High-Performance Decimal Floating-Point Arithmetic: Algorithms and Implementation in C++

MATTHEW BORLAND* and CHRISTOPHER KORMANYOS*

This article presents a comprehensive, portable C++ implementation of decimal floating-point arithmetic conforming to IEEE 754-2019 standards[8]. Our system offers three IEEE 754-compliant types, and three additional types that prioritize performance over strict standard adherence, providing flexible options for various computational needs. We describe in detail the Decimal system architecture, its standard library, and usage guidelines. The implementation incorporates novel algorithms for key operations, significantly improving performance in common use cases. Rigorous testing results demonstrate the system's correctness and IEEE 754 compliance. Performance benchmarks show competitive or superior results compared to existing implementations. This work addresses the growing demand for precise decimal arithmetic in financial, scientific, and engineering applications, offering a robust, efficient solution for C++ developers.

CCS Concepts: • **General and reference** → **Design**; **Performance**; • **Mathematics of computing** → **Mathematical software performance**; • **Software and its engineering** → *Software architectures.*

Additional Key Words and Phrases: C++, Decimal Floating Point, Object Oriented, Special Functions, System Architecture

ACM Reference Format:
Matthew Borland and Christopher Kormanyos. 2018. High-Performance Decimal Floating-Point Arithmetic: Algorithms and Implementation in C++. *ACM Trans. Math. Softw.* 37, 4, Article 111 (August 2018), 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Decimal floating-point arithmetic is crucial for many applications[4], particularly in financial and scientific computing. While several C++ decimal floating-point packages exist, they often lack IEEE 754[8] conformance, interoperability with the C++ Standard Template Library (STL), or both, and have limited portability. This paper presents a novel decimal system that addresses these limitations and advances decimal floating-point technology. Our system is standalone, relies on a minimal subset of the C++ STL, and has been tested on a wide range of devices, from S390X mainframes to AVR boards. It provides seamless interoperability with existing C++ standard types and full IEEE 754 conformance, features not collectively offered by any known package. This work significantly enhances the toolset available for high-precision decimal computations across diverse computing environments.

*Both authors contributed equally to this research.

Authors' address: Matthew Borland, matt@mattborland.com; Christopher Kormanyos, e_float@yahoo.com.

ACM 0098-3500/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

2 THE DECIMAL SYSTEM

2.1 Background

Decimal floating point is a method of representing floating point numbers using base-10 instead of base-2 like most are familiar with. Decimal floating point is not a new concept per se. Mechanical devices such as the abacus and slide rule use base-10 numbers. Early computers such as the IBM 650 used decimal floating point numbers prior to IEEE standardization[2]. Some modern architectures support decimal floating point in hardware such as IBM System Z[15]. The advantage of decimal floating point is that it can represent human readable floating point numbers exactly unlike binary floating point.

The major difference between the layout of binary and decimal floating-point numbers lies in their internal structure and how they represent the same numerical value. Figure 1 illustrates this difference by showing the bit layout of the same number in both formats.

Binary floating-point numbers consist of three parts:

- The sign bit (shown in blue)
- The exponent (shown in green)
- The significand (shown in red)

Decimal floating-point numbers, on the other hand, are composed of four distinct parts:

- The sign bit (shown in blue)
- The combination field (shown in orange)
- The exponent continuation (shown in green)
- The coefficient continuation (shown in red)

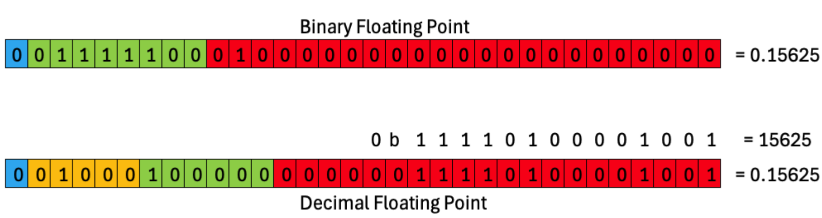

Fig. 1. Bit layouts of Binary and Decimal Floating Point Numbers

As depicted in Figure 1, these different structures allow the same numerical value to be represented in two fundamentally different ways. The binary format uses a base-2 system, while the decimal format uses a base-10 system, which affects how precisely certain decimal values can be represented and how rounding errors propagate in calculations. Due to these differences decimal can represent numbers exactly that binary cannot, for example the number 0.3 which with a C++ `double` would actually be: 0.30000000000000004.

