Jean-Michel Muller
Nicolas Brunie
Florent de Dinechin
Claude-Pierre Jeannerod
Mioara Joldes
Vincent Lefèvre
Guillaume Melquiond
Nathalie Revol
Serge Torres

Handbook of Floating-Point Arithmetic

Second Edition

Birkhäuser

Jean-Michel Muller • Nicolas Brunie
Florent de Dinechin • Claude-Pierre Jeannerod
Mioara Joldes • Vincent Lefèvre
Guillaume Melquiond • Nathalie Revol
Serge Torres

Handbook of Floating-Point Arithmetic

Second Edition

Birkhäuser

Jean-Michel Muller
CNRS - LIP
Lyon, France

Florent de Dinechin
INSA-Lyon - CITI
Villeurbanne, France

Mioara Joldes
CNRS - LAAS
Toulouse, France

Guillaume Melquiond
Inria - LRI
Orsay, France

Serge Torres
ENS-Lyon - LIP
Lyon, France

Nicolas Brunie
Kalray
Grenoble, France

Claude-Pierre Jeannerod
Inria - LIP
Lyon, France

Vincent Lefèvre
Inria - LIP
Lyon, France

Nathalie Revol
Inria - LIP
Lyon, France

ISBN 978-3-319-76525-9 ISBN 978-3-319-76526-6 (eBook)
https://doi.org/10.1007/978-3-319-76526-6

Library of Congress Control Number: 2018935254

Mathematics Subject Classification: 65Y99, 68N30

Printed on acid-free paper

This book is published under the imprint Birkhäuser, www.birkhauser-science.com by the registered company Springer International Publishing AG part of Springer Nature.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

List of Figures

List of Tables

Preface

FLOATING-POINT ARITHMETIC is by far the most widely used way of approximating real-number arithmetic for performing numerical calculations on modern computers. A rough presentation of floating-point arithmetic requires only a few words: a number x is represented in radix β floating-point arithmetic with a sign s, a significand m, and an exponent e, such that $x = s \times m \times \beta^e$. Making such arithmetic reliable, fast, and portable is however a very complex task. Although it could be argued that, to some extent, the concept of floating-point arithmetic (in radix 60) was invented by the Babylonians, or that it is the underlying arithmetic of the slide rule, its first modern implementation appeared in Konrad Zuse's Z1 and Z3 computers.

A vast quantity of very diverse arithmetics was implemented between the 1960s and the early 1980s. The radix (radices 2, 4, 8, 16, and 10, and even radix 3, were then considered), and the sizes of the significand and exponent fields were not standardized. The approaches for rounding and for handling underflows, overflows, or "forbidden operations" (such as $5/0$ or $\sqrt{-3}$) were significantly different from one machine to another. This lack of standardization made it difficult to write reliable and portable numerical software.

Pioneering scientists including Brent, Cody, Kahan, and Kuki highlighted the relevant key concepts for designing an arithmetic that could be both useful for programmers and practical for implementers. These efforts resulted in the IEEE 754-1985 standard for radix-2 floating-point arithmetic. The standardization process was expertly orchestrated by William Kahan. The IEEE 754-1985 standard was a key factor in improving the quality of the computational environment available to programmers. A new version, the IEEE 754-2008 standard, was released in August 2008. It specifies radix-2 and radix-10 floating-point arithmetic. At the time of writing these lines, a working group is preparing a new version of IEEE 754, which should be released in 2018. It will not be very different from the 2008 version.

By carefully specifying the behavior of the arithmetic operators, the IEEE 754-1985 and 754-2008 standards allowed researchers and engineers to design extremely powerful yet portable algorithms, for example, to compute very accurate sums and dot products, and to formally prove some critical parts of

programs. Unfortunately, the subtleties of the standard are little known by the nonexpert user. Even more worrying, they are sometimes overlooked by compiler designers. As a consequence, floating-point arithmetic is sometimes conceptually misunderstood and is often far from being exploited to its full potential.

This led us to the decision to compile into a book selected parts of the vast knowledge of floating-point arithmetic. This book is designed for programmers of numerical applications, compiler designers, programmers of floating-point algorithms, designers of arithmetic operators (floating-point adders, multipliers, dividers, ...), and more generally students and researchers in numerical analysis who wish to more accurately understand a tool that they manipulate on an everyday basis. During the writing, we tried, whenever possible, to illustrate the techniques described by an actual program, in order to allow a more direct practical use for coding and design.

