Floating-Point Arithmetic

Nicolas Brisebarre Florent de Dinechin
Claude-Pierre Jeannerod Vincent Lefèvre
Guillaume Melquiond Jean-Michel Muller Nathalie Revol
Damien Stehlé
Serge Torres
Laboratoire LIP, Projet Arénaire
CNRS, INRIA, École Normale Supérieure de Lyon
46 Allée d'Italie
69364 Lyon Cédex 07
FRANCE

March 31, 2009

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 [20, 37] and semi-logarithmic [31] number systems, continued-fractions [23, 40], rational numbers [22] and possibly infinite strings of rational numbers [27], level-index number systems [9, 33], fixed-slash and floating-slash number systems [26], 2-adic numbers [41].

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: clever compromises must be found between, e.g., speed, accuracy, dynamic range, ease of use and implementation, and memory. 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 **??**, 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.

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 [21], Donald Knuth presents a short history of floating-point arithmetic. He views the radix-60 number system of the Babylonians as some kind of early floating-point system. Since the Babylonians did not invent the 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 of w.

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 [45], and a very interesting analysis of the Babylonian mathematics related to YBC 7289 was done by Fowler and Robson [16].

The arithmetic of the slide rule, invented around 1630 by William Oughtred [44], 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 electro-mechanical implementation of Babbage's Analytical Engine with floating-point arithmetic [34]. And yet, the first real, modern implementation of floating-point arithmetic was in Konrad Zuse's Z3 computer, built in 1941 [8]. It used a radix-2 floating-point number system, with 14-bit significands, 7-bit exponents and one sign bit. The Z3 computer 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 [35]. Photographs of Konrad Zuse and the Z3 can be viewed at `http://www.computerhistory.org/projects/zuse_z23/` and `http://www.konrad-zuse.de/`.

Reader interested in the history of computing devices should have a look at the excellent book by Aspray et al. [4].

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. 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 [7, 11, 12, 13, 15, 39, 38, 42, 43]. The computer algebra system Maple also uses radix 10 for its internal representation of numbers. 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 has been a very odd exception. The Russian SETUN computer, built in Moscow University in 1958, represented numbers in radix 3, with digits -1, 0, and 1. This "balanced ternary" system has several advantages. One of them is the fact that rounding to nearest is equivalent to truncation [21]. Another one [17] 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.

Various studies (see references [6, 10, 24] and Chapter **??**) have shown that radix 2 with the *implicit leading bit convention* (see Chapter **??**) gives better worst-case or average accuracy than all other radices. This and the ease of implementation explain the current prevalence of radix 2.

The world of numerical computation changed much in 1985, when the IEEE 754-1985 Standard for Binary Floating-Point Arithmetic was released [2]. This standard specifies various formats, the behavior of the basic operations and conversions, and exceptional conditions. As a matter of fact, the Intel 8087 mathematic 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. Now, most systems of commercial significance offer compatibility[3] with IEEE 754-1985. This has resulted in significant improvements in terms of accuracy, reliability, and portability of numerical software. William Kahan 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 web page[4] contains much useful information.

IEEE 754-1985 only dealt with radix-2 arithmetic. Another standard, re-

[1]Financial calculations frequently require special rounding rules that are very tricky to implement if the underlying arithmetic is binary.

[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 $e = 2.7182818\cdots$

[3]Even if sometimes you need to dive into the compiler documentation to find the right options: see Section **??** and Chapter **??**.

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

leased in 1987, the IEEE 854-1987 Standard for Radix Independent Floating-Point Arithmetic [3], is devoted to both binary (radix-2) and decimal (radix-10) arithmetic.

IEEE 754-1985 and 854-1987 have been under revision since 2001. The new revised standard, called IEEE 754-2008 in this book, merges the two old standards and brings significant improvements. It was adopted in June 2008 [18].

1.2 Desirable Properties

Specifying a floating-point arithmetic (formats, behavior of operators, etc.) requires us to 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 about as bad as getting the correct one too late;
- **Range**: We may need to represent big as well as tiny numbers;
- **Portability**: The programs we write on a given machine must 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 close to 10^{-15}. This of course 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). But this also means that we might sometimes be able to carry out computations that must 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 smart "tricks" such as those presented in Chapter **??**.