Another idiosyncrasy that is found in decimal floating point types are called cohorts[8]. A cohort is all the different ways to represent the same value. For example one can represent the number 10 as: 0.1×10^2, 1×10^1, 10×10^0, 100×10^{-1}, and so on. The ramification of this is we must normalize the significand and exponents of a number before we can attempt doing comparisons or other basic mathematical operations.

2.2 Provided Types

The decimal system provides 6 total types: `decimal32`, `decimal64`, `decimal128`, `decimal32_fast`, `decimal64_fast`, and `decimal128_fast`. The first three types are conformant to IEEE-754 standards while the latter three provided identical numerical results without the constraints of IEEE-754.

The three conformant types is the design specifications provided in IEEE-754 namely their properties:

Parameter	**decimal32**	**decimal64**	**decimal128**
Storage Width	32	64	128
Precision (decimal digits)	7	16	34
Max Exponent	96	384	6144
Exponent Bias	101	398	6176
Sign Width	1	1	1
Combination Field Width	11	13	17
Significand Continuation Width	20	50	110

Table 1. Decimal Floating-Point Format Parameters

The three so-called fast types (i.e. `decimalXX_fast`) have the same values of precision, range, and exponent as their analogous conformant type, but require more space.

2.3 System Architecture

The decimal system architecture is robust and flexible. Each type is implemented completely independently of each other, but they share many of the same function implementations from the STL.

2.4 Using the System

Every effort was made during design and implementation to ensure that using the decimal system was straightforward and intuitive. The library contains zero dependencies, and uses only a small subsection of the C++ STL. The required language standard for the library is C++14.

```
#include <boost/decimal.hpp>
#include <iostream>
#include <iomanip>

int main()
{
    using namespace boost::decimal;

    constexpr decimal32 val_1 {100};          // Construction from an integer
    constexpr decimal32 val_2 {10, 1};        // Construction from an integer and exponent
    constexpr decimal32 val_3 {1, 2, false}; // Construction from an integer, exponent, and sign

    std::cout << "Val_1: " << val_1 << '\n'
              << "Val_2: " << val_2 << '\n'
```

```
              << "Val_3: " << val_3 << '\n';

    if (val_1 == val_2 && val_2 == val_3 && val_1 == val_3)
    {
        std::cout << "All equal values" << std::endl;
    }

    constexpr decimal64 val_4 {decimal64{2, -1} + decimal64{1, -1}};
    constexpr double float_val_4 {0.2 + 0.1};
    const decimal64 val_5 { float_val_4 }; // Explicit Conversion from double

    std::cout << std::setprecision(17) << "Val_4: " << val_4 << '\n'
              <<  "Float: " << float_val_4 << '\n'
              <<  "Val_5: " << val_5 << '\n';

    if (val_4 == val_5)
    {
        std::cout << "Floats are equal" << std::endl;
    }
    else
    {
        std::cout << "Floats are not equal" << std::endl;
    }

    return 0;
}
```

Listing 1. Basic Usage

In Listing 1 we show how to construct the type and that it works just like a built-in flowing point type. The following is an example that leverages one of decimal-floating points main strengths and one that we expect many people will use.

```
#include <boost/decimal.hpp>
#include <iostream>
#include <cassert>

int main()
{
    using namespace boost::decimal;

    decimal64 val {0.25}; // Construction from a double (not recommended but explicit construction is allowed)

    char buffer[256];
    auto r_to = to_chars(buffer, buffer + sizeof(buffer) - 1, val);
    assert(r_to); // checks std::errc()
    *r_to.ptr = '\0';

```

```
    decimal64 return_value;
    auto r_from = from_chars(buffer, buffer + std::strlen(buffer), return_value);
    assert(r_from);

    assert(val == return_value);

    std::cout << " Initial Value: " << val << '\n'
              << "Returned Value: " << return_value << std::endl;

    return 0;
}
```

Listing 2. Basic Usage

In Listing 2 we show how to serialize and parse numbers. These techniques and functions are an extension of Boost.Charconv[3].