The first part of the book presents the history and basic concepts of floating-point arithmetic (formats, exceptions, correct rounding, etc.) and various aspects of the 2008 version of the IEEE 754 standard. The second part shows how the features of the standard can be used to develop effective and nontrivial algorithms. This includes summation algorithms, and division and square root relying on a fused multiply-add. This part also discusses issues related to compilers and languages. The third part then explains how to implement floating-point arithmetic, both in software (on an integer processor) and in hardware (VLSI or reconfigurable circuits). It also surveys the implementation of elementary functions. The fourth part presents some extensions: complex numbers, interval arithmetic, verification of floating-point arithmetic, and extension of the precision. In the Appendix, the reader will find an introduction to relevant number theory tools and a brief presentation of the standards that predated IEEE 754-2008.

Acknowledgments

Some of our colleagues around the world, in academia and industry, and several students from École normale supérieure de Lyon and Université de Lyon greatly helped us by discussing with us, giving advice and reading preliminary versions of this edition, or by giving comments on the previous one that (hopefully) helped us to improve it: Nicolas Brisebarre, Jean-Yves L'Excellent, Warren Ferguson, Christian Fleck, John Harrison, Nicholas Higham, Tim Leonard, Dan Liew, Nicolas Louvet, David Lutz, Peter Markstein, Marc Mezzarobba, Kai Torben Ohlhus, Antoine Plet, Valentina Popescu, Guillaume Revy, Siegfried Rump, Damien Stehlé, and Nick Trefethen. We thank them all for their suggestions and interest.

Our dear colleague and friend Peter Kornerup left us recently. The computer arithmetic community misses one of its pioneers, and we miss a very kind and friendly person.

Grenoble, France	Nicolas Brunie
Villeurbanne, France	Florent de Dinechin
Lyon, France	Claude-Pierre Jeannerod
Toulouse, France	Mioara Joldes
Lyon, France	Vincent Lefèvre
Orsay, France	Guillaume Melquiond
Lyon, France	Jean-Michel Muller
Lyon, France	Nathalie Revol
Lyon, France	Serge Torres

Part I

Introduction, Basic Definitions, and Standards

Chapter 1

Introduction

REPRESENTING AND MANIPULATING real numbers efficiently is required in many fields of science, engineering, finance, and more. Since the early years of electronic computing, many different ways of approximating real numbers on computers have been introduced. One can cite (this list is far from being exhaustive): fixed-point arithmetic, logarithmic [337, 585] and semi-logarithmic [444] number systems, intervals [428], continued fractions [349, 622], rational numbers [348] and possibly infinite strings of rational numbers [418], level-index number systems [100, 475], fixed-slash and floating-slash number systems [412], tapered floating-point arithmetic [432, 22], 2-adic numbers [623], and most recently unums and posits [228, 229].

And yet, floating-point arithmetic is by far the most widely used way of representing real numbers in modern computers. Simulating an infinite, continuous set (the real numbers) with a finite set (the "machine numbers") is not a straightforward task: preserving all properties of real arithmetic is not possible, and clever compromises must be found between, e.g., speed, accuracy, dynamic range, ease of use and implementation, and memory cost. It appears that floating-point arithmetic, with adequately chosen parameters (radix, precision, extremal exponents, etc.), is a very good compromise for most numerical applications.

We will give a complete, formal definition of floating-point arithmetic in Chapter 3, but roughly speaking, a radix-β, precision-p, floating-point number is a number of the form

$$\pm m_0.m_1m_2\cdots m_{p-1}\times\beta^e,$$

where e, called the *exponent*, is an integer, and $m_0.m_1m_2\cdots m_{p-1}$, called the *significand*, is represented in radix β. The major purpose of this book is to explain how these numbers can be manipulated efficiently and safely.

J.-M. Muller et al., *Handbook of Floating-Point Arithmetic*,
https://doi.org/10.1007/978-3-319-76526-6_1

1.1 Some History

Even if the implementation of floating-point arithmetic on electronic computers is somewhat recent, floating-point arithmetic itself is an old idea. In *The Art of Computer Programming* [342], Knuth presents a short history. He views the radix-60 number system of the Babylonians as some kind of early floating-point system. That system allowed the Babylonians to perform arithmetic operations rather efficiently [498]. Since the Babylonians did not invent the number zero, if the ratio of two numbers is a power of 60, then their representation in the Babylonian system is the same. In that sense, the number represented is the *significand* of a radix-60 floating-point representation.