An example of a huge calculation that requires much care was carried-out by Laskar's team at the Paris observatory [25]. 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

2002, Kanada's group computed 1241 billion decimal digits of π [5], using the two formulas

$$\begin{aligned}\pi &= 48\arctan\frac{1}{49} + 128\arctan\frac{1}{57} - 20\arctan\frac{1}{239} + 48\arctan\frac{1}{110443}\\ &= 176\arctan\frac{1}{57} + 28\arctan\frac{1}{239} - 48\arctan\frac{1}{682} + 96\arctan\frac{1}{12943}.\end{aligned}$$

These last examples are extremes. One should never forget that with 50 bits, one can express the distance from the Earth to the Moon with an error less than the thickness of a bacterium. It is very uncommon to need such an accuracy on a final result and, actually, very few physical quantities are defined that accurately.

1.3 Some Strange Behaviors

Designing efficient and reliable hardware or software floating-point systems is a difficult and somewhat risky task. Some famous bugs have been widely discussed; we recall some of them below. Also, even when the arithmetic is not flawed, some strange behaviors can sometimes occur, just because they correspond to a numerical problem that is intrinsically difficult. All this is not surprising: mapping the continuous real numbers on a finite structure (the floating-point numbers) cannot be done without any trouble.

1.3.1 Some famous bugs

- The divider of the first version of the Intel Pentium processor, released in 1994, was flawed [29, 14]. In extremely rare cases, one would get three correct decimal digits only. For instance, the computation of

  $$8391667/12582905$$

  would give $0.666869\cdots$ instead of $0.666910\cdots$.

- With release 7.0 of the computer algebra system Maple, when computing

  $$\frac{1001!}{1000!},$$

  we would get 1 instead of 1001.

- With the previous release (6.0) of the same system, when entering

  $$21474836480413647819643794$$

  you would get

  $$413647819643790)+' --.(--.($$

- Kahan [19] mentions some strange behavior of some versions of the Excel spreadsheet. They seem to be due to an attempt to mimic a decimal arithmetic with an underlying binary one.

  An even more striking behavior happens with some early versions of Excel 2007: When you try to compute

  $$65536 - 2^{-37}$$

  the displayed result is 100001. This is an error in the binary-to-decimal conversion used for displaying that result: the internal binary value is correct, if you add 1 to that result you get 65537. An explanation can be found at `http://blogs.msdn.com/excel/archive/2007/09/25/calculation-issue-update.aspx`, and a patch is available from `http://blogs.msdn.com/excel/archive/2007/10/09/calculation-issue-update-fix-available.aspx`

- Some bugs do not require any programming error: they are due to poor specifications. For instance, the Mars Climate Orbiter probe crashed on Mars in September 1999 because of an astonishing mistake: one of the teams that designed the numerical software assumed the unit of distance was the meter, while another team assumed it was the foot [1, 32].

  Very similarly, in June 1985, a space shuttle positioned itself to receive a laser beamed from the top of a mountain that was supposedly 10,000 miles high, instead of the correct 10,000 feet [1].

1.3.2 Difficult problems

Sometimes, even with a correctly implemented floating-point arithmetic, the result of a computation is far from what could be expected.

A sequence that seems to converge to a wrong limit

Consider the following example, due to one of us [28] and analyzed by Kahan [19, 30]. Let (u_n) be the sequence defined as

$$\left\{\begin{array}{lcl} u_0 & = & 2 \\ u_1 & = & -4 \\ u_n & = & 111 - \dfrac{1130}{u_{n-1}} + \dfrac{3000}{u_{n-1}u_{n-2}} \end{array}\right. . \tag{1.1}$$

One can easily show that the limit of this sequence is 6. And yet, on any system with any precision, the sequence will seem to go to 100.

For example, Table 1.1 gives the results obtained by compiling Program 1.1 and running it on a Pentium4-based workstation, using the GNU Compiler Collection (GCC) and the Linux system.

```
#include <stdio.h>

int main(void)
{
  double u, v, w;
  int i, max;

  printf("n =");
  scanf("%d",&max);
  printf("u0 = ");
  scanf("%lf",&u);
  printf("u1 = ");
  scanf("%lf",&v);
  printf("Computation from 3 to n:\n");
  for (i = 3; i <= max; i++)
    {
      w = 111. - 1130./v + 3000./(v*u);
      u = v;
      v = w;
      printf("u%d = %1.17g\n", i, v);
    }
  return 0;
}
```

Program 1.1: A C program that is supposed to compute sequence u_n using double-precision arithmetic. The obtained results are given in Table 1.1.