3 IMPLEMENTATION DETAILS

3.1 IEEE 754 Conformant Decimal Types

The decimal system provides three IEEE-754 compliant types: `decimal32`, `decimal64`, and `decimal128`. Each of these are constructed using a single unsigned integer of the same width to hold the bits. For example `decimal32` consists of a single `std::uint32_t`. `decimal128` uses a custom implementation of a 128-bit unsigned integer for portability reasons rather than relying on the existence of `unsigned __int128`. The discussion of implementing big integers is outside the scope of the paper, but papers and implementations can be found in numerous places[3][12][10].

The decoding step of `decimal32` is somewhat involved because the values of the significand and exponent depend on the bits in the combination field. Our first case is where the bits in the combination field are in the form `00XXX`, `01XXX`, and `10XXX`.

```
//    Comb.  Exponent          Significand
// s 00 TTT (00)eeeeee (0TTT)[ssssssssss][ssssssssss]
// s 01 TTT (01)eeeeee (0TTT)[ssssssssss][ssssssssss]
// s 10 TTT (10)eeeeee (0TTT)[ssssssssss][ssssssssss]
```

Listing 3. Combination Field Pattern 1

In Listing 3 we can see that with these combination field bit patterns the first two bits of the combination field are appended to the front of the exponent, the next three bits are appended to the front of the significand. The second case is where the bits in the combination field are in the form `1100X`, `1101X`, and `1110X`.

```
//    Comb.  Exponent          Significand
// s 1100 T (00)eeeeee (100T)[ssssssssss][ssssssssss]
// s 1101 T (01)eeeeee (100T)[ssssssssss][ssssssssss]
// s 1110 T (10)eeeeee (100T)[ssssssssss][ssssssssss]
```

Listing 4. Combination Field Pattern 2

In Listing 4 we can see that with these combination field bit patterns there is now an implied leading bit to the significand, followed by `00T` bits. Bits 2 and 3 are now what represents the leading bits of the exponent as opposed to 0 and one like in Listing 3.

The final case is where the bits in the combination field are in the form `1111X` which is used for non-finite numbers:

```
//    Comb.  Exponent          Significand
// s 1111 0 eeeeee [ssssssssss][ssssssssss] = inf
// s 1111 1 eeeeee [ssssssssss][ssssssssss] = qnan
// s 1111 1 1eeeee [ssssssssss][ssssssssss] = snan
```

Listing 5. Combination Field Pattern 3

Since this combination field pattern is only for non-finite numbers it makes the implementation of `isnan`, `isinf`, etc. straightforward. We can also see the signbit remains unused so we can represent signed infinities. One can also use the remaining exponent and significand bits to provide a payload to NaN analogous to the use of `std::nan`[6].

Now that we have seen what each bit pattern of the combination field corresponds to; the full implementation of decoding the significand then becomes:

```
constexpr auto decimal32::full_significand() const noexcept -> significand_type
{
    significand_type significand {};

    if ((bits_ & detail::d32_comb_11_mask) == detail::d32_comb_11_mask)
    {
        // Only need the one bit of T because the other 3 are implied
        significand = (bits_ & detail::d32_comb_11_significand_bits) == detail::d32_comb_11_significand_bits ?
        static_cast<std::uint32_t>(0b1001'0000000000'0000000000) :
        static_cast<std::uint32_t>(0b1000'0000000000'0000000000);
    }
    else
    {
        // Last three bits in the combination field, so we need to shift past the exp field
        // which is next
        significand |= (bits_ & detail::d32_comb_00_01_10_significand_bits) >> detail::d32_exponent_bits;
    }

    significand |= (bits_ & detail::d32_significand_mask);

    return significand;
}
```

Listing 6. Decoding `decimal32` significand

Encoding the value is a similar to the decoding process found in Listing 6 , but more complex process in that we need to work backwards through the steps to figure out what the value of the combination field needs to be,and then we use a series of masks to write the value.