A famous tablet from the Yale Babylonian Collection (YBC 7289) gives an approximation to $\sqrt{2}$ with four sexagesimal places (the digits represented on the tablet are 1, 24, 51, 10). A photo of that tablet can be found in [633], and a very interesting analysis of the Babylonian mathematics used for computing square roots, related to YBC 7289, was carried out by Fowler and Robson [205].

Whereas the Babylonians invented the *significands* of our floating-point numbers, one may reasonably argue that Archimedes invented the *exponents*: in his famous treatise Arenarius (The Sand Reckoner) he invents a system of naming very large numbers that, in a way, "contains" an exponential representation [259]. The notation a^n for $a \times a \times a \times \cdots \times a$ seems to have been coined much later by Descartes (it first appeared in his book *La Géométrie* [169]).

The arithmetic of the slide rule, invented around 1630 by William Oughtred [632], can be viewed as another kind of floating-point arithmetic. Again, as with the Babylonian number system, we only manipulate significands of numbers (in that case, radix-10 significands).

The two modern co-inventors of floating-point arithmetic are probably Quevedo and Zuse. In 1914 Leonardo Torres y Quevedo described an electromechanical implementation of Babbage's Analytical Engine with floating-point arithmetic [504]. Yet, the first real, modern implementations of floating-point arithmetic were in Konrad Zuse's Z1 [514] and Z3 [92] computers. The Z3, built in 1941, used a radix-2 floating-point number system, with 15-bit significands (stored on 14 bits, using the *leading bit convention*, see Section 2.1.2), 7-bit exponents, and a 1-bit sign. It had special representations for infinities and indeterminate results. These characteristics made the real number arithmetic of the Z3 much ahead of its time.

The Z3 was rebuilt recently [515]. Photographs of Konrad Zuse and the Z3 can be viewed at http://www.konrad-zuse.de/.

Readers interested in the history of computing devices should have a look at the excellent books by Aspray et al. [20] and Ceruzzi [93].

When designing a floating-point system, the first thing one must think about is the choice of the radix β. Radix 10 is what humans use daily for representing numbers and performing paper and pencil calculations. Therefore,

to avoid input and output radix conversions, the first idea that springs to mind for implementing automated calculations is to use the same radix.

And yet, since most of our computers are based on two-state logic, radix 2 (and, more generally, radices that are a power of 2) is by far the easiest to implement. Hence, choosing the right radix for the internal representation of floating-point numbers was not obvious. Indeed, several different solutions were explored in the early days of automated computing.

Various early machines used a radix-8 floating-point arithmetic: the PDP-10 and the Burroughs 570 and 6700 for example. The IBM 360 had a radix-16 floating-point arithmetic (and on its mainframe computers, IBM still offers hexadecimal floating-point, along with more conventional radix-2 and radix-10 arithmetics [213, 387]). Radix 10 has been extensively used in financial calculations[1] and in pocket calculators, and efficient implementation of radix-10 floating-point arithmetic is still a very active domain of research [87, 89, 116, 121, 122, 124, 191, 614, 613, 627, 628]. The computer algebra system Maple also uses radix 10 for its internal representation of the "software floats." It therefore seems that the various radices of floating-point arithmetic systems that have been implemented so far have almost always been either 10 or a power of 2.

There was a very odd exception. The Russian SETUN computer, built at Moscow State University in 1958, represented numbers in radix 3, with digits -1, 0, and 1 [630]. This "balanced ternary" system has several advantages. One of them is the fact that rounding to a nearest number the sum or product of two numbers is equivalent to truncation [342]. Another one [250] is the following. Assume you use a radix-β fixed-point system, with p-digit numbers. A large value of β makes the implementation complex: the system must be able to "recognize" and manipulate β different symbols. A small value of β means that more digits are needed to represent a given number: if β is small, p has to be large. To find a compromise, we can try to minimize $\beta \times p$, while having the largest representable number $\beta^p - 1$ (almost) constant. The optimal solution[2] will almost always be $\beta = 3$. See http://www.computer-museum.ru/english/setun.htm for more information on the SETUN computer.

Johnstone and Petry have argued [306] that radix 210 could be a sensible choice because it would allow exact representation of many rational numbers.

Various studies (see references [63, 104, 352] and Chapter 2) have shown that radix 2 with the *implicit leading bit convention* gives better worst-case and average accuracy than all other radices. This and the ease of implementation explain the current prevalence of radix 2.

[1] For legal reasons, financial calculations frequently require special rounding rules that are very tricky to implement if the underlying arithmetic is binary: this is illustrated in [320, Section 2].

[2] If p and β were real numbers, the value of β that would minimize $\beta \times p$ while letting β^p be constant would be $e = 2.7182818\cdots$.

