SELECTED_REAL_KIND(2, 4)
    REAL(KIND=myreal) :: x, y, z

    x = 2.1_myreal
    y = 1.3_myreal

    z = x * y
    PRINT *, myreal, z
END PROGRAM test
```

```
4     2.72999978
```

Fortran - Requirement Selecting - IEEE 754 Only

`IEEE_SELECTED_REAL_KIND(P, R)`

- In the `IEEE_ARITHMETIC` module since Fortran 2003
- Works same way but restricted to IEEE 754 types

```fortran
PROGRAM test
    USE IEEE_ARITHMETIC

    INTEGER, PARAMETER :: myreal = IEEE_SELECTED_REAL_KIND(2, 4)
    REAL(KIND=myreal) :: x, y, z

    x = 2.1_myreal
    y = 1.3_myreal

    z = x * y
    PRINT *, myreal, z
END PROGRAM test
```

```
4    2.72999978
```

Fortran - Fixed Size Floating Point

`ISO_FORTRAN_ENV` Module

Module introduced in Fortran 2003, but floating point types introduced later

Size (bits)	Kind	Fortran Standard
16	REAL16	2023
32	REAL32	2008
64	REAL64	2008
128	REAL128	2008

- If not supported, Kind is -2 if a larger size is supported and -1 otherwise
- Could be any format that has the right size

The End

- Many floating point formats available
- Many languages provide more than one
- Important to pick the best one
- Languages can make it easy or hard to pick the format you want
- Hardware support is faster than pure software implementation

References

IEEE Floating-Point Working Group. “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019). Ed. by Mike Cowlishaw, pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture*. Order Number: 253665-077US. Apr. 2022. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.