The explanation of this weird phenomenon is quite simple. The general solution for the recurrence

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}$$

is

$$u_n = \frac{\alpha \cdot 100^{n+1} + \beta \cdot 6^{n+1} + \gamma \cdot 5^{n+1}}{\alpha \cdot 100^n + \beta \cdot 6^n + \gamma \cdot 5^n},$$

where α, β, and γ depend on the initial values u_0 and u_1. Therefore, if $\alpha \neq 0$ then the limit of the sequence is 100, otherwise (assuming $\beta \neq 0$), it is 6. In the present example, the starting values $u_0 = 2$ and $u_1 = -4$ were chosen so that $\alpha = 0$, $\beta = -3$, and $\gamma = 4$. Therefore, the "exact" limit of u_n is 6. And yet, when computing the values u_n in floating-point arithmetic using (1.1), due to the various rounding errors, even the very first computed terms become slightly different from the exact terms. Hence, the value α corresponding to these computed terms is very tiny, but nonzero. This suffices to make the computed sequence "converge" to 100.

n	Computed value	Exact value
3	18.5	18.5
4	9.378378378378379	9.3783783783783783784
5	7.8011527377521679	7.8011527377521613833
6	7.1544144809753334	7.1544144809752493535
11	6.2744386627644761	6.2744385982163279138
12	6.2186967691620172	6.2186957398023977883
16	6.1661267427176769	6.0947394393336811283
17	7.2356654170119432	6.0777223048472427363
18	22.069559154531031	6.0639403224998087553
19	78.58489258126825	6.0527217610161521934
20	98.350416551346285	6.0435521101892688678
21	99.898626342184102	6.0360318810818567800
22	99.993874441253126	6.0298473250239018567
23	99.999630595494608	6.0247496523668478987
30	99.999999999998948	6.0067860930312057585
31	99.999999999999943	6.0056486887714202679

Table 1.1: Results obtained by running Program 1.1 on a Pentium4-based workstation, using GCC and the Linux system, compared to the exact values of sequence u_n.

The Chaotic Bank Society

Recently, Mr. Gullible went to the Chaotic Bank Society, to learn more about the new kind of account they offer to their best customers. He was told:

> You first deposit $\$e - 1$ on your account, where $e = 2.7182818\cdots$ is the base of the natural logarithms. The first year, we take $\$1$ from your account as banking charges. The second year is better for you: We multiply your capital by 2, and we take $\$1$ of banking charges. The third year is even better: We multiply your capital by 3, and we take $\$1$ of banking charges. And so on: The n-th year, your capital is multiplied by n and we just take $\$1$ of charges. Interesting, isn't it ?

Mr. Gullible wanted to secure his retirement. So before accepting the offer, he decided to perform some simulations on his own computer to see what his capital would be after 25 years. Once back home, he wrote a C program (Program 1.2).

```
#include <stdio.h>

int main(void)
{
  double account = 1.71828182845904523536028747135;
  int i;
  for (i = 1; i <= 25; i++)
    {
      account = i*account - 1;
    }
  printf("You will have $%1.17e on your account.\n", account);
}
```

Program 1.2: Mr. Gullible's C program.

On his computer (with an Intel Xeon processor, and GCC on Linux, but strange things would happen with any other equipment), he got the following result:

```
You will have $1.20180724741044855e+09 on your account.
```

So he immediately decided to accept the offer. He will certainly be sadly disappointed, 25 years later, when he realizes that he actually has around $\$0.0399$ on his account.

What happens in this example is easy to understand. If you call a_0 the amount of the initial deposit and a_n the capital after the end of the n-th year, then

$$\begin{aligned} a_n &= n! \times \left(a_0 - 1 - \frac{1}{2!} - \frac{1}{3!} - \cdots - \frac{1}{n!} \right) \\ &= n! \times \left(a_0 - (e-1) + \frac{1}{(n+1)!} + \frac{1}{(n+2)!} + \frac{1}{(n+3)!} + \cdots \right), \end{aligned}$$

so that:

- if $a_0 < e - 1$, then a_n goes to $-\infty$;
- if $a_0 = e - 1$, then a_n goes to 0;
- if $a_0 > e - 1$, then a_n goes to $+\infty$.

In our example, $a_0 = e - 1$, so the *exact* sequence a_n goes to zero. This explains why the exact value of a_{25} is so small. And yet, even if the arithmetic operations were errorless (which of course is not the case), since $e - 1$ is not exactly representable in floating-point arithmetic, the *computed* sequence will go to $+\infty$ or $-\infty$, depending on rounding directions.