The world of numerical computation changed a great deal in 1985, when the IEEE 754-1985 Standard for Binary Floating-Point Arithmetic was published [12]. This standard specifies various formats, the behavior of the basic operations and conversions, and exception conditions. As a matter of fact, the Intel 8087 mathematics co-processor, built a few years before (in 1980) to be paired with the Intel 8088 and 8086 processors, was already extremely close to what would later become the IEEE 754-1985 standard. And the HP35 pocket calculator (a landmark: it was the machine that killed the slide rule!), launched in 1972, already implemented related ideas. Now, most systems of commercial significance offer compatibility[3] with IEEE 754-1985 or its successor IEEE 754-2008. This has resulted in significant improvements in terms of accuracy, reliability, and portability of numerical software. William Kahan [553] played a leading role in the conception of the IEEE 754-1985 standard and in the development of smart algorithms for floating-point arithmetic. His website[4] contains much useful information. He received the Turing award in 1989.

IEEE 754-1985 only dealt with radix-2 arithmetic. Another standard, released in 1987, the IEEE 854-1987 Standard for Radix Independent Floating-Point Arithmetic [13], was devoted to both binary (radix-2) and decimal (radix-10) arithmetic.

In 1994, a number theorist, Thomas Nicely, who was working on the twin prime conjecture,[5] noticed that his Pentium-based computer delivered very inaccurate results when performing some divisions. The reason was a flaw in the choice of tabulated constants needed by the division algorithm [108, 183, 437]. For instance, when dividing 4195835.0 by 3145727.0, one would get 1.333739068902 instead of 1.3338204491. This announcement of a "Pentium FDIV bug" provoked a great deal of discussion at the time but has had very positive long-term effects: most arithmetic algorithms used by the manufacturers are now published (for instance a few years after, the division algorithm used on the Intel/HP Itanium [120, 242] was made public) so that everyone can check them, and everybody understands that a particular effort must be made to build formal proofs of the arithmetic algorithms and their implementation [240, 243].

A revision of the standard, which replaced both IEEE 754-1985 and 854-1987, was adopted in 2008 [267]. That IEEE 754-2008 standard brought significant improvements. It specified the Fused Multiply-Add (FMA) instruction—which makes it possible to evaluate $ab + c$, where a, b, and c are floating-point numbers, with one final rounding only. It resolved some ambiguities in IEEE 754-1985, especially concerning expression evaluation and exception

[3] Even if sometimes you need to dive into the compiler documentation to find the right options; see Chapter 6.

[4] http://www.cs.berkeley.edu/~wkahan/.

[5] That conjecture asserts that there are infinitely many pairs of prime numbers that differ by 2.

handling. It also included "quadruple precision" (now called binary128 or decimal128), and recommended (yet did not mandate) that some elementary functions should be correctly rounded (see Chapter 10).

At the time of writing these lines, the IEEE 754 Standard is again under revision: the new standard is to be released in 2018.

1.2 Desirable Properties

Specifying a floating-point arithmetic (formats, behavior of operators, etc.) demands that one find compromises between requirements that are seldom fully compatible. Among the various properties that are desirable, one can cite:

- **Speed**: Tomorrow's weather must be computed in less than 24 hours;
- **Accuracy**: Even if speed is important, getting a wrong result right now is not better than getting the correct one too late;
- **Range**: We may need to represent large as well as tiny numbers;
- **Portability**: The programs we write on a given machine should run on different machines without requiring modifications;
- **Ease of implementation and use**: If a given arithmetic is too arcane, almost nobody will use it.

With regard to accuracy, the most accurate current physical measurements allow one to check some predictions of quantum mechanics or general relativity with a relative accuracy better than 10^{-15} [99].

This means that in some cases, we must be able to represent numerical data with a similar accuracy (which is easily done, using formats that are implemented on almost all current platforms: for instance, with the binary64 format of IEEE 754-2008, one represents numbers with relative error less than $2^{-53} \approx 1.11 \times 10^{-16}$). But this also means that we must sometimes be able to carry out long computations that end up with a relative error less than or equal to 10^{-15}, which is much more difficult. Sometimes, one will need a significantly larger floating-point format or clever "tricks" such as those presented in Chapter 4.

An example of a huge calculation that requires great care was carried out by Laskar's team at the Paris Observatory [369]. They computed long-term numerical solutions for the insolation quantities of the Earth (very long-term, ranging from -250 to $+250$ millions of years from now).

In other domains, such as number theory, some multiple-precision computations are indeed carried out using a very large precision. For instance, in