3.2 IEEE 754 Fast Types

Now that we have discussed the three IEEE-754 conformant types we will turn our attention to much more performant, but non-compliant types. The library offers: `decimal32_fast`, `decimal64_fast`, and `decimal128_fast`. These types are similar to how instead of `std::uint32_t` you can use `std::uint_fast32_t` which on x86_64 platforms generally aliases to `std::uint64_t`. This is the classic optimization of trading space for time. Again we will focus on the 32-bit type for clarity and simplicity. Unlike `decimal32` consisting of a data `std::uint32_t`, `decimal32_fast` actually contains its internal state in a structure.

```
class decimal32_fast final
{
public:
    using significand_type = std::uint_fast32_t;
    using exponent_type = std::uint_fast8_t;
    using biased_exponent_type = std::int_fast32_t;

private:
    // In regular decimal32 we have to decode the 24 bits of the significand and the 8 bits of the exp
    // Here we just use them directly at the cost of at least 2 extra bytes of internal state
    // since the fast integer types will be at least 32 and 8 bits respectively

    significand_type significand_ {};
    exponent_type exponent_ {};
    bool sign_ {};

    // Continued
};
```

Listing 7. Internal state of `decimal32_fast`

In Listing 7 we see that now instead of having to encode and decode the number every time we perform an operation we can now just access the value directly:

```
constexpr auto full_significand() const noexcept -> significand_type
{
    return significand_;
}
```

Listing 8. Decoding `decimal32_fast` significand

Now compare the decoding process found in the conformant type in Listing 6 to that in Listing 8. Encoding the value is similarly trivial. Each of the fast types use the same approach, but with different types used for the internal state to ensure that the range and precision is equal to that of the IEEE 754 conformant types.

3.3 Standard Library

3.3.1 Special Functions. The implementation of transcendental, and C++17 mathematical special functions[5][9] extensively uses Páde Approximations and Remez Polynomials.

4 RESULTS

4.1 Testing and Precision

4.2 Performance

As this system is implemented in software it will never be as performant as binary floating point is on computing systems with floating-point hardware. For example the table below shows the runtime difference for comparison operations between `float`, `double`, and the types provided by this system:

Type	Runtime (μs)	Ratio to double
`float`	8,587	1.376
`double`	6,240	1.000
`decimal32`	275,597	44.166
`decimal64`	296,929	47.587
`decimal32_fast`	99,664	15.972
`decimal64_fast`	102,132	16.367

Table 2. Comparison of runtime and ratio for comparison operations

The gap in performance increases even further for the multiplication operation which is typically only a few cycles in hardware[1]:

Type	Runtime (μs)	Ratio to double
`float`	1,646	0.957
`double`	1,720	1.000
`decimal32`	313,219	182.104
`decimal64`	583,818	339.429
`decimal32_fast`	86,093	50.054
`decimal64_fast`	333,582	193.943

Table 3. Comparison of runtime and ratio for multiplication operations

For floating-point operations that are implemented in software like parsing and serializing numbers the performance gap is quite smaller:

Type	Runtime (μs)	Ratio to double
`float`	235,816	0.953
`double`	247,307	1.000
`decimal32`	366,682	1.483
`decimal64`	485,965	1.965

Table 4. Comparison of runtime and ratio for `from_chars`

5 CONCLUSION AND OUTLOOK

The portable C++ decimal system has been presented. The system allows for users to easily employ decimal-floating point numbers in their existing code bases. For the first time a complete and interoperable system has been fully provided.