Rump's example

Consider the following function, designed by Siegfried Rump in 1988 [36],

$$f(a,b) = 333.75b^6 + a^2\left(11a^2b^2 - b^6 - 121b^4 - 2\right) + 5.5b^8 + \frac{a}{2b},$$

and try to compute $f(a,b)$ for $a = 77617.0$ and $b = 33096.0$. On an IBM 370 computer, the results obtained by Rump were

- 1.172603 in single precision;
- 1.1726039400531 in double precision; and
- 1.172603940053178 in extended precision.

Anybody looking at these figures would feel that the single precision result is certainly very accurate. And yet, the exact result is $-0.8273960599\cdots$. On more recent systems, we do not see the same behavior exactly. For instance, on a Pentium4-based workstation, using GCC and the Linux system, the C program (Program 1.3) which uses double-precision computations, will return 5.960604×10^{20}, whereas its single-precision equivalent will return 2.0317×10^{29} and its double-extended precision equivalent will return $-9.38724 \times 10^{-323}$. We still get totally wrong results, but at least, the clear differences between them show that something weird is going on.

```
#include <stdio.h>
int main(void)
{
  double a = 77617.0;
  double b = 33096.0;
  double b2,b4,b6,b8,a2,firstexpr,f;
  b2 = b*b;
  b4 = b2*b2;
  b6 = b4*b2;
  b8 = b4*b4;
  a2 = a*a;
  firstexpr = 11*a2*b2-b6-121*b4-2;
  f = 333.75*b6 + a2 * firstexpr + 5.5*b8 + (a/(2.0*b));
  printf("Double precision result: $ %1.17e \n",f);
}
```

Program 1.3: Rump's example.

Bibliography

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. Steele Jr. Object-oriented units of measurement. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 384–403, New York, NY, USA, 2004. ACM Press.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754-1985, 1985.

[3] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Radix Independent Floating-Point Arithmetic*. ANSI/IEEE Standard 854-1987, 1987.

[4] W. Aspray, A. G. Bromley, M. Campbell-Kelly, P. E. Ceruzzi, and M. R. Williams. *Computing Before Computers*. Iowa State University Press, Ames, Iowa, 1990. Available at `http://ed-thelen.org/comp-hist/CBC.html`.

[5] D. H. Bailey. Some background on Kanada's recent pi calculation. Technical report, Lawrence Berkeley National Laboratory, 2003. Available at `http://crd.lbl.gov/~dhbailey/dhbpapers/dhb-kanada.pdf`.

[6] R. P. Brent. On the precision attainable with various floating point number systems. *IEEE Transactions on Computers*, C-22(6):601–607, June 1973.

[7] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough. The IBM z900 decimal arithmetic unit. In *Thirty-Fifth Asilomar Conference on Signals, Systems, and Computers*, volume 2, pages 1335–1339, November 2001.

[8] P. E. Ceruzzi. The early computers of Konrad Zuse, 1935 to 1945. *Annals of the History of Computing*, 3(3):241–262, 1981.

[9] C. W. Clenshaw and F. W. J. Olver. Beyond floating point. *Journal of the ACM*, 31:319–328, 1985.

[10] W. J. Cody. Static and dynamic numerical characteristics of floating-point arithmetic. *IEEE Transactions on Computers*, C-22(6):598–601, June 1973.

[11] M. Cornea, C. Anderson, J. Harrison, P. T. P. Tang, E. Schneider, and C. Tsen. A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. In Kornerup and Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, pages 29–37. IEEE Computer Society Conference Publishing Services, June 2007.

[12] M. F. Cowlishaw. Decimal floating-point: algorism for computers. In Bajard and Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 104–111. IEEE Computer Society Press, Los Alamitos, CA, June 2003.

[13] M. F. Cowlishaw, E. M. Schwarz, R. M. Smith, and C. F. Webb. A decimal floating-point specification. In Burgess and Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 147–154, Vail, CO, June 2001.

[14] A. Edelman. The mathematics of the Pentium division bug. *SIAM Rev.*, 39(1):54–67, 1997.

[15] M. A. Erle, M. J. Schulte, and B. J. Hickmann. Decimal floating-point multiplication via carry-save addition. In Kornerup and Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, pages 46–55. IEEE Computer Society Conference Publishing Services, June 2007.

[16] D. Fowler and E. Robson. Square root approximations in old Babylonian mathematics: YBC 7289 in context. *Historia Mathematica*, 25:366–378, 1998.

[17] B. Hayes. Third base. *American Scientist*, 89(6):490–494, November-December 2001.

[18] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008. available at `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.

[19] W. Kahan. How futile are mindless assessments of roundoff in floating-point computation? Available at `http://http.cs.berkeley.edu/~wkahan/Mindless.pdf`, 2004.

[20] N. G. Kingsbury and P. J. W. Rayner. Digital filtering using logarithmic arithmetic. *Electronic Letters*, 7:56–58, 1971. Reprinted in E. E. Swart-

zlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[21] D. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[22] P. Kornerup and D. W. Matula. Finite-precision rational arithmetic: an arithmetic unit. *IEEE Transactions on Computers*, C-32:378–388, 1983.

[23] P. Kornerup and D. W. Matula. Finite precision lexicographic continued fraction number systems. In *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1985. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[24] H. Kuki and W. J. Cody. A statistical study of the accuracy of floating-point number systems. *Communications of the ACM*, 16(14):223–230, April 1973.

[25] J. Laskar et al. A long term numerical solution for the insolation quantities of the earth. *Astronomy and Astrophysics*, 428:261–285, 2004.

[26] D. W. Matula and P. Kornerup. Finite precision rational arithmetic: Slash number systems. *IEEE Transactions on Computers*, 34(1):3–18, 1985.

[27] V. Ménissier. *Arithmétique Exacte*. PhD thesis, Université Pierre et Marie Curie, Paris, December 1994. In French.

[28] J.-M. Muller. *Arithmétique des Ordinateurs*. Masson, Paris, 1989. In French.

[29] J.-M. Muller. Algorithmes de division pour microprocesseurs: illustration à l'aide du "bug" du pentium. *Technique et Science Informatiques*, 14(8), October 1995.

[30] J.-M. Muller. A few results on table-based methods. *Reliable Computing*, 5(3):279–288, August 1999.

[31] J.-M. Muller, A. Scherbyna, and A. Tisserand. Semi-logarithmic number systems. *IEEE Transactions on Computers*, 47(2), February 1998.

[32] J. Oberg. Why the Mars probe went off course. *IEEE Spectrum*, 36(12), 1999.

[33] F. W. J. Olver and P. R. Turner. Implementation of level-index arithmetic using partial table look-up. In *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, May 1987.

[34] B. Randell. From analytical engine to electronic digital computer: the contributions of Ludgate, Torres, and Bush. *IEEE Annals of the History of Computing*, 04(4):327–341, 1982.

[35] R. Rojas, F. Darius, C. Göktekin, and G. Heyne. The reconstruction of Konrad Zuse's Z3. *IEEE Annals of the History of Computing*, 27(3):23–32, 2005.

[36] S. Rump. Algorithms for verified inclusion. In R. Moore, editor, *Reliability in Computing, Perspectives in Computing*, pages 109–126. Academic Press, New York, 1988.

[37] E. E. Swartzlander and A. G. Alexpoulos. The sign-logarithm number system. *IEEE Transactions on Computers*, December 1975. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[38] A. Vázquez. *High-Performance Decimal Floating-Point Units*. PhD thesis, Universidade de Santiago de Compostela, 2009.

[39] A. Vázquez, E. Antelo, and P. Montuschi. A new family of high performance parallel decimal multipliers. In *18th Symposium on Computer Arithmetic*, pages 195–204. IEEE, 2007.

[40] J. E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8), 1990.

[41] J. E. Vuillemin. On circuits and numbers. *IEEE Transactions on Computers*, 43(8):868–879, August 1994.

[42] L.-K. Wang and M. J. Schulte. Decimal floating-point division using Newton-Raphson iteration. In *Application-specific Systems, Architectures and Processors*, pages 84–95. IEEE, 2004.

[43] L.-K. Wang and M. J. Schulte. Decimal floating-point adder and multifunction unit with injection-based rounding. In Kornerup and Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, pages 56–65. IEEE Computer Society Conference Publishing Services, June 2007.

[44] Wikipedia. Slide rule — wikipedia, the free encyclopedia, 2008. [Online; accessed 25-August-2008].

[45] Wikipedia. Square root of 2 — wikipedia, the free encyclopedia, 2008. [Online; accessed 25-August-2008].