Type	Runtime (μs)	Ratio to double
float	316,300	1.040
double	304,272	1.000
decimal32	406,053	1.335
decimal64	678,451	2.230

Table 5. Comparison of runtime and ratio for to_chars

During the course of our research and implementation we have limited ourselves to 32, 64, and 128-bit types. Further research can be conducted into making higher precision types as the properties of such numbers are specified in IEEE 754. We could also expand our range of compatibility to allow the types to be massively parallelized such as providing support for CUDA devices.

ACKNOWLEDGMENTS

To the C++ Alliance for sponsoring the development of this library.

REFERENCES

[1] ARM Limited. 2023. *ARM Architecture Reference Manual.* ARM Limited. https://developer.arm.com/documentation/ddi0487/latest ARM DDI 0487J.a (ID050623).

[2] Nelson H. F. Beebe. 2017. Historical floating-point architectures. In *The Mathematical-Function Computation Handbook - Programming Using the MathCW Portable Software Library* (1 ed.). Springer International Publishing AG, Salt Lake City, UT, USA, Chapter H, 948. https://doi.org/10.1007/978-3-319-64110-2

[3] Matt Borland, Peter Dimov, Junekey Jeon, Alexander Grund, Andrzej Krzemieński, Dmitry, Vinnie Falco, and Sam Darwin. 2024. *boostorg/charconv: Boost 1.86.0.* https://doi.org/10.5281/zenodo.13323694

[4] Michael F. Cowlishaw. 2003. Decimal Floating-Point: Algorism for Computers. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic.* IEEE, 104–111. https://doi.org/10.1109/ARITH.2003.1207670

[5] cppreference.com contributors. 2024. Special mathematical functions. https://en.cppreference.com/w/cpp/numeric/special_functions. https://en.cppreference.com/w/cpp/numeric/special_functions Accessed: 2024-08-23.

[6] cppreference.com contributors. 2024. std::nan, std::nanf, std::nanl. https://en.cppreference.com/w/cpp/numeric/math/nan. https://en.cppreference.com/w/cpp/numeric/math/nan Accessed: 2024-08-23.

[7] John F. Hart, E. W. Cheney, Charles L. Lawson, Hans J. Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thacher, and Christoph Witzgall. 1968. *Computer Approximations.* John Wiley & Sons, New York.

[8] IEEE. 2019. *IEEE Standard for Floating-Point Arithmetic.* Standard IEEE 754-2019. IEEE Computer Society, New York, NY, USA. https://doi.org/10.1109/IEEESTD.2019.8766229

[9] ISO/IEC. 2023. *Programming Languages – C++.* Standard ISO/IEC 14882:2023. International Organization for Standardization, Geneva, Switzerland. Sixth edition.

[10] Donald E. Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. 1-4B. Addison-Wesley, Reading, MA.

[11] Christopher Kormanyos. 2011. Algorithm 910: A Portable C++ Multiple-Precision System for Special-Function Calculations. *ACM Transactions on Mathematical Software (TOMS)* 37, 4, Article 45 (feb 2011), 27 pages. https://doi.org/10.1145/1916461.1916469

[12] Daniel Lemire. 2021. Number Parsing at a Gigabyte per Second. *Software: Practice and Experience* 51, 8 (2021), 1766–1785. https://doi.org/10.1002/spe.2914

[13] Jean-Michel Muller. 2016. *Elementary Functions: Algorithms and Implementation* (3 ed.). Birkhäuser, Boston. https://doi.org/10.1007/978-1-4899-7983-4

[14] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic* (2 ed.). Birkhäuser, Cham, Switzerland. https://doi.org/10.1007/978-3-319-76526-6

[15] Eric M Schwarz, John M Kapernick, and Mike F Cowlishaw. 2009. Decimal floating-point support on the IBM System z10 processor. *IBM Journal of Research and Development* 53, 1 (2009), 4:1–4:10. https://doi.org/10.1147/JRD.2009.5388557

Received XX February XXXX; revised XX March XXXX; accepted X June XXXX